ENHANCED DETAIL FLUID SIMULATION USING IMPLICIT DENSITY-INVARIANCE ALGORITHM

BY

Shuo-Wei Chang

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
awarded by DigiPen Institute of Technology
Redmond, Washington
United States of America

August
2017

Thesis Advisor: Barnabas Bede

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS TITLED

Enhanced Detail Fluid Simulation using Implicit Density-Invariance Algorithm

BY

Shuo-Wei Chang

HAS BEEN SUCCESSFULLY COMPLETED ON September 12, 2017.

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

APPROVED:

Dmitri Volper date	Erik Mohrman date
Graduate Program Director	Dean of Faculty
Pushpak Karnick date	Claude Comair date
Department Chair, Computer Science	President

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

THESIS APPROVAL

DATE: September 12, 2017

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Shuo-Wei Chang

ENTITLED

Enhanced Detail Fluid Simulation using Implicit Density-Invariance Algorithm

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Barnabas Bede date	Erik Mohrman date	Dmitri Volper date
Thesis Committee Chair	Thesis Committee Member	Thesis Committee Member

Xi Li date

Thesis Committee Member

ABSTRACT

In this thesis, we propose an efficient approach to fluid simulation on the GPU, by directly solving Navier-Stokes equations, using implicit numerical approaches. Our method is based on a density-variance algorithm and semi-Lagrangian approach. Solving directly the mass conversation equation, using central finite difference and explicit Euler method, leads to a conditionally stable result. We propose a solution with semi-Lagrangian and an implicit Euler method, using a splitting algorithm to release the time step restriction based on grid resolution, in addition to removing unnecessary clamping function that was added to previous methods due to numerical instability. We also apply a similar technique on the momentum conservation equation using Jacobi iterations to solve the viscous part of Navier-Stokes equations, this way reducing visual artifacts.

To my family

ACKNOWLEDGMENTS

I would like to thank my thesis advisor for all the supports.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

Introduction

One of the reasons video games are appealing is the ability to create desirable virtual world and allowing players to interact with. Fluids appear in several natural phenomena commonly seen in daily life, such as camp fire, ocean and smoke for instance. And adding this phenomenon to games allow developers to create realistic, immersive, and interesting virtual environments. Real-time fluid simulation has been a hot topic for research over the last decades. Let us recall here that many years ago, fluids were just simple particles, or animated textures, however, fluids are more complicated than the systems that these systems can capture. The resulting fluid behavior using these systems is very limited. Creating fluid effects can be very challenging and time-consuming, and also, matching real-world fluid behavior is impossible. In real life, fluids like smoke are governed by physics equations. In order to achieve realistic simulation in virtual world, we need to model them based on these equations. Thanks to the improving of hardware and computational power, and the effort of numerous researchers, we are able to approximate these equations in interactive frame time with the help of numerical methods.

1. Types of Simulation

There are two major fields in fluid simulation. One represents the whole movement of fluids [2], while the other one only discusses surface movement [12]. Surface movement like wave simulation is commonly done by generating a height map procedurally [12]. In contrast, tracking the whole fluid body uses a fixed or irregular grid approach [2,12]. However, above all, these are just different ways to describe the motion of the fluid. There are studies on other various aspects related to fluid simulation, for instance, how to render it is another different problem when it comes to shading the fluid. However, in this thesis, we will primarily focus on solving the governing equations that describe fluid motion than other extensions. Terminology we have used so far, we will define later in the thesis.

2. Thesis Overview

Before jumping into the topic, let's have an overview of this thesis' structure. We will first review essential mathematics used in fluid dynamics. Then we will go through algorithms in the order of solving N-S equations in Lagrangian and Eulerian point of view, and finally we will present our improvement based on Eulerian's viewpoint. We establish necessary material needed through out the thesis before we introduce our methods, our improvement is based on these ideas, so we will recall the background first. Our algorithm inspired by [5] and [1] is based on the idea of temporarily relaxing incompressibility to reduce equation complexity, compared to [5], but also preserving stability. We will also introduce Cole-Hopf Transformation to solve N-S equations. Finally let us bring up before we start the goals that we want to meet when thinking of fluid simulation for games.

3. Goals

Again, we want to emphasize that when we talk about fluid simulation in this thesis, we are talking about putting the simulation in games. Our goal is not only to run simulations in real-time but to have an algorithm that can be integrated to games. Here are the goals we want to achieve:

3.1. Fast. First of all, we want our simulation to be fast. It should run in 30 fps or above, and be stable under various time steps.

3.2. Scalable. We want it to be controllable. We want to create different fluid behaviors. We also want it to be responsive models to player interaction so that we can create unique user experience.

3.3. Realistic. The approximate result should retain as many details as possible and be convincing to players at first glance.

That is it for the big picture. Next section we will start with a review on vector calculus.

CHAPTER 2

Vector Calculus

In this section, basic vector calculus knowledge will be reviewed. This section is not for thorough review as we will not go through all the proof details, but we show important concepts, principles and theorems. This review covers the material needed for the following sections or chapters. We show our examples in a Cartesian coordinate system.

1. Scalar Field

In mathematics and physics, by definition, when every single point in either two or three dimensions is associated to a scalar value, we call it a scalar field. These values are defined by scalar valued functions of positions in space. In fluid dynamics, these values normally stand for a physical quantity. Take $p(x, y)$ as an example, it represents pressure at a given point on a two dimension space.

2. Vector Field

Before defining vector field, let's define what is a vector function. Similar to scalar functions, a vector function is a function that takes a position and returns a vector, for example $\mathbf{v} = f(x, y)$ (We use bold font to represent vectors in this thesis). And these vectors construct the result of a vector field just as a scalar field. In fluid dynamics, we have to generate a velocity field, which is a vector field, to track the flow of fluids.

Next we will review three important vector calculus concepts, gradient, divergence, and curl. They are important in fluid dynamics. For example, pressure force can be described as a gradient. Their main role is to help us analyze the net forces working on fluid.

3. Gradient

A del operator, noted as ∇, describes spatial partial derivatives. Gradient is given

$$\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}) \tag{2.1}$$

in two dimensions. You can also extend it to three dimension by adding a z term. Gradient takes spatial partial derivatives of a given function. Consider a scalar function f in two dimensions, gradient is defined as the following,

$$\nabla f(x, y) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) \tag{2.2}$$

and it can measure the rate change of the example function along x and y axis, it points in the increasing direction of the result scalar field. In later chapter, you will see how it is used to measure pressure force in Navier-Stokes equations. And let's have a concrete example before we move on. Let $f(x, y) = x^2y$, find $\nabla f(3, 1)$. The gradient of f is:

$$\frac{\partial f}{\partial x}(x, y) = 2xy \tag{2.3}$$

$$\frac{\partial f}{\partial y}(x, y) = x^2 \tag{2.4}$$

Therefore, the gradient is $(6, 9)$. We can also apply gradient on vector-valued function, the answer is in a matrix form. Given a vector field $\mathbf{f} = (u, v, w)$, the gradient of it is,

$$\nabla \mathbf{f} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \tag{2.5}$$

sometimes known as Jacobian.

4. The Divergence

The concept of divergence is needed to measure how much of the field flows outward from a given point of a vector-valued function. Given the idea of taking the dot-product between Del operator and a vector $\mathbf{u} = (u, v)$, the result is a scalar value. Divergence in two dimensions is,

$$\begin{aligned}\nabla \cdot \mathbf{u} &= (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}) \cdot (u, v) \\ &= \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\end{aligned} \tag{2.6}$$

5. The Curl

Curl is used to measure the amount of rotation at a given point. Using the same example in divergence, curl is given as,

$$\nabla \times \mathbf{u} = \frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} \tag{2.7}$$

you will see how to use curl to measure turbulence of fluid in this paper.

CHAPTER 3

The Navier–Stokes Equations

In chapter 2, we have reviewed some vector calculus basics and how individual operators are defined. In this chapter, we will introduce the famous Navier-Stokes equations that govern the motion of viscous fluids. They are derived from Newton's Second Law of Motion and by Navier, Poisson, Saint-Venant, and Stokes between 1827 and 1845. Navier-Stokes Equations hold throughout the entire simulation in time and space. We will break equations into pieces to further understand what each term stands for.

1. Incompressible Navier–Stokes equations

Most fluid simulation done in computer graphics is governed by the famous incompressible Navier–Stokes equations. We assume incompressible, homogeneous fluid, which means that the density of fluids stays constant both in time and space. The following equations describe the flow of incompressible fluid over time,

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F} \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{3.2}$$

Equation 3.1 is called the momentum equation. It represents the conservation of momentum. Equation 3.2 is the incompressibility condition for fluid and stands for conservation of mass. It is originally derived from the continuity equation,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{3.3}$$

Since we assume incompressible and homogeneous fluid, the first term becomes zero.

$$\frac{\partial \rho}{\partial t} = 0 \tag{3.4}$$

and also because ρ is a constant value, Equation 3.2 becomes

$$\rho \nabla \cdot \mathbf{u} = 0 \tag{3.5}$$

divided by ρ, we reached equation 3.2. Now let's look at the definition of symbols in equation 3.1. $\mathbf{u}$ is the fluid velocity, it plays an important role in transporting substances throughout the simulation. And we can see from the equations that our goal is to calculate how the velocity field evolves during the simulation and use it to move fluid such as smoke or fire. ρ is fluid density, for example, water is roughly $1000kg/m^3$. p is the fluid pressure in the unit of force per unit area. ν is the fluid kinematic viscosity, it describes how sticky fluid is, in other words, how fast fluid diffuses. $\mathbf{F}$ are the external net body forces on a fluid, they can be regional forces affecting part of fluid or gravity that affects whole fluid. The most common one is gravity, we can also apply some forces on fluid when it travels through certain area during the simulation.

2. Terms in the Navier–Stokes equations

So far we have some basic ideas of how Navier–Stokes equations look like and what each symbol stands for. Differential equation 3.1 appears to be difficult to solve at first glance. We can see that the partial derivative of $\mathbf{u}$ respective with t is the sum up of four elements on the right side of equation 3.1. In this section, we will break it down into pieces and explain each term in order. Considering equation 3.1 again we can see that each term is a result caused by a certain force written in acceleration. After describing what each term stands for, it will give us better idea how to derive equation 3.1 from Newton's Second Law of Motion.

2.1. Advection. Advection is a mechanism of transporting substance by the fluid. In Navier–Stokes equations, the advection term transports velocity from one location to another on the velocity field. In other words, this term helps velocity field evolve over time. It represents self-advection of the velocity field.

2.2. Pressure. Fluid might not be spread equally in space. The unit used to measure how much fluid per given space is called density. The difference of density throughout the space causes pressure force. High-pressure areas push on low-pressure areas. Take the negative gradient of the pressure field then we calculate pressure force to determine which way the fluid should flow. In mathematics, gradient is pointing to ascent direction. Then, it is obvious that we have to put a negative sign to correct the result.

2.3. Diffusion. The third term is called viscosity. This term tells how sticky the fluid is. This property can be imagined easily in real life. Think about one pours different types of liquid on to a wall. Liquids like water would only remain on the wall for a short time, while liquids like honey would remain on the wall for quite long time. Another way to think of this phenomenon is by observing how quickly would a type of fluid diffuses to the environment. We can also think in an way that this is a force that tries to make a fluid particle to move in average speed of nearby particles. This way it can prevent fluid from deforming.

2.4. External Forces. The last force that contributes to net forces is due to external forces. They can be body or local force. Wind force is an example of a local force, it can be applied to the fluid in some regions of the simulation domain. Gravity, which is a body force, should be applied to every fluid particles in most cases.

3. Material Derivative

Before we start to derive equation 3.1, we need to know what is material derivative. For understanding material derivative, we need to learn about the difference between Lagrangian and Eulerian point of view.

3.1. Lagrangian and Eulerian Viewpoints. These are two different approaches to track continuous motion. The Lagrangian point of view is what you are probably most familiar with. It treats fluid dynamics like a particle system. It considers each point in the fluid simulation as a particle that carries position, velocity, density and so on. Smoothed Particle Hydrodynamics (SPH) is one of the famous numerical solvers in a Lagrangian framework. In [10], SPH is used to simulate nonviscous fluids using a particle system where each fluid is treated as a rigid body carries its own properties. Simulating advection in a Lagrangian framework is straightforward since we know that equation 4.7 holds true in the momentum equation, basically, equation 4.7 states that the value carried by a particle remains the same through out the simulation, so the remaining thing for advection is really nothing but to do position integration based on velocity. The fact that each fluid is treated as a particle also helps solving boundary condition and apply body forces to the fluid. They are merely collision between rigid bodies and numerically integration. However, enforcing incompressibility is not as easy as advection in Lagrangian point of view. In addition, handling large numbers of particles is challenging to achieve real time performance. A hash grid is a common approach to optimize the update for a large numbers of particles.

One the other hand, the Eulerian approach, commonly used for fluids, takes a different tactic. Instead of thinking in particles, Eulerian approach measures fluid properties, such as velocity, density, temperature, etc., in a fixed grid space. Quantities at those points change in time, they are contributed by fluid flow that passes through those points. It's might be not entirely intuitive at first, but we will show an example to explain it in a better way. Imagine you are measuring a fixed point's temperature in air. As warm air moves past followed by cold air, the temperature reading at that position will raise and then decrease, even though the temperature of any particle in air does not change at all! With the same idea, one can also measure the wind direction.

Numerically, the Eulerian viewpoint makes it easier to approximate those spatial derivatives on a fixed grid than on a particle system with arbitrary moving points, which normally corresponds to the Lagrangian viewpoint. Moreover, as we progress through the paper, you will find that the data representation in an Eulerian framework can be parallel processed more easily than in a Lagrangian framework. However, it is hard to choose a grid approach that works for all resolutions in an Eulerian framework to achieve significant precision for advection. It requires re-sampling approximation, which is lossy in general, on a grid during the simulation.

Material Derivative connects the two viewpoints. Consider we want to measure how a quantity, say density q, changes over time at a fixed point in the body of the fluid. In Eulerian point of view, given a function of space $q(t, \mathbf{x})$ tells us the value of q at time t and position $\mathbf{x}$. Since we want to measure the change rate over time, we take the total derivative:

$$\begin{aligned} \frac{d}{dt}q(t, \mathbf{x}) &= \frac{\partial q}{\partial t} + \nabla q \cdot \frac{d\mathbf{x}}{d\mathbf{t}} \\ &= \frac{\partial q}{\partial t} + \nabla q \cdot \mathbf{u} \\ &= \frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q \\ &= \frac{Dq}{Dt} \end{aligned} \tag{3.6}$$

This is the material derivative! We can also explain this in Lagrangian viewpoint by considering that a particle that carries quantity q that has a given rate of change over time currently at location $\mathbf{x}$. The first term $\partial q/\partial t$ is how fast q is changing at that fixed point in space. The second term $\mathbf{u} \cdot \nabla q$, also known as advection, adds how much of that change is due to differences caused by neighbor fluid. Numerically, advection is effectively automatic in Lagrangian point of view, while it is harder to solve in Eulerian measurement.

3.2. Material Derivative on Vector Quantities. At this point it might be still unclear what does the material derivative mean when it applies to vector quantities mathematically. The meaning is simple: we calculate it with each component separately. Let's consider the advection term in Navier-Stokes equations for practice. We assume the fluid is moving on a velocity field, $\mathbf{u}$, in three dimension. Taking the material derivative on velocity $\mathbf{u} = (u, v, w)$ results in a self-advection formula.

$$\frac{D\mathbf{u}}{Dt} = \begin{bmatrix} Du/Dt \\ Dv/Dt \\ Dw/Dt \end{bmatrix} = \begin{bmatrix} \partial u/\partial t + \mathbf{u} \cdot \nabla u \\ \partial v/\partial t + \mathbf{u} \cdot \nabla v \\ \partial w/\partial t + \mathbf{u} \cdot \nabla w \end{bmatrix} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \tag{3.7}$$

Notice that we simply treat to each component separately.

4. Incompressibility

In physics, there are two types of fluids, incompressible and compressible. Incompressible fluid corresponds to liquid, such as water. On the other hand, compressible fluid is a gas such as smoke or air in general. We keep in mind that there's no perfectly incompressible flow, it is simply impossible, otherwise we would not be able to hear underwater. In real-time computer graphics, or games, it is common to treat the fluid as incompressible flow. Compressible flow normally adds more complexity and is more expensive to simulate. Furthermore, compressible flow contributes little visually and impacts fluid motion in microlevel. Hence, in general, it is practical to just consider incompressible fluid for both liquid and gas for animation.

In earlier section, we have showed how to reach equation 3.2 by applying incompressibility condition mathematically, making ρ a constant value. In this chapter, we will show more concrete details deriving the equation.

Let Ω be an arbitrary and fixed region of space. We define the mass of fluid M as

$$M = \int\int\int_{\Omega} \rho \tag{3.8}$$

Taking the integral around the boundary of the fluid speed along the surface normal. The rate at which mass flows in or out of Ω is given by

$$\frac{\partial M}{\partial t} = -\int\int_{\partial\Omega} \rho\mathbf{u}\cdot\mathbf{n} \tag{3.9}$$

Expanding M and transforming the right hand side with the help of divergence theorem, we get

$$\int\int\int_{\Omega} \frac{\partial \rho}{\partial t} = -\int\int\int_{\Omega} \nabla\cdot(\rho\mathbf{u}) \tag{3.10}$$

Since the equation should hold in any given region Ω, the following equation must also hold true,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{3.11}$$

We observe that this is equation 3.3, termed continuity equation. Assuming incompressible flow, or in other words, flow with fixed density, we reach the final incompressibility condition

$$\nabla \cdot \mathbf{u} = 0 \tag{3.12}$$

5. The Momentum Equation

Now let us gain a deeper understanding of equation 3.1. As we just went through previously, equation 3.1 can be separated into four pieces and each piece is not hard to understand. It tells us how different types of forces that form net force acting on fluid acceleration. We observe that this is in fact just Newton's equation $\mathbf{F} = m\mathbf{a}$. Starting with material acceleration in two dimensions, which is defined as following,

$$\mathbf{a} \equiv \frac{D\mathbf{u}}{Dt} \tag{3.13}$$

In fluid dynamics, it is the acceleration following a fluid particle by definition. Expanding it in two dimensions using the definition of material derivatives, we have,

$$\mathbf{a} \equiv \frac{\partial \mathbf{u}}{\partial t} + u\frac{\partial \mathbf{u}}{\partial x} + v\frac{\partial \mathbf{u}}{\partial y} \tag{3.14}$$

Applying material acceleration, Newton's law is now,

$$\mathbf{F} = m(\frac{D\mathbf{u}}{Dt}) \tag{3.15}$$

There are other three types of forces that contribute to fluid motion, pressure, viscosity, and body forces. We start with body forces, and in our case, we use gravity as an example. The force of gravity acts the same way as in solid mechanics,

$$\mathbf{F}_{gravity} = m\mathbf{g} = \rho\mathbf{g}dxdydz \tag{3.16}$$

Define $dxdydz$ as V, rewrite it to,

$$\rho\mathbf{g}dxdydz = \rho\mathbf{g}V \tag{3.17}$$

Next is the pressure force, in fluid dynamics, high pressure area pushes low pressure area. The way to measure imbalance pressure is to take negative gradient of pressure, define pressure as P, we have,

$$\mathbf{F}_{pressure} = -\nabla p dxdydz = -\nabla pV \tag{3.18}$$

The other fluid force is due to viscosity. Using the rate of stress and rate of strain tensors, we write the viscosity force in each component,

$$\frac{F_i}{V} = \frac{\partial}{\partial x_j}\Big[\eta(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}) + \lambda\delta_{ij}\nabla\cdot\mathbf{u}\Big] \tag{3.19}$$

where η is the dynamic viscosity coefficient, λ is the second viscosity coefficient, δ_{ij} is the Kronecker delta and j is used to sum up from j = 1, to the number of dimensions we are working on. Using the incompressibility condition, we rewrite 3.19 into a vector form, given,

$$\frac{\mathbf{F}_{viscosity}}{V} = \eta \nabla^2 \mathbf{u} \tag{3.20}$$

∇^2 is called Laplacian operator. It is defined as follows,

$$\nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.21}$$

Applying this operator on a vector is simply to input each component into the same calculation. Finally, we almost reach equation 3.1, but first, let's put everything together. We then have,

$$\rho V \frac{D\mathbf{u}}{Dt} = \rho \mathbf{g} V - V \nabla p + V \eta \nabla \cdot \nabla \mathbf{u} \tag{3.22}$$

Rearrange it by dividing each side of equation by V and ρ,

$$\frac{D\mathbf{u}}{Dt} = \mathbf{g} - \frac{1}{\rho} \nabla p + \frac{1}{\rho} \eta \nabla \cdot \nabla \mathbf{u} \tag{3.23}$$

Define the kinematic viscosity as $\nu = \frac{\eta}{\rho}$ to get,

$$\frac{D\mathbf{u}}{Dt} = \mathbf{g} - \frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla \mathbf{u} \tag{3.24}$$

Expand material acceleration 3.14, we reach the final momentum equation describing incompressible Newtonian fluid [7]. Notice that we have **g** instead of **F**, and as you might expect, this term can be the sum of all external body forces. In this section, we only list one of the common forces, gravity, that always works on fluid.

CHAPTER 4

Numerical Methods

We have basically covered all the essential materials in mathematics, physics and some basic numerical concepts. Beginning from this chapter, we will start to solve Navier-Stokes equations with incompressibility conditions numerically. There are many existing algorithms developed by researchers in the past decades. In [20], they write Navier-Stokes equations in different form using Reynolds number to describe viscous forces and alter the number in order to increase numerical stability or certain effect whereas in true physics, it is a constant value used to describe different flows. In addition, they solve them with finite difference approach. In [Foster and Metaxas], their approach also utilize finite difference and they use particles to model surface tracking. In [5], Jos Stam introduced a real time method that is stable under any given time step. Based on [5], [15] developed level set method to help track the surface of the water. Besides this, Smoothed Particle Hydrodynamics solves the incompressible Navier-Stokes equations in Lagrangian point of view. It is a very natural numerical approach applying incompressibility condition.

Our approach is based on the idea of [5] and [1] by temporarily loose incompressible condition to solve incompressible Navier-Stokes equations. That means we actually solve for density and use it as a variable plug-in to the momentum equation to update velocity that carries fluid around. In this chapter, we will build up the theory behind [5] and [1] in order to introduce our improvement. However, instead of talking about [5], we will talk about [9]. [9] is a more modern approach and runs on GPU, its implementation is alike [5] but on GPU. The main difference between the two methods is to use finite difference. If we think of the simulation happening on a 2-D array then we can store the information we need to move fluid, for example, density, velocity, or pressure. We will have to iterate these arrays by a certain time step to run the simulation. Furthermore, the reason behind choosing a GPU implementation is because GPU is good at processing 2-D array like data in parallel that can make the simulation highly efficient compared with a CPU approach. It requires more setup on the CPU side in order to access and manipulate data on GPU. We won't go too much into details on that except for showing you shader code snapshot. Usually it is recommended that one implements a new method first on CPU and move on to GPU.

Before introducing [9] and [1], we will cover common numerical methods used by the two algorithms. In the beginning of this chapter, we are going to first introduce how vector calculus is approximated by finite difference method. And then we are going to introduce two methods for solving Naiver-Stokes equations based on finite difference method.

1. Finite Difference Method

Finite difference method is a popular numerical solution to vector calculus due to it's a rather fast and simple approach deriving from Taylor's polynomial. The precision of the simulated finite difference method depends on time and space steps. Generally, smaller steps produce better results. In chapter 2, we covered important calculus knowledge in fluid simulation. And now, we write them in finite difference form to show how calculus is approximated by finite difference method. The method we are about to introduce is called central finite difference, there are other two methods called backward/forward finite difference methods exists. We will show them on a two dimension Cartesian grid.

The definition of gradient is

$$\nabla f(i,j) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) \tag{4.1}$$

Its central finite difference form is

$$(\frac{f_{i+1,j} - f_{i-1,j}}{2\delta x}, \frac{f_{i,j+1} - f_{i,j-1}}{2\delta y}) \tag{4.2}$$

where i,j refer to locations on a Cartesian grid, and $\delta x, \delta y$ is grid's size in x,y dimension. Given in similar format, divergence is given as

$$\nabla \cdot \mathbf{f}(i,j) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{4.3}$$

Where $\mathbf{f}(i,j) = (u,v)$, and the central finite difference form is given as

$$(\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\delta y}) \tag{4.4}$$

Last but not least, Laplacian in calculus is given as,

$$\nabla^2 f(i,j) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{4.5}$$

Where $\nabla^2 = \nabla \cdot \nabla$, and the central finite difference form is

$$(\frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{(\delta x)^2}, \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{(\delta y)^2}) \tag{4.6}$$

These form can be extended to three dimension easily by similar ideas. Finite difference method is a great method to solve fluid dynamics in Eulerian point of view with second-order accuracy. In other words, we can have a fixed grid that holds all the data we need during the simulation.

2. Advection Algorithm

The second numerical algorithm that would be used in both approaches is advection. Recall in Navier-Stokes equations 3.1, the first term on the right side derived from material derivative refers to advection, which means it helps to transport velocity on the simulation grid. In order to do so in Eulerian point of view is nontrivial because we have a fixed grid instead of bunch of free moving particles bouncing around. It is straightforward to track velocity field through the Lagrangian point of view because it is carried by particles, so is incompressible condition. On the contrary, in Eulerian point of view, it takes more effort and analysis to track velocity. In this section, we will introduce a semi-Lagrangian approach to solve any given advection equation in a stable manner, which means that it is stable under any time step size.

2.1. Semi-Lagrangian. Semi-Lagrangian approach was first introduced by [3] in the meteorological literature. It is proved that semi-Lagrangain is a first-order and unconditionally stable numerical method in [6] with the drawback of dissipation problem. We will borrow some of the notation [4] used to present the algorithm in this section. Given the material derivative,

$$Dq/Dt = 0 \tag{4.7}$$

In order to solve it, we first write out the PDE, for example, in one dimenstion:

$$\frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} = 0 \tag{4.8}$$

Then replace the above equation with forward Euler and central finite difference for the spatial derivative, we get:

$$\frac{q_i^{n+1} - q_i^n}{\Delta t} + u_i^n \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x} = 0 \tag{4.9}$$

Rearrange it into an explicit schema to get the new value of q:

$$q_i^{n+1} = q_i^n - \Delta t u_i^n \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x} \tag{4.10}$$

Explicit formula is a straightforward and easy to understand and implement, but probably the least one you want if there's obviously a better choice with a little more effort. It is because forward Euler is unconditionally unstable in equation 4.10 regardless of how small Δt is. Even if switching to a more stable integration technique, like, Eulerian Leapfrog, will only give us a conditionally stable result (more specifically, Courant–Friedrichs–Lewy condition, a.k.a CFL condition). There are still other issues remain with this spatial discretization. However, we will not go deeper on this topic but a good reference in this area is [4].

Instead, we take a different approach motivated by physics, unconditionally stable and was introduced to graphics by [5] called semi-Lagrangian method. The basic idea of using semi-Lagrangian for advection integration is based on the approximation of Lagrangian time derivative. We want to find the new value of q at its current grid point. In order to achieve it, we run backwards through the velocity field at current point to figure out where it started. And then bring the new value over to the current point. Let's define the current position is $\mathbf{x}_i$, we would like to find the new value of q at $\mathbf{x}_i$, written as q_i^{n+1}. The first step is to find the previous location $\mathbf{x}_p$ by backtracking along the velocity field.

$$\mathbf{x}_p = \mathbf{x}_i - \Delta t \mathbf{u}(\mathbf{x}_i) \tag{4.11}$$

And then look up q value at location $\mathbf{x}_p$

$$q_i^{n+1} = q(\mathbf{x}_p) \tag{4.12}$$

Most likely $\mathbf{x}_p$ won't be exactly on the grid, so we won't have exact value to look up on the grid. In general, trilinear (bilinear in two dimensions) interpolation is used to approximate the result. Let's illustrate this, in one dimension, for completeness before we move on to the next topic. Assuming $\mathbf{x}_p$ lies between $[\mathbf{x}_j, \mathbf{x}_{j+1}]$, $\alpha = (\mathbf{x}_P - \mathbf{x}_j)/\Delta x$ be the coefficient for linear interpolation. Then we look up the new value by

$$q_i^{n+1} = q(\mathbf{x}_p) = (1-\alpha)q_j^n + \alpha q_{j+1}^n \tag{4.13}$$

In the application, we apply this to velocity field to solve Navier-Stokes equations and update density field to move the fluid around.

3. Fast Fluid Dynamics Simulation on the GPU

So far we have discussed common numerical schemes used in solving fluid dynamics equations. Next, we are going to show you some previous works utilizing the power of modern GPU for fluid calculation. We start from Mark Harris' work published in GPU GEM VOLUME 1 [9]. Their work was based on [5]. It is an unconditionally stable solver using many passes on GPU. The second one is Martin Guay's work published in GPU PRO 2 [1]. They presented an efficient way to simulate Newtonian fluid that only requires a single pass on GPU but the result is conditionally stable only, with the constraint on choosing a proper time step depending on grid resolution.

3.1. The approach. To begin with the main idea we just mentioned, this paper is highly based on the [5]. While [5] has the implementation on CPU, this paper has their algorithm implemented on modern GPU because this type of computation is well suited on these hardwares. When we simulate fluid in Eulerian's point of view, it is important to recall that the computation is performed on a fixed grid of cells. Array is a great data representation to represent the data, on GPU, we have texture that is similar to an array as basic data storage unit, and we normally refer a cell as a pixel. In fact, GPU achieve high performance calculation through parallelism: they are capable of processing multiple pixels simultaneously.

3.2. The Helmholtz-Hodge Decomposition.

The Helmholtz-Hodge Decomposition, HHD for short, plays an important role applying Incompressible condition to equation 3.1. It transforms equation 3.1 into a form that is suitable to apply numerical computation.

3.2.1. *Helmholtz-Hodge Decomposition Theorem:*

Definition: Given any vector field, it can be decomposed into the sum of two other vector fields: a divergence-free vector field, and a gradient of a scalar field. In mathematics, it means that given a vector field $\mathbf{w}$ on D, there exist $\mathbf{u}$, p such that

$$\mathbf{w} = \mathbf{u} + \nabla p \tag{4.14}$$

where $\mathbf{u}$ has zero divergence and is parallel to ∂D [define], that is, $\mathbf{u} \cdot \mathbf{n} = 0$ on ∂D. It also says that the divergence-free field approaches zero at the boundary. For the proof of the theorem, please refer to [7].

3.2.2. *The Projection Operator:*

After solving equations 3.1, we get new values of the velocity field, which is not divergent free. But this result contradicts equation 3.2. So we correct it by rearranging equation 4.14 into,

$$\mathbf{u} = \mathbf{w} - \nabla p \tag{4.15}$$

Secondly, HHD leads to an equation for solving the pressure field, by applying divergence operator to both side of equations 4.14, we obtain:

$$\nabla \cdot \mathbf{w} = \nabla \cdot (\mathbf{u} + \nabla p) = \nabla \cdot \mathbf{u} + \nabla^2 p \tag{4.16}$$

With equation 3.2, it can be simplified to:

$$\nabla^2 p = \nabla \cdot \mathbf{w} \tag{4.17}$$

which is a Poisson-pressure equation of the fluid. And at this point of the paper, we have all we need to reach the final divergence-free field **u**. Lastly, we can define a projection operator, $\mathbb{P}$, that projects a vector field **w** onto its divergence-free component, **u**. By definition,

$$\mathbb{P}\mathbf{w} = \mathbb{P}\mathbf{u} = \mathbf{u} \tag{4.18}$$

based on this definition, applying the projection operator,

$$\mathbb{P}\mathbf{w} = \mathbb{P}\mathbf{u} - \mathbb{P}(\nabla p) \tag{4.19}$$

Therefore,

$$\mathbb{P}(\nabla p) = 0 \tag{4.20}$$

Now, applying projection operator to the Navier-Stokes equations, we get

$$\mathbb{P}\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}(-(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{F}) \tag{4.21}$$

Since $\mathbf{u}$ is divergence-free, also realizing that $\mathbb{P}(\nabla p) = 0$. We are left with the final equation,

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}(-(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{F}) \tag{4.22}$$

The paper solves it with typical split technique (More information about this technique is given in [4]). It means that each component in the final equation is a step that takes a field as input, and produces a new field as output. The author defines an operator $\mathbb{S}$ that is equivalent to the right side of equation 4.22 over a single time step. Furthermore, $\mathbb{S}$ is defined as the composition of operators (using the same idea) for advection ($\mathbb{A}$), diffusion ($\mathbb{D}$), force application $\mathbb{F}$, and projection ($\mathbb{P}$):

$$\mathbb{S} = \mathbb{P}\circ\mathbb{F}\circ\mathbb{D}\circ\mathbb{A} \tag{4.23}$$

The solver then starts from the rightmost operator to leftmost one, each component takes velocity from last frame and iterate it over a time step and pass the intermediate value to the next component as input. For projection operator, updating the pressure field is required as an additional input as it is shown in equation 4.17. Now, since we have already seen how semi-Lagrangian works, and it is not hard to imagine how to implement it with code. Next, we want to take a look at how to solve diffusion and Poisson equation with Laplacian operator numerically using the idea of finite difference.

3.3. Viscous Diffusion. This property is given as:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u} \tag{4.24}$$

This can be solved using explicit integration but we know that it is not as stable as we want. The paper provides an implicit solution that is stable for arbitrary time steps and viscosities, given as:

$$(\mathbf{I} - \nu \delta \nabla^2)\mathbf{u}(\mathbf{x}, t + \delta t) = \mathbf{u}(\mathbf{x}, t) \tag{4.25}$$

where $\mathbf{I}$ is an identity matrix. This equation shares the same format as equation 4.17, it is a Poisson equation for velocity. It can be with an iterative relaxation technique. It is solved with Jacobi iteration, the simplest iterative technique, in this paper.

3.3.1. *Solution of Poisson Equations using Jacobi Iteration:*

The Poisson equation is a matrix equation in the form of $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is the vector of values we are solving (for fluid simulation, we need to solve for pressure and diffusion), $\mathbf{b}$ is a vector of constants, and $\mathbf{A}$ is a matrix. [8] provides proof of Jacobi iteration for general matrix equations. Equation 4.17 and 4.25 can be discretized using the formulation of finite difference and rewritten in the following form.

$$x_{i,j}^{k+1} = \frac{x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k} + \alpha\beta_{i,j}}{\beta} \tag{4.26}$$

where α and β are constants. Solving Poisson-pressure and viscous diffusion equation accordingly. For the Poisson-pressure equation, x represents p, b represents $\nabla \cdot \mathbf{w}$, $\alpha = -(\delta x)^2$, and $\beta = 4$. For the viscous diffusion equation, both x and b represent $\mathbf{u}$, $\alpha = (\delta x)^2/\nu\delta t$, and $\beta = 4 + \alpha$.

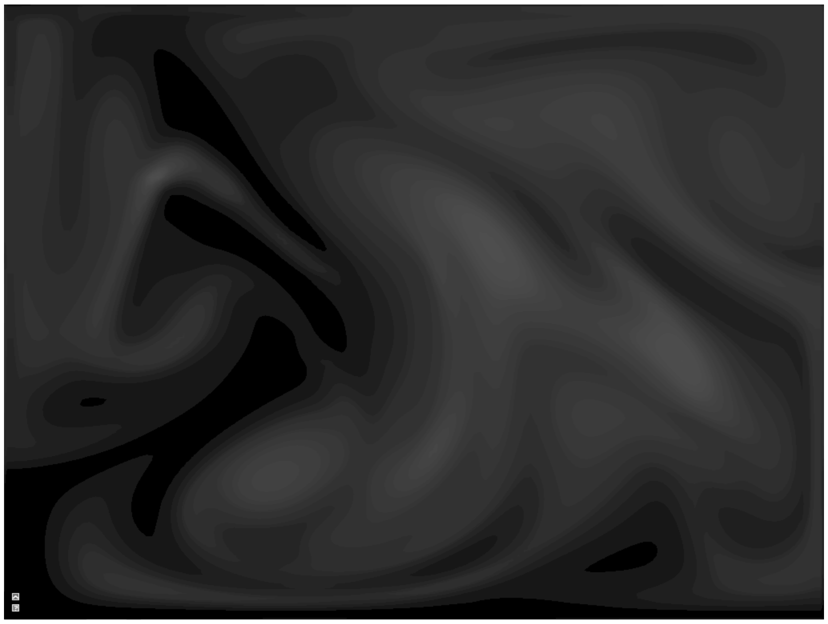

Figure 1. Result

4. Simple and Fast Fluids

In this chapter, we are going to introduce a simple and efficient algorithm also runs on the GPU but only requires a single pixel shader. This algorithm was introduced by Martin Guay in [1]. He proposed a method that allows temporarily relaxing the incompressibility condition, and then solve the full Naiver-Stokes equations over the domain in a single pass. A single pass algorithm on GPU means we only need to draw one time to solve the equation.

4.1. Density-Invariance Algorithm. The solver algorithm is based on density invariance, which has proven to be stable for the SPH method. First of all, observe that Equation 3.11 can be rewritten as

$$\frac{\partial \rho}{\partial t} = -\nabla \rho \cdot \mathbf{u} - \rho \nabla \cdot \mathbf{u} \tag{4.27}$$

It shows the link between the divergence of the vector field and the variation of the local density. A corrective pressure field is given as

$$P = K(\rho^n - \rho_0) \tag{4.28}$$

where ρ_0 is the rest (initial) density and where the constant K is based on the gas-state equation. And since we only care about P's derivative, the corresponding correction is then given by

$$\nabla P = K \nabla \rho \tag{4.29}$$

4.2. Numerical Approach. The paper solves the equations and runs the simulation by following steps, we will skip boundary conditions because its not our focus:

1. Solve the mass conservation equation for density by computing the differential operators with central finite differences and integrating the solution with the forward Euler method.

2. Solve the momentum conservation equation for velocity in two conceptual steps: (a) Solve the transport equation using the semi-Lagrangian scheme. (b) Solve *the rest of the momentum conservation equation using the same framework as in Step 1.*

Conservation of Mass:

This is solved by using central finite difference method, and performing a forward Euler integration technique in the paper.

$$\rho_{i,j,k}^{n+1} = \rho_{i,j,k}^{n} + \Delta t(-\mathbf{u}_{i,j,k}^{n} \cdot \nabla \rho_{i,j,k}^{n} - \rho_{i,j,k}^{n} \nabla \cdot \mathbf{u}_{i,j,k}^{n}) \tag{4.30}$$

Conservation of momentum:

The velocity advection is solved exactly the same it is in the previous chapter. The rest is solved using the same idea for conversation of mass.

$$\mathbf{u}_{i,j,k}^{n+1} = \mathbf{u}_{i,j,k}^{n} + \Delta t(-S\nabla \rho_{i,j,k}^{n} + \mathbf{g} + \nu \nabla^2 \mathbf{u}_{i,j,k}^{n}) \tag{4.31}$$

And the following coefficient is given by the author in the paper. $\nu := \frac{\mu}{\rho_0}$ and $S := K\frac{(\Delta x)^2}{\Delta t \rho_0}$. S is an experimental scale value that generates better results in 2 dimension. The stability condition is given

$$\Delta t < max\{|\frac{\Delta x}{u}|, |\frac{\Delta y}{v}|, |\frac{\Delta z}{w}|\} \tag{4.32}$$

Named as Courant-Friedrichs-Lewy (CFL) condition. And this condition must be satisfied everywhere in the domain.

CHAPTER 5

Proposed Method

1. Stablize Simple and Fast Fluid GPU Solver

1.1. Drawbacks of using explicit integration. The purpose of this chapter is to stabilize the GPU solver given in [1]. While a simple solution is represented in [1], and can be simulated by only one pass on GPU, the algorithm suffers from numerical instability because of using explicit Euler integration. In order to integrate such a solver into games might be troublesome because it is only conditionally stable. Recall the constraint in [1], a stable time step has to be based on grid size. This means that in order to create higher resolution results, the smaller time steps are needed to achieve stability. So for example, if we are creating a game with $512 * 512$ resolution and we want to put smoke in certain regions, regardless of how big is the simulation domain, the solver's time step has to be below $\frac{1}{512}$ otherwise it becomes unstable. This instability can also lead to graphical artifacts in the simulation domain [figure 1]. Maintaining such high frame rate is impractical for games. A single frame rate drop could fail the simulation. Also, a game has to fulfill the constraint in different resolutions it supports, which means the time step might vary. One way to solve this problem is to relax such condition by reducing resolution (increasing grid size in other words) in simulation domain, but doing so could potentially introduce inconsistency throughout the scene, for example, the smoke might look blurry compared to the character who interacts with it. A more sophisticated way is to fix the simulation time step in the game loop. This is a very famous and robust technique used constantly in game physics. However, using this method may harm the performance under some situations where we keep the same resolution as our previous example, and the simulation needs to catch up due to a sudden frame rate drop or merely having high resolution (recall that we just mentioned that normally games do not have such a high frame rate). In order to consume accumulated time, it might need to run several integrations to sync with global game state.

The other issue worth mentioning is the unnatural result from the numerical solution of the continuity equation using explicit integration. Equation 4.30 can lead to negative density no matter how small time step is given. That is why the author in [1] has to clamp the final value in the implementation to prevent it from going to negative values. Explicit Euler integration also accumulate error over time, so in practice, we will have to clamp the value from being too big.

1.2. Implicit Solver to increase stability. As mentioned earlier, we observed that equation 4.30 may lead to negative density value, which is why the implementation in simple and fast fluid involves clamping after solving the continuity equation. So our goal is to stabilize [1] using more sophisticated solver and we also want to maintain the simplicity as we rewrite the equations in different forms. Our algorithm successfully removes the clamp function solving continuity equation and adds more detail to the fluid compared to [2]. First of all, we follow Stam's result in [5] to calculate the viscous diffusion acceleration using Jacobi iteration for 40-80 times. And then, instead of using forward Euler integration to solve the continuity equation, we present a method that uses the splitting technique in [5] to cut the equation into two parts,

$$\frac{\partial \rho}{\partial t} = -\mathbf{u} \cdot \nabla \rho \tag{5.1}$$

$$\frac{\partial \rho}{\partial t} = -\rho \nabla \cdot \mathbf{u} \tag{5.2}$$

1.3. Solving continuity equation using Semi-Lagrangian and Implicit Euler Integration. Equation 5.1 shows the same form, as equation 4.8, so we can perform an unconditionally stable Semi-Lagrangian scheme on it.

$$\rho^{temp} = \rho(\mathbf{x} - \mathbf{u}\delta t) \tag{5.3}$$

Numerical dissipation caused by Semi-Lagrangian can be resolved using higher order advection algorithms, such as Back and Forth Error Compensation and Correction (BFECC) [16]. In our approach, we use this intermediate velocity as input of equation 5.2 by following a splitting technique. Equation 5.2 can be solved by using implicit Euler scheme. The equation can be derived as:

$$\frac{\rho^{n+1} - \rho^{temp}}{\delta t} = -\rho^{n+1} \nabla \cdot \mathbf{u} \tag{5.4}$$

$$\rho^{n+1} = \rho^{temp} - \rho^{n+1} \nabla \cdot \mathbf{u} \delta t \tag{5.5}$$

which gives,

$$\rho^{n+1} = \frac{\rho^{temp}}{1 + (\nabla \cdot \mathbf{u}) \delta t} \tag{5.6}$$

This solution allows us to completely remove clamping.

1.4. Solving viscous diffusion equation using Jacobi iteration. After solving the continuity equation, in order to simulate viscous fluid, we have to solve for the viscosity term in order to get accurate results. We followed the results in [5] by solving the equation in implicit form and using Jacobi iteration (equation 4.26). Doing this will introduce more computational time, and according to [9] that normally requires 40-80 iterations to get the result to converge.

1.5. Summary of Algorithm. So far in the previous sections in this chapter, we again broke down N-S equations into parts and solved them individually. This section, we are going to summarize the algorithm and show a step-by-step pseudo code. We will then do an analysis of our method compare to the original method in [1] in the experimental section.

1.5.1. *Algorithm Overview.* During initialization, it is important to set up a rest density value for the type of fluid we want to simulate, then we can proceed to the simulation loop. First of all, we calculate the new density value using semi-Lagrangian followed by the implicit Euler method.

$$\rho^{temp} = \rho(\mathbf{x} - \mathbf{u}\delta t) \tag{5.7}$$

$$\rho^{n+1} = \frac{\rho^{temp}}{1 + (\nabla \cdot \mathbf{u})\delta t} \tag{5.8}$$

Second, given the new value of density, we use it in the equation below to solve for the current velocity within the grid.

$$\mathbf{u}^{n+1}_{i,j,k} = \mathbf{u}^{n}_{i,j,k} + \Delta t(-S\nabla\rho^{n}_{i,j,k} + \mathbf{g}) \tag{5.9}$$

Next, we solve for the viscosity forces contribution using Jacobi iterations:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u} \tag{5.10}$$

For smoke simulation, we need to track an additional temperature field T to apply Buoyancy defined as

$$f_{buoyancy} = (-kd + \sigma(T - T_0))\hat{j} \tag{5.11}$$

Where T_0 is a given ambient temperature, and d stands for the smoke density advected by the velocity field. k and σ are constants scale factors. In order to simulate obstacles, we use Neumann boundary conditions:

$$\frac{\partial f}{\partial n} = 0 \tag{5.12}$$

To keep fluid from entering the obstacle cell. Lastly, we need to visualize the fluid. In order to do so, we simply create another density field tracking moving fluids. Since we already solved the velocity field, we update fluids using semi-Lagrangian algorithm and write to the final texture and draw it.

Algorithm 1 Implicit GPU Solver

```
 1: Update loop
 2: for all grid cell do
 3:     Initialize simulation domain
 4: for all gird cell do
 5:     Solve continuity equation
 6: if is viscous fluid then
 7:     for all $i$ such that $0 \leq i \leq 40 \sim 80$ do
 8:         for all gird cell do
 9:             Solve diffusion term if you are simulating viscous fluids
10: for all gird cell do
11:     Solve momentum equation using corrective pressure, viscous forces and
    external forces
12: if is smoke then
13:     for all gird cell do
14:         Solve temperature and add buoyancy
15: for all grid cell do
16:     Update density field and output as a texture for visualization
```

1.5.2. *The Proposed Method Pseudo Algorithm.*

1.6. Experimental Results. We tested our method using Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz 4.01GHz with Geforce GTX980 Ti. We found that using this method to solve the continuity equation we ended up having similar result as [5]. The simulation has to follow CFL condition (equation 4.32) as it is mentioned in [1]. It is only stable under the condition using correct time step constraint. The following table is a quick comparison of our proposed method and the other two.

Table 1. Algorithm comparison

Method	GPU Memory Usage	**GPU Passes**
Stable Fluid	8*floating point	4
Simple and Fast Fluid	6*floating point	1
Proposed Method	6*floating point	2

By a visual comparison, we can see how Jacobi iteration smooths out the result, but it doesn't loose much details, Figure 2.

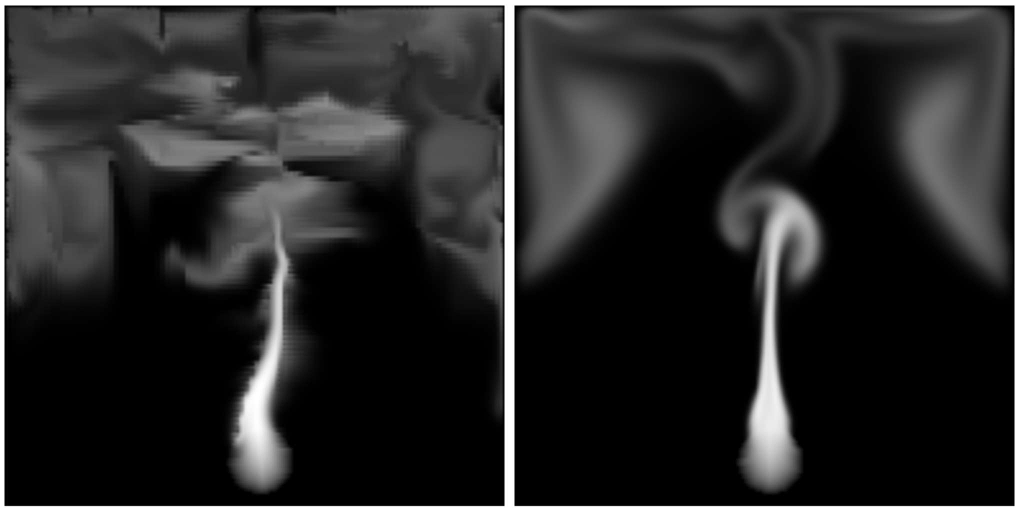

Figure 2. Jacobian smooth out the result

Figure 3 shows dye in water simulation with constant gravity pulling the water down using our proposed method compare to simple and fast fluid, and Stable fluid by capturing the same frame in our framework. Our proposed method provides more details than Stable fluid [2] and the fluids stay more active than simple and fast fluid [1]. But our method is only stable under CFL condition, if it fails to meet the constraint it will dissipate and become unstable. Figure 4 shows average frame time comparison simulating viscous fluids as you can see our method runs two times faster than stable fluid at the same time capturing more details of fluids. Figure 5 shows that the proposed method also allocates less video memory than Stable fluid. Furthermore, in order to compare how much of the high frequency details each algorithm can preserve, we used Fourier analysis. First, we apply Fourier transform on the image we captured [Figure 8], to generate a frequency spectrum image. Secondly, apply a high pass filter on the frequency spectrum image to filter out the low frequency components. Finally, apply inverse Fourier transform [Figure 8] and compare the results [Figure 10]. We can see from the remaining white color, that both our algorithm and simple and fast fluid preserve high frequency components better than stable fluid. Table 2 shows how much details get preserved through Fourier Analysis with approximately 80 pixels filter radius. The measurement takes places by directly analyzing the result images' red, green and blue channel.

Table 2. Fourier Analysis measured in how much black spaces left out before and after applying high frequency filter to calculate preserved rates.

Method	Before	After	Preserved Rate(%)
Stable Fluid	0.43	0.85	∼25
Simple and Fast Fluid	0.29	0.61	∼55
Proposed Method	0.34	0.67	∼50

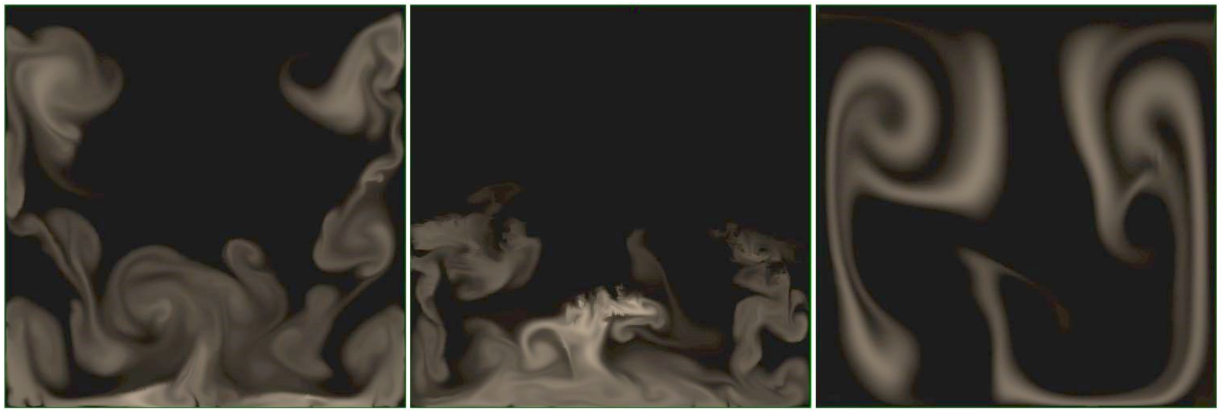

Figure 3. Starting from left is proposed method, simple and fast fluid, and stable fluid simulating dye in water using a 1/512 time step in a 512*512 simulation domain

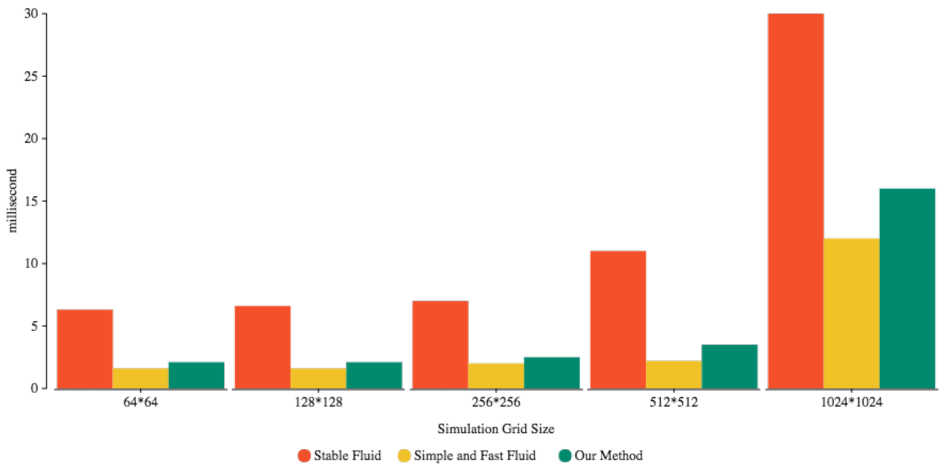

Figure 4. Average frame rate in multiple resolution

1.7. Error Analysis. First we perform error analysis for semi-Lagrangian. Consider the continuity equation,

$$\frac{\partial \rho}{\partial t} = -\nabla \rho \cdot \mathbf{u} - \rho \nabla \cdot \mathbf{u} \tag{5.13}$$

The proposed semi-Lagrangian scheme in equation 5.3, according to [25], obeys the following error estimate:

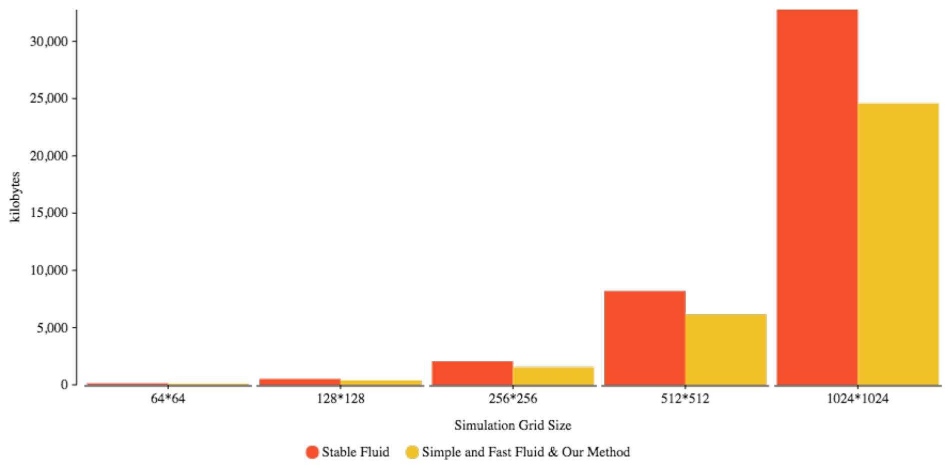

Figure 5. VRAM footage under different resolution measured based on table 1

$$|\rho^{temp} - \rho^{true}| \leq C_1(\delta t + \frac{\Delta x^2}{u^2 \delta t}) \tag{5.14}$$

if $\frac{\Delta x}{u} \leq C_2(\delta t)$ then we get:

$$|\rho^{temp} - \rho^{true}| \leq C\delta t \tag{5.15}$$

Secondly, we take a look at proposed Implicit Euler 5.6, well known results in numerical differential equation prove that local truncation error for Implicit Euler (backward) satisfy

$$|\rho_{n+1} - \rho^{true}| \leq C\delta t \tag{5.16}$$

In conclusion, if $\frac{\Delta x}{u}$ is comparable with δt, then the local truncation error converges to zero for this step, and the proposed algorithm provides a good approximation for the solution of the continuity equation.

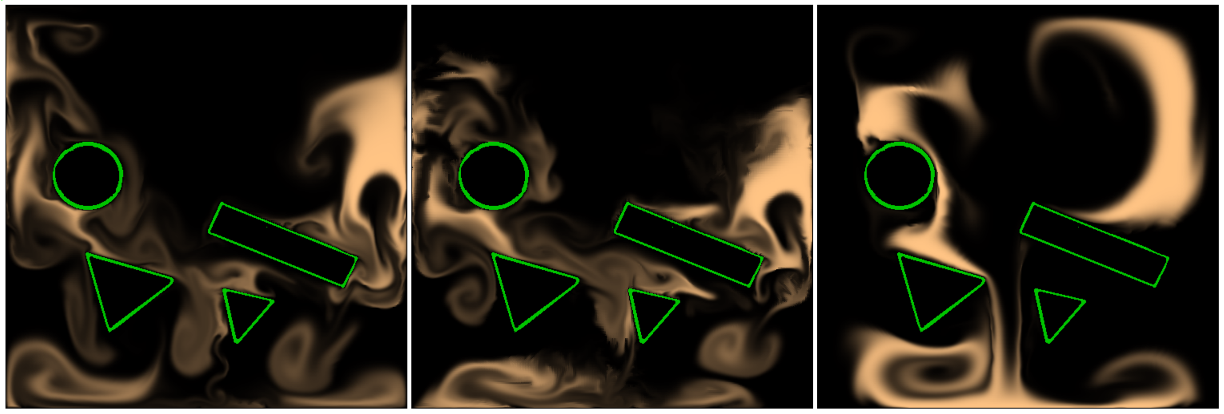
Figure 6. Starting from left is proposed method, simple and fast fluid, and stable fluid simulating dye in water using a 1/512 time step in a 512*512 simulation domain, testing against obstacles in water

1.8. Limitations and Future work. Our approach successfully smooth out the final image and achieve similar result as [5] by using less computational time and a more straightforward solver compared to [5]. We also reduced memory footage for the simulation [figure 5]. However, numerical instability caused by temporary relaxing incompressible condition cannot be resolved by simply using implicit integration. The problem comes down to integrate such solver into games as we mentioned previously. Under high resolution such as 512*512 in 2D, our method is restricted to proper time step. However, our best result shows that under a 200*200 grid, we can run the simulation in 60fps. In order to integrate our method properly into a game engine, we need to apply a proper and fixed time step integration to keep our solver stable. To do so in high resolution grid like 256*256, developers might have to accumulate delta time from each game update loop, and if the accumulated time is bigger than the fixed-time step defined for the simulation, we update the fluid one time or several times by subtracting the simulation time step until accumulated time is smaller than

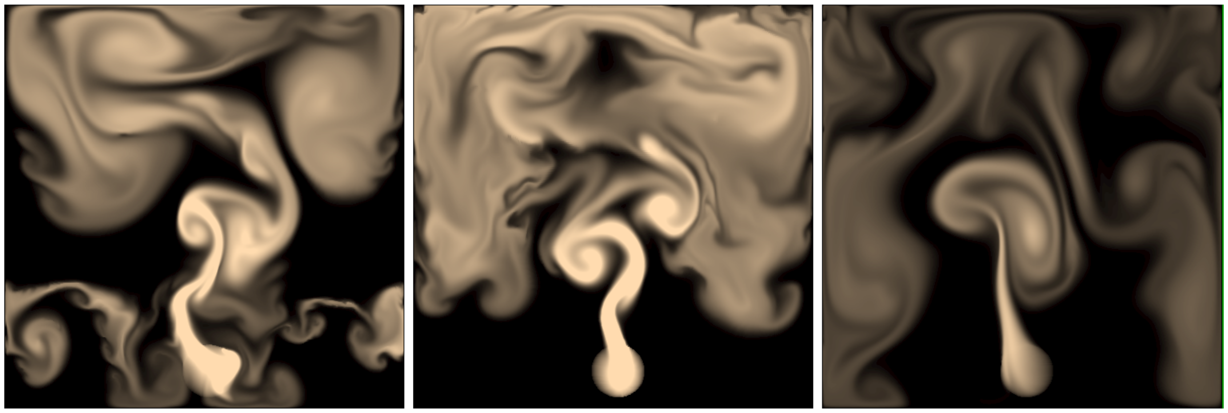

Figure 7. Starting from left is proposed method, simple and fast fluid, and stable fluid simulating smoke using a 1/512 time step in a 512*512 simulation domain

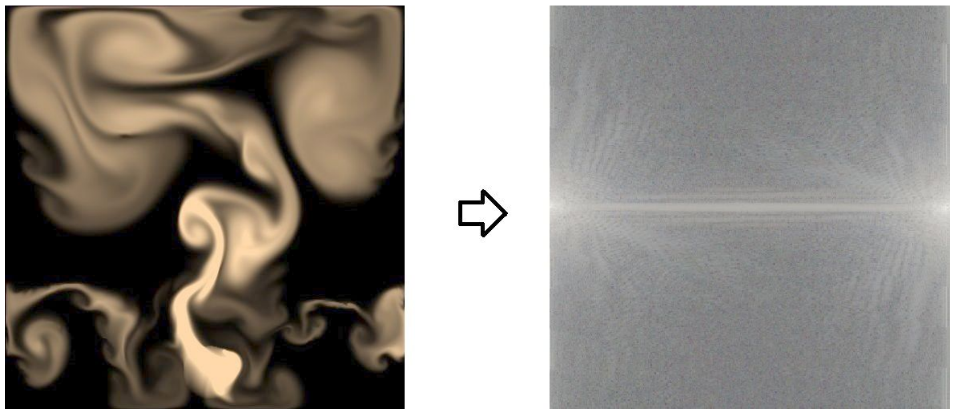

Figure 8. Result after applying Fourier transform

the simulation time step. This might cause issue to maintain the frame rate for a game, for example, here we keep the same environment 512*512 in 2D, if the rest of the game state has advanced 1/60 millisecond, the simulation has to catch up the game state by integrating the solver about 8-9 times. This might hurt the performance as we saw in the above table describing how much time each update can take in average. In contrast, Stam's stable fluid can be stable under any time step. For future work, we would like to see if we can find a better numerical method that helps us solve N-S

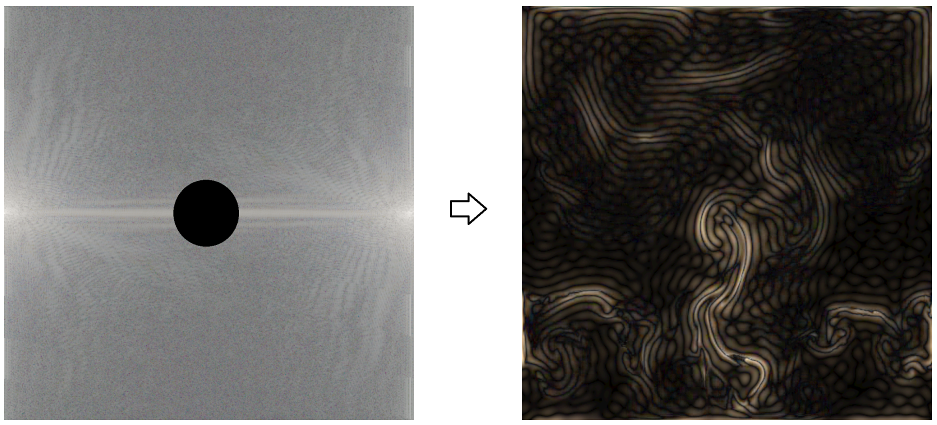

Figure 9. Applying a high pass filter and do a inverse Fourier transform

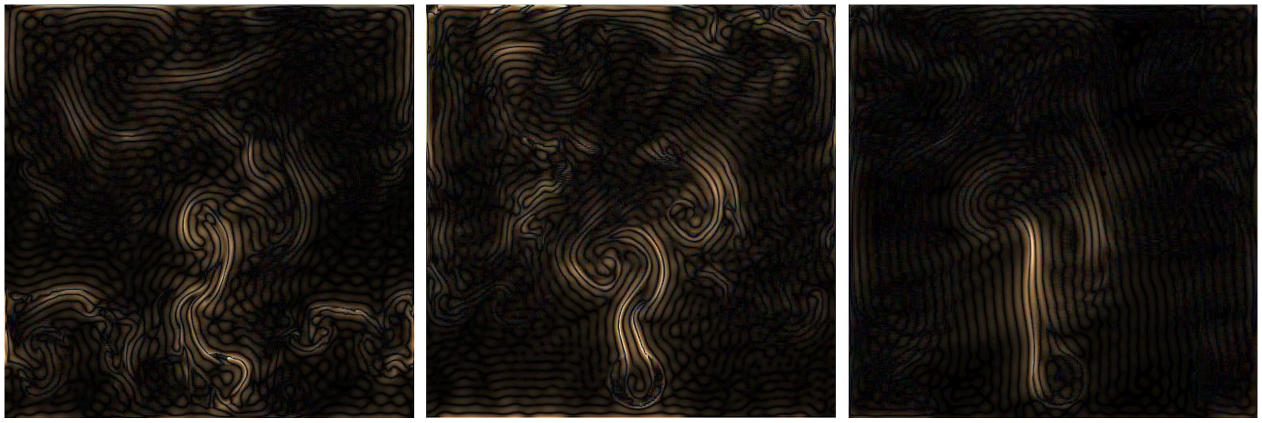

Figure 10. Result from figure 7

equations directly in a better way.

1.9. Summary. Although we did not succeed in making the solver unconditionally stable. Our method does not need clamping function under correct time step. By observation, our final result is similar to [5] but requires less computational time by fusing the advantages in [5] and [1]. We are also able to smooth the result in [1]. Our method successfully sustain stability by solving continuity equation using Semi-Lagrangian and implicit Euler integration although this step requires one extra pass when compared to [1] because we need to wait until all the density get solved then use the result to get correct pressure force by using central finite difference. For viscous liquids, we apply Jacobi iteration on implicit diffuse term just like what it is done in [9]. The result [figure 4] shows that we are desired to sacrifice some computational time to trade off stability and smoothness for the final rendered image.

CHAPTER 6

Conclusions and Future Research

Several numerical methods of simulating fluids have been presented in this paper. Mainly we focus on physically based fluid simulations in Eulerian point of view. Our work starts from Stable Fluids [5], and Simple and Fast Fluid [1] and compared our approach with above discussed methods. Experimental results show that our method preserves high frequency details at the same time removes unnatural clamping when solving the continuity equation. Our method shares the same amount of memory footprint as simple and fast fluid, which is less than required by the method in Stable fluid, and it is more efficient compared to Stable fluid. We also showed how important the splitting technique is when solving complicated partial differential equations. However, this is not the end of the line, there are other directions of fluid simulation that should be studied. For example, level set method [15] is good at tracking water surface while vortex particles [24] are used to simulate turbulence in fluids. Furthermore, in our simulation, we only talked about water and smoke but there's also fire and viscous fluid types. In order to simulate those types of fluids, [4] gives a nice overview of how each type of them works.

For future work, we would like to make the simulation more stable under arbitrary time steps. Currently if the time step fails to meet the stability condition, we get the final result with some obvious artifact. We would also like to explore betters numerical methods to help solve numerically Navier-Stokes equations in the future.

REFERENCES

[1] M. Guay, F. Colin, and R. Egli, *Simple and Fast Fluid*, GPU Pro 2, 2011.

[2] J. Stam, *Real-Time Fluid Dynamics for Games*, GDC, 2003.

[3] R. Courant, E. Isaacson, and M. Rees, *On the solution of nonlinear hyperbolic differential equations by finite differences,* Communications on Pure and Applied Mathematics, 5:243255, 1952.

[4] R. Bridson, *Fluid Simulation for Computer Graphics*, 2008

[5] J. Stam, *Stable Fluids*, In Proceedings of SIGGRAPH, 1999.

[6] M. Lentine, J. Gretarsson, and R. Fedkiw, *An Unconditionally Stable Fully Conservative Semi-Lagrangian Method*, 2010

[7] Chorin, A.J., and J.E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, 1993, 3rd ed. Springer.

[8] Golub, G.H., and C.F. Van Loan, *Matrix Computations*, 1996, 3rd ed. The Johns Hopkins University Press.

[9] Mark J. Harris, *Fast Fluid Dynamics Simulation on the GPU*, GPU GEMS VOLUME 1

[10] M. Kelager, *Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics*, 2006

[11] Kim, Theodore, et al., *Wavelet turbulence for fluid simulation*, ACM Transactions on Graphics (TOG). Vol.27. No.3. ACM, 2008

[12] J. Tessendorf, *Simulating Ocean Water*, SIGGRAPH, 2001

[13] A. Gorguis, *A reliable approach to the solution of Navier-Stokes equations*, Appl. Math. Lett. 25 (2012) 2015-2017

[14] TAREK M. A. EL-MISTIKAWY, *Note on "A. Gorguis, A reliable approach to the solution of Navier-Stokes equations, Appl. Math. Lett. 25 (2012) 2015-2017"* , Department of Engineering Mathematics and Physics, Cairo University, Giza 12211, Egypt

[15] SONG, O.-Y., SHIN, H., and Ko, H.-S. Stable but nondissipative water. *ACM Trans Graph* 24, 1 (2005), 81-97.

[16] Kim, B., Liu, Y., Llamas, I., Rossignac, J., Flowfixer: Using BFECC for fluid simulation, *Proceedings of the First Eurographics Conference on Natural Phenomena NPH05*, pp. 5156. Eurographics Association, Aire-la-Ville (2005)

[17] Braley, C., Sandu, A., and Tech, V., Fluid Simulation for Computer Graphics: a Tutorial in Grid Based and Particle Based Methods, 2009

[18] Mller, M., Charypar, D., and Gross, M. (2003). Particle-Based Fluid Simulation for Interactive Applications. *Proceedings Of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation,* 154159.

[18] Mark F., Cyan W., *Effective Water Simulation from Physical Models*, GPU GEMS VOLUME 1

[19] Kim, Theodore and Thürey, Nils and James, Doug and Gross, Markus, *Wavelet Turbulence for Fluid Simulation*, ACM SIGGRAPH 2008 Papers, 2008

[20] Chen, Jim X. and Lobo, Niels da Vitoria and Hughes, Charles E. and Moshell, J. Michael, *Real-Time Fluid Simulation in a Dynamic Virtual Environment*, IEEE Comput. Graph. Appl., May 1997

[21] Ronald Fedkiw and Jos Stam and Henrik Wann Jensen, *Visual simulation of smoke*, SIGGRAPH, 2001

[22] Clavet, Simon and Beaudoin, Philippe and Poulin, Pierre, *Particle-based Viscoelastic Fluid Simulation*, Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2005

[23] Takamatsu, Kenji; Kanai, Takashi, *A Fast and Practical Method for*

Animating Particle-Based Viscoelastic Fluids, International Journal of Virtual Reality . 2011, Vol. 10 Issue 1, p29-35.

[24] Selle, Andrew and Rasmussen, Nick and Fedkiw, Ronald, *A vortex particle method for smoke, water and explosions*, ACM Trans. Graph., SIGGRAPH '05, 2005, p910-914.

[25] Maurizio Falcone and Roberto Ferretti, *Semi-Lagrangian Approximation Schemes for Linear and HamiltonJacobi Equations*, SIAM, 2013