Real Time Multiple Fluid Simulation and Rendering on GPUs

Dunfan Lu

Somerville College

University of Oxford

Supervised by: Joe Pitt-Francis

3rd Year Project Report for

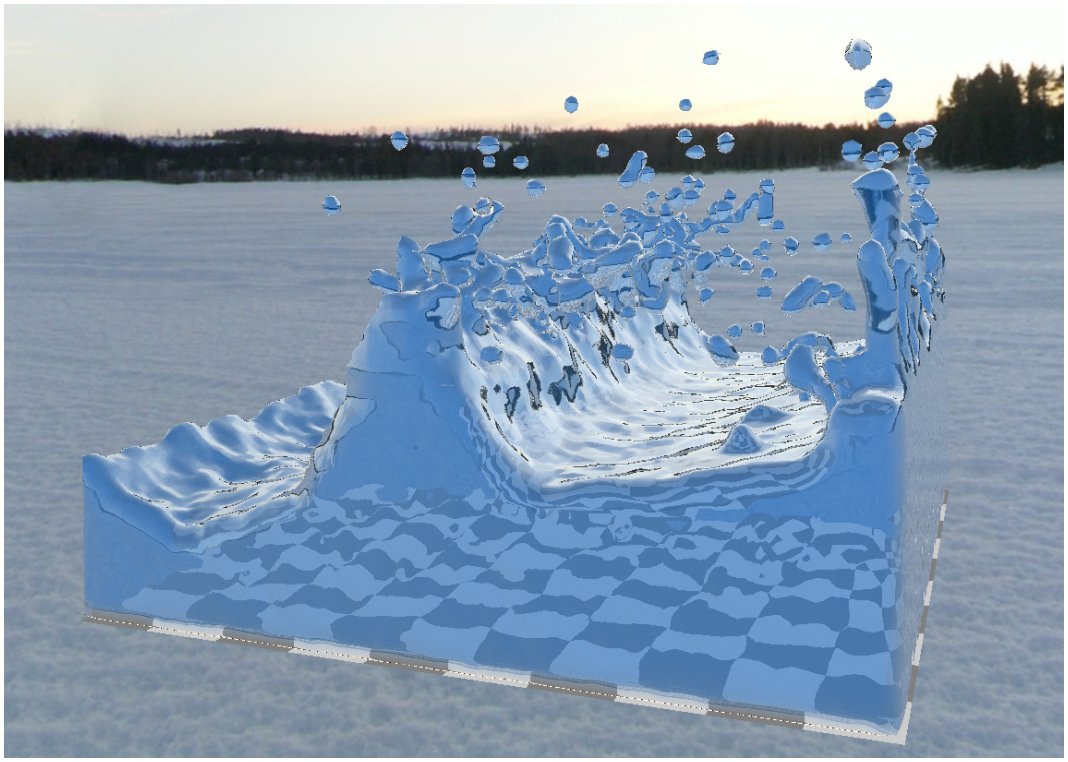
MCompSci

Trinity 2020

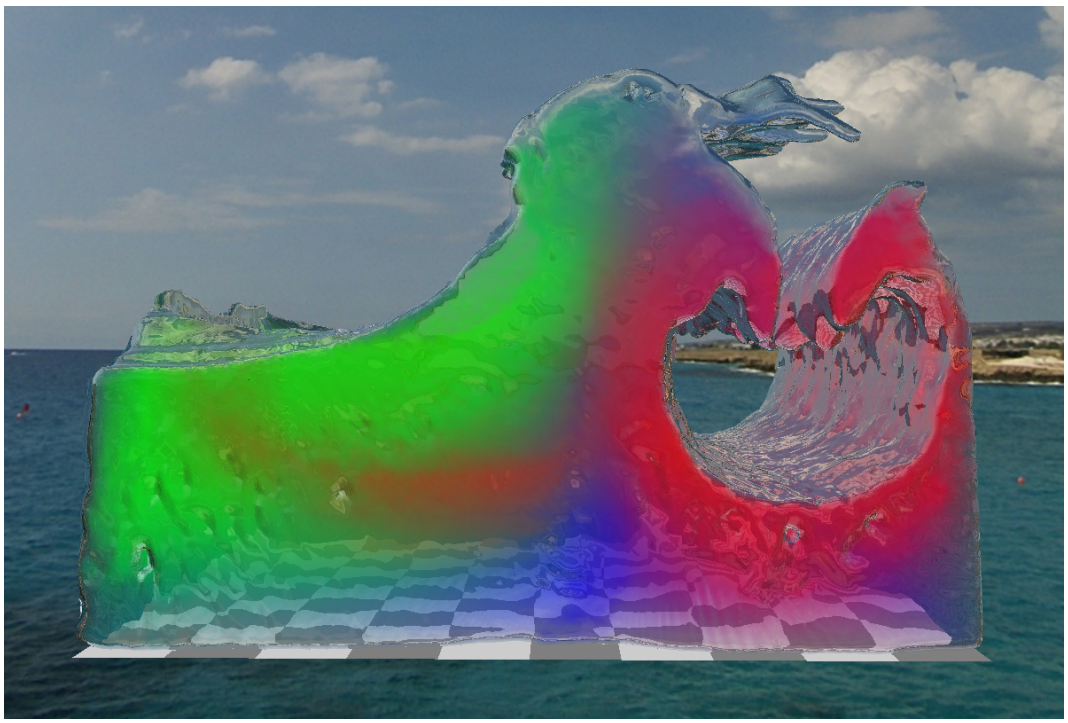
Abstract

Fluid simulation is an extremely common and important computational task. These simulations are often heavily expensive and require a large amount of CPU time, hence difficult to apply in real time applications. Fortunately, modern GPUs, equipped with massively parallel general purpose computing architectures, provide a solution to this problem. This project explores methods to perform fluid simulations on GPUs, and to realistically render the simulated fluids, both in real time.

This project focuses on two of the most widely used fluid simulation algorithm: FLIP (Fluid-Implicit-Particle) and PBF (Position Based Fluids). The project studies how each algorithm can be parallelized, and creates efficient GPU implementations of these algorithms using NVIDIA's CUDA programming model. This project also extends the FLIP algorithm to support multiphase fluid simulation, thereby capturing the diffusion phenomenon between different fluids of different colors and densities. Alongside the simulation, a real time liquid rendering scheme is implemented, which includes a novel algorithm for rendering the varying concentrations of colored fluids inside the volume of liquid. A program that integrates all of these algorithms was created, which accepts user-provided simulation parameters, and performs the simulation in real time while rendering and displaying realistic animations.



(a) A PBF simulation



(b) A multi-fluid FLIP simulation

Figure 1: Screenshots of the fluid simulated and rendered by the software created in this project. More images can be found at `https://github.com/AmesingFlank/Aquarius`

Contents

1 Introduction

1.1 Motivation

Fluids can be seen everywhere. The smoke rising from the chimney and spreading in the wind, the milk in a cup mixing with the coffee, the calm flow of a river with tiny ripples under the rain, and the violent waves of the ocean shooting up and splashing down. Many of these these phenomenons have stunning visual effects, and quite often, realistic images of these fluids need to be computationally generated for purposes such as cinematics and video games.

Due to the mathematical complexity underlying the motion of fluids, accurate numerical simulations often require a huge amount of computation time. However, real time computer graphics applications, such as video games, usually require the simulation to be computed in approximately the same amount of time as the physical process it represents. Moreover, in these applications, it is often needed to also realistically render and display the results of simulation (i.e, the shape and motion of the fluid) to the user. This project studies how these demands can be met on modern machines – by utilizing the parallel computing abilities of GPUs.

1.2 Related Work

The study of the behavior of fluids dates back to 18th century, when Leonhard Euler proposed a set of partial differential equations (PDEs), known as *Euler Equations*, that governs the behavior of an idealized incompressible and inviscid fluid. In the 19th century, these equations were extended by Claude-Louis Navier and George Gabriel Stokes into the famous *Navier-Stokes Equations*, which describe a much wider class of fluids that occur in the real world. These equations are explained in greater details in chapter 2, and they are exactly what most fluid simulation softwares, including the one implemented in this project, are trying to solve.

Somewhat unfortunately, the Euler and Navier-Stokes equations have extremely difficult mathematical properties, and general analytical solutions are yet to be found

even today. As a result, softwares resort to numerical methods to approximate solutions. In computer graphics applications, there are two main families of numerical methods for solving the fluid equations: the grid-based methods and the particle-based methods. Each approach comes with its own benefits and drawbacks, but could both be implemented efficiently on GPUs to achieve real time simulation.

The grid-based methods relies on spatial discretizations of the scalar and vector fields that represent the fluids. The most widely used discretization method, known as the **MAC** (Marker and Cell) **grid**, was proposed by Harlow and Welch [5] in 1965. This scheme offers second order accuracy, and is used as a basis of most grid-based fluid simulation algorithms.

A significantly important step during a grid-based simulation is to move all the physical quantities stored in the grid (e.g concentration) according to the velocity field. This step, known as *advection*, essentially determines how the shape of the fluid evolves over time, thus is key to a high-fidelity simulation. A few popular advection algorithms include **MacCormack**[14] and **BFECC**[7], both of which have efficient GPU implementations[4][17]. This project chooses to implement the advection algorithm known as **FLIP** (Fluid Implicit Particle)[18], developed by Zhu and Bridson. This algorithm, interestingly enough, makes uses of particles to move quantities within the MAC grid. FLIP has various advantages over the purely grid-based algorithms, and is likely the most widely used advection method nowadays.

As an addition to the traditional single phase fluid simulation, Kang et el.[6] showed how to extend the grid-based algorithms to capture the diffusion between multiple miscible fluid phases (e.g red ink spreading in transparent water). This project implements a modified version of the proposed algorithm, where FLIP, rather than BFECC, is used to advect the concentration of different fluid phases.

Apart from grid-based simulations, there is also a family of particle-based fluid simulation algorithms called **SPH** (Smoothed Particle Hydrodynamics), which does not rely on a grid. Originally developed for astronomical simulations by Lucy[9] and Monaghan [11] in 1977, SPH was introduced to computer graphics in 2003 by Müller[12]. The SPH method represent the fluid by a moving cloud of particles, which carry the physical quantities of the fluid with them. This project chooses to study and implement an extension of SPH, called **PBF**(Position Based Fluids), developed by Macklin and Müller[10] in 2013. This extended algorithm improve upon the plain SPH in that it enforces the incompressibility constraint of fluids, which is important for visual fidelity.

On the rendering side, this project follows the proposal by Zhu and Bridson[18], who showed how a particle representation of a fluid can be used to compute a signed distance field, which represents the distance to the fluid surface of each point in the 3D space. An algorithm known as Marching Cubes, invented by Wyvill[16] and Lorensen[8], can then use this field to reconstruct the surface of the fluid into a triangle mesh representation, which is suitable for rendering. Alternatively, screen space algorithms[15] can be used, which does not generate a triangle mesh and directly uses the particles for rendering.

1.3 Project Outline

This project focuses on investigating and producing highly performant GPU implementations of the most widely used fluid simulation and rendering algorithms. An extended version of the FLIP algorithm, which supports multiple fluid simulation and diffusions between fluids of different colors, is studied and implemented, with the details elaborated in chapter 4. Similarly, GPU versions of the PBF algorithm is also created, as described in chapter 5.

To visualize the simulations, the project implements a fast surface reconstruction algorithm, which transforms a particle cloud representation of fluids into a renderable triangle mesh. A real time renderer is implemented to render the mesh while capturing all the reflection and refraction phenomenons that occur in the real world. Furthermore, the renderer incorporates a novel algorithm that computes the different levels of attenuation of light caused by fluids of different colors, thereby also realistically rendering the liquid diffusion effects. The details of the renderer are given in chapter 6.

These implementations are based from their original descriptions in the papers, but many additional considerations and optimizations were taken to enable efficient parallelization. Specifically, the project utilizes NVIDIA's general purpose GPU programming interface known as CUDA, and tailors the implementation code to exploit the full potential of CUDA GPUs. The results are showcased by a fully featured program, which allows the user to easily configure the starting state of a simulation. These include the shapes and sizes of the fluid before the simulation starts, as well as the initial color and transparency of each fluid volume. The program can then carry out the simulation and render realistic results to the user in real time.

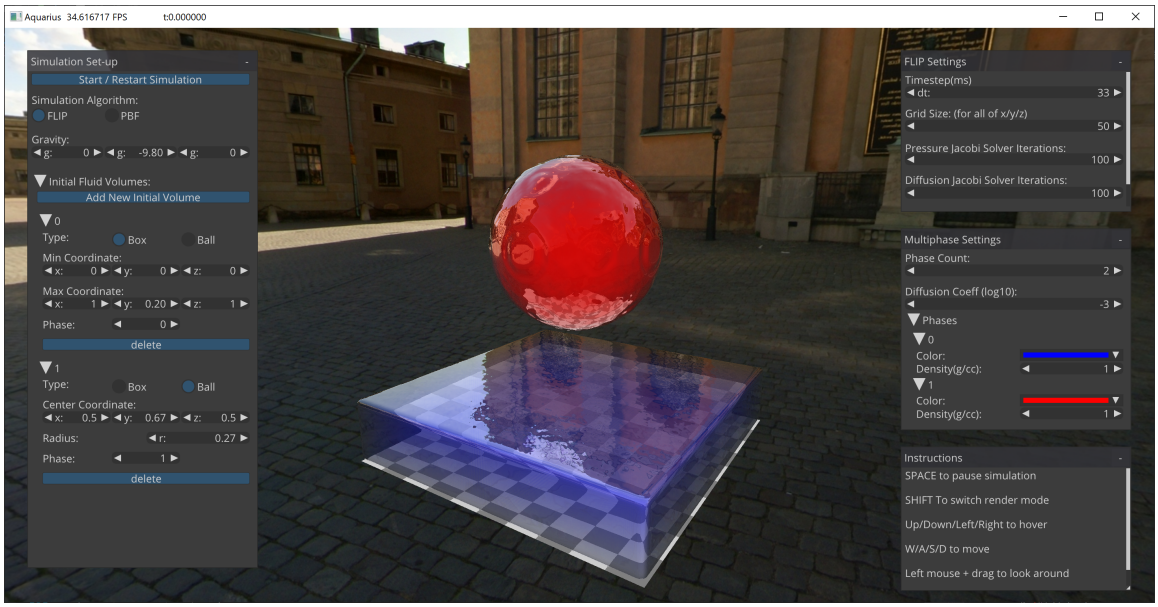


Figure 1.1: The full user interface of the software

2 Physics of Fluids

The mechanics of fluids are governed by the partial differential equations (PDEs) known as the *Incompressible Navier-Stokes Equations*, or in case of inviscid fluids, the *Euler Equations*. This chapter explains the meaning and intuition behind these equations, which are key to designing and implementing numerical simulation algorithms.

2.1 Vector Calculus

The fluid equations are commonly written in the language of vector calculus. A brief introduction of the main concepts and operators involved is given in this chapter.

Scalar Field

A *scalar field* on $\mathbb{R}^3$ is a mapping $\phi : \mathbb{R}^3 \to \mathbb{R}$ from 3D cartesian coordinates to scalar values. Example scalar fields include fluid density, or pressure, where a scalar value can be sampled in each point of the 3D space.

Vector Field

A *vector field* on $\mathbb{R}^3$ is a mapping $\phi : \mathbb{R}^3 \to \mathbb{R}^3$ from 3D cartesian coordinates to 3D vectors. A very commonly used vector field is the velocity field $\mathbf{u}$, which describes the direction and speed of the fluid's movement at each point in the 3D space

The grad

Given a scalar field $\phi : \mathbb{R}^3 \to \mathbb{R}$, the *gradient* or *grad* of the field is a vector field written as $\nabla\phi$, and it is defined by:

$$\nabla\phi = \begin{pmatrix} \dfrac{\partial\phi}{\partial x} \\ \dfrac{\partial\phi}{\partial y} \\ \dfrac{\partial\phi}{\partial z} \end{pmatrix}$$

The grad of a scalar quantity ϕ represents the rate of change of ϕ across each dimension. Moreover, $\nabla\phi$ computes the direction of movement which causes the greatest increase of ϕ.

The ∇ operator can also be extended to scaler fields of higher dimensions: let $\phi : \mathbb{R}^N \to \mathbb{R}$ be an N-dimensional scalar field, $\nabla\phi$ is then defined as:

$$\nabla\phi \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} \dfrac{\partial\phi}{\partial x_1} \\ \dfrac{\partial\phi}{\partial x_2} \\ \vdots \\ \dfrac{\partial\phi}{\partial x_N} \end{pmatrix}$$

The div

Given a vector field $\mathbf{u} : \mathbb{R}^3 \to \mathbb{R}^3$, the *divergence* or *div* of the field is a scalar field written as $\nabla \cdot \mathbf{u}$, and it is defined by:

$$\nabla \cdot \mathbf{u} = \nabla \cdot \begin{pmatrix} \mathbf{u}_x \\ \mathbf{u}_y \\ \mathbf{u}_z \end{pmatrix} = \frac{\partial \mathbf{u}_x}{\partial x} + \frac{\partial \mathbf{u}_y}{\partial y} + \frac{\partial \mathbf{u}_z}{\partial z}$$

If $\mathbf{u}$ is the velocity field, then the scalar field $\nabla \cdot \mathbf{u}$ represents the speed at which the fluid is expanding or shrinking at each 3D location. Thus, a velocity field that satisfies $\nabla \cdot \mathbf{u} = 0$ would keep the fluid in constant volume, which is how most fluids behave in the real world.

The curl

Given a vector field $\mathbf{u} : \mathbb{R}^3 \to \mathbb{R}^3$, the *curl* of the field is a scalar field written as $\nabla \times \mathbf{u}$, and it is defined by:

$$\nabla \times \mathbf{u} = \nabla \times \begin{pmatrix} \mathbf{u}_x \\ \mathbf{u}_y \\ \mathbf{u}_z \end{pmatrix} = \begin{pmatrix} \dfrac{\partial \mathbf{u}_z}{\partial y} - \dfrac{\partial \mathbf{u}_y}{\partial z} \\ \dfrac{\partial \mathbf{u}_x}{\partial z} - \dfrac{\partial \mathbf{u}_z}{\partial x} \\ \dfrac{\partial \mathbf{u}_y}{\partial x} - \dfrac{\partial \mathbf{u}_x}{\partial y} \end{pmatrix}$$

Informally, the curl of the velocity field is a measure of the local rotation of the fluid. Though not directly used in the equations and algorithms presented in this project, it is at the heart of a different class of algorithms, called the vortex methods[1].

The Laplacian

The *Laplacian* operator, written $\nabla \cdot \nabla$, is defined to be the divergence of the gradient. For scalar field ϕ, it can be computed that:

$$\nabla \cdot \nabla \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2}$$

The Laplacian describes the difference between the average value of ϕ in the neighborhood of a certain point and the value of ϕ at that point. As defined, this operator takes a scalar field and returns a scalar field. However, The Laplacian is also often extended to be applied to vector fields, where

$$\nabla \cdot \nabla \mathbf{u} = \begin{pmatrix} \nabla \cdot \nabla \mathbf{u}_x \\ \nabla \cdot \nabla \mathbf{u}_y \\ \nabla \cdot \nabla \mathbf{u}_z \end{pmatrix}$$

2.2 The Eulerian and Lagrangian Viewpoints

For any physical quantity that represents some property of a fluid, the field of that quantity, either scalar or vector, could be constantly evolving as time passes. There are two different approaches to tracking this rate of change: the Eulerian viewpoint and the Lagrangian viewpoint.

The Eulerian viewpoint considers the time derivative of quantities at fixed locations in the 3D space. For a scalar field ϕ which varies through time, its *Eulerian derivative* is simply $\frac{\partial \phi}{\partial t}$. To be more precise, the Eulerian derivative $\frac{\partial \phi}{\partial t}$, evaluated at point $\mathbf{x}$, is the rate of change of ϕ of the fluid at the fixed position $\mathbf{x}$, despite the fact that the fluid could be in motion. This has the immediate consequence that the concept of Eulerian derivative fails to capture the fact that physical quantities are carried around (i.e advected) by the fluid.

The Lagrangian viewpoint, on the other hand, tracks the rates of changes of quantities as it moves along the velocity field $\mathbf{u}$. In this approach, for a scalar field ϕ, its derivative with respect to time is written as $\frac{D\phi}{Dt}$, and defined to be

$$\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + \nabla \phi \cdot \mathbf{u}$$

This derivative, known as the *Lagrangian derivative* or *material derivative*, can be justified by treating the fluid as a collection of infinitesimal particles, each carrying some quantities and moving along the velocity field. At time t, for each particle p

with position $\mathbf{x}$, the quantity of ϕ it carries is $\phi_p = \phi(t, \mathbf{x}(p))$. The derivative with respect to t of this term computes the rate of change of ϕ_p:

$$\begin{aligned}\frac{d}{dt}\phi_p &= \frac{d}{dt}\phi(t, \mathbf{x}(t)) \\ &= \frac{\partial \phi}{\partial t} + \nabla\phi \cdot \frac{d\mathbf{x}}{dt} \\ &= \frac{\partial \phi}{\partial t} + \nabla\phi \cdot \mathbf{u} \\ &= \frac{D\phi}{Dt}\end{aligned}$$

Which is precisely the Lagrangian derivative.

When formalizing the Euler and Navier-Stokes equations, the Lagrangian derivative $\dfrac{D}{Dt}$ will be automatically extended to be applied to vector fields, where each component of the vector field is differentiated separately. This allows the term $\dfrac{D\mathbf{u}}{Dt}$ to be written, representing the acceleration of the infinitesimal fluid particles:

$$\frac{D\mathbf{u}}{Dt} = \frac{\partial \mathbf{u}}{\partial t} + \begin{pmatrix}\nabla \mathbf{u}_x \cdot \mathbf{u} \\ \nabla \mathbf{u}_y \cdot \mathbf{u} \\ \nabla \mathbf{u}_z \cdot \mathbf{u}\end{pmatrix} \tag{2.1}$$

As a flashforward to chapter 4 and 5, the grid-based methods mainly employ the Eulerian viewpoint, storing quantities on a fixed grid, and using an explicit computational step called *advection* to move around the quantities. In contrast, the particle-based methods always use the Lagrangian viewpoint, with quantities being recorded solely on particles.

2.3 The Euler and Navier-Stokes Equations

Using the previously defined notations, the Euler equations, which governs the motion of an incompressible and inviscid fluid , can be written as

$$\begin{cases} \dfrac{D\mathbf{u}}{Dt} = -\dfrac{\nabla p}{\rho} + \mathbf{g} \\ \nabla \cdot \mathbf{u} = 0 \end{cases} \qquad \text{(Euler Equations)}$$

where $\mathbf{u}$ is the velocity field, p is pressure, ρ is the fluid's density, and $\mathbf{g}$ the acceleration caused by an external force field (e.g gravity).

A generalized version of these equations is the famous incompressible Navier-Stokes equations, in which a term that describes viscosity is added:

$$\begin{cases} \dfrac{D\mathbf{u}}{Dt} = -\dfrac{\nabla p}{\rho} + \mathbf{g} + \nu\nabla \cdot \nabla \mathbf{u} \\ \nabla \cdot \mathbf{u} = 0 \end{cases} \qquad \text{(Navier-Stokes Equations)}$$

where ν is the kinematic viscosity coefficient.

As described in the last section, the quantity $(\nabla \cdot \mathbf{u})$ represents the rate at which the fluid is expanding or shrinking. Fluids in the real world usually remains in constant volume, unless in extreme conditions. This motivates the equation $\nabla \cdot \mathbf{u} = 0$, included in both Euler and Navier-Stokes.

Besides the incompressibility condition, both Euler and Navier-Stokes include another equation known as the momentum equation (which is in fact a set of equations, because the quantities are vectors). The momentum equation essentially specifies Newton's 2nd law: $\mathbf{a} = \dfrac{\mathbf{F}}{m}$, i.e the acceleration is the force divided by the mass.

As previously explained, the quantity $\dfrac{D\mathbf{u}}{Dt}$ represents the acceleration of the infinitesimal fluid particles. Thus, to explain the momentum equations, it remains to demonstrate that the right hand side correctly computes the force divided by the mass. Let the mass of the infinitesimal particle be m, and let the force be separated into the internal forces within the fluid F_{in} and the external forces F_{ext}:

$$\frac{D\mathbf{u}}{Dt} = \frac{F_{in} + F_{ext}}{m}$$

With $\mathbf{g}$ representing the acceleration caused by an external force field (e.g gravity), this can be rewritten as

$$\frac{D\mathbf{u}}{Dt} = \frac{F_{in}}{m} + \mathbf{g}$$

The internal forces within a fluid is caused by an imbalance in pressure. Specifically, if one side of a infinitesimal particle has a greater pressure than the opposite side, then the particle will be pushed towards the low pressure region. This justifies why the pressure forces are in the direction of $-\nabla p$, which computes the direction of fastest decrease of pressure. From a dimensional analysis point of view, the unit of p is $\dfrac{force}{length^2}$, thus the unit of ∇p is $\dfrac{force}{length^3}$. This indicates that to obtain the pressure force, it's necessary to multiply by the volume V of the infinitesimal particle, which produces

$$\frac{D\mathbf{u}}{Dt} = -\frac{V\nabla p}{m} + \mathbf{g}$$

Using $\rho = \dfrac{m}{V}$, this becomes:

$$\frac{D\mathbf{u}}{Dt} = -\frac{\nabla p}{\rho} + \mathbf{g}$$

Which is Euler's momentum equation. It is important to note that the justifications of the fluid equations given in this section is far from a rigorous mathematical derivation, which would not fit into this report due to its complexity.

The Navier-Stokes momentum equation extends the Euler momentum equation by considering viscosity. In a viscous fluid, the velocity of a particle tends to diffuse into its surrounding particles, causing the velocity in the neighborhood to converge into its average. The difference between the average of $\mathbf{u}$ in the neighborhood and the value of $\mathbf{u}$ of the particle is captured by the Laplacian of the velocity: $\nabla \cdot \nabla \mathbf{u}$, thus adding a positive multiple of this quantity creates a viscous effect:

$$\frac{D\mathbf{u}}{Dt} = -\frac{\nabla p}{\rho} + \mathbf{g} + \nu \nabla \cdot \nabla \mathbf{u}$$

where ν is a constant property, known as the kinematic viscosity of the fluid. For water, which is a rather inviscid fluid, this quantity is almost negligible. When simulating water, considering the effects of viscosity requires considerable extra computation, while bringing little improvements to the visual fidelity. As a result, this project chooses to only solve the Euler equations during simulation.

2.4 Boundary Conditions

For a fluid region that is not the entirety of $\mathbb{R}^3$, boundary conditions must be specified, which defines the behavior of the fluid in the physical boundary of the fluid region.

When simulating liquids, there are two types of boundary conditions: the solid boundaries and the free surface boundaries. At a solid boundary, the condition is

$$\mathbf{u} \cdot \mathbf{n} = 0$$

where $\mathbf{n}$ is the normal of the solid surface. This condition ensures that the fluid cannot flow into a solid.

The second type of boundary is the free surface boundary, which is the boundary between the liquids and some region of space that is not occupied by anything. In this case, that region of space will not exert any force, and therefore pressure, to the fluid, which motivates the condition

$$p = 0$$

This free surface condition can be also applied to the boundary between liquid and air, which is because air is significantly lighter than liquid, and hence does not influence the motion of the liquid.

The liquid simulated in this project is contained within a cubic box. Moreover,it doesn't fill the box entirely and thus has a free surface. Consequently, both boundary condition will be applied during the numerical simulation.

2.5 Multiple Fluids

Finally, this section introduces an equation that governs the concentration changes in a mixture of more than one type of miscible fluids, for example, ink and water. In physics, the different types of fluids are sometimes referred to as *phases*.

Concentration of different fluid phases will be represented using *volume fractions*. Specifically, for an infinitesimal fluid element at location $\mathbf{x}$ with volume $\mathbf{V}(\mathbf{x})$, and let the portion of this volume occupied by the ith fluids phase be $\mathbf{V}^i(\mathbf{x})$, then the concentration of the ith phase at $\mathbf{x}$ will be $\alpha^i(\mathbf{x}) = \dfrac{\mathbf{V}^i(\mathbf{x})}{\mathbf{V}(\mathbf{x})}$.

The diffusion among multiple fluid phases is a result of the random Brownian motion of the fluid particles. However, it's possible to model this process from a macroscopic viewpoint, where an equation can be written that governs the expectation of how the concentrations change:

$$\frac{D\alpha^i}{Dt} = C\nabla \cdot \nabla\alpha^i \qquad \text{(Advection-Diffusion Equation)}$$

where C is the diffusion coefficient. Informally, if the concentration of a fluid region has a different concentration than its surroundings, this difference will tend to "diffuse" into the neighborhood, so that the concentrations tends to become more similar. This explains why the rate of change of the concentration is proportional to the Laplacian of the concentration.

3 GPU Programming

To achieve maximum performance, this project chooses to implement the fluid simulation software on GPUs. Originally built for graphics applications, GPUs were designed to handle a massive amount of geometries and pixels in parallel, because in graphics applications the computation for different geometries and pixels are largely independent. The ability to do massively parallel computation motivated GPGPU (General Purpose GPU) programming models to arise, which became significantly useful for scientific computing purposes. The implementation code in this project is written using the CUDA programming model, developed by the NVIDIA Corporation.

3.1 The CUDA Programming Model

CUDA employs the executing model known as SIMT (Single Instruction Multiple Thread). In this model, a large amount of GPU threads can be spawned simultaneously, each running the same sequence of code on different sets of data. GPU threads are organized in groups of 32, known as *warps*, and the instructions running on threads of the same warp must be synchronized. However, different warps does not need to remain in sync. When the threads within a warp access the memory, the entire warp can be swapped out, so that a different warp can start executing before the memory access finishes. Using this mechanism, the GPU hides memory access latencies by allowing very fast context switching. As a result, each physical core in the GPU (known as a CUDA core) can simultaneously handle multiple logical threads.

As an example, the GPU used for development of this project is an NVIDIA GTX1060 Mobile, which contains 10 *Streaming Multiprocessors*, each of which consists of 128 CUDA cores. Each streaming multiprocessors can have up to 2048 resident threads, giving a total of 20480 threads that can be simultaneously handled. Even though each GPU thread is not as fast as a CPU thread, a well adjusted CUDA program can still be up to 100 times faster than the same CPU program.

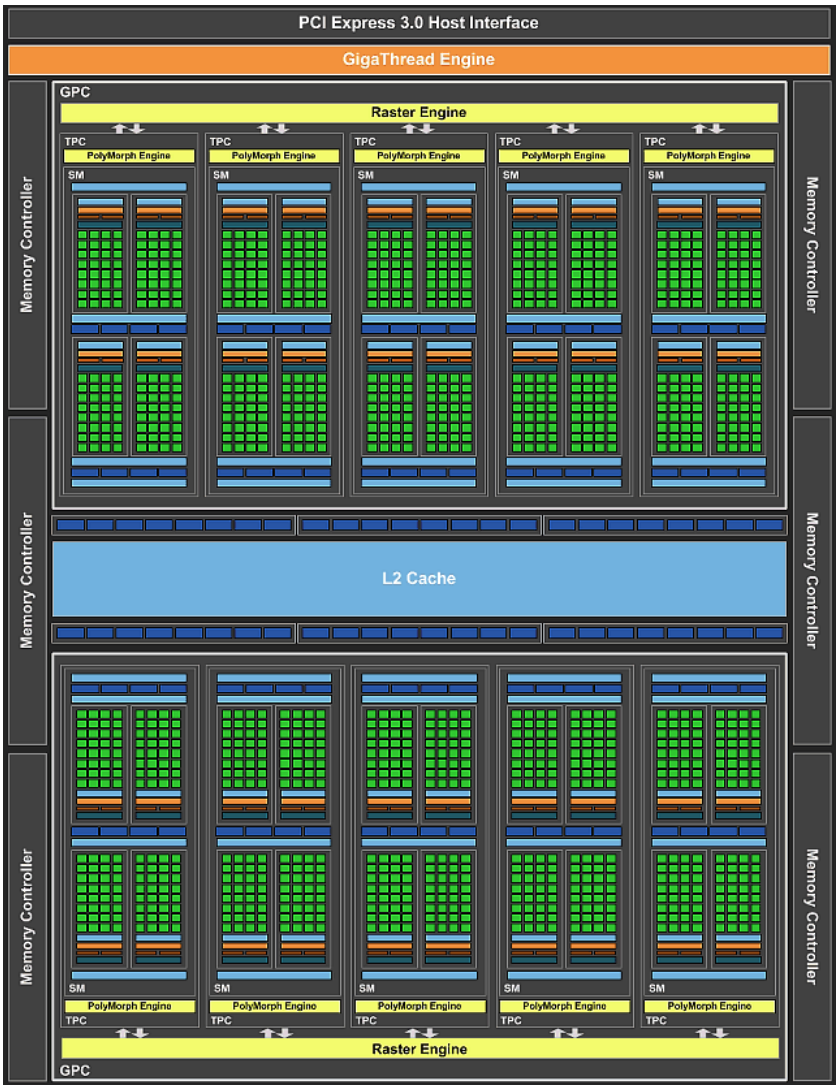


Figure 3.1: The architecture of a GTX 1060 (GP106), with 10 streaming multiprocessors, each containing 128 CUDA cores. Image courtesy to NVIDIA Corporation.

3.2 The OpenGL Render Pipeline

Since this project implements not only the simulation but also the rendering of fluids, the graphics functions of modern GPUs is also extensively utilized. Specifically, this program uses the OpenGL API for rendering.

In OpenGL (as well as other APIs such as Direct3D and Vulkan), the only types of geometries that can be rendered are points, lines, and triangles. For this reason, in order to render complex 3D objects, a mesh of triangles is often used to represent the surface of the object. After being fed into OpenGL's pipeline, these triangles go through 4 main stages:

1. *The Vertex Shader*

   The vertex shader is a piece of GPU program, and its executed for each vertex of every triangle to be rendered. Usually, the vertex shader computes essential geometric information (e.g position on screen, normal) of the vertices, which are passed on to the next stage.

2. *Rasterization*

   In the rasterization stage, a piece of hardware in the GPU uses the screen positions of vertices to determine, for each triangle, which pixels does the triangle cover. Each pixel of each triangle is then passed on to the next stage for coloring.

3. *The Pixel Shader*

   Also referred to as the *fragment shader*, the pixel shader is also a piece of GPU program. The pixel shader culminates the job of the entire graphics pipeline: it computes of the color of each pixel to be displayed.

4. *Compositing*

   In 3D scene, objects might hide behind each other, and some pixels might be covered by multiple objects at different depth. The compositor's job is to determine which shading of the pixel to should be used, and in some cases (e.g opaque objects), how the differently shaded colors should be mixed together to give the final output. This process is essential to the multiphase fluid rendering algorithm invented in this project, to be described in subsection 6.2.2.

The full OpenGL pipeline includes a few extra stages, such as the geometry shader and the tesselation shader. However, these are optional stages and is not used in this project. In chapter 6, more details will be given on the graphics pipeline of this project, and how it performs realistic liquid rendering in real time.

4 Grid-Based Simulations

This chapter introduces a grid-based multiphase fluid simulation scheme and its CUDA implementation in this project. This scheme has three key components: a **MAC** (Marker and Cell) grid for discretizing the Euler equations, a **FLIP** (Fluid Implicit Particle) algorithm for advection, and a Jacobi linear solver for solving the diffusion equation and the Poisson pressure equation (which ensures incompressibility).

4.1 Operator Splitting

A common way for numerically solving differential equations is the *operator splitting* approach. As a simple example, consider the simple differential equation:

$$\frac{dx}{dt} = f(x) + g(x) \quad \text{With initial condition } x(0) = x_0$$

To numerically solve this, decide on some small time step $\triangle t$, and let $x_{[n]}$ be the value of x at the nth time step. The goal is to find $x_{[n]}$ for increasing larger n. To do this, start with $x_{[0]} = x_0$ and consider the two differential equations:

$$\frac{dx}{dt} = f(x)$$
$$\frac{dx}{dt} = g(x)$$

Suppose there exists some good solutions (either analytical or numerical) for these two equations, then these solutions can be used to find a good solution for the original equation. Specifically, suppose $F_{f_0}(t)$ is a solution of $\dfrac{dx}{dt} = f(x)$ with initial condition $x(0) = f_0$, and $G_{g_0}(t)$ is a solution of $\dfrac{dx}{dt} = g(x)$ with initial condition $x(0) = g_0$, then, the original equation can be solved as

$$\widetilde{x} = F_{x_{[n]}}(\triangle t)$$
$$x_{[n+1]} = G_{\widetilde{x}}(\triangle t)$$

In essence, this approach splits the equation into a few more easily solved differential equations, and accumulates the solution of each over a small time step.

This same splitting approach can be applied to the Euler equations. To do so, the Euler momentum equation is first written in a form where the material derivative is expanded using equation (2.1):

$$\frac{\partial \mathbf{u}}{\partial t} = -\begin{pmatrix} \nabla \mathbf{u}_x \cdot \mathbf{u} \\ \nabla \mathbf{u}_y \cdot \mathbf{u} \\ \nabla \mathbf{u}_z \cdot \mathbf{u} \end{pmatrix} + \mathbf{g} - \frac{\nabla p}{\rho}$$

This then allows the equation, and therefore the simulation algorithm, to be split into three parts:

1. $$\frac{\partial \mathbf{u}}{\partial t} = -\begin{pmatrix} \nabla \mathbf{u}_x \cdot \mathbf{u} \\ \nabla \mathbf{u}_y \cdot \mathbf{u} \\ \nabla \mathbf{u}_z \cdot \mathbf{u} \end{pmatrix}$$

   Again using equation (2.1), this can be rewritten back into the material derivative form:

   $$\frac{D\mathbf{u}}{Dt} = 0$$

   Intuitively, solving this equation means to move the fluid according to its velocity field, in a way such that the velocity of each infinitesimal fluid partial remains unchanged. This is the step known as *advection*.

2. $$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{g}$$

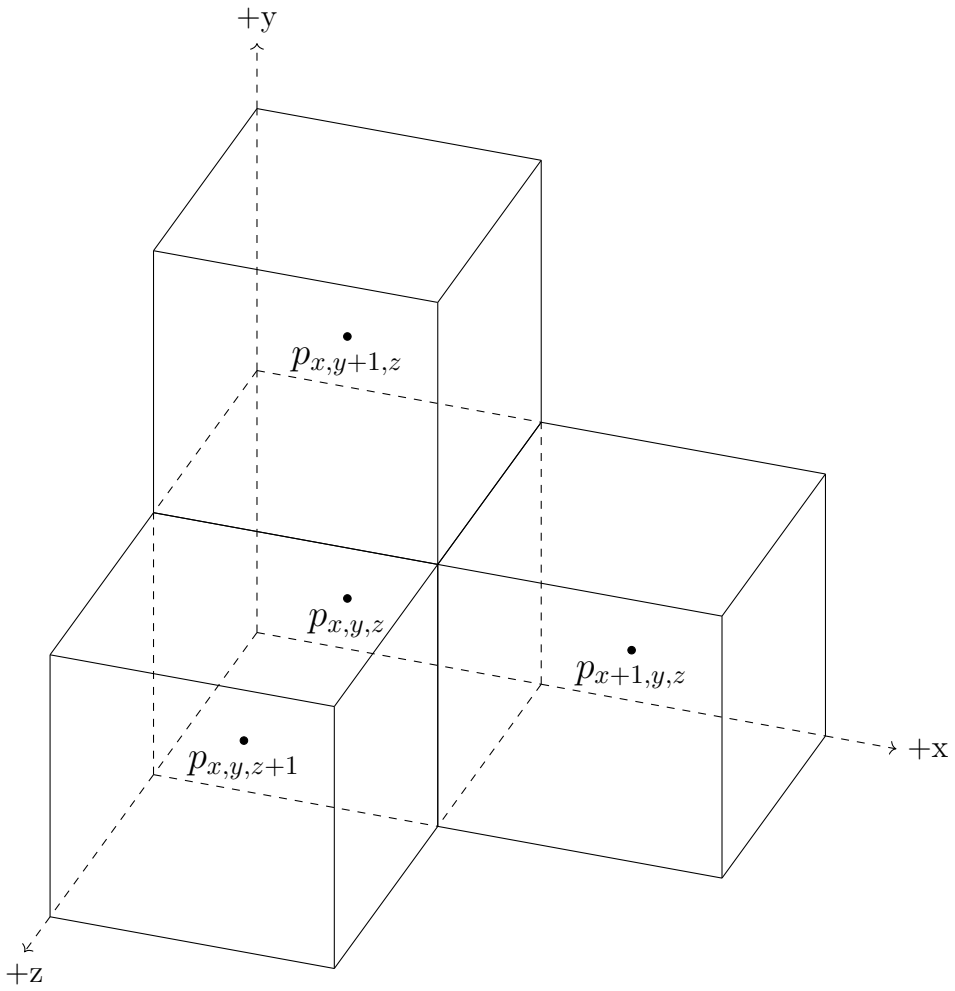
   Solving this equation is the process of exerting external forces (e.g gravity) on the fluid. The solid boundary conditions can also be enforced in this step.

3. $$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\nabla p}{\rho}$$

   Since this is the last step of the splitting, it is essential to make sure that the results of solving this equation satisfies the incompressibility condition $\nabla \cdot \mathbf{u} = 0$. This amounts to finding a pressure field p such that, subtracting by $\triangle t \frac{\nabla p}{\rho}$ makes the velocity have zero divergence. This step enforces the incompressibility of the fluid.

4.2 Discretization

The Euler equations involve two important quantities: the the pressure scalar field p, and the velocity vector field $\mathbf{u}$. For a numerical simulation, discretized versions of both fields need to maintained. A straightforward choice, which is used for the pressure field, is to maintain a 3d grid, where each cubic grid cell stores the pressure value sampled at the center of the cell. As an example, this figure shows the cell with location (x, y, z), and 3 of its neighbors:



Other than being simple to understand and implement, this discretization scheme also has the advantage that the finite difference approximation of the Laplacian of the pressure field, sampled at the center of the cells, can be easily computed:

$$
\begin{aligned}
\nabla \cdot \nabla p &= \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} \\
&\approx \frac{p_{x+1,y,z} + p_{x-1,y,z} - 2p_{x,y,z}}{(\triangle x)^2} + \\
&\quad \frac{p_{x,y+1,z} + p_{x,y-1,z} - 2p_{x,y,z}}{(\triangle x)^2} + \\
&\quad \frac{p_{x,y,z+1} + p_{x,y,z-1} - 2p_{x,y,z}}{(\triangle x)^2} \\
&= \frac{p_{x+1,y,z} + p_{x-1,y,z} + p_{x,y+1,z} + p_{x,y-1,z} + p_{x,y,z+1} + p_{x,y,z-1} - 6p_{x,y,z}}{(\triangle x)^2}
\end{aligned}
\tag{4.1}
$$

where $\triangle x$ is the edge length of the cubic cell.

For the velocity field $\mathbf{u}$, a slightly more sophisticated method known as the **MAC**(Marker and Cell) grid is used. Instead of storing the value of $\mathbf{u} = (\mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_z)$ sampled at the cell center, an MAC grid stores different components of $\mathbf{u}$ sampled at different locations. Specifically, the grid cell at position (x, y, z) stores the value of $\mathbf{u}_x$ sampled at the center of its left face, the value of $\mathbf{u}_y$ sampled at its lower face, and the value of $\mathbf{u}_z$ sampled at its back face, as illustrated in this figure:

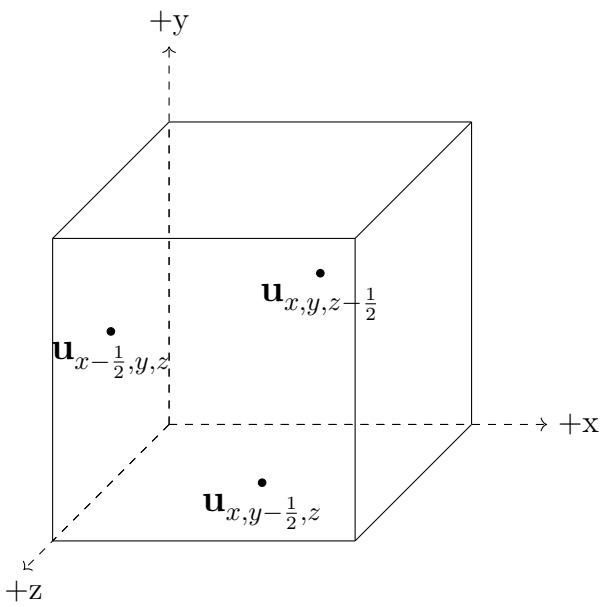
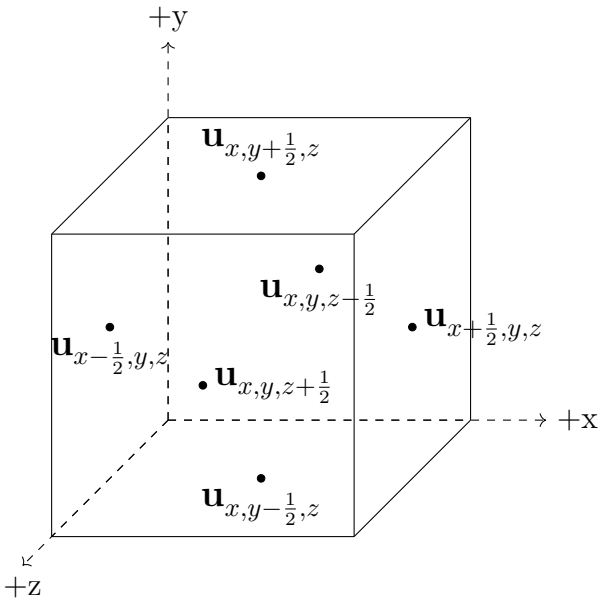


Figure 4.1: a 3D MAC grid cell and the velocity data it stores

The quantities $\mathbf{u}_{x,y,z-\frac{1}{2}}$, $\mathbf{u}_{x,y-\frac{1}{2},z}$, $\mathbf{u}_{x-\frac{1}{2},y,z}$ are all scalars, representing the velocity pointing at the x, y, and z direction, respectively. Furthermore, notice that the values of $\mathbf{u}_{x+\frac{1}{2},y,z}$, $\mathbf{u}_{x,y+\frac{1}{2},z}$, and $\mathbf{u}_{x,y,z+\frac{1}{2}}$, which are respectively sampled at the centers of

the right, upper, and front faces, will also be available. This is because $\mathbf{u}_{x+\frac{1}{2},y,z} = \mathbf{u}_{x+1-\frac{1}{2},y,z}$, that is, the value of $\mathbf{u}_x$ sampled at the right face of the cell is exactly the value of $\mathbf{u}_x$ sampled at the left face of the neighboring cell on the right. The same can be applied for the upper and front faces. As a result, there are 6 velocity values associated with each grid cell:



Using these quantities, an approximation of the divergence of the velocity, $\nabla \cdot \mathbf{u}$, sampled at cell centers, can be easily computed:

$$\begin{aligned}
\nabla \cdot \mathbf{u} &= \frac{\partial \mathbf{u}_x}{\partial x} + \frac{\partial \mathbf{u}_y}{\partial y} + \frac{\partial \mathbf{u}_z}{\partial z} \\
&\approx \frac{\triangle \mathbf{u}_x}{\triangle x} + \frac{\triangle \mathbf{u}_y}{\triangle y} + \frac{\triangle \mathbf{u}_z}{\triangle z} \\
&= \frac{\mathbf{u}_{x+\frac{1}{2},y,z} - \mathbf{u}_{x-\frac{1}{2},y,z}}{\triangle x} + \frac{\mathbf{u}_{x,y+\frac{1}{2},z} - \mathbf{u}_{x,y-\frac{1}{2},z}}{\triangle x} + \frac{\mathbf{u}_{x,y,z+\frac{1}{2}} - \mathbf{u}_{x,y,z-\frac{1}{2}}}{\triangle x}
\end{aligned} \tag{4.2}$$

During the incompressibility step of the simulation, the velocity field will be updated according the gradient of the pressure field. Thus, it's also important to compute the approximation of ∇p at the velocity field sample points, i.e the centers of faces of the cells. This is made easy by the fact that, the pressure field is sampled at the centers of the cells:

$$(\nabla p)_{x-\frac{1}{2},y,z} = \frac{p_{x,y,z} - p_{x-1,y,z}}{\triangle x} \tag{4.3}$$

The numerical approximations to $\nabla \cdot \nabla p$, $\nabla \cdot \mathbf{u}$, and ∇p will all be used during the incompressibility step, as will be explained in section 4.5.

4.3 Advection

As previously explained, the first step in each time step of the simulation is to solve the advection equation $\frac{D\mathbf{u}}{Dt} = 0$. Intuitively, this equation demands that the velocity of each infinitesimal particle in the fluid remains unchanged (but the velocity field itself will change because the positions of the particles will change).

A once widely used advection algorithm is called **PIC**(Particle in Cell), which is closely based on the intuition behind the material derivative. Instead of infinitely many infinitely small particles, the fluid is approximately represented using a finite but large cloud of particles, each storing its own velocity. Using the velocity field $\mathbf{u}_{[n]}$ in nth time step, the PIC advection at the $n+1$ th time step work in these following steps:

1. For each particle p with position $\mathbf{x}_p$, sample and interpolate the MAC grid to obtain the value of $\mathbf{u}_{[n]}$ at $\mathbf{x}_p$. Assign this as the particle's velocity, $\mathbf{u}_p$.

2. Move the particle in the velocity field $\mathbf{u}_{[n]}$. This can be as simple as computing $\mathbf{x}_p^{new} = \mathbf{x}_p + \triangle t \mathbf{u}_{[n]}(\mathbf{x}_p)$. For higher accuracy, this project performs this using a 3rd-order Runge-Kutta integration:

$$\begin{aligned}
\mathbf{u}_{temp1} &= \mathbf{u}_{[n]}(\mathbf{x}_p) \\
\mathbf{u}_{temp2} &= \mathbf{u}_{[n]}(\mathbf{x}_p + \frac{1}{2}\triangle t \mathbf{u}_{temp1}) \\
\mathbf{u}_{temp3} &= \mathbf{u}_{[n]}(\mathbf{x}_p + \frac{3}{4}\triangle t \mathbf{u}_{temp2}) \\
\mathbf{x}_p^{new} &= \mathbf{x}_p + \triangle t(\frac{2}{9}\mathbf{u}_{temp1} + \frac{3}{9}\mathbf{u}_{temp2} + \frac{4}{9}\mathbf{u}_{temp3})
\end{aligned}$$

   For particles near the boundaries of the fluid, some of the $\mathbf{u}_{temp}$ values might be sampled outside the fluid region, which is slightly problematic because the velocity isn't defined outside the fluids. This is fixed by a simple *extrapolation* step, which extends the velocity to a few grid cells outside its original region.

3. For each MAC grid cell, and for each of its 3 sample points where a component of $\mathbf{u}$ is stored, find all particles within a certain small radius (usually $\triangle x$), and interpolate their value of $\mathbf{u}_p$. Save these values as a temporary velocity field, $\mathbf{u}_{[n+1]}^{advected}$.

In short, the PIC algorithm first transfers the velocity field from the MAC grid to the particles, then move the particles, and then transfers the velocity from the particles back to the MAC grid.

The PIC algorithm is largely superseded by another algorithm known as **FLIP**(Fluid Implicit Particle), which is implemented in this project. FLIP is very similar to PIC, with only a slightly different 1st step:

1′. For each particle p with position $\mathbf{x}_p$, sample and interpolate the MAC grid to obtain the value of $\mathbf{u}_{[n]} - \mathbf{u}_{[n-1]}$ at $\mathbf{x}_p$. Add this to the particle's velocity, $\mathbf{u}_p$.

That is, instead of interpolating the value of $\mathbf{u}$ on to the particles, FLIP interpolates the change of $\mathbf{u}$ in the last time step, and add that to the particles' velocities. Zhu and Bridson [18] showed that this method reduces the undesirable effect called *numerical dissipation*, where visually interesting details in the fluid are smoothed away due to excessive interpolation.

4.4 External Forces

After obtaining the temporary velocity field $\mathbf{u}_{[n+1]}^{advected}$, the next step is to apply external forces. Two types of external forces will be considered: the forces arising from an external force field such as gravity, and the forces exerted by a solid boundary.

Let $\mathbf{g}$ denote the acceleration caused by the external force field, (for gravity, $\mathbf{g} \approx [0, -0.98, 0]^T$), applying the forces is then achieved by adding $\triangle t\mathbf{g}$. In a MAC grid, this is done by updating the components of $\mathbf{u}$ sampled at different faces using the different components of $\triangle t\mathbf{g}$.

To apply the solid boundary condition $\mathbf{u} \cdot \mathbf{n} = 0$, as mentioned in section 2.4, components of $\mathbf{u}$ sampled at faces that represent solid-fluid boundaries need to be set to 0. For example, if the solid region is considered to be exactly the region of space outside the MAC grid, then the leftmost faces of the leftmost cells (and rightmost faces of rightmost cells...etc) will be considered as a solid-fluid boundary. For all such boundary faces, the velocity component there will be set to 0.

Starting from an incompressible velocity field $\mathbf{u}_{[n]}$, performing advection to obtain $\mathbf{u}_{[n+1]}^{advected}$, and then applying external forces, the resulting velocity will likely not be incompressible anymore. Let this field be called $\mathbf{u}_{[n+1]}^{compressible}$, and the next step will be to apply pressure within the fluid, so that the incompressibility is restored.

4.5 Enforcing Incompressibility

To enforce the incompressibility condition $\nabla \cdot \mathbf{u}_{[n+1]} = 0$, the algorithm needs to find a pressure field p such that,

$$\nabla \cdot \mathbf{u}_{[n+1]} = \nabla \cdot (\mathbf{u}_{[n+1]}^{compressible} - \triangle t \frac{\nabla p}{\rho}) = 0$$

Rearranging the equation on the right gives

$$-\frac{\triangle t}{\rho} \nabla \cdot \nabla p = -\nabla \cdot \mathbf{u}_{[n+1]}^{compressible}$$

Using the discretization formulas 4.1 and 4.2, the discrete version of this equation can be written:

$$\begin{aligned} &\frac{\triangle t}{\rho \triangle x}(6p_{x,y,z} - p_{x+1,y,z} - p_{x-1,y,z} - p_{x,y+1,z} - p_{x,y-1,z} - p_{x,y,z+1} - p_{x,y,z-1}) \\ &\quad = -(\mathbf{u}_{x+\frac{1}{2},y,z} - \mathbf{u}_{x-\frac{1}{2},y,z} + \mathbf{u}_{x,y+\frac{1}{2},z} - \mathbf{u}_{x,y-\frac{1}{2},z} + \mathbf{u}_{x,y,z+\frac{1}{2}} - \mathbf{u}_{x,y,z-\frac{1}{2}}) \end{aligned}$$

One such equation exists for every cell that contains fluid (i.e contains FLIP particles), and together, they form a system of linear equations, called the *Poisson pressure equation*. The unknowns, $p_{i,j,k}$, correspond to the pressure at the centers of cells.

Since each equation involves not only the pressure of the fluid cell itself, but also the pressure of its 6 adjacent cells, extra care needs to be taken for cells that are at the boundaries of the fluid. Specifically, if a variable $p_{i,j,k}$ in the equations corresponds to the pressure within a air cell, it should automatically be assigned 0, which satisfies the free surface boundary conditions. If the variable $p_{i,j,k}$ corresponds to the pressure within solid, then it suffices to replace it with the pressure of the fluid cell next to the boundary, because the velocities at solid-fluid boundaries were already fixed in the external forces step.

With the boundary conditions satisfied, the equation become a system of N linear equations with N variables, where N is the total amount of fluid cells. Solving this system hence results in a discrete representation of the pressure field p that satisfies

$$\nabla \cdot (\mathbf{u}_{[n+1]}^{compressible} - \triangle t \frac{\nabla p}{\rho}) = 0$$

Then, to retrieve the incompressible velocity field, it only remains to compute

$$\mathbf{u}_{[n+1]} = \mathbf{u}_{[n+1]}^{compressible} - \triangle t \frac{\nabla p}{\rho}$$

using discretization formula 4.2. This completes the simulation of one time step.

To summarize, using a grid and FLIP advection, the simulation of each time step follows the following procedure:

Algorithm 1: Single phase fluid FLIP simulation step

```
// At time step [n+1]
```
foreach *particle* p **do**
  $\mathbf{u}_p := \mathbf{u}_p + \mathbf{u}_{[n]}(\mathbf{x}_p) - \mathbf{u}_{[n-1]}(\mathbf{x}_p)$;
  Move p inside the velocity field using Runge-Kutta;
end
foreach *grid cell at location* (x, y, z) **do**
  Find all particles within a radius of $\triangle x$;
  Compute $\mathbf{u}_{[n+1]}^{advected}$, as an interpolation of the $\mathbf{u}_p$ of nearby particles;
end
Apply external forces, $\mathbf{u}_{[n+1]}^{compressible} = FixSolidBoundary(u_{[n+1]}^{advected} + \triangle t\mathbf{g})$;
Construct and solve the Poisson pressure equation, obtain pressure p;
Compute $\mathbf{u}_{[n+1]} = \mathbf{u}_{[n+1]}^{compressible} - \triangle t\frac{\nabla p}{\rho}$

4.6 Simulating Diffusion

While algorithm 1 is only for single phase fluid simulation, it is possible to extend it to support multiple fluid phases. As explained in section 2.5, the changes in the concentration α^i of a fluid phase i is governed by the Advection-Diffusion equation:

$$\frac{D\alpha^i}{Dt} = C\nabla \cdot \nabla\alpha^i$$

To incorporate this equation to the simulation algorithm, the first step is again to apply splitting. The equation is split into two parts:

$$\frac{D\alpha^i}{Dt} = 0$$
$$\frac{\partial\alpha^i}{\partial t} = C\nabla \cdot \nabla\alpha^i$$

Just like the Euler momentum equation, this first equation that splitting produces is an advection equation. Thus, the same FLIP advection that was applied for the velocity field can be used to advect the concentration quantities: first transfer the quantities from the grid to the particles by interpolating and adding $\alpha_{[n]}^i - \alpha_{[n-1]}^i$, then move the particles in the velocity field, then transfer the α^i back to the grid.

The second equation is the "diffusion" part of the advection-diffusion equation, and is sometimes referred as the diffusion equation by itself. It was shown that

using forward Euler scheme for this equation is unstable for large time steps[6], so a discretized implicit equation is used:

$$\begin{aligned} -\lambda\alpha^i_{[n+1]\ x-1,y,z} - \lambda\alpha^i_{[n+1]\ x+1,y,z} \\ -\lambda\alpha^i_{[n+1]\ x,y-1,z} - \lambda\alpha^i_{[n+1]\ x,y+1,z} \\ -\lambda\alpha^i_{[n+1]\ x,y,z-1} - \lambda\alpha^i_{[n+1]\ x,y,z+1} \\ +(1+6\lambda)\alpha^i_{[n+1]x,y,z} \end{aligned} \quad = \quad \alpha^{i,advected}_{[n]x,y,z}$$

where $\lambda = \dfrac{C\triangle t}{(\triangle x)^2}$. With the value of α^i_{n+1} at each fluid cell as unknown, this is again a linear equation. Solving the equation produces the new concentration field, $\alpha^i_{[n+1]}$.

Incorporating the FLIP concentration advection and solving the diffusion equation into the previous algorithm gives an algorithm for multiphase simulation:

Algorithm 2: Multiphase phase fluid FLIP simulation step

```
   // At time step [n+1]
```
1 **foreach** *particle* p **do**
2 | $\mathbf{u}_p := \mathbf{u}_p + \mathbf{u}_{[n]}(\mathbf{x}_p) - \mathbf{u}_{[n-1]}(\mathbf{x}_p)$;
3 | $\alpha^i_p := \alpha^i_p + \alpha^i_{[n]}(\mathbf{x}_p) - \alpha^i_{[n-1]}(\mathbf{x}_p)$, for all fluid phases i ;
4 | Move p inside the velocity field using Runge-Kutta;
5 **end**
6 **foreach** *grid cell at location* (x, y, z) **do**
7 | Find all particles within a radius of $\triangle x$;
8 | Compute $\mathbf{u}^{advected}_{[n+1]}$, as an interpolation of the $\mathbf{u}_p$ of nearby particles;
9 | Compute $\alpha^{advected}_{[n+1]}$, as an interpolation of the α_p of nearby particles;
10 **end**
11 Apply external forces, $\mathbf{u}^{compressible}_{[n+1]} = FixSolidBoundary(u^{advected}_{[n+1]} + \triangle t\mathbf{g})$;
12 Construct and solve the Poisson pressure equation, obtain pressure p;
13 Compute $\mathbf{u}_{[n+1]} = \mathbf{u}^{compressible}_{[n+1]} - \triangle t\dfrac{\nabla p}{\rho}$;
14 Construct and solve the discretized diffusion equation to obtain α^i_{n+1} for all fluid phases i.

4.7 Implementation

The pseudocode presentation of algorithm 2 takes a sequential form, and the challenge remains to parallelize this algorithm on GPU.

For certain parts of the algorithm, parallelization is straightforward. This includes line 2-4, line 8-9, which are all operations performed within a loop body. In different loop iterations, the data being operated on are completely different, and does not depend on previous iterations, which means it is safe to use parallel threads instead

of loops to perform these operations. Similarly, the velocity field updates in line 11 and 13 are also easy to parallelize, with each thread operating on one grid cell of the discretized velocity field.

The rest of the algorithm, line 7, 12 and 14, requires much more attention. Line 7 performs a task called *spatial indexing*: associating each grid cell with all the particles that are within radius $\triangle x$ at each time step. A naive implementation would require a linear search on all particles, and this has to be performed for all grid cells, which is intolerable because the simulation could involve up to around 1,000,000 particles and 100,000 grid cells. In line 12 and 14, a linear equation needs to be solved, where there's an unknown for each fluid cell. As a result, the total amount of unknowns is in the order of 100,000. A naive linear solver have a $O(N^3)$ complexity, which is also too costly. In order to achieve real-time simulation and rendering, each of these operations, indexing 1,000,000 million particles and solving linear equations with 100,000 unknowns, needs to be performed at least around 20 times per second. This section will focus on how this is made possible in this project.

4.7.1 Spatial Indexing

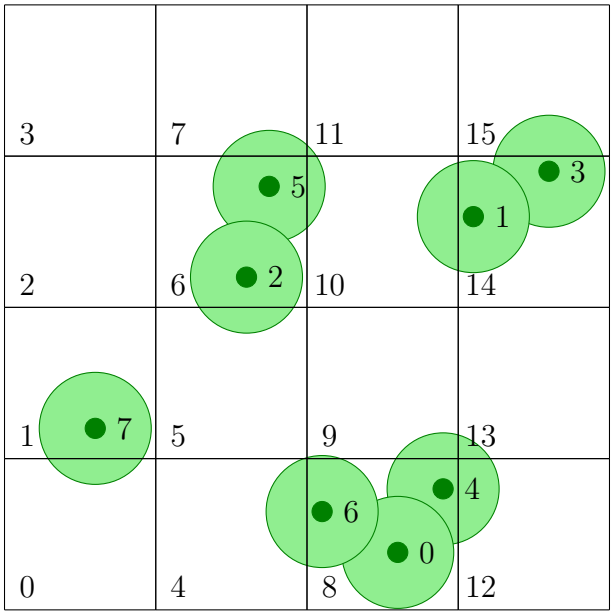
With $\triangle x$ being the edge length of each cubic grid cell, finding all particles within a radius $\triangle x$ of each cell can be reduced to finding the particles that are *inside* each cell. Then, for a certain cell, it suffices to check all the 27 cells in the neighborhood, because all particles within a radius $\triangle x$ must be contained inside these 27 cells.

To create an index from each cell to the particles inside the cell, this project uses the parallel algorithm proposed by Green [5]. The algorithm proceeds in the following steps:

1. Decide on a hash function for 3D grid coordinates. For example, in an $N*N*N$ grid, the hash of the coordinate (x, y, z) can be xN^2+yN+z, which fully avoids hash collision.

2. Create an array of hashes for particles. For each particle, use its physical position to compute the cell that it is in, and compute the hash of that cell as the hash of the particle. Store the hashes of all particles in this array. Since these steps is independent for each particle, it can be efficiently parallelized.

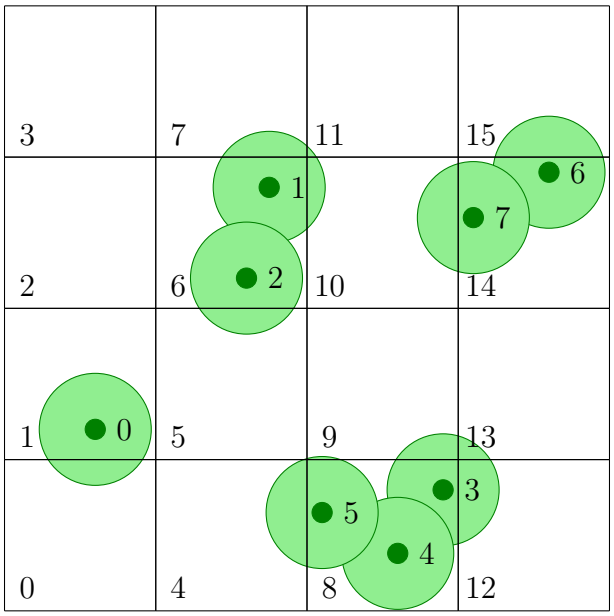
3. Sort this array of particle hashes, and sort the array of particles into the same order.



(a) The indices and positions of the particles in the grid before spatial indexing.

particle	hash
0	8
1	14
2	6
3	14
4	8
5	6
6	8
7	1

(b) The array of hashes of particles, computed in step 2.



(c) The indices and positions of the particles, after they are sorted according to their hashes, in step 3.

cell	*cellStart*	*cellEnd*
0		
1	0	0
2		
3		
4		
5		
6	1	2
7		
8	3	5
9		
10		
11		
12		
13		
14	6	7
15		

(d) The final result of spatial indexing, represented as the *cellStart* and *cellEnd* array.

Figure 4.2: Example of spatial indexing in 2D

4. Create two arrays *cellStart* and *cellEnd*, which denote, for each cell, the first and the last particle inside the cell. To compute elements of these arrays, for the ith particle, use the hash array to check if the $i-1$th particle is in the same cell, if not, the *cellStart* of this cell should be i. Similarly, if the $i+1$th particle is not in the same cell, the *cellEnd* of this cell should be i. This can also be done for all particles in parallel.

Having created the *cellStart* and *cellEnd* arrays, for each cell, the particles inside it is them simply the particles with index $\geq$ *cellStart* and $\leq$ *cellEnd*. A 2D example of this procedure is illustrated in figure 4.2.

With all other steps being completely parallelizable, the only complicated step is sorting the array of particle hashes. The implementation of this project uses the thoroughly optimized sorting library provided with the CUDA API. The resulting cost of the spatial indexing is almost negligible compared to other tasks, such as solving systems of linear equations.

4.7.2 Jacobi Linear Solver

The two linear systems to be solved in each simulation step, the Poisson pressure equation and the diffusion equation, both have the special property of being *symmetric positive-definite*. Many advanced approaches have been proposed on how to solve these types of matrices, such as ICPCG [3] and *Geometric Multigrid*[4]. This project chooses to implement a simpler algorithm, called the Jacobi solver. Though not as fast as the most advanced methods, its efficiency and accuracy is found to be sufficient for the real time simulations in this project.

In the Jacobi Solver, given a system of linear equations written in matrix form:

$$A\mathbf{x} = \mathbf{b}$$

the matrix A is decomposed into $D + C$, where D is a diagonal matrix, and C has only 0s on the diagonal:

$$(D + C)\mathbf{x} = \mathbf{b}$$

The system is then rewritten as

$$D\mathbf{x} = \mathbf{b} - C\mathbf{x}$$

Thus,

$$\mathbf{x} = D^{-1}(\mathbf{b} - C\mathbf{x})$$

which motives an iterative scheme: begin with an initial guess $\mathbf{x}_0$, and then iteratively compute

$$\mathbf{x}_{i+1} = D^{-1}(\mathbf{b} - C\mathbf{x}_i)$$

For a certain amount of iterations. Each iterations is simple to do, because D is only a diagonal matrix.

4.7.3 Results

Performances of the FLIP algorithm implemented in this project highly depend on the simulation parameters used. For a grid of size 50^3, approximately 100 Jacobi iterations are required at each time step to ensure incompressibility. And with roughly 8 particles inside each grid cell, the simulation alone (without rendering) runs at over 50 FPS(frames per second), which comfortably meets the requirements of real time applications. With rendering added, the frame rate drops to around 20FPS, which is caused by the rather expensive volume rendering(section 6.2.2) and surface reconstruction(section 6.1). Screenshots of some example simulations follow below.

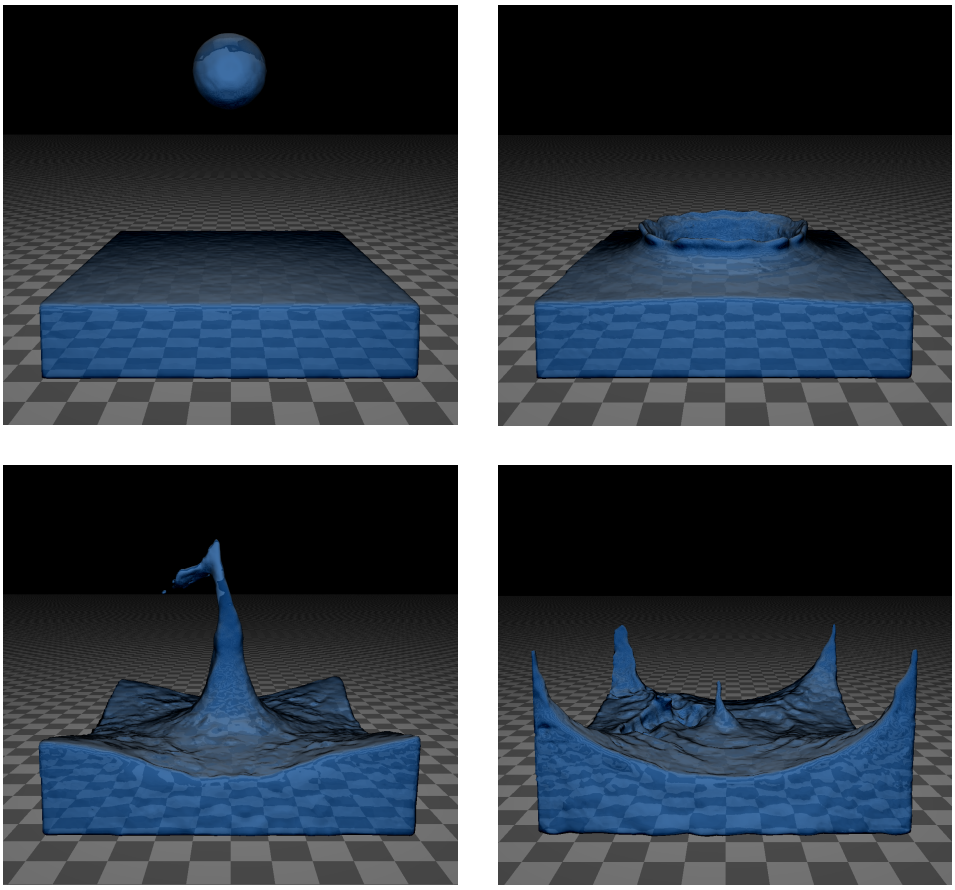


Figure 4.3: Ball drop. 50^3 grid, $208k$ FLIP particles

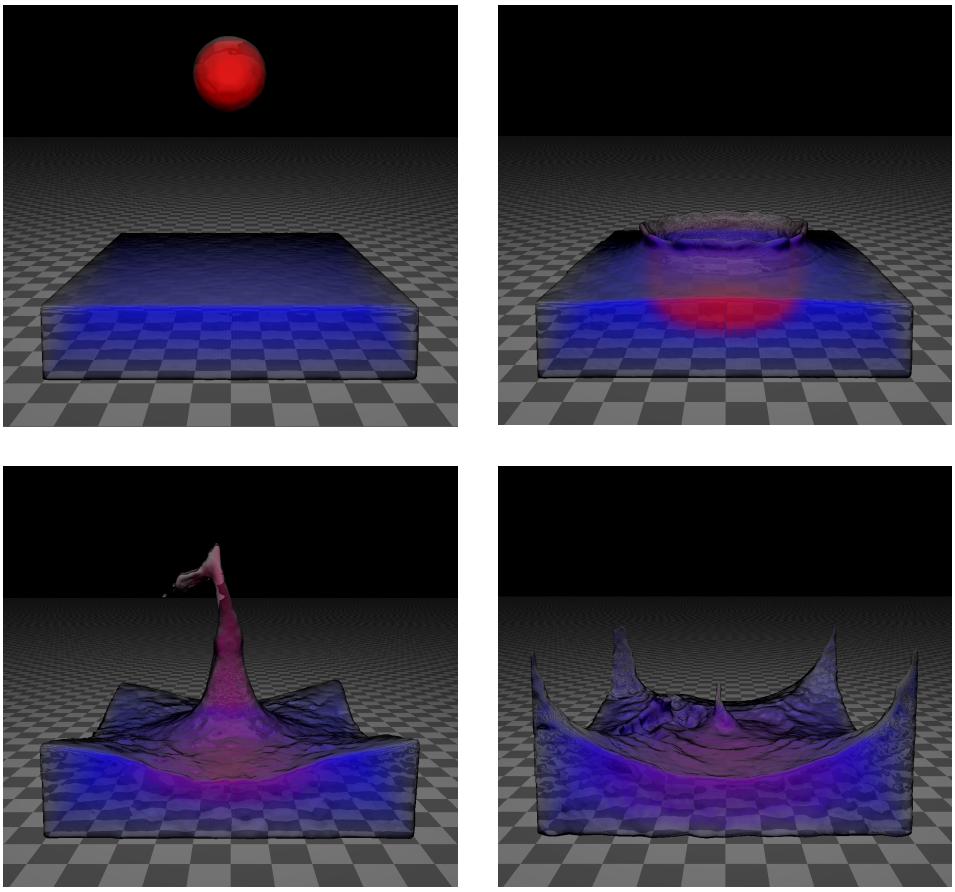


Figure 4.4: Same simulation as figure 4.3, except with 2 phases

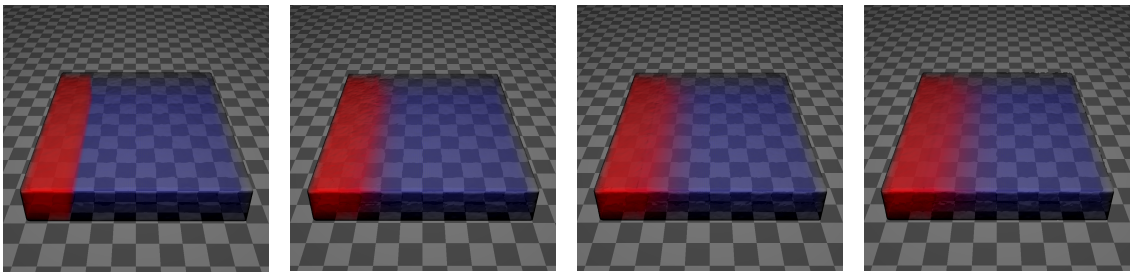


Figure 4.5: Diffusion with coefficient 10^{-3}

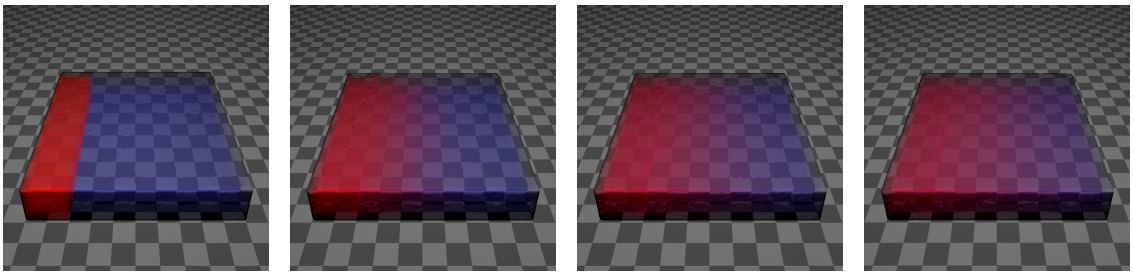


Figure 4.6: Diffusion with coefficient 10^{-2}

5 Particle-Based Simulations

Besides the grid-based fluid simulations algorithms, this project also studied and implemented a fundamentally different algorithm, known as **PBF** (Position-Based Fluids), which only uses particles to represent the fluid and its relevant scalar and vector fields. The principles of this algorithm is given in this chapter.

5.1 Smoothed Particle Hydrodynamics

PBF belongs to a family of algorithms called SPH(Smoothed Particle Hydrodynamics). SPH does not utilize a grid to discretize scalar and vector fields, and instead only uses a large cloud of particles to represent to the fluid. All quantities that are involved in the simulation are carried by these particles. For some quantity Q, either scalar or vector, and some location $\mathbf{x}$, SPH approximates Q at $Q(\mathbf{x})$ as a weighted average of Q carried by the nearby particles.

$$Q(\mathbf{x}) = \sum_{j=1}^{N} m_j \frac{Q_j}{\rho_j} W(\mathbf{x} - \mathbf{x}_j, h) \tag{5.1}$$

where N is the total amount of particles, m_j is the mass of the jth particle, $\mathbf{x}_j$ its position, ρ_j its density, and Q_j the value of Q it carries. Most importantly, W is a *smoothing kernel function*, and h its *smoothing radius*, which satisfies the properties:

$$\begin{aligned} \forall \mathbf{r} \quad s.t \quad ||\mathbf{r}|| > h, W(\mathbf{r}, h) = 0 \\ \int W(\mathbf{r}, h) d\mathbf{r} = 1 \end{aligned} \tag{5.2}$$

W is used to decide the contribution weight of each particle, which is greater if $\mathbf{x}_j$ is closer to $\mathbf{x}$. Thus, normally the value of $W(\mathbf{r}, h)$ only depends on $||r||$, and is 0 if $||r|| > h$.

A convenient and important property of the SPH framework is that, computing the gradient/Laplacian of a quantity can be done by only applying the gradient/Laplacian

operator on the smoothing kernel:

$$\begin{aligned} \nabla Q(\mathbf{x}) &= \sum_j m_j \frac{Q_j}{\rho_j} \nabla W(\mathbf{x} - \mathbf{x}_j, h) \\ \nabla \cdot \nabla Q(\mathbf{x}) &= \sum_j m_j \frac{Q_j}{\rho_j} \nabla \cdot \nabla W(\mathbf{x} - \mathbf{x}_j, h) \end{aligned} \tag{5.3}$$

A common choice of W is the "poly6" function:

$$W_{poly6}(\mathbf{r}, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - ||\mathbf{r}||^2)^3 & \text{if } 0 \leq ||\mathbf{r}|| \leq h \\ 0 & \text{otherwise} \end{cases}$$

However, this function has the problem that, its gradient vanishes as $\mathbf{r}$ approaches $\mathbf{0}$. Thus, when computing ∇W, a alternative kernel called the "spiky" kernel is used:

$$\nabla W_{spiky}(\mathbf{r}, h) = \begin{cases} \frac{-45}{\pi h^6 ||\mathbf{r}||}(h - ||\mathbf{r}||)^2 \mathbf{r} & \text{if } 0 \leq ||\mathbf{r}|| \leq h \\ \mathbf{0} & \text{otherwise} \end{cases}$$

The follwing figure illustrates the difference between the two kernels:

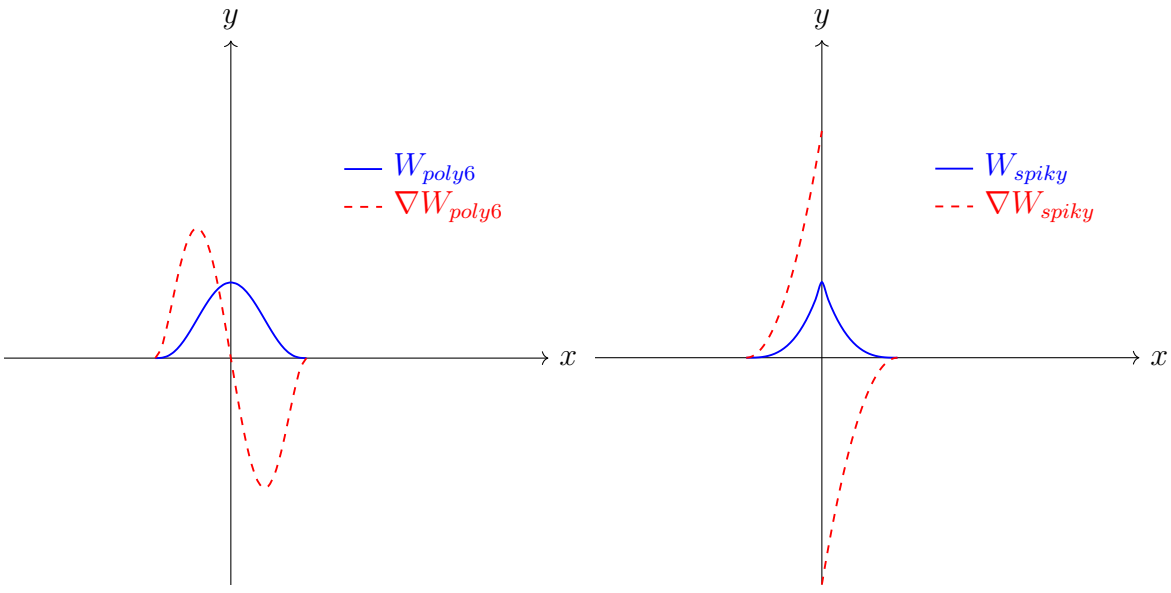


Figure 5.1: Scalar versions of W_{poly6} and W_{spiky}

In the first paper where SPH is used for computer graphics, written by Matthias Müller[12] in 2003, each quantity involved in the Navier-Stokes momentum equation is explicitly written in the forms of formula 5.1 or 5.3. An explicit time stepping integration is then used to update the velocity field and particle positions. However, the incompressibility condition was ignored in that paper. Since then, many extensions

and modifications to the original SPH scheme was proposed, which enforces incompressibility. The newest and currently most popular one of these extensions is proposed in 2013, again developed by Müller and his colleague Macklin in NVIDIA[10]. This extension is the Position Based Fluids algorithm.

5.2 Position Based Fluids

In Position Based Fluids, or PBF, the incompressibility condition is enforced by putting an explicit constraint on the density of the fluid. To begin with, for each particle i at position $\mathbf{x}_i$ the density at each particle can be derived using formula 5.1:

$$\begin{aligned}\rho_i &= \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{x}_i - \mathbf{x}_j, h) \\ &= \sum_j m_j W(\mathbf{x}_i - \mathbf{x}_j, h)\end{aligned}$$

PBF treats all particles as having equal mass. This mass can be chosen to be 1, which simplifies the formula:

$$\rho_i = \sum_j W(\mathbf{x}_i - \mathbf{x}_j, h) \tag{5.4}$$

Note that ρ_i is in fact a function of the positions of all the particles in the simulation: $\rho_i = \rho_i(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N)$ where N is the total amount of particles. Using the density of each particle, PBF defines a constraint quantity C_i for each particle i:

$$C_i(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N) = \frac{\rho_i}{\rho_{rest}} - 1$$

where ρ_{rest} is the rest density of the fluid. The incompressibility constraint can then be explicitly written as:

$$C_i(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N) = 0$$

Collectively, using the concatenated vector $\mathbf{X} = [\mathbf{x}_1^T, \mathbf{x}_2^T, \ldots, \mathbf{x}^T]^T$ to denote the positions of all particles, the constraint of the ith particle can be written as:

$$C_i(\mathbf{X}) = 0 \tag{5.5}$$

In each time step of the PBF simulation, the algorithm begins by predicting a new position $\mathbf{x}_i^*$ for each particle i using its velocity $\mathbf{u}_i$:

$$\mathbf{x}_i^* = \mathbf{x}_i + \triangle t \mathbf{u}_i$$

If the particle happens to travel into a solid region, some basic collision handling should also be applied (e.g clamping $\mathbf{x}_i^*$ outside the solid).

Let $\mathbf{X}^* = [\mathbf{x}_1^{*T}, \mathbf{x}_2^{*T}, \ldots, \mathbf{x}_N^{*T}]^T$ be the collective predicted position, then, in all likelihood, $\mathbf{X}^*$ will not satisfy the density constraints. Thus, PBF aims to find a set of corrections for the particles' positions, $\triangle\mathbf{X} = [\triangle\mathbf{x}_1^T, \triangle\mathbf{x}_2^T, \ldots, \triangle\mathbf{x}_N^T]^T$, which restores the density constraints:

$$C_i(\mathbf{X}^* + \triangle\mathbf{X}) = 0 \quad \forall i \in [0 \ldots N] \tag{5.6}$$

PBF approximates $\triangle\mathbf{X}$ by restricting it in the direction of $\nabla C_i(\mathbf{X}^*)$:

$$\triangle\mathbf{X} \approx \lambda_i \nabla C_i(\mathbf{X}^*) \tag{5.7}$$

where λ_i is a scalar coefficient. To find a suitable λ_i, PBF considers the 1st order Taylor series of $C_i(\mathbf{X}^* + \triangle\mathbf{X})$:

$$\begin{aligned} C_i(\mathbf{X}^* + \triangle\mathbf{X}) &\approx C_i(\mathbf{X}^*) + (\nabla C_i(\mathbf{X}^*))^T \triangle\mathbf{X} \\ &\approx C_i(\mathbf{X}^*) + (\nabla C_i(\mathbf{X}^*))^T \nabla C_i(\mathbf{X}^*)\lambda_i \\ &= C_i(\mathbf{X}^*) + ||\nabla C_i(\mathbf{X}^*)||^2 \lambda_i \end{aligned} \tag{5.8}$$

Combining this with formula 5.6, it can be derived that

$$\begin{aligned} \lambda_i &\approx \frac{-C_i(\mathbf{X}^*)}{||\nabla C_i(\mathbf{X}^*)||^2} \\ &= \frac{-C_i(\mathbf{X}^*)}{\sum_j ||\nabla_{\mathbf{x}_j^*} C_i(\mathbf{X}^*)||^2} \end{aligned} \tag{5.9}$$

The summation in the denominator can be separated into two cases: when $j = i$ and $j \neq i$

$$\lambda_i \approx \frac{-C_i(\mathbf{X}^*)}{||\nabla_{\mathbf{x}_i^*} C_i(\mathbf{X}^*)||^2 + \sum_{j \neq i} ||\nabla_{\mathbf{x}_j^*} C_i(\mathbf{X}^*)||^2} \tag{5.10}$$

And then the SPH formula for gradients 5.3 can be used to explicit compute each term:

$$\begin{aligned} \nabla_{\mathbf{x}_i^*} C_i(\mathbf{X}^*) &= \frac{1}{\rho_{rest}} \sum_j \nabla_{\mathbf{x}_i^*} W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) \\ &= \frac{1}{\rho_{rest}} \sum_j \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) \\ \nabla_{\mathbf{x}_j^*} C_i(\mathbf{X}^*) &= \frac{1}{\rho_{rest}} \nabla_{\mathbf{x}_j^*} W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) \\ &= \frac{-1}{\rho_{rest}} \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) \end{aligned}$$

Substituting these back into formula 5.10 gives

$$\begin{aligned} \lambda_i &\approx \frac{-C_i(\mathbf{X}^*)}{\frac{1}{\rho_{rest}^2}(||\sum_j \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h)||^2 + \sum_j ||\nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h)||^2)} \\ &= \frac{-(\frac{\rho_i}{\rho_{rest}} - 1)}{\frac{1}{\rho_{rest}^2}(||\sum_j \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h)||^2 + \sum_j ||\nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h)||^2)} \end{aligned} \tag{5.11}$$

which is now in a form ready to be translated into CUDA code.

Having obtained λ_i for each particle constraint C_i, the corrections to particle locations can be computed as a sum of the corrections from all constraints:

$$\begin{aligned}
\triangle \mathbf{x}_i &= \sum_j \lambda_j \nabla_{\mathbf{x}_i^*} C_j(\mathbf{X}^*) \\
&= \lambda_i \nabla_{\mathbf{x}_i^*} C_i(\mathbf{X}^*) + \sum_{j \neq i} \lambda_j \nabla_{\mathbf{x}_i^*} C_j(\mathbf{X}^*) \\
&= \frac{1}{\rho_{rest}} \sum_j \lambda_i \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) + \frac{-1}{\rho_{rest}} \sum_j \lambda_j \nabla W(\mathbf{x}_j^* - \mathbf{x}_i^*, h) \\
&= \frac{1}{\rho_{rest}} \sum_j \lambda_i \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) + \frac{1}{\rho_{rest}} \sum_j \lambda_j \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h) \\
&= \frac{1}{\rho_{rest}} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*, h)
\end{aligned} \tag{5.12}$$

Since the $\triangle \mathbf{X}$ computed in this manner is only an approximation, it will not perfectly restore the density constraints. Therefore, more than one iterations of position corrections will be needed. Typically, around 5 iterations of corrections are sufficient to give visually plausible results.

With the density constraints at its core, the PBF simulation algorithm works as follows:

Algorithm 3: PBF simulation step

```
foreach particle i do
    apply external forces: u_i := u_i + Δt g ;
    predict position: x*_i := x_i + Δt u_i and handle solid boundary collisions;
end
for i := 0 to solverIterations do
    foreach particle i do
        calculate ρ_i using formula 5.4;
        calculate λ_i using formula 5.11;
    end
    foreach particle i do
        calculate Δx_i using formula 5.12;
        update prediction x*_i := x*_i + Δx_i and handle solid boundary
          collisions;
    end
end
foreach particle i do
    update velocity: u_i := 1/Δt (x*_i − x_i) ;
    update position: (x_i := x*_i) ;
end
```

5.3 Implementation

To efficiently implement algorithm 3 on GPU, the spatial indexing algorithm, which was used in FLIP and described in subsection 4.7.1, must again be applied. Specifically, in line 7, 8, and 11, the computation for ρ_i, λ_i, and $\triangle \mathbf{x}_i$ all involve summations over all particles, where the terms being summed are weighted by W or ∇W. This indicates that it suffices to only sum over the neighboring particles within a radius of h, which can be founded using spatial indexing.

After incorporating spatial indexing, the rest of the PBF algorithm can be straightforwardly to parallelized in CUDA. Each iteration of the "foreach" loops can be designated to a CUDA thread, because different iterations operate on different particles, and there are no data dependencies. In the implementation of this project, with roughly 40000 particles, the PBF simulation and rendering runs at approximately 20FPS.

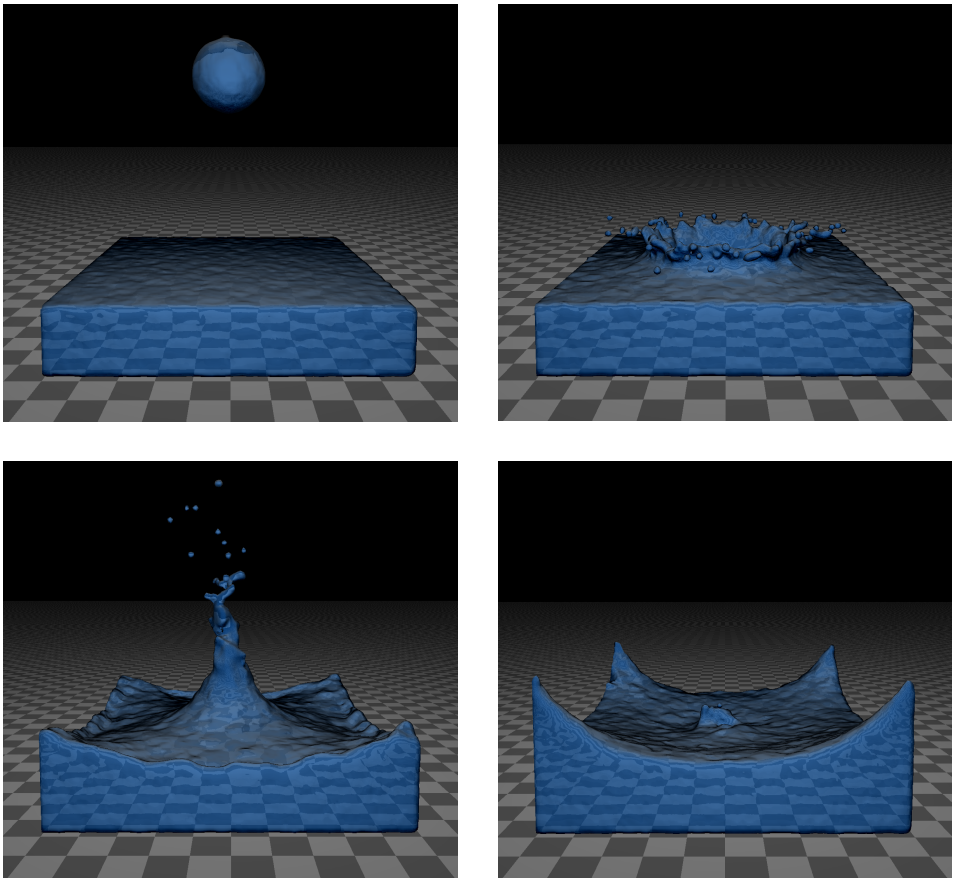


Figure 5.2: Same simulation as figure 4.3, except in PBF

6 Rendering

In computer graphics applications, the ultimate goal for fluid simulation is usually so that the fluid can be rendered and displayed to the human user. Thus, in addition to the FLIP and PBF algorithm, this project also implements a rendering pipeline, so that the simulation and rendering and be conducted together in real time. This chapter introduces the key rendering facilities that are implemented and the algorithms used.

6.1 Surface Reconstruction

To render liquids, the first step is to represent the surface of the liquid as a mesh of triangles, which can then be fed into the GPU rendering pipeline. Since this triangle mesh is not maintained by the simulation algorithms, it needs to be generated in each frame. Fortunately, both FLIP and PBF maintains a cloud of particles that occupy the entire fluid region, so if an algorithm can reconstruct a triangle mesh from particles, it can be used to render the results of both simulation algorithms.

In the paper where they also described FLIP[18], Zhu and Bridson utilized a concept called *Signed Distance Field* to perform surface reconstruction. The signed distance field $\phi : \mathbb{R}^3 \to \mathbb{R}$ is a scalar field, with the property that, given a 3D cartesian coordinate $\mathbf{x}$:

- $\phi(\mathbf{x}) = 0$ if $\mathbf{x}$ is on the boundary of the fluid region.
- $\phi(\mathbf{x}) = d$ if $\mathbf{x}$ is outside the fluid region, and the shortest distance between $\mathbf{x}$ and any point on the surface is d.
- $\phi(\mathbf{x}) = -d$ if $\mathbf{x}$ is inside the fluid region, and the shortest distance between $\mathbf{x}$ and any point on the surface is d.

Thus, the magnitude of $\phi(\mathbf{x})$ indicates the distance between $\mathbf{x}$ and the surface, and the sign of $\phi(\mathbf{x})$ indicates whether $\mathbf{x}$ is in the inside or outside. The fluid surface is

then precisely the set of points where ϕ evaluates to 0. In literature, this is often called the *zero isocontour* or *zero isosurface*.

Zhu and Bridson's method[18] begins by considering the special case where there is only a single particle in the fluid. Assume the particle is centered at $\mathbf{x}_0$ and has radius r_0, the signed distanced field of this spherical fluid region must then be:

$$\phi(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_0|| - r_0$$

This is then generalized to N particles, where $\mathbf{x}_0$ is replaced by a weighed average of the center positions of nearby particles:

$$\begin{aligned} \phi(\mathbf{x}) &= ||\mathbf{x} - \overline{\mathbf{x}}|| - r \\ \overline{\mathbf{x}} &= \frac{\sum_{i=1}^{N} W(\mathbf{x} - \mathbf{x}_i, h)\mathbf{x}_i}{\sum_{i=1}^{N} W(\mathbf{x} - \mathbf{x}_i, h)} \end{aligned} \tag{6.1}$$

where it is assumed that all particles have the same radius r. The W and h are respectively a smoothing kernel and a smoothing radius, which behaves similarly as the ones used during the PBF simulation. The specific one proposed by Zhu and Bridson and used in this project is

$$W(\mathbf{r}, h) = max(0, (1 - \frac{||\mathbf{r}||^2}{h^2})^3)$$

Using formula 6.1, a discretized signed distance field can be computed and stored on a 3D grid. Notice that since the summation in $\overline{\mathbf{x}}$ again sums over neighbor particles, the spatial indexing technique described in section 4.7.1 will again be needed when implementing the formula. The 3D grid of discrete ϕ values is often called a *level set* in literature[3].

Having computed the discrete signed distance field ϕ, it remains to extract its zero isosurface as a triangle mesh. This is done using a famous algorithm called *Marching Cubes*, invented by Wyvill[16] and Lorensen[8]. The algorithm considers each cubic grid cell in the level set grid, and puts a triangle vertex at each edge of the cube if one end of the edge is outside the fluid($\phi > 0$) and the other end is inside the fluid ($\phi \leq 0$). Since for each cube has 8 vertices, and each vertex can be either inside or outside the fluid, there're a total of $2^8 = 256$ different cases for each cube. A selection of them is shown in this figure:

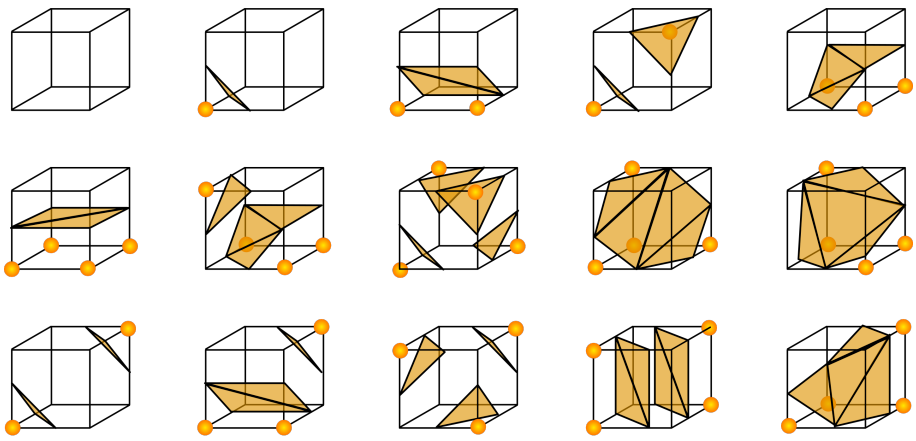
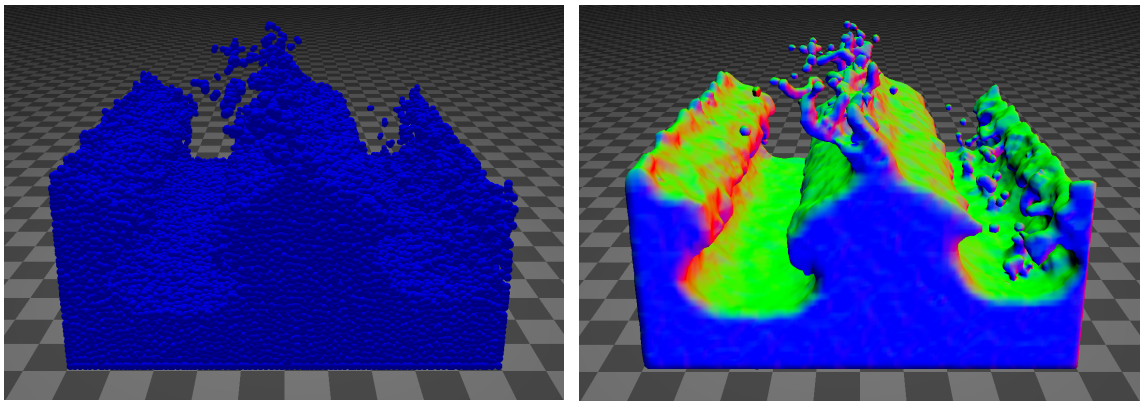


Figure 6.1: 15 of the 256 cases for each cube. The colored vertices are the ones inside the fluid. The orange triangles are the results generated for each cube. Image courtesy of the Wikipedia page on Marching Cubes.

The triangles generated for all the cells connect together into a watertight mesh, which can be fed into the OpenGL pipeline for rendering. This project implements the Marching Cubes algorithm by using a precomputed look-up table, which maps each of the 256 cases into an array of corresponding triangle vertices.

When the triangle mesh of the fluid surface is being rendered, the normal vectors of the surface is needed. This project chooses to approximate the normal vector using the discrete gradient of the signed distance field, $\nabla\phi$. This vector represents the normal because, intuitively, $\nabla\phi$ points in the direction where ϕ increases the most, which is also the direction that perpendicularly points away from the liquid surface.



(a) PBF Particles

(b) Constructed mesh and its normal. Direction represented by colors: Red:$+x$ Green:$+y$ Blue:$+z$

Figure 6.2: Surface reconstruction from particles

6.2 Surface Shading

After the triangle mesh representing the liquid surface is constructed, it is put through the OpenGL pipeline for shading. The most important task falls upon the fragment shader, where the color of each pixel on the mesh is computed. In order to generate realistic coloring, it is necessary to model how rays of light interact with the liquid.

6.2.1 Reflection And Refraction

As a ray of light hits the boundary between air and liquid, part of its energy is bounced away from the surface, while the rest of enters the liquid and continues to travel inside. The two resulting rays are respectively called the *reflection* and the *refraction*.

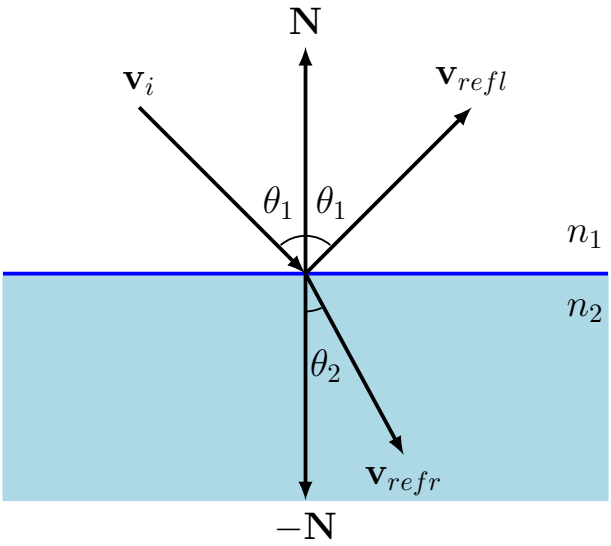


Figure 6.3: Reflection and Refraction.

Let $\mathbf{v}_i$ be the direction of the incident ray, and let $\mathbf{v}_{refl}$ be the direction of the reflected ray. The angle θ_i between $\mathbf{v}_i$ and the surface normal $\mathbf{N}$ will always be same as the angle between $\mathbf{N}$ and $\mathbf{v}_{refl}$. Knowing $\mathbf{v}_i$ and $\mathbf{N}$, $\mathbf{v}_{refl}$ can be computed as:

$$\mathbf{v}_{refl} = 2\mathbf{N}(-\mathbf{v}_i \cdot N) + \mathbf{v}_i \tag{6.2}$$

The direction of the refracted ray, $\mathbf{v}_{refr}$, is slightly more complicated to compute. The angle θ_2 between $\mathbf{v}_{refr}$ and the inverse of $\mathbf{N}$ is governed by the Snell's law:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

where n_1 is the *Index of Refraction* of air, and n_2 that of the liquid. Usually, n_1 is roughly equal to 1, and the n_2 for water is around 1.333. Based on Snell's law, Bec[2] derived a fast formula for computing $\mathbf{v}_{refr}$:

$$\begin{aligned} \mathbf{v}_{refr} &= (w-k)\mathbf{N} + n\mathbf{v}_i \qquad \text{where} \\ n &= \frac{n_1}{n_2} \\ w &= n(-\mathbf{v}_i \cdot N) \\ k &= \sqrt{1+(m-n)(m+n)} \end{aligned} \tag{6.3}$$

Besides the direction of the reflected and refracted ray, it's also necessary to know the ratio of the incident energy that is reflected. Let this ratio be F, then, due to conservation of energy, the ratio of the refracted light must be $1-F$. The exact value of F is determined by the Fresnel equation, which is also dependent on the polarization and spectral distribution of the light. Due to the complexity of these equations, real time computer graphics applications often use an approximation given by Schlick[13]:

$$\begin{aligned} F &= F_0 + (1-F_0)(1-(-\mathbf{v}_i \cdot \mathbf{N}))^5 \quad \text{,where} \\ F_0 &= \left(\frac{n_2-n_1}{n_2+n_1}\right)^2 \end{aligned} \tag{6.4}$$

In the OpenGL fragment shader, the final out put color will be F multiplied by the color of the reflection, and $1-F$ multiplied by the color of the refraction. The reflected color is straightforward to compute, by tracing the ray in the direction of $\mathbf{v}_{refl}$. However, for the refracted ray, because its path lies within the liquid, it's necessary to consider the behavior of the light ray within a colored medium. To make things more complicated, since this project supports simulating multiple phases of fluids, different fluids of different color could be mixed together in the same region. This is explored in the next subsection.

6.2.2 Multiple Fluids Rendering

As a ray of light travels past a region of colored liquid, the energy of the light ray diminishes as a result of scattering and absorption by the liquid particles. This process is quantified by the *Beer-Lambert Law*, which defines the ratio T_r of the light that remains after it travels through the fluid region:

$$\begin{aligned} T_r &= e^{-\tau} \quad \text{where} \\ \tau &= \int_0^d \sigma_t(\mathbf{o}+x\mathbf{v})dx \end{aligned}$$

In this formula, $\mathbf{o}$ is the point where the ray enters the liquid, $\mathbf{v}$ its direction of travel, and d the length of the path of the ray inside the water. The function $\sigma_t(\mathbf{x})$ indicates how much of the light becomes extinct at location $\mathbf{x}$, and the integral of the extinction across the entire path is τ, the *optical thickness*.

For a fluid consisting of only one type of fluid, the parameter σ_t stays constant, so τ can be computed simply as $d\sigma_t$. However, in a region where multiple fluids are mixed together, the integral must be explicitly evaluated, usually numerically.

Moreover, the extinction parameter σ_t is highly dependent on the wavelength of the light, which is exactly the cause for different fluids to have different color. For example, a red liquid has the lowest σ_t for the wavelength of red, thereby allowing more red light to pass through. As a result, the numerical integration of τ needs to be performed for each of the RGB channels.

A frequently used method for computing τ is *ray marching*, where the ray in question is progressed a small step at a time, accumulating the extinction along the way. Alternatively, this project invented a novel algorithm for accumulating σ_t, which is more efficient and directly makes use of the FLIP particles, which carry the concentration information:

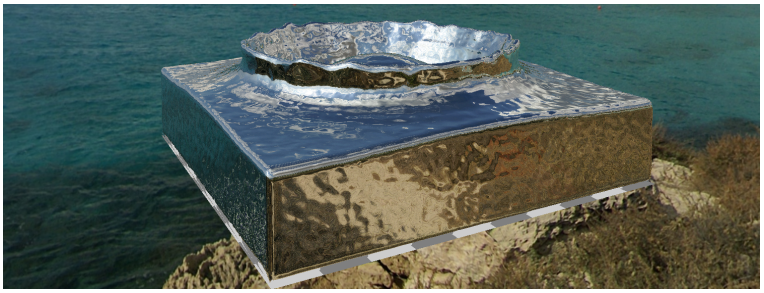
1. Create a texture image, where each color channel in the texture will correspond to the thickness of a specific fluid phase. Note that this limits the maximum number of different fluids to 4.

2. Configure OpenGL's compositing step (see section 3.2) to perform simple addition for overlapping pixels. This means that, if a pixel is covered by more than one particles, the results from rendering those particles will be added together to form the final result. On the GPU, compositing is not computed by ALUs, but hardware accelerated by a dedicated unit called the ROP(Render Output Unit).

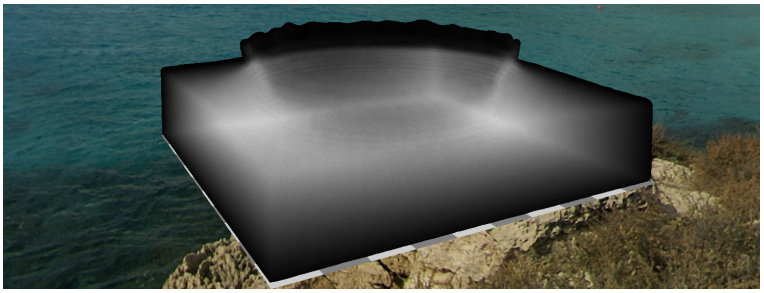
3. Render the FLIP particles and direct the output to the texture created in step 1. For each fluid particle p, and for each fluid phase f, render the concentration of f that p carries into the channel designated for f. The configuration from step 2 will ensure that the concentrations for all particles are accumulated.

4. While rendering the mesh, at each pixel, sample the concentration texture, and use the accumulated concentration to determine τ. The correct coloring can then be computed.

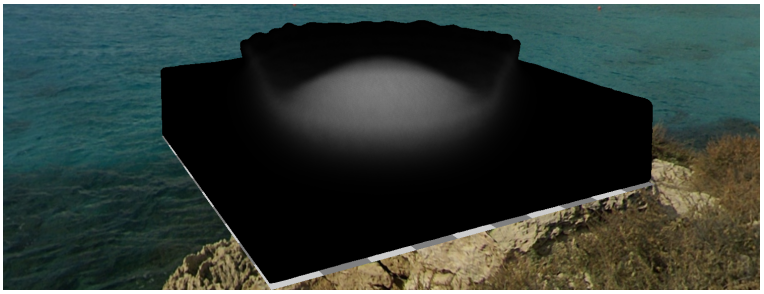
Example results of the intermediary render stages is shown in this figure:



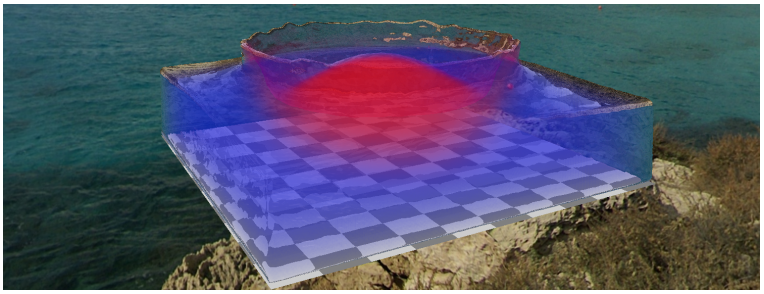
(a) The reflection



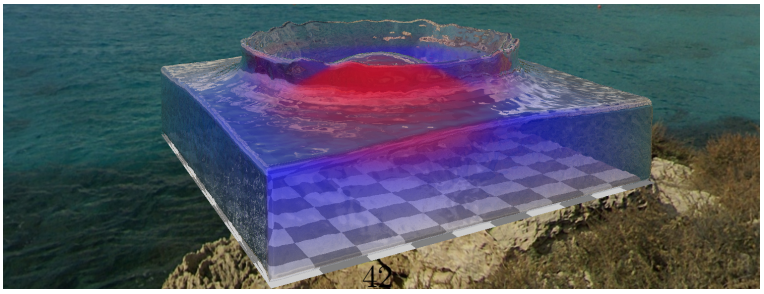
(b) The thickness texture of the blue liquid.
Brighter color indicates greater thickness.



(c) The thickness texture of the red liquid



(d) The refraction color.



(e) Final result

Figure 6.4: Intermediary and final outputs of the renderer. The images show the moment after a large ball of red liquid falls into a box of blue liquid

7 Conclusions

This project explored how modern GPUs can be programmed to efficiently simulate and render multiphase fluids. A software was built that was able to perform real time fluid simulation, and simultaneously render realistic images to be displayed. The project mostly focuses on parallelizing existing algorithms, but invents an original algorithm for performing fast multiphase fluid volume rendering. The software created is named *Aquarius*, and its full source code is available at `https://github.com/AmesingFlank/Aquarius`.

During the development of this project, many ideas emerged which would all bring more value to this project. Unfortunately, the time permitted did not allow these ideas to be explored. The next section gives a brief summary of these.

7.1 Future Work

- Adding viscosity.

  This project focused on fluids that follow the Euler equations, which ignored the effect of viscosity as described in the full Navier-Stokes equations. While not problematic for water-like liquids, many other interesting visual effects of fluids (e.g melting chocolate) are only possible with viscosity.

- Solid-fluid coupling

  In the real world, the motion of fluids almost always induces or is induced by the motion of solids. Modelling the coupling between solid and fluid allows more useful simulations.

- Caustics, shadows, and foams

  The renderer built in this project is far from complete, there are still a lot of optical features of fluids that was not captured. The curved shapes of liquids can cause the refracted light to be focused onto certain areas, creating an alternating pattern of brightness and darkness, known as caustics and shadows. Also, fast

moving liquid sometimes have small regions that traps volumes of air, creating foams. These phenomenons are common in real life, and supporting them in the renderer can significantly increase the realism of the images generated.

- Performance Improvements

  The FLIP simulation code in this project is very efficient. However, other parts of the program, especially the renderer, still has a large room for improvements. In fact, currently, the rendering is taking around 5 times the computation time as the simulation. With more work, the performance of the renderer could definitely be improved.

- Publishing as a library

  In its current form, the software created by this project has limited practical use. However, with some extra engineering, the code could be turned into a distributable library that can be used to perform fluid simulation in other projects. Specifically, it could be adapted into a plugin for popular game engines such as Unity or Unreal, so that game programmers can use it to easily incorporate real time fluids in their work.

7.2 Personal Reflections

Working on this project has been a gratifying experience. The topic is very exciting, and there was so much to that I learnt during the process. Apart from the improvements mentioned in 7.1, which I wish I could have made more progress in, I am in general very happy with and proud of the results I achieved.

When I submitted the proposal of this project in February 2019, it was purely based on interest as I had no previous experience and understanding of fluid simulation. So, the first thing I had to do in this project was to learn about the physics of fluids. As a pure computer science student, this proved challenging at first, because the mathematics courses that CS students took virtually did not discuss any vector calculus. In the Easter vacation of 2019, I spent a substantial amount of time catching up with vector calculus and fluid dynamics, which was difficult, but quite educational and layed the ground for my future efforts.

Before I started reading academic papers on the topic, I read a book called *Fluid Simulation for Computer Graphics*[3], written by Robert Bridson, one of the leading experts in the field. This book gave a really good advice: always start by writing a 2D simulator, before moving on to 3D. I did follow this advice, and took it one step

further: I started by writing code that only runs on CPU, before moving on to a GPU implementation. This was definitely a good idea because, considering the level of complexity of my GPU implementation right now, it would have been impossible for me to implement it correctly when I was still very unfamiliar with the problem at hand. A lesson learnt here is that for complicated tasks, a good way to start is by building a simple prototype.

One of the biggest frustrations that I faced in this project is the performance of my program. When I first ported my CPU 2D fluid simulator code to GPU, the simulation was only running at an embarrassing 7 frames per second, without even incorporating any sophisticated rendering. I was quite anxious about this, because I really wished to achieve real time simulation and rendering, and that is for 3D. At that time, the performance problem was caused by the pressure solver (4.5), which was taking tremendously long to solve the pressure Poisson equation. After weeks of careful analysis, a few optimization tricks, and some helpful directions pointed out by my supervisor, Professor Joe Pitt-Francis, I eventually resolved this issue. There isn't a specific moral in overcoming this obstacle, but I certainly learnt a lot about numerical computation and GPU optimization because of it.

I absolutely enjoyed learning about fluids and coding a simulator, but, somewhat surprisingly, I also enjoyed writing this report very much. Not only was I able to talk about a project that I worked passionately on, but as I reiterated the methods I used, I cleared up a few technical details which I did not completely understand. Although, I did find it quite stressful to obey the 10,000 word limit. There are a few mathematical details and quite a lot of GPU optimizations, which I spent a considerable amount of time working on, but sadly could not fit into this report.

This project is possibly the biggest intellectual endeavor I have made in my years at Oxford(though waiting to be surpassed by my 4th year project), and I find it to be an unmissable part of my education. Through this project, I was exposed to many spectacular areas of computer graphics that I did not know existed, and I was familiarized with the techniques that is essential to these studies. I have grown as a computer science student because of this project.

Bibliography

[1] Alexis Angelidis and Fabrice Neyret. Simulation of smoke based on vortex filament primitives. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 87–96, 2005.

[2] Xavier Bec. Faster refraction formula and transmission color filtering. *Ray Tracing News*, 10(1):396, 1997.

[3] Robert Bridson. *Fluid simulation for computer graphics*. CRC press, 2015.

[4] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. In *ACM Siggraph 2011 Papers*, pages 1–10. 2011.

[5] Francis H Harlow and J Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The physics of fluids*, 8(12):2182–2189, 1965.

[6] Nahyup Kang, Jinho Park, Junyong Noh, and Sung Yong Shin. A hybrid approach to multiple fluid simulation using volume fractions. In *Computer Graphics Forum*, volume 29, pages 685–694. Wiley Online Library, 2010.

[7] ByungMoon Kim, Yingjie Liu, Ignacio Llamas, and Jaroslaw R Rossignac. Flowfixer: Using bfecc for fluid simulation. Technical report, Georgia Institute of Technology, 2005.

[8] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[9] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.

[10] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.

[11] Joe J Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

[12] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003.

[13] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[14] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3):350–371, 2008.

[15] Wladimir J van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98, 2009.

[16] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Soft objects. In *Advanced Computer Graphics*, pages 113–128. Springer, 1986.

[17] Shibiao Xu, Xing Mei, Weiming Dong, Zhiyi Zhang, and Xiaopeng Zhang. Interactive visual simulation of dynamic ink diffusion effects. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, pages 109–116, 2011.

[18] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*, 24(3):965–972, 2005.