ONE-STEP FLOW MAPS FOR REAL-TIME FLUID SIMULATION WITH DYNAMIC BOUNDARIES

A Dissertation
Presented to
The Academic Faculty

By

Yutong Sun

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Interactive Computing
College of Computing

Georgia Institute of Technology

December 2025

ONE-STEP FLOW MAPS FOR REAL-TIME FLUID SIMULATION WITH DYNAMIC BOUNDARIES

Thesis committee:

Dr. Bo Zhu
School of Interactive Computing
Georgia Institute of Technology

Dr. Sehoon Ha
School of Interactive Computing
Georgia Institute of Technology

Dr. Yingjie Liu
School of Mathematics
Georgia Institute of Technology

Date approved: November 25, 2025

Shoot for the moon. Even if you miss, you'll land among the stars.

Norman Vincent Peale

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my advising Professor, **Bo Zhu**, for his invaluable academic guidance. He taught me how to conduct research independently, guiding me through reading papers, holding regular academic sharing sessions, and conducting biweekly one-on-one meetings. Without his mentorship, this dissertation would not have been accomplished. I also would like to thank the members of my thesis committee, Professor **Sehoon Ha** and Professor **Yingjie Liu**, for reviewing this dissertation.

Special thanks to my lab mates, **Yuchen Sun** and **Junlin Li**, for their open-source simulation and rendering code framework. This resource relieved me from writing boilerplate code, allowing me to focus on implementing my main novel contributions. I am also grateful to **Zhiqi Li** for sharing his math notes on flow map methods.

Finally, I want to express my heartfelt gratitude to my parents, **Yanhui Sun** (father) and **Hong Ma** (mother), for their unconditional and continuous support in every aspect of my life.

I also wish to thank everyone who has helped me, your small moments of kindness lifted me up and made this journey possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

AMGPCG Algebraic Multigrid Preconditioned Conjugate Gradient

APIC Affine Particle-in-Cell

BFECC Back and Forth Error Compensation and Correction

CG Conjugate Gradient

CPU Central Processing Unit

FLIP Fluid-Implicit-Particle

GPU Graphics Processing Unit

ICF Incomplete Cholesky Factorization

LFM Leapfrog Flow Maps

MCM Method of Characteristic Mapping

NFM Neural Flow Maps

OFM One-Step Flow Map

PCG Preconditioned Conjugate Gradien

PFM Particle Flow Maps

PIC Particle-in-Cell

RBGS Red-Black Gauss-Seidel

RK2 2th Order Runge-Kutta

RK4 4th Order Runge-Kutta

SOR Successive Over-Relaxation

SPD Symmetric Positive Definite

TVD-RK3 Total Variation Diminishing 3rd Order Runge-Kutta

UAAMG Unsmoothed Aggregation Algebraic Multigrid

SUMMARY

As modern Graphics Processing Unit (GPU) evolve, video games are adopting more advanced rendering technologies to produce high-fidelity image synthesis. To match this visual aesthetic, the physical simulation of dynamic effects is also required to achieve a higher degree of realism. This thesis introduces a high-precision fluid simulation method, flow map-based fluids, into the real-time graphics pipeline. The goal is to produce high-quality, high-vorticity smoke simulations suitable for video games, all while maintaining interactive frame rates. To achieve this, we first simplify the original, computationally expensive method by proposing a One-Step Flow Map (OFM). This modification reduces the cost of the high-precision simulation by using only a single integration sub-step per timestep. Furthermore, we combine our OFM solver with a real-time mesh voxelization pipeline. This allows solid objects, represented by standard triangle meshes, to apply dynamic boundary conditions to the fluid. By enabling the fluid to realistically interact with moving objects in the scene, this method significantly enhances the physical immersion of the simulation.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Dynamic visual effects, such as fluid flow, smoke, and fire, significantly enhance the visual fidelity and immersion of video games or any real-time interactive graphics application. However, their real-time simulation remains a formidable challenge due to the high computational cost associated with high-fidelity fluid dynamics. To circumvent this performance bottleneck, many development pipelines rely on pre-computed (or "baked") simulation data, which sacrifices dynamic interaction. An alternative common approach involves 2D sprite-based particle systems, while efficient, this method requires extensive artist-driven tuning to achieve realistic-looking results and often fails to capture true volumetric behavior. Although some modern engines (e.g., Unreal Engine 5, as shown in Figure 1.1) implement real-time 3D fluid simulation, these systems must still make significant compromises. They are often limited to low-resolution grids and employ non-physically-based heuristics—such as vorticity confinement [1, 2] or procedural noise fields [3] (e.g., Simplex noise [4])—to mitigate the characteristic loss of vorticity. Consequently, these methods can lack physical plausibility, temporal consistency, and realistic motion.

In real-time applications, the Semi-Lagrangian scheme [5] is often employed for gas and smoke simulation due to its ease of implementation and relative computational efficiency. However, the Semi-Lagrangian scheme is inherently diffusive. Its reliance on linear interpolation during the advection step smooths out high-frequency details, causing fluid vorticity to dissipate quickly. To counteract this and enhance visual realism, practitioners often apply heuristics. These include vorticity confinement [1, 2], which artificially magnifies existing vortices, or the injection of artificial vorticity from noise functions [2].

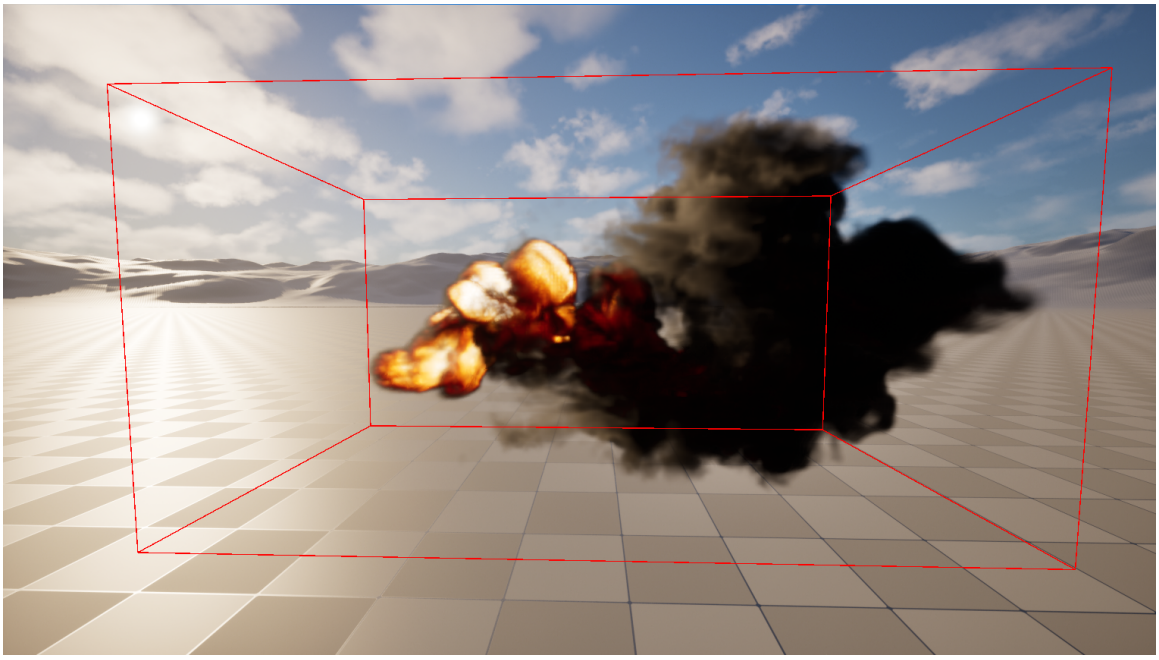

Figure 1.1: A 3D gas simulation which sources from a simple particle emitter in Unreal Engine 5.

While these techniques address the visual symptoms of dissipation, they are not physically grounded and require significant artistic tuning to produce convincing results.

Methods to preserve vorticity by reducing advection error also exist. For example, Back and Forth Error Compensation and Correction (BFECC) can elevate the first-order semi-Lagrangian scheme to second-order accuracy [6], which helps preserve vorticity features. Yet, this method often remains insufficient for highly realistic smoke as it still suffers from numerical dissipation. Hybrid methods, such as the Affine Particle-in-Cell (APIC) [7] and Fluid-Implicit-Particle (FLIP) [8] methods, exhibit significantly lower dissipation. While they perform well for liquid simulation, they are less frequently used for gas or smoke. Unlike liquids, gases lack a free surface and fill their entire domain, meaning a particle-based approach would be prohibitively memory-intensive.

This thesis addresses these challenges by exploring a real-time fluid simulation scheme for gas. The proposed method is designed to be computationally efficient, preserve high-frequency details such as vortices, and support dynamic interaction with solid objects in

the scene.

1.2 Related Work

To achieve smoke simulation that is both realistic and interactive, two primary components must be considered.

1.2.1 Incompressible Fluid Simulation

First is the accurate modeling of the fluid's physical behavior. We model the smoke as an inviscid, incompressible fluid of constant density. This model is physically justified for typical real-time scenarios. The effects of physical viscosity and molecular diffusion are negligible, as they are dominated by numerical dissipation inherent in the simulation schemes. Furthermore, when the smoke's velocity is well below the speed of sound, compressibility effects are also negligible. Consequently, the incompressible Euler equations (also known as the inviscid Navier-Stokes equations) are used to describe the time-evolution of the smoke's velocity field, $\mathbf{u} = (u, v, w)$ [9]. These equations will be presented in the next chapter.

Non-dissipative Advection Methods. Following the "Stable Fluids" solver introduced by Stam [10], significant research has been proposed to address the low-order accuracy of its Semi-Lagrangian advection, which causes significant numerical dissipation. Kim *et al.* introduced BFECC to computer graphics, which elevates the first-order Semi-Lagrangian scheme to a second-order method [11]. Later, Selle *et al.* adapted the MacCormack method for graphics, maintaining second-order accuracy with lower computational cost. Hybrid particle-in-cell methods, such as Particle-in-Cell (PIC), FLIP [8], and APIC [7], also exhibit low dissipation by transferring momentum between grids and particles, effectively preserving vorticity.

Flow Map Methods. The flow map, initially known as Method of Characteristic Mapping (MCM) [12], reduces numerical dissipation by tracking long-term flow maps

instead of advecting fluid quantities directly, thus decreasing the number of interpolations. Tessendorf's work [13] presents a detailed analysis and derivation of MCM. Nabizadeh *et al.* first introduced flow maps to advect covector fields, reducing the diffusion error of Semi-Lagrangian advection [14]. Neural Flow Maps (NFM) [15] further extends this by marching flow maps both forward and backward with a high-order Runge-Kutta scheme to represent accurate, consistent bidirectional mappings and their Jacobians. More recently, Zhou *et al.* proposed Particle Flow Maps (PFM) [16], which uses particles to generate bidirectional flow maps without a costly neural network. Chen *et al.* achieved fluid-solid interaction using a unified particle flow map representation. Leapfrog Flow Maps (LFM) [17], a recently proposed Eulerian solver, also avoids a neural buffer for storing flow maps and implements a fast, GPU-based matrix-free Algebraic Multigrid Preconditioned Conjugate Gradient (AMGPCG) Poisson solver for the projection step.

1.2.2 3D Mesh Voxelization

The second critical component for an interactive simulation is the handling of solid boundaries. The fluid must interact with arbitrary scene geometry, typically represented by triangle meshes. To facilitate this interaction, these continuous boundary representations must be converted into the simulation's discrete, grid-based domain. This process is known as voxelization or 3D scan-conversion.

Mathematically, the problem is to determine, for every voxel in a uniform 3D grid, whether it lies inside, outside, or on the boundary of a solid defined by a closed triangle mesh. This is a 3D extension of the 2D rasterization problem. Foundational work by Kaufman and Shimony established methods for 3D scan-converting geometric primitives, including polyhedra, into a voxel representation [18].

However, for simulation, a simple surface voxelization is insufficient, as it can contain gaps or "leaks" that violate the solver's boundary conditions. A key challenge is ensuring the voxelized surface is topologically correct and "separating" (i.e., watertight). Huang *et*

al. proposed an accurate method that guarantees separability and avoids common artifacts at edges and vertices, which is critical for robust boundary handling [19]. For real-time applications, this process must also be fast. GPU-based methods became common, such as the method using slice-by-slice rasterization introduced by Crane *et al.* [20]. This voxelized boundary is what the fluid solver ultimately uses to detect collisions and enforce boundary conditions.

1.3 Thesis Contributions

This thesis introduces a high-performance framework for real-time interactive fluid simulation, uniquely combining the high-fidelity advection of flow maps with fully dynamic, velocity-aware solid boundaries. The primary contributions of this work are:

- **One-Step Leapfrog Flow Map Advection Scheme**

  We build upon the recently proposed LFM method [17], which preserves high-frequency vorticity by marching grid-based flow maps. Our primary contribution is the development of a **one-step integration method**, which modifies the original multi-step flowmap marching in leapfrog scheme. This modification simplifies the marching step, reduces computational complexity, while retaining the excellent vorticity preservation of the flow map approach.

- **Framework for Integrating a Flow Map Solver with Voxelized Boundaries**

  This thesis presents a real-time pipeline that successfully **combines a flow-map-based solver with a dynamic mesh voxelization pipeline**. By integrating our one-step fluid solver with an on-the-fly voxelization module, we can handle arbitrary, moving triangle meshes. This module provides a complete, per-frame boundary condition by generating a voxel representation and—most critically—**sampling the solid's velocity** into the grid. This novel combination enables high-fidelity, one-way

momentum transfer from moving solids to detail-preserving fluid in a real-time context.

CHAPTER 2

MATH BACKGROUND

We first establish the notational conventions used throughout this thesis: scalars are denoted by lowercase letters, vectors by bold lowercase letters, and matrices by calligraphic symbols. A complete summary of all notations, including temporal and spatial derivatives, is provided in Table 2.1. We will then elaborate on the fluid evolution equations for flow maps, present the mathematical foundation of the flow map method, and apply this method to solve the governing fluid equations.

Table 2.1: Important symbols and notations used in this paper

Notation	Type	Meaning
ρ	scalar	density
$\mathbf{u}$	vector	velocity
p	scalar	pressure
$\mathbf{f}$	vector	external force
$\mathbf{m}$	vector	impulse
$\mathbf{X}$	vector	material particle location at initial time
$\mathbf{x}$	vector	material particle location at current time
$\mathbf{\Phi}$	vector	forward flow map
$\mathbf{\Psi}$	vector	backward flow map
$\mathcal{F}$	matrix	Jacobian of forward flow map
$\mathcal{T}$	matrix	Jacobian of backward flow map
ξ	scalar	gauge variable
$\mathbf{n}$	vector	normal direction
$\mathbf{p}$	vector	vertex position

2.1 Impulse Fluid Dynamics

The dynamics of an ideal, incompressible fluid are commonly described by the **Euler equations**. In their standard formulation, they consist of a momentum equation and an incom-

pressibility constraint:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \mathbf{f} \\ \nabla \cdot \mathbf{u} = 0 \end{cases} \tag{2.1}$$

Here, $\mathbf{u}$ is the fluid's velocity field, p is the pressure, and $\mathbf{f}$ represents external body forces like gravity.

An alternative and powerful way to express the Euler equations is through the **impulse formulation** [21]. This approach recasts the dynamics in terms of an intermediate velocity field, the impulse $\mathbf{m}$. Before presenting the equations, it's useful to define the **material derivative**, $\frac{D}{Dt}$, which describes the rate of change of a quantity following a fluid particle:

$$\frac{D(\cdot)}{Dt} = \frac{\partial(\cdot)}{\partial t} + (\mathbf{u} \cdot \nabla)(\cdot) \tag{2.2}$$

The impulse form is then given by the following system of equations:

$$\begin{cases} \dfrac{D\mathbf{m}}{Dt} = -(\nabla \mathbf{u})^T \mathbf{m} & (2.3) \\ \nabla^2 \varphi = \nabla \cdot \mathbf{m} & (2.4) \\ \mathbf{u} = \mathbf{m} - \nabla \varphi & (2.5) \end{cases}$$

This formulation separates the evolution of the fluid's momentum from the constraint of incompressibility. Each part of the equation will be explained below:

- **Impulse ($\mathbf{m}$):** The impulse can be thought of as a temporary velocity field that contains the fluid's rotational information (vorticity). Equation 2.3 describes how this impulse is advected and stretched by the velocity field $\mathbf{u}$. However, $\mathbf{m}$ is not guaranteed to be divergence-free.

- **Projection to a Divergence-Free Field:** To obtain the final, physical velocity $\mathbf{u}$, the impulse field $\mathbf{m}$ must be projected onto a divergence-free space. This step is crucial for satisfying the incompressibility condition ($\nabla \cdot \mathbf{u} = 0$).

- **The Projection Mechanism:** This projection is achieved using a Helmholtz-Hodge decomposition [22], as shown in Equation 2.5. The impulse $\mathbf{m}$ is split into a divergence-free component ($\mathbf{u}$) and a curl-free component (the gradient of a scalar potential, $\nabla\varphi$). By subtracting this gradient term, we enforce the incompressibility of $\mathbf{u}$. The scalar potential φ acts as a gauge to remove any divergent parts from $\mathbf{m}$ [15, 16].

The relationship between these equations becomes clear when we enforce the incompressibility of $\mathbf{u}$. By taking the divergence of Equation 2.5:

$$\nabla \cdot \mathbf{u} = \nabla \cdot \mathbf{m} - \nabla \cdot (\nabla\varphi)$$

Since we require an incompressible flow, $\nabla \cdot \mathbf{u} = 0$. This directly leads to the Poisson equation for the scalar potential φ shown in Equation 2.4:

$$0 = \nabla \cdot \mathbf{m} - \nabla^2\varphi \quad \Longrightarrow \quad \nabla^2\varphi = \nabla \cdot \mathbf{m}$$

Thus, the entire system provides a clear workflow: first, evolve the impulse, then solve for the potential φ that projects the impulse into a divergence-free velocity field $\mathbf{u}$.

2.2 Flow Maps Foundation

Consider a velocity field $\mathbf{u}(\mathbf{x}, \tau)$ within a fluid domain Ω. We define a virtual material point, initially positioned at $\mathbf{X} \in \Omega$, which advects with the velocity field $\mathbf{u}$ over time. At a later time t, this point will have traveled to a new location $\mathbf{x}$. We define the **forward flow map** $\Phi(\cdot, t) : \Omega_0 \to \Omega_t$ as the function that maps the initial position to the current position:

$$\begin{cases} \frac{\partial\Phi(\mathbf{X},\tau)}{\partial\tau} = \mathbf{u}[\Phi(\mathbf{X}, \tau), \tau], \\ \Phi(\mathbf{X}, 0) = \mathbf{X}, \\ \Phi(\mathbf{X}, t) = \mathbf{x}, \end{cases} \tag{2.6}$$

This map, $\Phi(\mathbf{X}, t) = \mathbf{x}$, describes the trajectory of the material point, mapping its initial location $\mathbf{X}$ at time 0 to its current location $\mathbf{x}$ at time t. This is computed by integrating the velocity field $\mathbf{u}$ from 0 to t.

Similarly, we define the **backward flow map** $\Psi(\cdot, t) : \Omega_t \to \Omega_0$:

$$\begin{cases} \frac{\partial \Psi(\mathbf{x},\tau)}{\partial \tau} = \mathbf{u}(\Psi(\mathbf{x}, \tau), \tau), \\ \Psi(\mathbf{x}, t) = \mathbf{x}, \\ \Psi(\mathbf{x}, 0) = \mathbf{X}. \end{cases} \tag{2.7}$$

This map, $\Psi(\mathbf{x}, 0) = \mathbf{X}$, traces the particle's current position $\mathbf{x}$ at time t back to its origin $\mathbf{X}$ at time 0. This is achieved by integrating the velocity field $\mathbf{u}$ backward in time from t to 0.

This bidirectional relationship between the material particle's initial coordinate $\mathbf{X}$ and final coordinate $\mathbf{x}$ is illustrated in Figure 2.1.

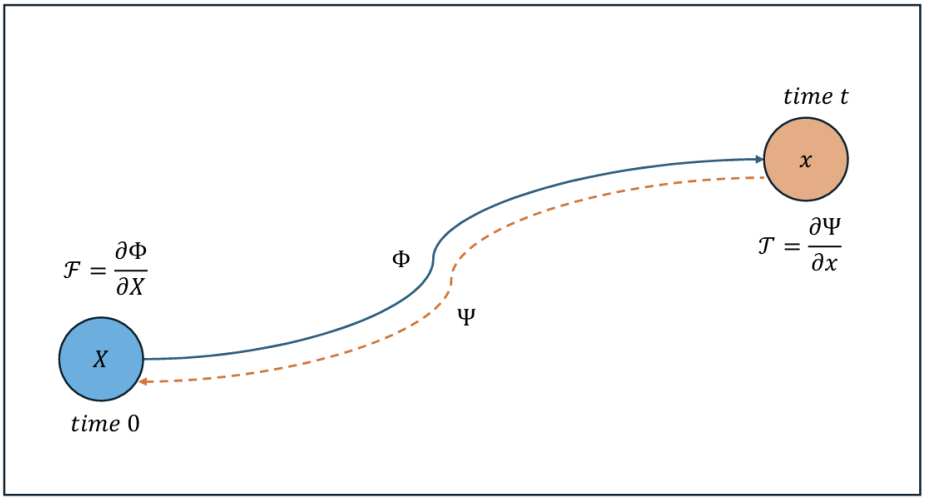

Figure 2.1: Illustration of the forward flow map (Φ) and backward flow map (Ψ). The forward map $\mathbf{\Phi}$ advects a material point from its initial position $\mathbf{X}$ at time 0 to its current position $\mathbf{x}$ at time t. The backward map Ψ traces the point back to its origin. The Jacobians $\mathcal{F}$ and $\mathcal{T}$ represent the deformation gradients of these respective maps.

The Jacobian matrices of Φ and Ψ are denoted as:

$$\begin{cases} \mathcal{F} = \frac{\partial \Phi}{\partial \mathbf{X}}, \\ \mathcal{T} = \frac{\partial \Psi}{\partial \mathbf{x}}, \end{cases} \tag{2.8}$$

The governing evolution equations for $\mathcal{F}$ and $\mathcal{T}$ are:

$$\begin{cases} \frac{D\mathcal{F}}{Dt} = \nabla \mathbf{u} \mathcal{F}, \\ \frac{D\mathcal{T}}{Dt} = -\mathcal{T} \nabla \mathbf{u}. \end{cases} \tag{2.9}$$

$\mathcal{F}$ and $\mathcal{T}$ represent the deformation between the initial and current reference frames symbolically, also known as deformation gradient [15]. For a detailed derivation of flow map evolution equations, please refer to the work by Junwei *et al.* [16].

2.3 Flow Maps for Fluid simulation

To solve the impulse form of the Euler equation, Nabizadeh *et al.* demonstrated that the impulse $\mathbf{m}$ can be reconstructed from a flow-map perspective [14]:

$$\mathbf{m}(\mathbf{x}, t) = \mathcal{T}_{t,0}^{T} \mathbf{m}(\Psi_{t,0}(\mathbf{x}), 0). \tag{2.10}$$

Intuitively, Equation 2.10 reconstructs the current impulse by first backtracing the current location $\mathbf{x}$ at time t to its initial location $\mathbf{X} = \Psi_{t,0}(\mathbf{x})$, then reading the initial impulse $\mathbf{m}(\mathbf{X}, 0)$, and finally transforming it by the transposed Jacobian of the backward flow map, $\mathcal{T}_{t,0}^{T}$.

At the initial time $t = 0$, the impulse field is identical to the initial velocity field:

$$\mathbf{m_0}(\mathbf{X}) = \mathbf{u}_0(\mathbf{X}) \tag{2.11}$$

Therefore, Equation 2.10 can be rewritten by substituting this initial condition:

$$\mathbf{m}(\mathbf{x}, t) = \mathcal{T}_{t,0}^{T}\mathbf{u_0}(\Psi_{t,0}(\mathbf{x}), 0). \tag{2.12}$$

The divergence-free velocity field $\mathbf{u}$ is then reconstructed from the impulse $\mathbf{m}$. As shown by [17], this is a multi-step process. First, the effect of external forces is accumulated via a path integral along the particle's trajectory. Second, a gauge term is subtracted to enforce incompressibility:

$$\mathbf{u}_t(\mathbf{x}) = \mathbf{m}_t(\mathbf{x}) - \frac{1}{\rho}\nabla\xi_{0,t}(\mathbf{x}) + \frac{1}{\rho}\mathcal{T}_{t,0}^{T}(\mathbf{x}) \int_0^t \mathcal{F}_{0,\tau}^{T}(\mathbf{X})\mathbf{f}_\tau(\Psi_{t,\tau}(\mathbf{x}))d\tau \tag{2.13}$$

In Equation 2.13, $\mathbf{X}$ represents the initial position of the material particle currently at $\mathbf{x}$ (i.e., $\mathbf{X} = \Psi_{t,0}(\mathbf{x})$). The term $\Psi_{t,\tau}(\mathbf{x})$ denotes that particle's intermediate position at time τ, and $\xi_{0,t}$ is the gauge term. This gauge term is ultimately found by solving a Poisson equation.

For completeness, we provide the formal definition of $\xi_{0,t}$ in terms of the backward flow map, although in practice this integral is not computed directly:

$$\xi_{0,t}(\mathbf{x}) = \int_0^t \left(p_\tau - \frac{1}{2}\rho|\mathbf{u}_\tau|^2\right)(\Psi_{t,\tau}(\mathbf{x}))d\tau \tag{2.14}$$

CHAPTER 3
ONE STEP FLOW MAP METHOD

3.1 Problem Statement

The primary advantage of flow map methods is their superior vorticity preservation over long-term simulations. By marching the flow map Ψ directly, rather than advecting fluid quantities, these methods avoid the repeated interpolations that cause significant numerical diffusion in traditional advection schemes.

Despite this advantage, existing flow map methods are computationally expensive. They require multiple marching sub-steps within each reinitialization cycle, each involving costly computations of flow map Jacobians and velocity gradients. Both NFM [15] and LFM [17] store flow maps on the faces of a staggered (MAC) grid [23]. For a reinitialization cycle with n sub-steps, NFM requires $(\frac{1}{2}n^2 + \frac{5}{2}n)$ marching steps per dimension and $2n$ projections. LFM optimizes this to $2n$ marching steps per dimension and $n + 1$ projections, achieving linear complexity with respect to n. Although LFM is a significant improvement, its cost remains prohibitive for real-time graphics applications.

To achieve real-time performance, we propose a modification based on a key insight: the most critical step for vorticity preservation is the reconstruction of the impulse, shown in Equation 2.12. This operation, which multiplies the initial velocity field by the transposed Jacobian of the backward map ($\mathcal{T}$), can be interpreted as a Lie advection [24] of the covector velocity field, adhering to the conservation of linear momentum [14].

Leveraging this insight, we trade the formal accuracy of a multi-step integration for a significant gain in efficiency. We set the number of sub-steps n to $\mathbf{1}$ for every reinitialization cycle. This simplification still produces visually appealing results with robust vorticity preservation. Our method requires only $\mathbf{2}$ marching steps per dimension and $\mathbf{2}$ projections

per timestep, a constant cost independent of n. We name this approach the **One-Step Flow Map Method (OFM)**.

3.2 Simulation Procedure

Our simulation scheme, inspired by LFM [17], is detailed in Algorithm 1. The procedure advances the simulation by one timestep, Δt, from state n to $n + 1$. Due to the numerical instability inherent in long-term flow map evolution, we reinitialize the flow maps at the beginning of each timestep. The key stages are as follows:

- **Initialization (Line 1):** All flow maps (Φ, Ψ) and their Jacobians $(\mathcal{F}, \mathcal{T})$ are initialized to identity.
- **Mid-Point Velocity (Lines 2-4):** A second-order accurate mid-point velocity, $\mathbf{u}_{n+1/2}$, is computed. This involves applying external forces (Line 2), advecting the velocity to the mid-point using 2th Order Runge-Kutta (RK2) (Line 3), and projecting the result to be divergence-free (Line 4).
- **Flow Map Marching (Lines 5 & 7):** The computed mid-point velocity, $\mathbf{u}_{n+1/2}$, is used to drive the forward (Line 5) and backward (Line 7) evolution of the flow maps (Φ, Ψ) and their Jacobians ($\mathcal{F}$, $\mathcal{T}$). This is performed using a Total Variation Diminishing 3rd Order Runge-Kutta (TVD-RK3) integrator, which solves their governing evolution equations (Equation 2.6, Equation 2.7, Equation 2.9). To evaluate these equations at arbitrary points, the velocity field and its gradient are sampled from the grid using quadratic B-spline interpolation [25].
- **Path Integral (Line 6):** The path integral for external forces is accumulated efficiently by reusing intermediate values from the forward TVD-RK3 march.
- **Impulse Reconstruction (Line 8):** The impulse $\mathbf{m}_{n+1}$ is reconstructed using the initial velocity $\mathbf{u}_n$ and the computed backward map, as defined in Equation 2.12.

- **Error Compensation (Lines 9-12):** A BFECC error compensation is applied. A round-trip error e is calculated (Lines 9-10), subtracted from the impulse (Line 11), and then clamped (Line 12).

- **Final Projection (Line 13):** The final, compensated impulse $\mathbf{m}_{n+1}$ is projected to produce the divergence-free velocity $\mathbf{u}_{n+1}$ for the new timestep.

Algorithm 1 Simulation Of One Time Step

Input: $\mathbf{u}_n$
Output: $\mathbf{u}_{n+1}$
1: $\Psi_{n+1,n+1} = \text{id}, \Phi_{n,n} = \text{id}, \mathcal{T}_{n+1,n+1} = I, \mathcal{F}_{n,n} = I$
2: $\mathbf{u}_n^\dagger = \mathbf{u}_n + \frac{\Delta t}{2\rho}\mathbf{f}_n$
3: $\tilde{\mathbf{u}}_{n+1/2} = \text{RK2-Advect}(\mathbf{u}_n^\dagger, \mathbf{u}_n, \Delta t/2)$
4: $\mathbf{u}_{n+1/2} = \text{Project}(\tilde{\mathbf{u}}_{n+1/2})$
5: $\Phi_{n,n+1}, \mathcal{F}_{n,n+1} = \text{TVD-RK3-March}(\Phi_{n,n}, \mathcal{F}_{n,n}, \mathbf{u}_{n+1/2}, \Delta t)$
6: $\mathbf{u}_n = \mathbf{u}_n + \frac{\Delta t}{\rho}\mathcal{F}^T_{n,n+1/2}\mathbf{f}_{n+1/2}(\Phi_{n,n+1/2})$ ▷ $\mathcal{F}_{n,n+1/2}$ and $\Phi_{n,n+1/2}$ are computed via previous step as intermediate variables
7: $\Psi_{n+1,n}, \mathcal{T}_{n+1,n} = \text{TVD-RK3-March}(\Psi_{n+1,n+1}, \mathcal{T}_{n+1,n+1}, \mathbf{u}_{n+1/2}, -\Delta t)$;
8: $\mathbf{m}_{n+1} = \mathcal{T}^T_{n+1,n}\mathbf{u}_n(\Psi_{n+1,n})$;
9: $\hat{\mathbf{u}}_n = \mathcal{F}^T_{n,n+1}\mathbf{m}_{n+1}(\Phi_{n,n+1})$;
10: $e = (\hat{\mathbf{u}}_n - \mathbf{u}_n)/2$;
11: $\mathbf{m}_{n+1} = \mathbf{m}_{n+1} - \mathcal{T}^T_{n+1,n}e(\Psi_{n+1,n})$;
12: $\mathbf{m}_{n+1} = \text{BFECCClamp}(\mathbf{m}_{n+1})$;
13: $\mathbf{u}_{n+1} = \text{Project}(\mathbf{m}_{n+1})$;

It should be noted that because we use only a single sub-step, the path integral from Equation 2.13 simplifies to a single multiplication at the mid-point:

$$\frac{1}{\rho}\int_{t_n}^{t_{n+1}} \mathcal{F}^T_{n,\tau}(\mathbf{X})\mathbf{f}_\tau(\Psi_{n+1,\tau}(\mathbf{x}))d\tau \approx \frac{\Delta t}{\rho}\mathcal{F}^T_{n,n+1/2}(\mathbf{X})\mathbf{f}_{n+1/2}(\Psi_{n+1,n+1/2}(\mathbf{x})) \tag{3.1}$$

Since $\Phi_{n,n+1/2}$ and $\Psi_{n+1,n+1/2}$ both represent the material point's position at the same intermediate time $t_{n+1/2}$, they are equivalent. Thus, Equation 3.1 can be expressed using the forward map:

$$\frac{\Delta t}{\rho}\mathcal{F}^{T}_{n,n+1/2}(\mathbf{X})\mathbf{f}_{n+1/2}(\Phi_{n,n+1/2}(\mathbf{x})) \tag{3.2}$$

This is exactly the operation performed in Line 6 of the algorithm.

Unlike the original LFM, which uses an 4th Order Runge-Kutta (RK4) integrator, we employ a TVD-RK3 integrator [26] combined with the mid-point velocity $\mathbf{u}_{n+1/2}$ to march the flow maps. This choice is motivated by the fact that TVD-RK3 is better at handling discontinuities—a situation we expect from dynamic boundaries, where there might be a sudden change in velocity at the solid interface. The details of our TVD-RK3 implementation are provided by Algorithm 4 in Appendix A.

In some literature, this leapfrog integration approach (using a mid-point velocity) is also referred to as semi-implicit [27]. We employ BFECC to further increase accuracy. For efficiency, intermediate variables from the forward TVD-RK3 march (Line 5) are reused for the path integral (Line 6), avoiding redundant computation.

CHAPTER 4
DYNAMIC BOUNDARY CONDITION

4.1 Problem Statement

Our goal is to support dynamic boundaries, where the fluid can be influenced by arbitrary, moving solid objects in the scene—for example, a moving wall compressing gas or fan blades dispersing smoke. We require a system capable of adapting to complex scene geometry, regardless of the shape, motion, or number of solid objects.

A natural approach to achieving this is to use a voxelization method. This process converts the continuous mesh representations of solid objects—standard in video games—into a discrete voxel representation. In this representation, each voxel stores data indicating whether it is occupied by a solid. This discrete occupancy information can then be efficiently used by the fluid simulation to enforce boundary conditions.

For our implementation, we adopt a GPU-based mesh voxelization method introduced by Crane *et al.* [20], which is well-suited for real-time applications.

4.2 Real-time Mesh Voxelization

4.2.1 Inside-Outside Voxelization

We represent the solid geometry within the fluid domain using a 3D occupancy texture. Each texel (or voxel) in this grid corresponds to a cell in the simulation, storing a binary value to indicate whether that cell is inside or outside a solid. The result of this process is a 3D grid of stencil slices, as illustrated in Figure 4.1.

To generate this 3D texture, we render the triangle meshes using a slice-based approach. As shown in Figure 4.2, each layer of the 3D texture is generated by setting an orthographic projection matrix where the near plane corresponds to the current slice's depth and the far

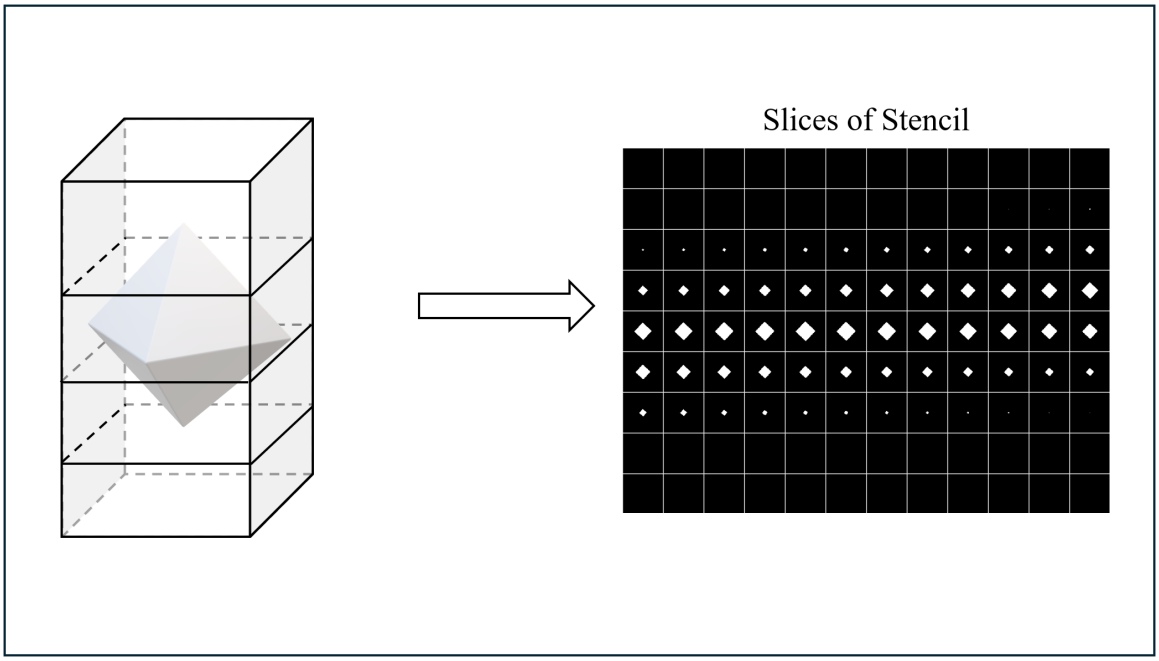

Figure 4.1: A 3D mesh (left) is voxelized into a series of 2D stencil slices (right), which form the 3D occupancy texture.

plane is set to infinity. This process is highly suitable for GPU instancing. We can draw the entire mesh geometry in a single call, instanced for each depth slice. Each instance is then configured to render into its corresponding layer of the 3D texture.

During this render pass, back-face culling is disabled. We configure the stencil buffer to increment for back-facing triangles and decrement for front-facing triangles. This 3D scan-conversion technique ensures that any pixel corresponding to a voxel inside the mesh will have a non-zero value in the stencil buffer. After rendering all geometry into a 2D slice, the resulting stencil data is copied to the corresponding layer of the 3D occupancy texture.

It is critical that the input meshes are watertight (i.e., closed). This algorithm relies on the assumption that every front-facing triangle has a corresponding back-facing triangle, ensuring that the stencil increments and decrements correctly balance to zero for all exterior voxels.

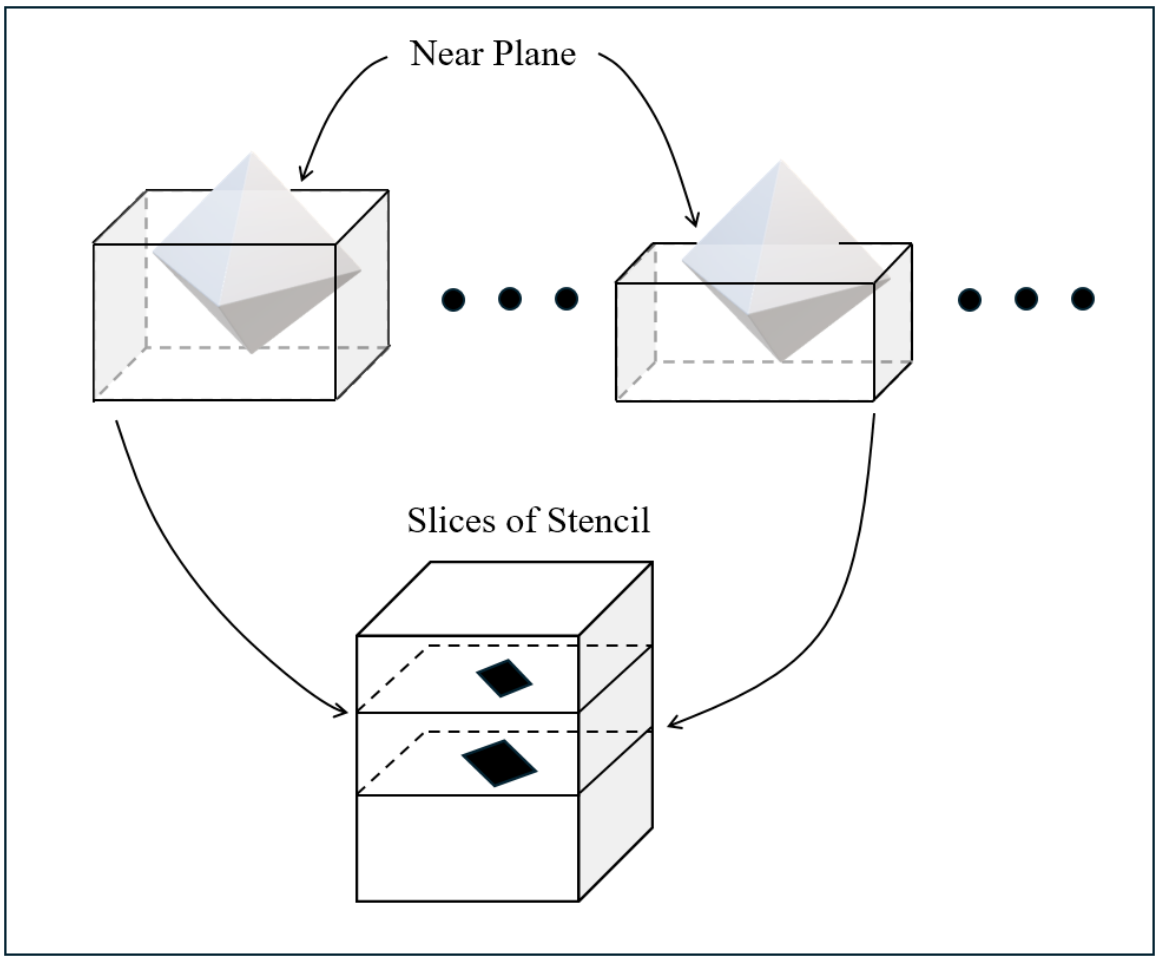

Figure 4.2: Illustration of the voxelization process. The geometry is rendered multiple times, each with a different near plane, to generate one 2D stencil slice per layer.

4.2.2 Velocity Voxelization

In a dynamic scene, the boundary condition requires not only the solid's position but also its velocity at every boundary voxel. This velocity data is stored in a separate 3D texture, as shown in Figure 4.3. In this texture, the RGB channels are remapped to represent the x, y, and z components of the velocity. This allows the fluid simulator to enforce the Neumann boundary condition[9]:

$$\mathbf{u}_{\text{fluid}} \cdot \mathbf{n} = \mathbf{u}_{\text{solid}} \cdot \mathbf{n} \tag{4.1}$$

This equation defines a "free-slip" boundary. It ensures that the fluid's velocity normal to the solid's surface matches the solid's own velocity, preventing the fluid from passing through the boundary while allowing it to slide along the surface without friction.

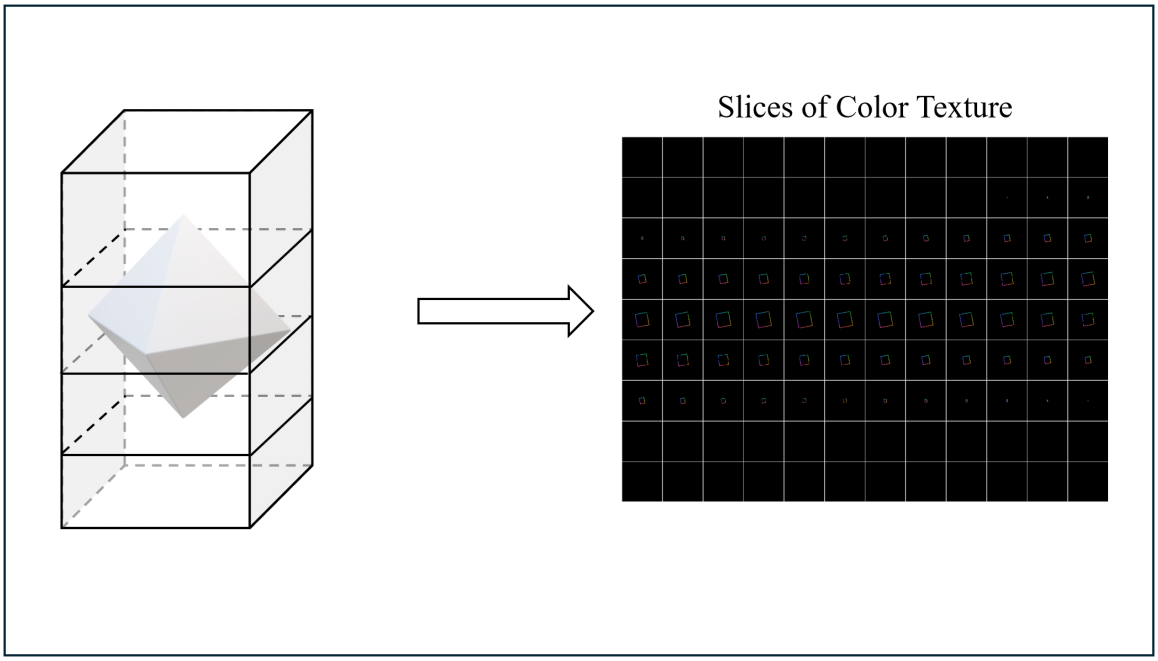

Figure 4.3: Boundary velocity is written into a 3D texture. The RGB channels, representing the velocity vector, are remapped to [0, 1] here for illustration.

Generation Method

The velocity voxelization is a multi-step process. First, the velocity of each mesh vertex is computed in the vertex shader using a simple forward difference and stored in a buffer:

$$\mathbf{u}_i = \frac{\mathbf{p}_i - \mathbf{p}_{i-1}}{\Delta t} \tag{4.2}$$

Next, the mesh is processed to generate the 3D velocity texture, slice by slice. This is achieved using a **geometry shader**. As illustrated in Figure 4.4, the geometry shader takes a triangle as input and tests for intersection against the current depth slice. If the triangle intersects the plane, the intersection forms a 1D line segment (e.g., from v_1 to v_2).

To ensure this 1D segment is captured by the 2D pixel grid of the slice, we "thicken" it by extruding it into a 2D quad that lies within the slice plane. The velocities at the vertices of this new quad are computed by linearly interpolating the velocities from the original triangle's vertices. This quad is then rasterized into the current slice of the velocity texture, with the fragment shader writing the interpolated solid velocities.

Handling Gaps

A critical problem with rasterizing thin line segments is that gaps can appear in the resulting voxels when the geometry is tilted, as shown in Figure 4.5 (left). This is a form of aliasing that would create "leaks" in the fluid boundary.

To solve this, the width w of the extruded quad is set to be **twice the diagonal length of a texel**. This conservative width ensures full coverage, filling any potential gaps between adjacent texels. This guarantees a watertight, gap-free velocity boundary, as seen in Figure 4.5 (right).

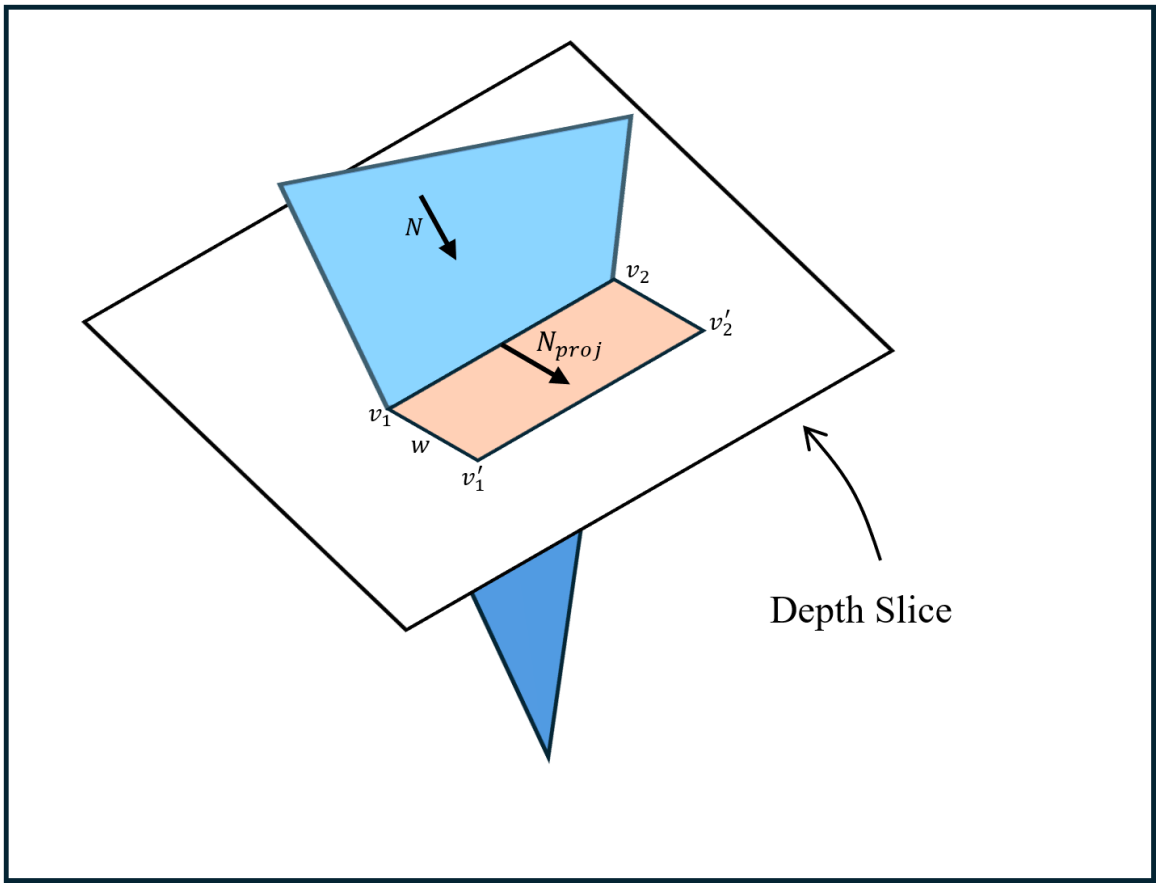

Figure 4.4: A triangle (blue) intersects a depth slice. The 1D intersection segment is extruded into a 2D quad (orange) which is then rasterized.

GPU Instancing

Just like the inside-outside voxelization, this entire procedure is heavily optimized using GPU instancing. The entire mesh geometry is drawn in a single call, instanced for each slice of the 3D velocity texture.

4.2.3 Integrating Boundaries with the Flow Map Solver

At the beginning of each frame, the voxelization pipeline executes, performing both the inside-outside and velocity voxelization steps. This process updates the 3D textures that encode the solid boundary's position and velocity. The fluid simulator then uses this boundary information to compute the divergence for each grid cell and construct the pressure

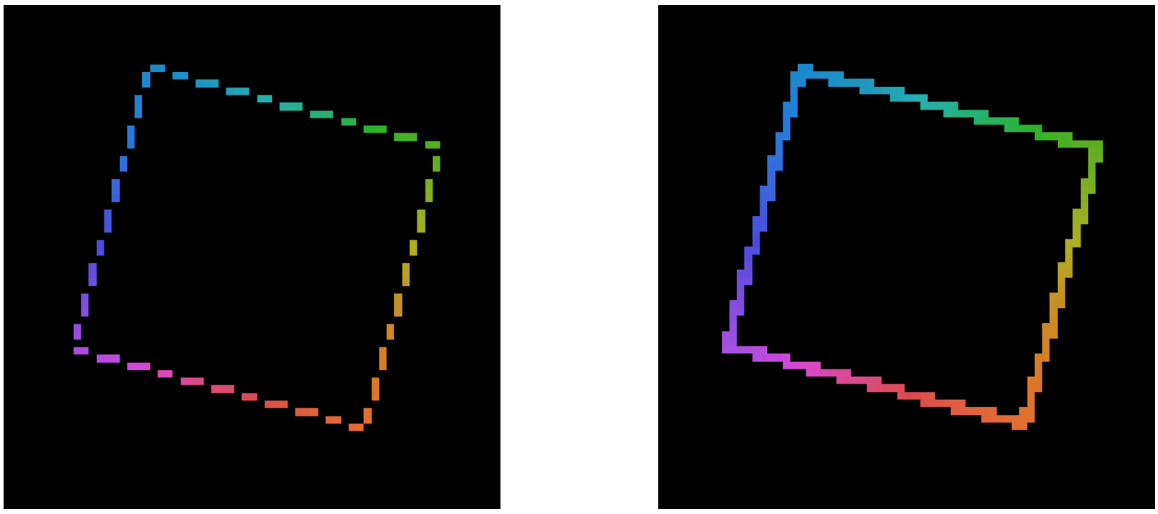

Figure 4.5: Gaps exist when the quad width is only one texel (left). These gaps are filled by using a width of two texels (right), ensuring a watertight boundary.

projection matrix. Crucially, this approach simplifies the boundary enforcement. Unlike older methods, such as the one in [20], we do not need to perform an extra, explicit step to enforce the ”free-slip” condition after the pressure solve. This is because the modern Poisson solver we use, which will be elaborated in the next chapter, has an exceptionally high convergence rate. It is designed to solve the pressure system with the Neumann boundary condition ($\mathbf{u}_{\text{fluid}} \cdot \mathbf{n} = \mathbf{u}_{\text{solid}} \cdot \mathbf{n}$) implicitly. Because the solver converges robustly, it finds the correct pressure solution that inherently produces a divergence-free field that already respects the solid boundaries, eliminating the need for a separate velocity correction step.

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 Pressure Projection and Poisson Solver

The pressure projection step is consistently the most computationally expensive part of an incompressible fluid simulation, as it requires solving a large, sparse Poisson equation. This equation forms a linear system of the form $A\vec{p} = \vec{b}$ where the matrix A is Symmetric Positive Definite (SPD) [28]. Therefore, the choice of an efficient solver is critical for achieving real-time performance.

The field of computer graphics has seen a clear evolution in solver technology. Early methods, such as Successive Over-Relaxation (SOR) [29], were common but suffered from slow convergence on large grids. Subsequently, Krylov subspace methods [30], particularly the Conjugate Gradient (CG) method, proved to be a more powerful alternative. The efficacy of CG is significantly enhanced when accelerated with a preconditioner, leading to the Preconditioned Conjugate Gradien (PCG) method [31]. A seminal contribution by Foster and Fedkiw [32] utilized an Incomplete Cholesky Factorization (ICF) preconditioner, which became a standard for some time.

However, methods like ICF are inherently serial and not well-suited for parallel architectures of multi-core Central Processing Unit (CPU) or GPU. This led to the adoption of Multigrid methods, which are highly parallelizable. Multigrid can be used either as a standalone solver or as a powerful preconditioner for a PCG algorithm [33, 34, 35].

For our framework, we leverage the open-source, matrix-free **Algebraic Multigrid Preconditioned Conjugate Gradient (AMGPCG)** solver developed by Sun *et al.* [17]. This state-of-the-art solver is implemented in CUDA [36] and specifically designed for large-scale, real-time fluid simulation on the GPU, making it the ideal choice for our

performance-critical projection step. Its efficacy was demonstrated in that work, where it was shown to converge to a relative error of 10^{-6} in only $\mathbf{3.45}$ **ms (14 iterations)**, confirming its suitability for real-time applications.

In the following sections, we will briefly review the foundations of the AMGPCG method. For a comprehensive discussion of its technical details and GPU-specific kernel optimizations, readers are referred to the original paper [17]. For a foundational treatment of multigrid methods, we refer readers to this book [28].

5.1.1 Preconditioned Conjugate Gradient

To solve a linear system $A\vec{x} = \vec{b}$ where A is SPD, the PCG method is often employed, as illustrated in Algorithm 2.

PCG accelerates the standard CG method by applying a preconditioner M, which is chosen to be a good approximation of A. The goal is to find an M that satisfies two conditions: It significantly reduces the condition number of the system (i.e., $\kappa(M^{-1}A) \ll \kappa(A)$), guaranteeing rapid convergence. The system $M\vec{z} = \vec{r}$ (solved in Lines 2 and 9) is computationally inexpensive to solve.

Algorithm 2 Preconditioned Conjugate Gradient (PCG)

Input: $A, \vec{b}, \vec{x}^{(0)}$
Output: $\vec{x}^{(n)}$
1: Compute $\vec{r}^{(0)} = \vec{b} - A\vec{x}^{(0)}$
2: Solve $M\vec{z}^{(0)} = \vec{r}^{(0)}$
3: Set $\vec{p}^{(0)} = \vec{z}^{(0)}$
4: **for** $k = 0, 1, \ldots, n-1$ **do**
5: $\quad \vec{q}^{(k)} = A\vec{p}^{(k)}$
6: $\quad \alpha_k = \frac{\langle \vec{r}^{(k)}, \vec{z}^{(k)} \rangle}{\langle \vec{p}^{(k)}, \vec{q}^{(k)} \rangle}$
7: $\quad \vec{x}^{(k+1)} = \vec{x}^{(k)} + \alpha_k \vec{p}^{(k)}$
8: $\quad \vec{r}^{(k+1)} = \vec{r}^{(k)} - \alpha_k \vec{q}^{(k)}$
9: $\quad$ Solve $M\vec{z}^{(k+1)} = \vec{r}^{(k+1)}$
10: $\quad \beta_k = \frac{\langle \vec{r}^{(k+1)}, \vec{z}^{(k+1)} \rangle}{\langle \vec{r}^{(k)}, \vec{z}^{(k)} \rangle}$
11: $\quad \vec{p}^{(k+1)} = \vec{z}^{(k+1)} + \beta_k \vec{p}^{(k)}$
12: **end for**

5.1.2 Unsmoothed Aggregation Algebraic Multigrid

The preconditioner M used in Algorithm 2 is an Unsmoothed Aggregation Algebraic Multigrid (UAAMG), introduced in the work of Sun *et al.* [17]. The operation Solve $M\vec{z} = \vec{r}$ is performed by executing a few steps of V-Cycle, as detailed in Algorithm 3.

The fundamental principle of multigrid is to efficiently reduce error components at different frequencies. High-frequency (rough) errors are dampened by an inexpensive iterative smoother, while low-frequency (smooth) errors are handled by doing error correction on coarser grid. In this method, the Red-Black Gauss-Seidel (RBGS) method [37] is chosen as the smoothing operator, as it is highly parallelizable on the GPU.

Algorithm 3 illustrates this recursive process. The V-Cycle begins at the finest level l with a pre-smoothing step (Line 6) using RBGS. A residual ($\vec{b_l} - A_l\vec{x}_l$) is then computed and restricted (down-sampled) to the next coarser level $l+1$ (Line 7). The V-Cycle is called recursively to solve this coarse-grid problem (Line 8). The resulting correction $\vec{x}_{l+1}$ is then prolongated (interpolated) back to the fine grid and added to the solution (Line 9). Finally, a post-smoothing step (Line 10) is applied to eliminate any high-frequency errors introduced by the prolongation. At the coarsest level (Lines 2-4), the system is small enough to be solved directly with multiple RBGS iterations.

Algorithm 3 V-Cycle

Input: $l, \vec{b_l}$
Output: $\vec{x}_l$
1: $\vec{x}_l = \vec{0}$
2: **if** $l ==$ nLevel $- 1$ **then** ▷ coarsest level
3: multiple RBGS to $A_l\vec{x}_l = \vec{b}_l$
4: **return** $\vec{x}_l$
5: **end if**
6: one RBGS to $A_l\vec{x}_l = \vec{b}_l$ ▷ smoothing
7: $\vec{b}_{l+1} = R_l(\vec{b}_l - A_l\vec{x}_l)$ ▷ restriction
8: $\vec{x}_{l+1} =$ V-Cycle$(l+1, \vec{b}_{l+1})$
9: $\vec{x}_l = \vec{x}_l + 2P_l\vec{x}_{l+1}$; ▷ prolongation with scaling
10: one RBGS to $A_l\vec{x}_l = \vec{b}_l$ ▷ smoothing
11: **return** $\vec{x}_l$

As noted by Stüben [38], when UAAMG is used as a preconditioner for PCG, convergence is improved by scaling the prolongated correction. This is why a scaling coefficient of 2 is applied in the prolongation step (Line 9). Furthermore, the restriction R and prolongation P operators are constructed such that $PR = I$.

5.2 Project Framework

Our framework is built on a hybrid graphics and compute pipeline, integrating two key APIs: Vulkan [39] and CUDA [36].

We use Vulkan to leverage GPU rasterization hardware for the dynamic mesh voxelization. The core fluid simulation is implemented in CUDA to take full advantage of its general-purpose compute capabilities.

These two APIs are bridged via CUDA-Vulkan interoperation. The 3D textures generated by the Vulkan voxelizer (encoding boundary position and velocity) are mapped into the CUDA address space and used directly by the simulator. Conversely, the final simulation data, a GPU buffer, is passed from CUDA back to a Vulkan-based raymarching renderer to generate the final image in real-time.

This interoperation requires careful synchronization to ensure resources are not read and written simultaneously (race conditions) and that the graphics and compute tasks execute in the correct order, as illustrated in Figure 5.1. We use a pair of binary semaphores [39] to manage this dependency.

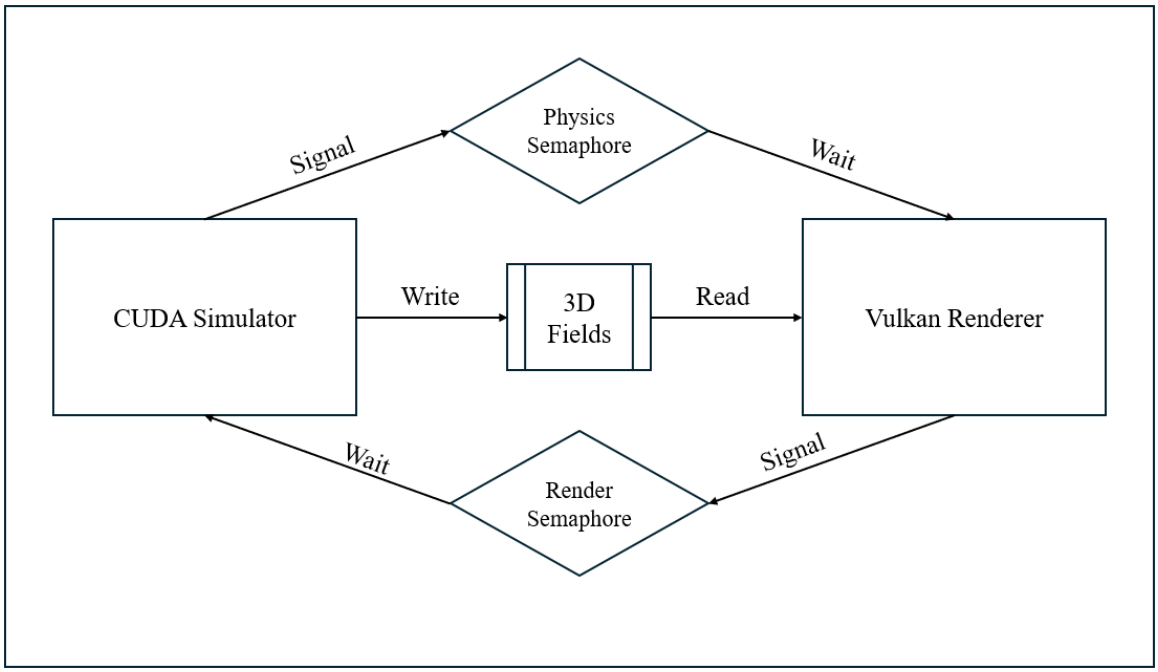

Figure 5.1: The synchronization data flow between the CUDA simulator and the Vulkan renderer. The Physics Semaphore is signaled by CUDA when 3D fields are ready to be read by the renderer, which waits on it. The Render Semaphore is signaled by Vulkan when rendering is complete, allowing CUDA (which waits on it) to safely write new simulation data for the next frame.

CHAPTER 6

RESULTS

In this chapter, we present the evaluation of our OFM framework. All experiments were conducted on a simulation grid resolution of $256 \times 128 \times 128$, running on a laptop equipped with an NVIDIA RTX 4080 GPU.

To visualize the vorticity preservation capabilities of our method, we compute the vorticity magnitude for each grid cell. This data is passed to a raymarching renderer, where it is mapped to a color gradient: low vorticity is rendered as periwinkle blue, medium vorticity as neutral grey, and high vorticity as reddish-orange.

6.1 Comparison with LFM

We benchmark our OFM method against the original LFM approach. As illustrated in Figure 6.1 and Figure 6.2, while our single-step method is theoretically less accurate than the multi-step LFM (which uses 5 substeps in these examples), the visual difference is minimal. The slight increase in numerical diffusion is visually negligible, particularly when weighed against the significant performance gains.

Table 6.1 details the runtime performance. Our OFM method achieves a speedup of approximately $3\times$ to $4\times$ compared to LFM, demonstrating that the trade-off between formal accuracy and computational efficiency is highly favorable for real-time applications.

Table 6.1: Runtime Performance Comparison between LFM and OFM

Scene	LFM (substeps = 5)	OFM	Speedup
Comet	128 ms	31 ms	**4.13**$\times$
Fireball	100 ms	30 ms	**3.33**$\times$

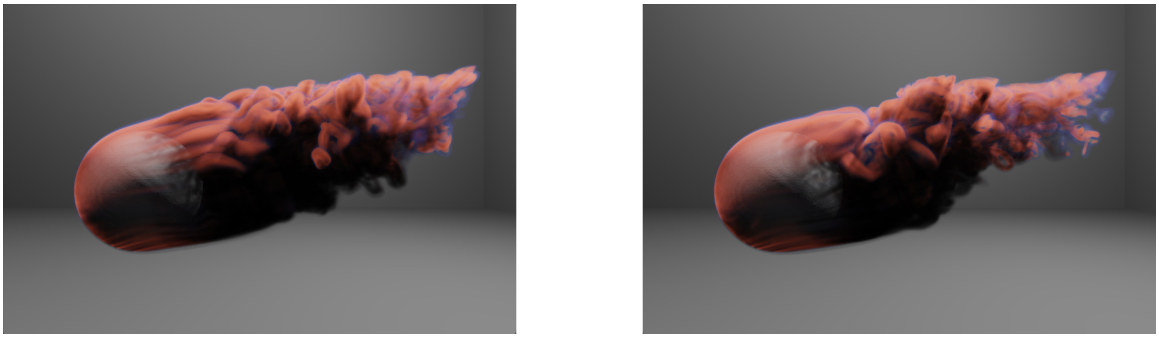

Figure 6.1: Comet Simulation. Airflow moves from left to right, generating vortices along the sphere's surface. A comparison between LFM (Left) and our OFM (Right) shows that while the OFM result is slightly more diffusive (blurrier), the visual fidelity of the vortices remains high.

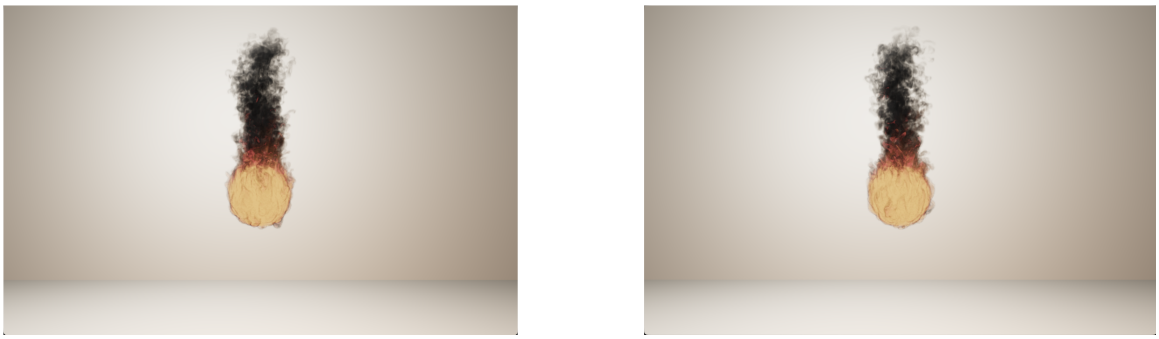

Figure 6.2: Fireball Simulation. Combustion generates turbulent vortices surrounding the sphere. Our real-time OFM (Right) exhibits slightly higher numerical dissipation compared to the LFM simulation (Left).

6.2 Runtime Analysis

To analyze the computational cost of individual components within our framework, we profile a scene featuring a rotating octahedron, as shown in Figure 6.3. This scene demonstrates the interaction between the fluid and a dynamic, rotating solid boundary.

The breakdown of the computation time for a single timestep is visualized in Figure 6.4. The analysis reveals that the voxelization pipeline (both inside-outside and velocity voxelization) is efficient, consuming only $\sim 8\%$ of the total frame time. The majority of the computational budget is dedicated to the fluid simulation itself. Specifically, the two pressure projection steps required by the OFM algorithm constitute the primary bottleneck, accounting for approximately 44% of the total execution time.

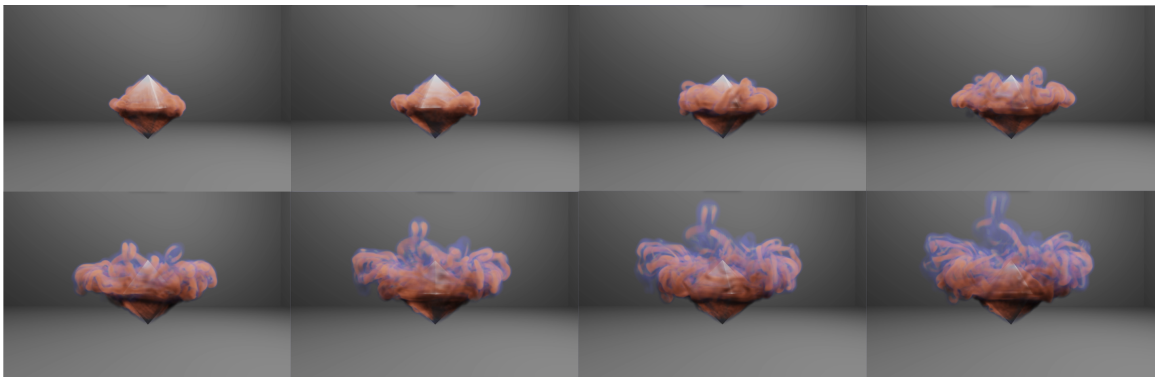

Figure 6.3: Time-lapse sequence of a rotating octahedron interaction. The solid object transfers angular momentum to the fluid via our dynamic velocity boundary condition, generating turbulent vortices that propagate outward into the fluid domain.

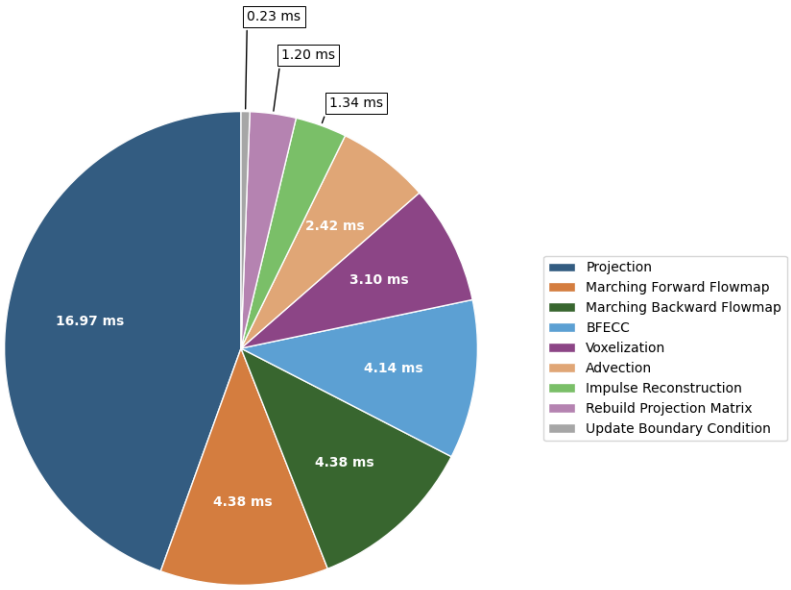

Figure 6.4: Runtime breakdown of a single simulation timestep. The pressure projection (blue) is the dominant cost.

CHAPTER 7
CONCLUSION

This thesis presented a real-time fluid simulation framework. We began in Chapter 2 by establishing the mathematical foundations of fluid dynamics and the flow map method. In Chapter 3, we detailed our primary contribution, the One-Step Flow Map Method (OFM). This method represents a deliberate trade-off, sacrificing formal accuracy for significant gains in computational efficiency and ease of implementation compared to previous flow map techniques.

Following this, in Chapter 4, we introduced a pipeline for dynamic boundary conditions. This system integrates the fluid solver with a real-time mesh voxelization method, enabling the simulation to react to arbitrary, moving geometry. Finally, in Chapter 5, we demonstrated that our method successfully produces fluid flow that is visibly and realistically affected by a dynamic environment, all while maintaining interactive frame rates.

While the computational cost of this framework may currently be high for immediate adoption in a commercial game production, we believe this work represents a step toward that goal. As the power of GPU continues to grow, the adoption of more advanced, physically-based simulation methods is inevitable. We are confident that the techniques presented in this thesis, particularly the integration of flow maps with dynamic voxelization, are a valuable contribution to that future.

CHAPTER 8
FUTURE WORK

This thesis has presented a high-performance framework for fluid simulation, but there are two primary avenues for future improvement: algorithmic optimization and engineering portability.

8.1 Algorithmic Optimizations

Our OFM method was a deliberate trade-off, prioritizing performance by simplifying the integration. This trade-off could be pushed even further in future iterations.

The BFECC error compensation (Algorithm 1, Lines 9-12) requires marching both a forward and backward flow map, which is still computationally significant. A more aggressive optimization would be to eliminate the forward flow map (Φ) and its Jacobian ($\mathcal{F}$) entirely. The round-trip error could instead be approximated using a standard advection result. While this would further sacrifice formal accuracy, it may be a worthwhile trade-off, as the primary goal of vorticity preservation is handled by the backward map's Jacobian ($\mathcal{T}$) during the impulse reconstruction (Line 8).

8.2 Engineering and Portability

From an engineering perspective, the current framework relies on a hybrid pipeline of Vulkan and CUDA. This design requires CUDA-Vulkan interoperation and restricts the hardware to NVIDIA GPUs.

A significant future goal would be to port the entire CUDA-based simulation stack—including the fluid solver and the AMGPCG Poisson solver—to a platform-agnostic graphics API. This would involve rewriting the compute kernels in a shader language such as HLSL [40]

or GLSL [41]. Such an effort would create a single, unified framework, eliminating the overhead and complexity of API interoperation. Most importantly, it would ensure vendor independence, allowing the simulation to run on any modern, standards-compliant GPU.

Additionally, the voxelization pipeline could be optimized by adopting modern geometry processing standards. Our current implementation relies on geometry shaders to perform the triangle-to-quad extrusion required for velocity voxelization. However, geometry shaders can become a performance bottleneck when processing input meshes with high geometric complexity, primarily due to their sequential execution model and variable output size. Future work could reimplement this stage using Mesh Shader [42]. This modern pipeline replaces the traditional vertex and geometry stages with a compute-centric approach, allowing for cooperative thread processing of meshlets. Adopting this architecture would provide higher geometric throughput and greater flexibility for voxelization technique.

Appendices

APPENDIX A

ADDITIONAL PSEUDOCODE

Algorithm 4 TVD RK3 Marching

Input: $\mathbf{u},\ \Phi,\ \mathcal{F},\ \Delta t$

Output: $\Phi_{\text{next}},\ \mathcal{F}_{\text{next}}$

$(\mathbf{u}_1, \nabla\mathbf{u}_1) = \text{Interpolate}(\mathbf{u}, \Phi)$

$\left.\frac{\partial \mathcal{F}}{\partial t}\right|_1 = \nabla\mathbf{u}_1\mathcal{F}$

$\Phi^{(1)} = \Phi + \Delta t \cdot \mathbf{u}_1$

$\mathcal{F}^{(1)} = \mathcal{F} + \Delta t \cdot \left.\frac{\partial \mathcal{F}}{\partial t}\right|_1$

$(\mathbf{u}_2, \nabla\mathbf{u}_2) = \text{Interpolate}(\mathbf{u}, \Phi^{(1)})$

$\left.\frac{\partial \mathcal{F}}{\partial t}\right|_2 = \nabla\mathbf{u}_2\mathcal{F}^{(1)}$

$\Phi^{(2)} = \frac{3}{4}\Phi + \frac{1}{4}\left(\Phi^{(1)} + \Delta t \cdot \mathbf{u}_2\right)$

$\mathcal{F}^{(2)} = \frac{3}{4}\mathcal{F} + \frac{1}{4}\left(\mathcal{F}^{(1)} + \Delta t \cdot \left.\frac{\partial \mathcal{F}}{\partial t}\right|_2\right)$

$(\mathbf{u}_3, \nabla\mathbf{u}_3) = \text{Interpolate}(\mathbf{u}, \Phi^{(2)})$

$\left.\frac{\partial \mathcal{F}}{\partial t}\right|_3 = \nabla\mathbf{u}_3\mathcal{F}^{(2)};$

$\Phi_{\text{next}} = \frac{1}{3}\Phi + \frac{2}{3}\left(\Phi^{(2)} + \Delta t \cdot \mathbf{u}_3\right);$

$\mathcal{F}_{\text{next}} = \frac{1}{3}\mathcal{F} + \frac{2}{3}\left(\mathcal{F}^{(2)} + \Delta t \cdot \left.\frac{\partial \mathcal{F}}{\partial t}\right|_3\right);$

REFERENCES

[1] M. Macklin and M. Müller, "Position based fluids," *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013.

[2] C. Horvath and W. Geiger, "Directable, high-resolution simulation of fire on the gpu," *ACM Trans. Graph.*, vol. 28, no. 3, Jul. 2009.

[3] R. Bridson, J. Houriham, and M. Nordenstam, "Curl-noise for procedural fluid flow," *ACM Transactions on Graphics (ToG)*, vol. 26, no. 3, 46–es, 2007.

[4] S. Gustavson, "Simplex noise demystified," *Linköping University, Linköping, Sweden, Research Report*, vol. 1, no. 2, p. 6, 2005.

[5] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01, New York, NY, USA: Association for Computing Machinery, 2001, pp. 15–22, ISBN: 158113374X.

[6] B. Kim, Y. Liu, I. Llamas, and J. Rossignac, "Flowfixer: Using bfecc for fluid simulation," in *Proceedings of the First Eurographics Conference on Natural Phenomena*, ser. NPH'05, Dublin, Ireland: Eurographics Association, 2005, pp. 51–56, ISBN: 3905673290.

[7] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, "The affine particle-in-cell method," *ACM Trans. Graph.*, vol. 34, no. 4, Jul. 2015.

[8] M. S. Nabizadeh, R. Roy-Chowdhury, H. Yin, R. Ramamoorthi, and A. Chern, "Fluid implicit particles on coadjoint orbits," *ACM Trans. Graph.*, vol. 43, no. 6, Nov. 2024.

[9] L. LANDAU and E. LIFSHITZ, "Chapter ii - viscous fluids," in *Fluid Mechanics (Second Edition)*, L. LANDAU and E. LIFSHITZ, Eds., Second Edition, Pergamon, 1987, pp. 44–94, ISBN: 978-0-08-033933-7.

[10] J. Stam, "Stable fluids," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '99, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128, ISBN: 0201485605.

[11] T. F. Dupont and Y. Liu, "Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function," *Journal of Computational Physics*, vol. 190, no. 1, pp. 311–324, 2003.

[12] D. C. Wiggert and E. B. Wylie, "Numerical predictions of two-dimensional transient groundwater flow by the method of characteristics," *Water Resources Research*, vol. 12, no. 5, pp. 971–977, 1976.

[13] J. Tessendorf, *Advection solver performance with long time steps, and strategies for fast and accurate numerical implementation*, 2015.

[14] M. S. Nabizadeh, S. Wang, R. Ramamoorthi, and A. Chern, "Covector fluids," *ACM Trans. Graph.*, vol. 41, no. 4, Jul. 2022.

[15] Y. Deng, H.-X. Yu, D. Zhang, J. Wu, and B. Zhu, "Fluid simulation on neural flow maps," *ACM Trans. Graph.*, vol. 42, no. 6, Dec. 2023.

[16] J. Zhou *et al.*, "Eulerian-lagrangian fluid simulation on particle flow maps," *ACM Transactions on Graphics (TOG)*, vol. 43, no. 4, pp. 1–20, 2024.

[17] Y. Sun *et al.*, "Leapfrog flow maps for real-time fluid simulation," *ACM Transactions on Graphics (TOG)*, vol. 44, no. 4, pp. 1–12, 2025.

[18] A. Kaufman and E. Shimony, "3d scan-conversion algorithms for voxel-based graphics," in *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, ser. I3D '86, Chapel Hill, North Carolina, USA: Association for Computing Machinery, 1987, pp. 45–75, ISBN: 0897912284.

[19] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion, "An accurate method for voxelizing polygon meshes," in *Proceedings of the 1998 IEEE symposium on Volume visualization*, 1998, pp. 119–126.

[20] K. Crane, I. Llamas, and S. Tariq, "Real-time simulation and rendering of 3d fluids," *GPU gems*, vol. 3, no. 1, 2007.

[21] R. Cortez, *Impulse-based particle methods for fluid flow*. University of California, Berkeley, 1995.

[22] H. Bhatia, G. Norgard, V. Pascucci, and P.-T. Bremer, "The helmholtz-hodge decomposition—a survey," *IEEE Transactions on visualization and computer graphics*, vol. 19, no. 8, pp. 1386–1404, 2012.

[23] F. H. Harlow, J. E. Welch, *et al.*, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *Physics of fluids*, vol. 8, no. 12, p. 2182, 1965.

[24] E. Grinspun, M. Desbrun, K. Polthier, P. Schröder, and A. Stern, "Discrete differential geometry: An applied introduction," *ACM Siggraph Course*, vol. 7, no. 1, 2006.

[25] H. Prautzsch, W. Boehm, and M. Paluszny, *Bézier and B-spline techniques*. Springer Science & Business Media, 2002.

[26] S. Gottlieb and C.-W. Shu, "Total variation diminishing runge-kutta schemes," *Mathematics of computation*, vol. 67, no. 221, pp. 73–85, 1998.

[27] J. Bender, M. Müller, and M. Macklin, "A survey on position based dynamics, 2017," *Proceedings of the European Association for Computer Graphics: Tutorials*, pp. 1–31, 2017.

[28] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*. SIAM, 2000.

[29] N. Foster and D. Metaxas, "Realistic animation of liquids," *Graphical models and image processing*, vol. 58, no. 5, pp. 471–483, 1996.

[30] J. Liesen and Z. Strakos, *Krylov subspace methods: principles and analysis*. Numerical Mathematics and Scie, 2013.

[31] R. Bridson, *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.

[32] N. Foster and R. Fedkiw, "Practical animation of liquids," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 23–30.

[33] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids.," in *Symposium on Computer Animation*, vol. 65, 2010, p. 74.

[34] N. Chentanez and M. Müller, "Real-time eulerian water simulation using a restricted tall cell grid," in *Acm siggraph 2011 papers*, 2011, pp. 1–10.

[35] H. Shao, L. Huang, and D. L. Michels, "A fast unsmoothed aggregation algebraic multigrid framework for the large-scale simulation of incompressible flow," *ACM Transactions on Graphics (TOG)*, vol. 41, no. 4, pp. 1–18, 2022.

[36] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[37] J. Zhang, "Acceleration of five-point red-black gauss-seidel in multigrid for poisson equation," *Applied Mathematics and Computation*, vol. 80, no. 1, pp. 73–93, 1996.

[38] K. Stüben, "A review of algebraic multigrid," *Numerical Analysis: Historical Developments in the 20th Century*, pp. 331–359, 2001.

[39] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[40] P. Varcholik, *Real-time 3D rendering with DirectX and HLSL: A practical guide to graphics programming*. Addison-Wesley Professional, 2014.

[41] R. Marroquim and A. Maximo, "Introduction to gpu programming with glsl," in *2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, IEEE, 2009, pp. 3–16.

[42] M. Englert, "Using mesh shaders for continuous level-of-detail terrain rendering," in *Special interest group on computer graphics and interactive techniques conference talks*, 2020, pp. 1–2.