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 three of the most widely used fluid simulation algorithm: FLIP (Fluid-Implicit-Particle), PBF (Position Based Fluids), and PCISPH (Predicative-Corrective Incompressible Smoothed Particle Hydrodynamics). 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 multiple fluid rendering, thereby capturing the diffusion phenomenon between different phases of fluids. Alongside the simulation, a real time liquid rendering scheme is implemented, which efficiently captures the refection and refraction effects, as well as the varying concentrations of colored fluid phases inside the liquid. These algorithms are integrated into a fully featured program, which accepts user-provided simulation parameters, then performs and renders the liquid simulation in a real time and visually plausible manner.

Contents

1 Introduction

1.1 Motivation

Fluids can be seen everywhere. The smoke coming out of chimney 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 enormous waves of the ocean splashing onto the surface. Many of these these phenomenons have interesting and even stunning visual effects, thus, quite often, plausible images of these fluids need to be computationally generated for purposes such as cinematics and video gaming.

Due to the complexity underlying the behavior of fluids, accurate numerical simulation of fluids often require a huge amount of computational resources. In areas such as aeronautic engineering, where the importance of accuracy completely overrides that of efficiency, hours of CPU time can be spent on the simulation of a few seconds of fluid motion. Real time computer graphics applications, such as video games, usually do not require this level of accuracy, but instead asks the simulation to be computed in roughly the same amount of time as the physical process it represents. This efficiency requirement is met with the help of modern GPUs, which have massively parallel computing abilities. This project demonstrates this by implementing three simulation algorithms on GPUs, all of which can perform simulations in real time.

For computer graphics applications, fluid simulation isn't the only task. It is equally necessary for the software to display the results of simulation (i.e, the shape and motion of the fluid) to the user. This is especially important for interactive video games, where the user experience highly depends on the quality of the rendering. This project thus studies and implements the real time photorealistic rendering of liquids, the most frequently simulated and visualized type of fluid.

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 fields (e.g the velocity field) that represents the fluids. The most widely used discretization method, known as the **MAC** (Marker and Cell) **grid**, was proposed by Harlow and Welch [4] 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**[10] and **BFECC**[6], both of which have efficient GPU implementations[3][12]. This project chooses to implement the advection algorithm known as **FLIP** (Fluid Implicit Particle)[13], 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.[5] showed how to extend the grid-based algorithms to capture the diffusion between multiple fluid phases (e.g red ink diffusing in transparent water). This project imple-

ments a modified version of the proposed algorithm, where FLIP, rather than BFECC, is used to advect the concentration of different fluid phases.

Parallel to the grid-based approach is the family of particle-based algorithms. For computer graphics, the most commonly used class of particle-based algorithms is known as **SPH** (Smoothed Particle Hydrodynamics). Introduced to computer graphics in 2003 by Müller [9], the SPH method represent the fluid by a moving cloud of particles, which carry the physical quantities with them. This project chooses to implement two extensions to SPH: **PCISPH**(Predicative-Corrective Incompressible SPH) by Solenthalar[11] and **PBF**(Position Based Fluids) by Macklin and Müller[8]. These extended algorithms improve upon the plain SPH in that they enforce the incompressibility constraint of fluids, which are important for visual fidelity.

Given a well performing simulation, either grid-based or particle-based, it remains a nontrivial task to visualize the fluid. This project follows the proposal by Zhu and Bridson [13], 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 [7], proposed by Lorensen, can then use this field to reconstruct the surface of the fluid into a triangle mesh representation, which is suitable 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 PCISPH and PBF algorithm are 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 takes into account 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 origin 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.

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 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:

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

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 $\dfrac{\partial \phi}{\partial t}$. To be more precise, the Eulerian derivative $\dfrac{\partial \phi}{\partial t}$, 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 $\dfrac{D\phi}{Dt}$, 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 $\frac{D}{Dt}$ will be automatically extended to be applied to vector fields, where each component of the vector field is differentiated separately. This allows the term $\frac{D\mathbf{u}}{Dt}$ 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 miscible fluids phases.

Concentration of different fluid phases will be represented using ratios.Specifically, let N be the total number of different fluid phases, the concentration of the ith fluid is represented by a scalar field α^i, where $\sum_{i=0}^{N} \alpha^i = 1$ at any point within the fluid.

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 then its surrounding, this difference will tend to "diffuse" into the surrounding, 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 GPUs 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. Thus, the GPU can hide the 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.

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's shader uses the interpolated output of the vertex shader as input, to determine the color of the pixel.

4. *Compositing* Since in the 3D scene, triangles might hide behind each other, some pixels might be covered by multiple triangles as well. The compositor's job is to determine which shading of the pixel to should be used, and in case of opaque objects, how the different shaded colors should be mixed together to give the final output.

In chapter 6, more details will be given on how this project utilizes this pipeline to realistically render liquid 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 *splitting* approach. As a simple example, consider the simple differential equation:

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

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 $\frac{dx}{dt} = f(x)$ with boundary condition $x(0) = f_0$, and $G_{g_0}(t)$ is a solution of $\frac{dx}{dt} = g(x)$ with boundary 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. $$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{g}$$

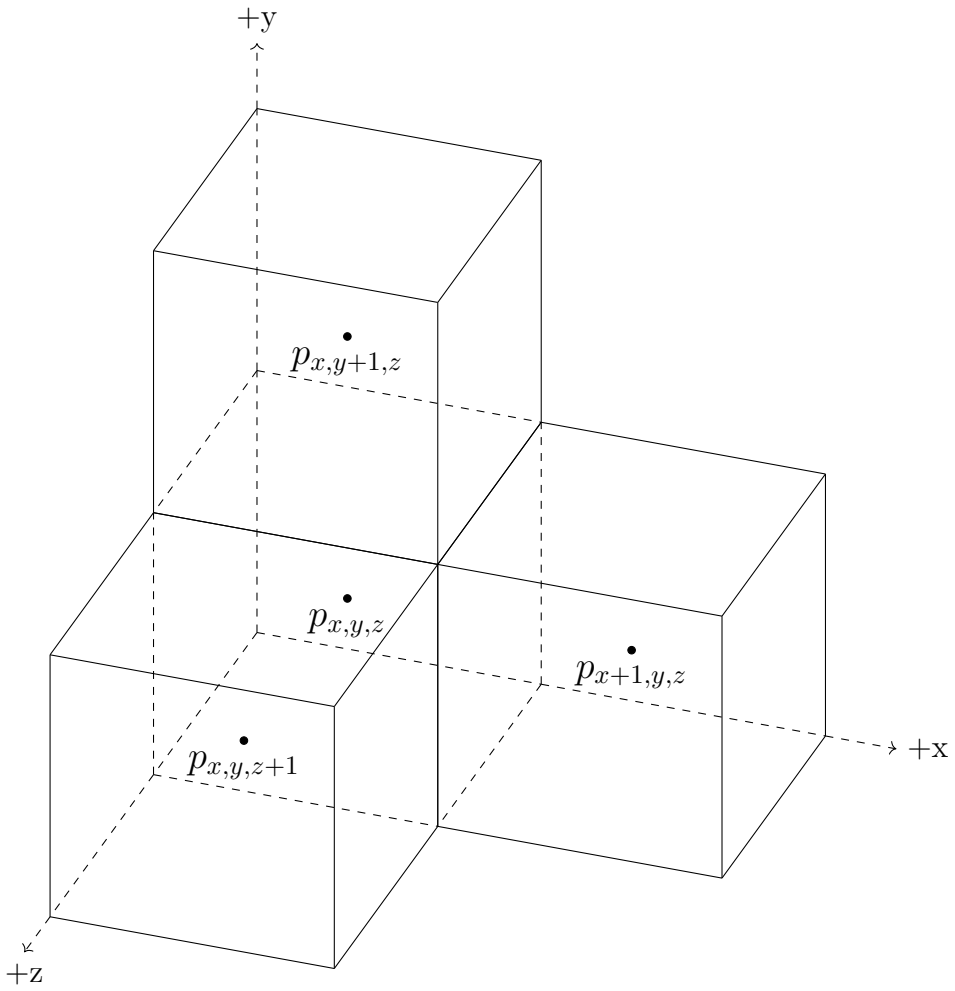
   Solving this equation is the process of exerting external forces (e.g gravity) on the fluid. The solid boundary conditions, as mentioned in section 2.4, can also be enforced in this step.

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, it stores the value of $\mathbf{u}_y$ sampled at its lower face, and it stores the value of $\mathbf{u}_z$ sampled at its back face, as illustrated in this figure:

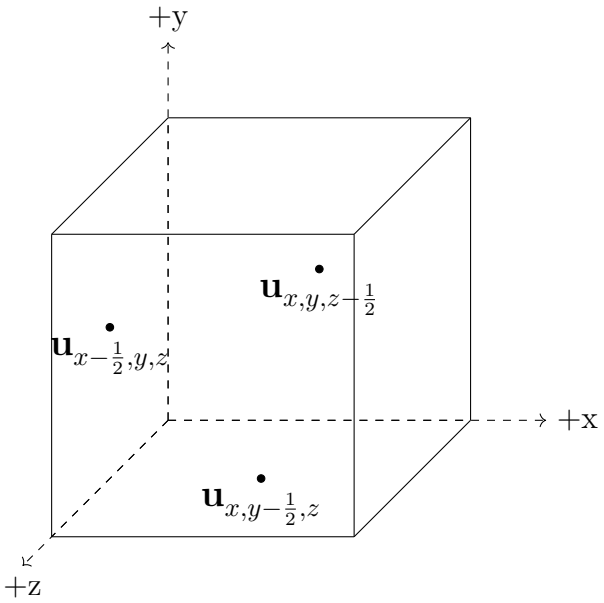
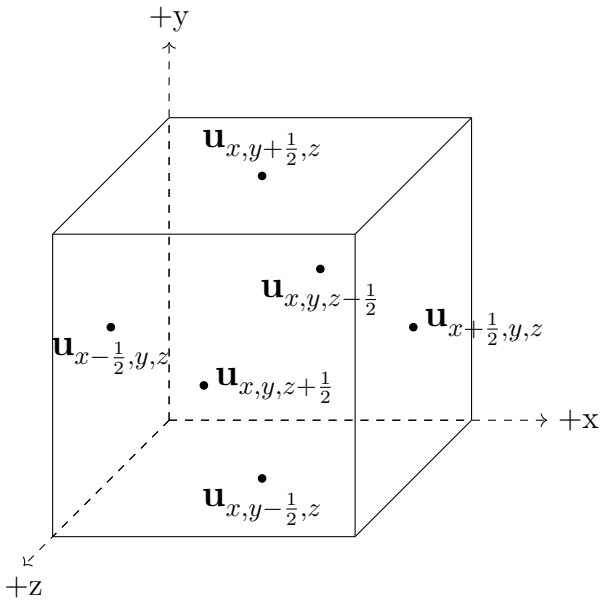


Figure 4.1: a 3D MAC grid cell and the velocity data it stores

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 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 to 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 when enforcing the incompressibility of the fluid, 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. Specifically, 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 according to the velocity field. Either by computing $\mathbf{x}_p^{new} = \triangle t \mathbf{u}_p$ to $\mathbf{x}_p$, or, for better precision, using a higher-order Runge-Kutta integration.

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 [13] 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, $\mathbf{g} \approx (0, -0.98, 0)$), applying the forces is then achieved by adding $\triangle t\mathbf{g}$. In a MAC grid, this needs to be 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 $\mathbf{u} \cdot \mathbf{n} = 0$, the components of $\mathbf{u}$ sampled at faces that represents 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 $\mathbf{u}_{[n+1]}^{compressible}$, the next step is to apply pressure within the fluid, so that this field becomes incompressible.

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 equations 4.1 and 4.2, the discrete version of this equation can be written:

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

Such an equation exists for all cells that containing some fluid (i.e contain some FLIP particles), and together, they form a system of linear equations, called the *Poisson*

pressure equation. The unknowns in the equations, $p_{i,j,k}$, correspond to the pressure values sampled at the center of the 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 fluid-air 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 solid-fluid boundary conditions were already satisfied in the external forces step.

With the boundary conditions satisfied, the equation become a system of N linear equations with N variables, with N being the amount of fluid cells. Solving this equations hence results in a discrete representation of the pressure field p, which 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 equation 4.2. This completes the simulation of this time step.

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

Algorithm 1: Single phase fluid FLIP simulation step

```
// At time step [n+1]
```
foreach *particle p* **do**
  $\mathbf{u}_p := \mathbf{u}_p + \mathbf{u}_{[n]}(\mathbf{x}_p) - \mathbf{u}_{[n-1]}(\mathbf{x}_p)$;
  $\mathbf{x}_p := \mathbf{x}_p + \mathbf{u}_p \triangle t$ (or using a higher order 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 average of the $\mathbf{u}_p$ carried by 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 not complicated 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, fortunately, the exact 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 $\alpha^i_{[n]} - \alpha^i_{[n-1]}$, 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 [5], 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  |  u_p := u_p + u_[n](x_p) − u_[n−1](x_p) ;
3  |  α^i_p := α^i_p + α^i_[n](x_p) − α^i_[n−1](x_p), for all fluid phases i ;
4  |  x_p := x_p + u_p△t (or using a higher order Runge-Kutta);
5  end
6  foreach grid cell at location (x, y, z) do
7  |  Find all particles within a radius of △x;
8  |  Compute u^advected_[n+1], as an average of the u_p carried by nearby particles;
9  |  Compute α^advected_[n+1], as an average of the α_p carried by nearby particles;
10 end
11 Apply external forces, u^compressible_[n+1] = FixSolidBoundary(u^advected_[n+1] + △tg);
12 Construct and solve the Poisson pressure equation, obtain pressure p;
13 Compute u_[n+1] = u^compressible_[n+1] − △t ∇p/ρ;
14 Construct and solve the discretized diffusion equation to obtain α^i_{n+1} for all
     fluid phases i.
```

Despite being a straightforward idea, using FLIP particles to advect the fluid concentrations is a novel invention by this project, because the related works only used other advection algorithms for concentration. The paper by Kang[5], for example, used BFECC. To achieve real time simulation, the grid used in this project is considerably more coarse then the grid used in Kang's offline simulation algorithm, but with the merits of FLIP, considerable visual results were achieved. This verifies Zhu and Bridson's comments that FLIP is able to preserve a lot of details even when the grid is coarse[13].

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 [4]. 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.

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*.

With all other steps being completely parallelizable, the only time-consuming step is sorting the array of particle hashes. However, this only needs to be done once per time step, so the cost is manageable. In fact, in the implementation of this project, by using the extremely optimized sorting library provided with the CUDA API, the cost of the sorting 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 of 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 [2] and *Geometric Multigrid*[3]. 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.

For the testing simulation configurations in this project, where the number of cells is in the order of 50^3, 100 to 200 Jacobi iterations suffices to simulate the fluid accurate enough to give plausible visual results.

5 Particle-Based Simulations

6 Rendering

A Sample Title

B Sample Title

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Barbara Solenthaler and Renato Pajarola. Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 papers*, pages 1–6. 2009.

[12] 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.

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