Interactive Simulation of Fluid Flow

Michael Fortin

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2011

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: Michael Fortin
Entitled: Interactive Simulation of Fluid Flow

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Hovhannes Harutyunyan

_____ Examiner
Dr. Eusebius J. Doedel

_____ Examiner
Dr. Adam Krzyzak

_____ Supervisor
Dr. Peter Grogono

_____ Co-supervisor
Dr. Sha Xin Wei

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

Abstract

Interactive Simulation of Fluid Flow

Michael Fortin

The simulation of fluid flow on rectangular grids using a discretized version of the Navier Stokes equations for incompressible fluid flow can be simultaneously described as an aesthetically pleasing and computationally intensive embarrassingly parallel problem.

Ideally, the aesthetics of the fluid simulation should, given some set of parameters, feel natural despite the synthetic nature of the underlying grids. This natural feel, paramount to the success of the system, should fool a person into believing that they are interacting with a real fluid.

The number of calculations and data accesses increases with the number of cells present in the rectangular grid upon which the fluid is simulated. An increased number of calculations are required for augmented accuracy, different external forces, and additional dimensions. Since it is a trivial task to increase the complexity of the simulation, interactivity becomes a challenge of balancing accuracy, stability, and detail against speed of execution.

A simple solution is to throw more processing power through increased instruction execution speeds or additional cores. Throwing additional cores at the problem strains the memory bus making it the point that slows down the simulation. Therefore for a given algorithm, respecting data locality and processor peculiarities can be used to minimize execution times.

This document introduces a means of caching corrected velocity fields, a task scheduler that attempts to maximize the usage of the cache on multi-core processors, and a naïve compression algorithm based on run-length encoding.

Acknowledgments

There are many people that we would like to thank for all their help with bringing this project to completion (in alphabetical order):

Sean Braithwaite Listened to an initial presentation on the equations of fluid flow. His confusion in my explanations lead to some rewording in the first chapter.

Josée-Anne Drolet Helped film the video for the ACM Multimedia 2009 submission.

Adrian Freed Sat down with me and threw a bunch of ideas on how to get different visual effects out of the simulation.

Peter Grogono Always giving excellent feedback on my usually sub-optimal writing.

Dominic Lam Ensured that I could attend the I.B.M. Cell workshop.

Michael Perrone He spent time with me looking over the project and helped with his emphasis to always run through the numbers before actually implementing something.

Harry Smoak For all the times I asked him what made more sense when I was searching for a way to explain a concept.

Sha Xin Wei For providing the challenging project, helping me plough through all of the mathematics, and giving comments on an early draft of this document.

I should not forget all the help from the folk who passed through the Topological Media Lab during my stay there.

The list could not be complete without mentioning my parents who blindly encouraged me during the realization of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The real-time simulation of fluid flow can augment interactive media through the addition of intuitive complexity[SFR09] and the aesthetics of fluid flow.

We present a fluid simulation that attempts to optimally use currently available parallel processing hardware with the goal of maximizing the visual complexity of the presented simulated fluid.

First: in chapter 2, we show how to obtain a discrete mathematical model describing fluid flow. Despite our focus on grid-based representations, we also provide details and references to alternative representations.

Second: we describe a means of obtaining a greater range of behaviours within our simulated flow by adding additional terms to the equations presented in the second chapter. These terms allow for effects that depend upon external factors, such as sources of temperature for smoke and fire.

Third: we explore different ways of obtaining visual representations of our synthetic flow. We look at means for warping reality based on live camera feeds, playing with the underlying temperature model, and using gradients to remap colours for added visual flair.

Fourth: we discuss machine-specific details that guided the final implementation. The explored hardware includes Intel's Core 2 Duo CPU, I.B.M.'s Cell Broadband Engine, and nVidia's geForce 8800M GTS GPU. For each piece of hardware, we explore issues regarding data and instruction throughput.

Fifth: we introduce our task scheduler that simplifies the writing of parallel code for successive image processing operations by providing a layer of abstraction above system threads.

Sixth: we present a compression algorithm that allows for rapid compression of image data based on Fast and Efficient Lossless Image Compression System (FELICS)[HV93] and partly inspired by Portable Network Graphics (PNG)[ABB+03].

Seventh: we describe our fluid simulation as a combination of the previously described parts.

Complementing the main text are three appendices. The first appendix provides a more thorough look at mathematical concepts. The second lists the symbols used within the physics equations and describes some physical properties that could be useful. The third appendix focuses on clarifying some used terminology.

Chapter 2

Fluid Flow

Consider a sheet of fluid confined to the plane, similar to figure Figure 2.1, at any time $t \in \mathbb{R}^+$ with set P containing all particles. Each particle $p \in P$ in the fluid can be described by a position $\boldsymbol{x}_p \in \mathbb{R}^2$ and velocity $\boldsymbol{u}_p \in \mathbb{R}^2$. Let $\boldsymbol{i} \in \mathbb{I}^2$ be a position, define $\boldsymbol{M_i} \in \mathbb{R}^2$ as an average of all $\boldsymbol{u}_l$, $l \in P$ bounded by $[\boldsymbol{i}^x, \boldsymbol{i}^y]\times]\boldsymbol{i}^x + 1, \boldsymbol{i}^y + 1[$. We will let $\boldsymbol{u}_d \in \mathbb{R}^2$ be a discrete grid of $\boldsymbol{M_i}$ for all $\boldsymbol{i}$ bounded by $[0, 0] \times [w, h]$ where $w, h \in \mathbb{I}^+$. Specifically, $\boldsymbol{u}_d$ is a discrete approximation of the velocities of the particles within P.

$\boldsymbol{u}_d$ describes the fluid flow over a discrete grid. In this chapter we show the equations that describe the evolution of $\boldsymbol{u}_d$ mimicking a fluid as t increases.

First, we describe a system of equations that models the evolution of $\boldsymbol{u}_d$. In addition, we attempt to impart the intuition as to why the equations model fluid flow.

Second, we provide a discretized form of the equations that could potentially be run on computational hardware.

Third, we discuss modelling the equations using particles in the place of rectangular grids.

2.1 The Equations Describing Fluid Flow

The mathematical model behind our fluid simulation yields fluid-like behaviour through the following four properties:

Advection describes how $\boldsymbol{u}_d$ evolves with respect to time reflecting the change of $\boldsymbol{x}_i$ for all particles $i \in P$ as time t increases. The fluid would be stationary without this attribute, which sounds quite boring compared to the motion of a perturbed liquid.

Viscosity is a type of drag on the fluid. Intuitively a sort of friction among particles. Rotational behaviour emerges from viscosity when faster streams of fluid meet with slower streams.

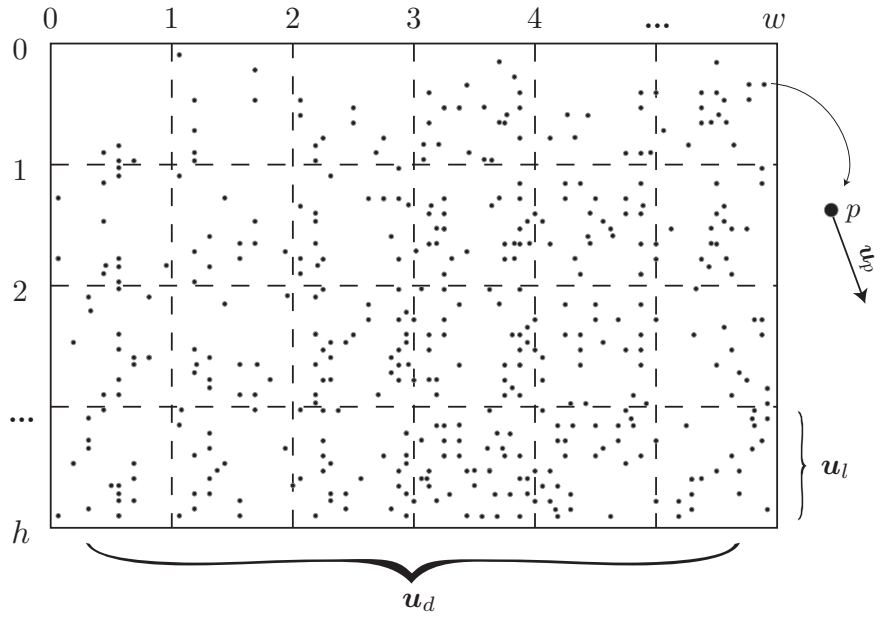

Figure 2.1: Macroscopic View of Variables
A grid showing what could be all particles $p \in P$. Each particle has a position $\boldsymbol{x}$ and a velocity $\boldsymbol{u}$. We are interested in the approximation $\boldsymbol{u}_d$ that is a grid consisting of multiple cells averaging the motion of the contained particles as a single vector.

Pressure is a force that builds up against the flow of fluid matter accumulating in one location. Pressure prevents the case where the fluid collapses into one location.

Incompressibility adds the restriction that the flow can not be compressed nor expanded, simplifying the resulting equations.

We will now explore in more detail the four properties. Our goal is to obtain the Navier Stokes Equations for incompressible flow.

2.1.1 Advection

Advection is "the transfer of heat or matter by the flow of a fluid..."[App07]. In figure 2.2, we assume a static velocity field and show a single particle with an inscribed timestep flowing along the field. Imagine colour flowing along the velocity field. Like the swirls that appear when white milk is poured into black coffee just before the colours blend together into a consistent light brown.

The underlying velocity field models the overall evolution of the underlying particles that make up the fluid. Consider a cup of water with large dark floating round objects within. If we were to stir the water, the particles would move. We could also say that the objects

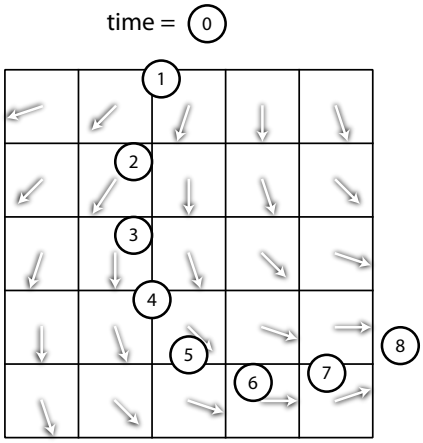

Figure 2.2: Advection
Given a static velocity field, the circle is advected along the vector field following the arrows that dictate direction and magnitude.

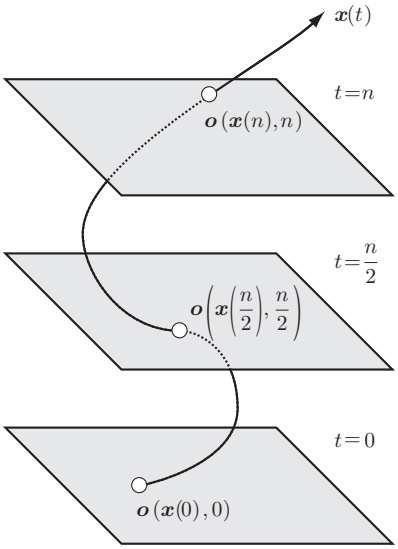

Figure 2.3: Material Derivative
$\boldsymbol{o}$ is an attribute of particle $l \in P$ positioned at $\boldsymbol{x}$. Within the field, at location $\boldsymbol{x}$ and time t, the value equals that of $\boldsymbol{o}$.

within the water approximate the flow of the fluid particles. Similarly, we can approximate the flow as vectors within a grid.

Let $\boldsymbol{u} \in \mathbb{R}^2$ be a 2D vector field identical to $\boldsymbol{u}_d$ except with infinite spatial resolution.

Then, we say that the advection term models the evolution of $\boldsymbol{u}$ to reflect how all the represented particles $\boldsymbol{x}_l$, $\forall\, l \in P$ would have moved.

Following Chorin and Marsden[CM03], we obtain the advection equation from the relations between acceleration, velocity, position, and time. We provide Figure 2.3 to clarify the relations among the variables.

Let $\boldsymbol{x}_l(t) = (x_l(t), y_l(t))$ be the position of any particle $l \in P$.

Let $\boldsymbol{o}(\boldsymbol{x}(t), t) \in \mathbb{R}^d, d \in \mathbb{I}, d > 0$ for all particles in P be a field of fluid attributes, such as velocity or density. We assume that the values at time t are known.

We wish to compute $\boldsymbol{o}$ at time $t + \Delta t$ without any knowledge of the represented particles in P.

Let $\boldsymbol{u}_l$ be the velocity of the particle l. $\boldsymbol{u}_l$ is defined as change in position:

$$\frac{d\boldsymbol{x}_l}{dt} = \left(\frac{dx_l}{dt}, \frac{dy_l}{dt}\right) \tag{2.1}$$

$$= \boldsymbol{u}_l \tag{2.2}$$

We can compute the change in $\boldsymbol{o}$ with respect to time using the chain rule as follows:

$$\begin{aligned}
\frac{D\boldsymbol{o}}{Dt} &= \frac{d\boldsymbol{o}(x(t), y(t), t)}{dt} && (2.3)\\
&= \frac{\partial \boldsymbol{o}}{\partial x}\frac{dx}{dt} + \frac{\partial \boldsymbol{o}}{\partial y}\frac{dy}{dt} + \frac{\partial \boldsymbol{o}}{\partial t}\frac{dt}{dt} && (2.4)\\
&= \left(\frac{dx}{dt}, \frac{dy}{dt}\right) \cdot \left(\frac{\partial \boldsymbol{o}}{\partial x}, \frac{\partial \boldsymbol{o}}{\partial y}\right) + \frac{\partial \boldsymbol{o}}{\partial t} && (2.5)\\
&= (\boldsymbol{u}) \cdot (\nabla \boldsymbol{o}) + \frac{\partial \boldsymbol{o}}{\partial t} && (2.6)\\
&= (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{o} + \frac{\partial \boldsymbol{o}}{\partial t} && (2.7)
\end{aligned}$$

$\frac{D\boldsymbol{o}}{Dt}$ is known as the material derivative. $\frac{\partial \boldsymbol{o}}{\partial t}$ represents the evolution of the field $\boldsymbol{o}$ with respect to time t compared to $\frac{D\boldsymbol{o}}{Dt}$ that describes the change of $\boldsymbol{o}$ based upon the movement of the underlying particles.

$\frac{D\boldsymbol{u}}{Dt}$ is the change in velocity, in other words the acceleration, of the fluid.

Recall that $\boldsymbol{F} = m\boldsymbol{a}$, where in the Navier Stokes Equations, $\boldsymbol{F} = m\frac{D\boldsymbol{u}}{Dt}$ [CM03].

An alternate form of the advection equation is known as the conservative form, written as $-\nabla \cdot \boldsymbol{uu}$[TLP06].

Armed with the knowledge of how we represent the evolution of a field with respect to a velocity field, we now look at how we would model the friction amongst particles.

2.1.2 Viscosity

Viscosity provides friction among particles that allows for rotational behavour to appear within the fluid[CM03]. Conceptually, as shown in figure 2.4, imagine two sheets of fluid flowing at different speeds. If the sheets of fluid affect each other, then they will curve. Like heat applied to a bimetallic strip.

The viscosity term, $\frac{\partial \boldsymbol{u}}{\partial t} = \upsilon \nabla^2 \boldsymbol{u}$, computes a type of friction between particles that inhibits flow. For this reason it is also known as drag[FM97b]. The higher the value of the viscosity constant υ, the greater the friction between the modeled particles.

Another inter-particle issue that we will encounter next is how we handle the case where the particles would compress themselves into any region given we focus on incompressible flow.

2.1.3 Pressure

A ball pit is a region filled with multi-colored plastic balls that people, stereotypically children, play in. Now consider a massive 1km^3 ball pit. The balls at the bottom support the weight of those on top. There is more pressure on the balls at the bottom of the pit.

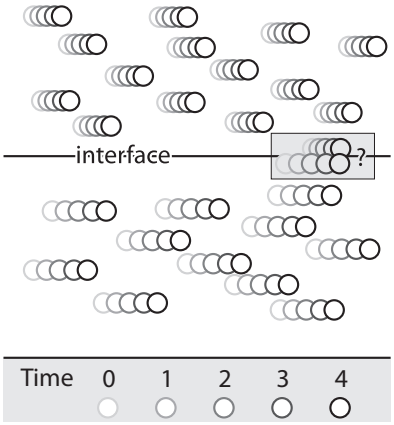

Figure 2.4: Viscosity
Note that where particles of different velocities meet at the interface, their velocity should be affects to yield rotational behavour.

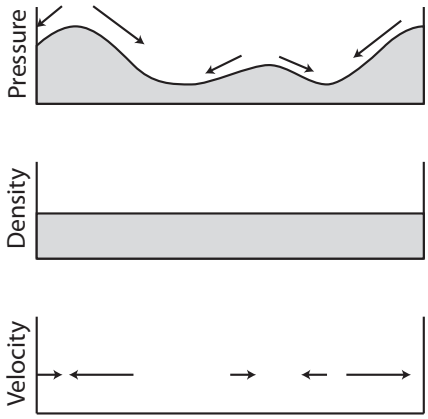

Figure 2.5: Pressure
Consider a flow in one dimension. The density is constant, yet the velocities might push densities together - essentially compress the flow. Pressure builds up from this potential compression, and adds an opposing force.

This pressure found in our imagined ball-pit is very similar to the pressure found in incompressible fluids. Pressure is a measure of the force resisting compression.

Pressure arises when fluid matter (referred to as density) is crushed into a region of the fluid. It is a force that pushes back when the fluid is compressed. Since this document deals with incompressible flow, pressure arises when there is an accumulation of fluid going in one spot. We use the resulting pressure to adjust the velocity field so that density is neither created nor destroyed.

Figure 2.5 shows an incompressible flow confined to a line. The amount of fluid is constant. However the velocity pushes the fluid in ways that would crush the fluid in given regions. Notice that where the density is getting crushed, there's a buildup in pressure. The negative gradient of the pressure is a series of vectors pointing away from high-pressure points. These vectors are the force due to pressure that is added – a force pushing cramped particles away from each other.

Further insight into pressure can be playing with equations. Following Chorin and Marsden's lead[CM03].

Let $\boldsymbol{W}$ be a region within the fluid with border $\partial \boldsymbol{W}$.

Let $\boldsymbol{s}(\boldsymbol{x}, t) \in \mathbb{R}^2$ be the force exerted from pressure at a point $\boldsymbol{x}$ on $\partial \boldsymbol{W}$ and time t.

Let $\boldsymbol{n}$ be a unit normal at point $\boldsymbol{x}$ on $\partial \boldsymbol{W}$.

Let $p(\boldsymbol{x}, t) \in \mathbb{R}$ be the amount of pressure at location $\boldsymbol{x}$ and time t.

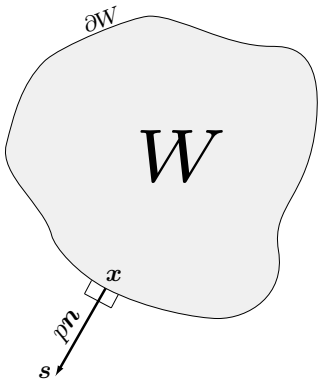

Figure 2.6: Pressure
For a given point $\boldsymbol{x}$ in region W with border ∂W, there is a force $\boldsymbol{s}$ perpendicular to the surface ∂W known as pressure with magnitude p.

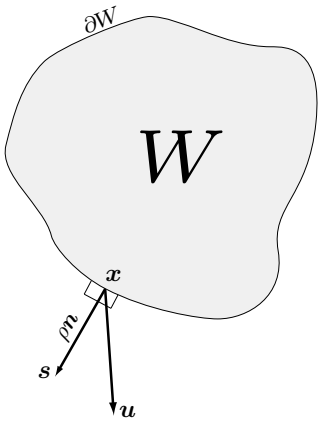

Figure 2.7: Incompressibility Condition
Diagram showing the relation between the variables.

The force due to pressure is normal to the surface for a given region in an ideal fluid[CM03]. Let $\boldsymbol{s}$ describe the force of pressure in an ideal fluid, as such it is a vector normal to ∂W with magnitude equal to the pressure:

$$\boldsymbol{s} \quad = \quad p(\boldsymbol{x}, t)\boldsymbol{n} \tag{2.8}$$

The relationship between the variables can be seen in figure 2.6.

Let $d\boldsymbol{A}$ be a small surface element.

Let $\boldsymbol{s_{\partial W}}$ be the force of pressure exerted upon the region W. Then the force $\boldsymbol{s_{\partial W}}$ for an ideal fluid is defined as the negative sum of all pressure acting on the border of W[CM03]. We use the negative sum since we want the force due to pressure to push into the region $\boldsymbol{W}$:

$$\boldsymbol{s_{\partial W}} \quad = \quad -\int_{\partial \boldsymbol{W}} p\boldsymbol{n}d\boldsymbol{A} \tag{2.9}$$

Let $\boldsymbol{q} \in \mathbb{R}^n$ be a vector field where $n \in \mathbb{I}^+$.

The divergence theorem is a mathematical statement of the fact that the increase or decrease of density within a region (for example W) equals the amount of density flowing through the boundaries[Weib]. The divergence theorem describes the following relation for the surface

integral of $\boldsymbol{q}$ over the boundary of W where $d\boldsymbol{A}$ refers to a small surface element[KC08][1][Weib]:

$$\int_{\boldsymbol{W}} \nabla \cdot \boldsymbol{q} d\boldsymbol{W} = \int_{\partial \boldsymbol{W}} \boldsymbol{q} \cdot \boldsymbol{n} d\boldsymbol{A} \tag{2.10}$$

Let $\boldsymbol{e} \in \mathbb{R}$ be any unit vector. Then, as shown by Chorin and Marsden[CM03], we can relate the change of pressure across the boundary of region $\boldsymbol{W}$ to the change in pressure within the region of $\boldsymbol{W}$:

$$\boldsymbol{e} \cdot \boldsymbol{s}_{\partial \boldsymbol{W}} = -\int_{\partial \boldsymbol{W}} p\boldsymbol{e} \cdot \boldsymbol{n} d\boldsymbol{A} \tag{2.11}$$

$$= -\int_{\boldsymbol{W}} \nabla \cdot (p\boldsymbol{e}) d\boldsymbol{V} \tag{2.12}$$

$$= -\int_{\boldsymbol{W}} \nabla p \cdot \boldsymbol{e} d\boldsymbol{V} \tag{2.13}$$

$$\boldsymbol{s}_{\partial \boldsymbol{W}} = -\int_{\boldsymbol{W}} \nabla p d\boldsymbol{V} \tag{2.14}$$

For a sufficiently small volume element dV, the force due to pressure is approximately $-\nabla p$. This force specifies how pressure affects the fluid. Specifically, that the fluid has a tendency to flow from high-pressure points to low-pressure points. Within the next section we will look at the incompressibility condition. Incompressibility, as we will show during discretization, will help us compute pressure values.

2.1.4 Incompressibility Condition

Constant density combined with conservation of mass results in incompressibility as a side-effect. In figure 2.5, we have a constant density with a velocity field that will potentially result with non-constant amounts of density unless we use the force of pressure as a counter-acting force. In the last section we showed the effect of pressure. Now we give an idea on how we could compute pressure.

Keep in mind that what we call pressure is, what we believe to be, a sufficient approximation given incompressible flow. For compressible flow, more complex methods would be used.

We will now show a mathematical equation that describes the conservation of mass with the aforementioned hint that incompressibility relates to pressure. Later, when we explain the discretization of the equations, we shall elaborate on the computational relationship between conservation of mass and pressure.

Let $m \in \mathbb{R}^+$ be the mass of a region at time t.

[1] Kundu[KC08] lets $dA = \boldsymbol{n} d\boldsymbol{A}$, thus writes the divergence theorem as:
$\int_{\boldsymbol{W}} \nabla \cdot \boldsymbol{q} dW = \int_{\partial \boldsymbol{W}} \boldsymbol{q} \cdot dA$

Let $\rho(\boldsymbol{x}, t) \in \mathbb{R}^+$ be a scalar field describing the density of the fluid, assumed to always be a constant of 1.

Recall that $m = V\rho$ where m is mass, V is a volume, and ρ is density. For our purposes, density is constant for any volume.

Letting Chorin and Marsden lead the way[CM03]; given a region $\boldsymbol{W}$, we can define the change of mass m in $\boldsymbol{W}$ with respect to time t as a function of the change in density in the same region:

$$\frac{dm(\boldsymbol{W}, t)}{dt} = \frac{d}{dt}\int_{\boldsymbol{W}} \rho d\boldsymbol{V} \tag{2.15}$$

The amount of density travelling out of the borders of $\boldsymbol{W}$, or flow rate, is defined as:

$$\text{flow rate} = \int_{\partial \boldsymbol{W}} \rho \boldsymbol{n} \cdot \boldsymbol{u} d\boldsymbol{A} \tag{2.16}$$

Change in density equals the amount of density flowing into $\boldsymbol{W}$, that is the negative of the flow rate:

$$\frac{d}{dt}\int_{\boldsymbol{W}} \rho d\boldsymbol{V} = -\int_{\partial \boldsymbol{W}} \rho \boldsymbol{n} \cdot \boldsymbol{u} d\boldsymbol{A} \tag{2.17}$$

Using the divergence theorem, (2.10):

$$\frac{d}{dt}\int_{\boldsymbol{W}} \rho d\boldsymbol{V} = -\int_{\boldsymbol{W}} \left(\nabla \cdot (\rho \boldsymbol{u})\right) d\boldsymbol{V} \tag{2.18}$$

$$\frac{d}{dt}\int_{\boldsymbol{W}} \rho d\boldsymbol{V} + \int_{\boldsymbol{W}} \left(\nabla \cdot (\rho \boldsymbol{u})\right) d\boldsymbol{V} = 0 \tag{2.19}$$

$$\int_{\boldsymbol{W}} \frac{\partial \rho}{\partial t} d\boldsymbol{V} + \int_{\boldsymbol{W}} \left(\nabla \cdot (\rho \boldsymbol{u})\right) d\boldsymbol{V} = 0 \tag{2.20}$$

$$\int_{\boldsymbol{W}} \left(\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u})\right) d\boldsymbol{V} = 0 \tag{2.21}$$

For an infinitely small region:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{2.22}$$

Since we assume a constant density of 1, we obtain the incompressibility condition:

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.23}$$

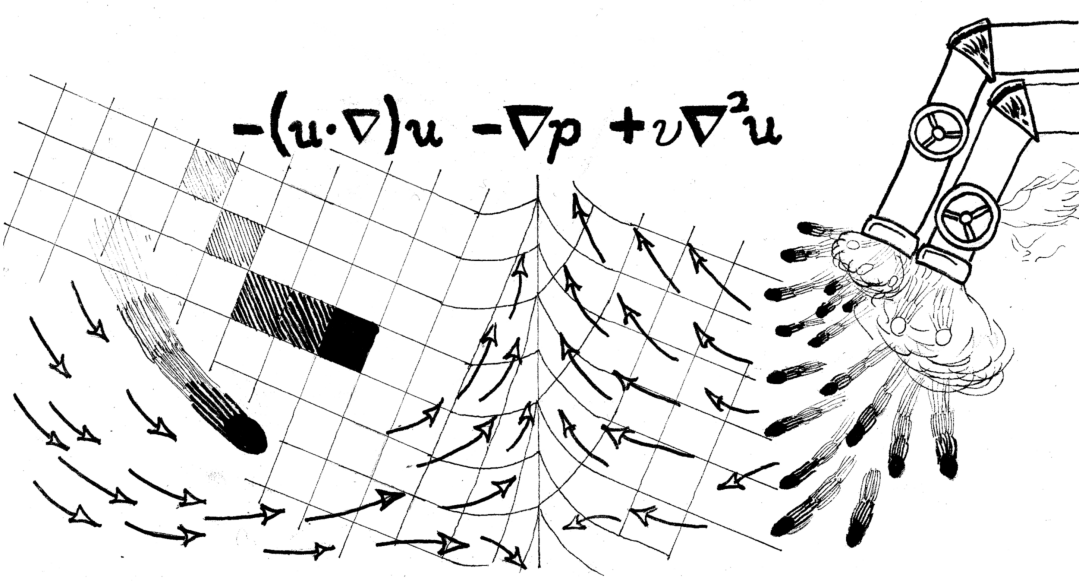

Figure 2.8: Image of Terms of Navier Stokes Equations
Image inspired by the equations. To the left we see a particle shooting through space with it's discretized counter-part. Velocities field at the bottom giving a hint to direction. To the right, two pipes spit out particles of a fluid at different speeds, diffusion causes them to curl towards the center. In the center, both flows collide, raising the grid as a way to represent the force of pressure.

The continuity equation[HBSL03], $\nabla \cdot \boldsymbol{u} = 0$ is implied since the fluid simulation deals with incompressible flow[GBO04]. $\nabla \cdot \boldsymbol{u} = 0$ states that any fluid that flows into one region of the fluid must have flowed out of another. As such, a fluid can not be compressed (or expanded) as it would violate this condition by taking less or more space without the fluid flowing elsewhere or coming in from somewhere else.

2.1.5 External Forces

In this chapter we focus on what could be called the "core equations" of incompressible fluid flow with respect to our simulation. The provided equations can be augmented through external forces which provide additional behaviours.

Let $\boldsymbol{F}_{\text{ext}}(\boldsymbol{x}, t) \in \mathbb{R}^2$ be a vector field known as the external forces. This vector field is a summation of individual forces, such as gravity, acting upon the fluid.

A sampling of possible external forces are provided in the next chapter starting on page 37.

2.1.6 The Navier-Stokes Equations for Incompressible Flow

The Navier Stokes equations for incompressible fluid flow[Sta99][GBO04], (2.30) and (2.32), are the focus of this chapter. The equations come from Claude Navier in 1822 and George

Stokes in 1845[MCG03]. These equations are also known as the momentum equations[FM96], as well as the incompressible Euler equations of fluid motion[HBSL03], or simply the incompressible Euler equations[FSJ01].

Consider Newton's second law[CM03]:

$$\boldsymbol{F} = m\boldsymbol{a} \tag{2.24}$$

Recall that the material derivative of $\boldsymbol{u}$ equals acceleration:

$$\boldsymbol{F} = m\frac{D\boldsymbol{u}}{Dt} \tag{2.25}$$

$$= m(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + m\frac{\partial \boldsymbol{u}}{\partial t} \tag{2.26}$$

Since we assume that the density of the fluid is constant, for any fixed volume the mass will always be the same:

$$\boldsymbol{F} = (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \frac{\partial \boldsymbol{u}}{\partial t} \tag{2.27}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \boldsymbol{F} \tag{2.28}$$

$$\tag{2.29}$$

And expressed as a system of equations with the forces of pressure and viscosity added in, the Navier Stokes equations for incompressible flow are[2]:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla p + \upsilon\nabla^2\boldsymbol{u} + \boldsymbol{F}_{\text{ext}} \tag{2.30}$$

$$\frac{\partial \boldsymbol{o}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{o} \tag{2.31}$$

$$\nabla\cdot\boldsymbol{u} = 0 \tag{2.32}$$

(2.30) describes how the velocity field $\boldsymbol{u}$ evolves with respect to time t. It is composed of an advection term $-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u}$, a pressure term $-\nabla p$, a viscosity term $\upsilon\nabla^2\boldsymbol{u}$, and external forces $\boldsymbol{F}_{\text{ext}}$.

(2.31) moves fluid attributes $\boldsymbol{o}$, ensuring that heat and colour, in addition to velocity, follow the approximated particles in the fluid flow.

[2]These are as found in [CM03], however Stam divides the pressure gradient by the density[Sta99], and there are other variations.

(2.32) is the continuity equation specifying that the fluid can not be compressed. It is a required condition for incompressible flow.

A fluid simulation based on these equations can potentially display rotational properties. As evidence, add in a single set of velocities pushing the fluid up on the right-half of the simulated region with walls bounding the simulation region. Once the fluid hits the top of the region, it will have no choice but to go left, then go down, and right back to the source so it may start over again. This would create a subregion in the middle of the vector field where the field is not irrotational.

We now have the basic equations that are at the heart of our fluid simulation. The rest of this chapter is spent massaging the system of equations until it is in a form that can more easily be translated into code for the underlying computational hardware to execute.

2.2 Boundary Conditions

Our simulated fluid does not exist in an infinite expanse of space, rather it is confined by the size of the underlying velocity and density fields – which unfortunately are finite due to computational constraints. In this section we describe what happens when the fluid attempts to flow out of its container, known as the simulation domain[Har04].

The special cases that arise when the simulated fluid meets a border of its simulation domain is known as a boundary condition. Boundary conditions are also required to ensure that the equations are well posed, "...that a unique solution exists and depends continuously on the initial data"[CM03].

Boundary conditions do not only apply to the edges of the simulation, but also to how the simulation reacts to objects within the fluid. We will first look at the simplest condition, a fluid flowing against a solid obstacle. Second, we will consider the case of an object moving of its own accord within the flow. Third, we look at letting fluid flow out of the edge of the simulation domain.

Consider figure 2.9. In this example we have the fluid flowing against a static obstacle. If we apply an opposing force normal to the surface of the obstacle that is equal to the force of the flowing fluid, we push back the fluid. At the same time, the velocity where the fluid meets the obstacle, known as the interface, is $\mathbf{0}$. This is known as a no-slip boundary condition[CM03]. Specifically, $\boldsymbol{u} = \mathbf{0}$ on static obstacles[CM03].

Before we look at how we deal with moving obstacles within the fluid, we will define a few variables:

Let $\boldsymbol{n}(\boldsymbol{x}, t)$ be a unit normal to the surface of the obstacle at point $\boldsymbol{x}$ and time t.

Let $k \in \mathbb{R}^+$ be a scalar where $k > 0$.

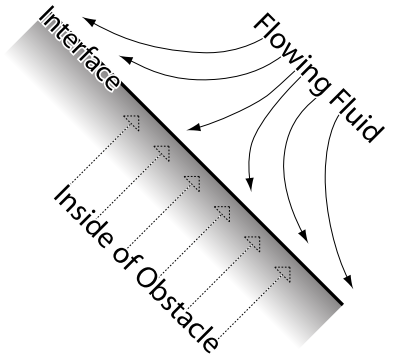

Figure 2.9: No-slip Boundary Conditions
The interface acts like a wall that pushes back the fluid with equally opposing velocities normal to the surface. Within the system, this results in an increase of pressure forcing the fluid to flow around the obstacles.

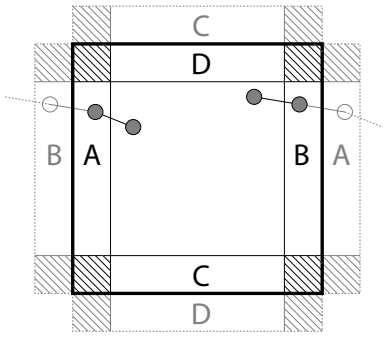

Figure 2.10: Periodic Boundary
Imagine the circle going out of side A gets teleported to B. The grey areas show the logical continuation of the fluid on the edges. The dash-filled boxes represent corner cases.

Let $\boldsymbol{u}_{\text{fluid}}(\boldsymbol{x}+k\boldsymbol{n},t)$ be the velocity of the fluid near the interface at the limit as k approaches 0.

Let $\boldsymbol{u}_{\text{solid}}(\boldsymbol{x}-k\boldsymbol{n},t)$ be the velocity of a moving obstacle within the fluid at the limit as k approaches 0.

We can extend the no-slip boundary condition for moving obstacles by specifying that $\boldsymbol{u}_{\text{fluid}} \cdot \boldsymbol{n} = \boldsymbol{u}_{\text{solid}} \cdot \boldsymbol{n}$[CLT08][3]. In this case, the velocity of the fluid is set to match that of the moving obstacle, allowing the obstacle to interact with the fluid flow.

Smoke should not be confined by the simulation domain. In such a case, we do not stop the flow of the fluid once it reaches the edge of the simulation domain, which is known as a free-slip boundary condition. Specifically, we do not set the velocity at the border. If $\boldsymbol{u}_{\text{edge}}$ is the velocity of the fluid just outside of the simulation domain and $\boldsymbol{u}_{\text{inside}}$ is a nearby velocity of the fluid close to the edge of the simulation domain, then a free-slip boundary condition can be defined as $\boldsymbol{u}_{\text{edge}} = \boldsymbol{u}_{\text{inside}}$[FM96].

No-slip boundary conditions can have problems when fluid information has to be taken from outside the simulated region. Wrapping the fluid domain, as seen in Figure 2.10, can be a solution depending upon the application. This wrapping of the fluid domain is known as a periodic boundary condition[Sta99]. Periodic boundary conditions can be used to provide the illusion of an endless ocean if the simulated region is wide enough, for example.

[3][CLT08] and [TLP06] calls $\boldsymbol{u}_{\text{fluid}} \cdot \boldsymbol{n} = \boldsymbol{u}_{\text{solid}} \cdot \boldsymbol{n}$ a free-slip boundary condition, contradicting [FM96][Har04]. For this reason, we have opted to speak of an "extended no-slip boundary condition" rather than a "free-slip boundary condition'.'

We have introduced some boundary conditions in this section which allow us to describe obstacles, either static or moving, within a fluid. With this information in hand, we will look at discretizing the equations on Eulerian grids.

2.3 Eulerian Grids

Our focus so far has been on continuous fields to describe fluid flow, whereas our simulated flow will operate on finite-sized discrete grids. In this section, we translate the Navier Stokes equations for fluid flow into a form understandable by the machine.

We use a Eulerian representation, also referred to as a field-based and grid-based representation, of the equations describing fluid flow. The term Eulerian is named after the Swiss[Weic] mathematician Leonhard Euler (1707-1783)[KC08].

First, we will look at previous work related to fluid simulations that use a grid-based representation, outlining the contributions to the literature.

Second, we define the constraints that are inherited from the interactive nature of the system.

Third, we examine numerical issues that could potentially plague the visual aesthetic of the fluid simulation.

Fourth, we look at the discretization of the advection equation, $\frac{\partial \boldsymbol{o}}{\partial t} = -(\nabla \cdot \boldsymbol{u})\boldsymbol{o}$.

Fifth, we discretize the viscosity term, $v\nabla^2\boldsymbol{u}$ in order to add a bit of friction between particles.

Sixth, we deal with the pressure term, $-\nabla p$, and the incompressibility condition, $\nabla \cdot \boldsymbol{u} = \boldsymbol{0}$, while emphasizing their relationship with regards to incompressible flow.

2.3.1 Previous Work

We attempt to tell a chronological story of the evolution of grid-based fluid simulations emphasizing what inspired us in our simulation or what we found interesting:

1928 In the German journal, Mathematische Annalen, Courant, Friedrichs, and Lewy determine that there is a ratio among the grid size and distance data can travel across the grid (through advection, for example) which allows for convergence of partial differential equations on rectangular grids. IBM published an English translation in 1967[CFL67]. The results from this paper have had an impact in fluid simulations, primarily in determining a maximum reasonable timestep.

1965 Harlow and Welch[HW65] introduced the marker and cell method. They used massless marker particles for the densities allowing them to have greater amounts of detail from the multitudes of particles despite the coarse underlying simulation grid. We would like to note that their simulation had stability issues if the time-step was too great.

1990 Kass and Miller [KM90] simplified the Navier Stokes equations to only deal with height-fields. The reduced complexity of the resulting equations allowed them to simulate the surface of a body of water fast enough for their purposes. Kass and Miller were not the only ones who attempted to simplify the problem around this time.

1996 Foster and Metaxas extended the simulation method introduced by Harlow and Welch for the use of animators. Since their focus was computer graphics, they only cared about the visual appeal of the results, therefore they had a coarser underlying simulation grid. For control, they allowed the user to add inflows, outflows, and modifying the pressure of the free surfaces[FM96].

1997 Foster and Metaxas continued their work to add more controls for animators allowing the animators to obtain the motion they desired out of the simulated fluid without expending effort to understand how the simulation worked[FM97a]. In another paper in the same year, Foster and Metaxas looked at the simulation of turbulent gases, again with a focus on visual flair rather than numerical accuracy. They noted different boundary conditions that could be applied to obtain different interactions between the fluid and obstacles[FM97b].

1999 Stam introduced an, interactive, unconditionally stable fluid solver that solved the complete incompressible Navier Stokes equations[Sta99]. Stam's solver's advection component was based on the method of characteristics which greatly reduced the number of calculations needed to move fluid matter around. Also, he used implicit equations for added stability in the pressure and viscosity solvers. Being from computer graphics, stability and speed came at the expense of accuracy. For example, the velocity and density fields would synthetically diffuse over time due to the advection scheme.

2001 Foster and Fedkiw describe a system for animating fluids, however they wanted more accuracy than what the advection scheme used by Stam provided. Therefore, they used variable time-steps based upon the velocity field when doing semi-lagrangian advection. For more accuracy, they used a level set to keep track of the evolution of the density field. Particles were used near the interface of the fluid for splashing effects. They described a system capable of simulating 3D fluids with free surfaces and moving objects.[FF01].

2003 Treuille, McNamara, Popović, and Stam improved the accuracy of the advection step by doing several small semi-lagrangian steps and showed an improved method to control animations of gases by specifying where the gas densities should go. For example, they could have smoke form arbitrary shapes[TMPS03].

2004 Lossaso, Gibou, and Fedkiw simulated fluids on an octree in order to unbound the simulated region and only apply computations to what is interesting at the moment[LGF04].

2006 Treuille, Lewis, and Popović decided to rethink how fluids are simulated in interactive environments, and preferred pre-computing the simulation for certain flows and using

model reduction techniques. By doing so, they have achieved higher resolutions at interactive frame-rates, at a loss of accuracy when the pre-computed data is unable to match the user interaction accurately enough[TLP06].

2008 Selle, Fedkiw, Kim, Liu, and Rossignac introduced an unconditionally stable MacCormack method[SFK+08]. The MacCormack method helps improve the accuracy of the semi-lagrangian advection step of Stam's 1999 fluid solver. [CLT08] used this method, which looks like predictor corrector.

2009 Wicke, Stanton, and Treuille extended the work started by Treuille et al. in 2006 to enable the simulation of fluids on large-scale scenes by tiling multiple simulations grids[WST09].

2010 Lentine, Zheng, and Fedkiw increase the performance of their fluid simulation by doing the pressure-projection step on coarse grids[LZF10].

Our basic simulation presented in this section is derived from Stam's stable fluid solver[Sta99] with a few notes from other sources as necessary. Stam's solver has the advantage of being able to work with arbitrarily large velocities and time-steps – ideal for creative geniuses that enjoy the aesthetic of unrealistically fast fluid flow.

2.3.2 Interactivity Constraints

Interactivity places harsh time constraints on the system. Before delving into details of the discretization, we will build a vocabulary which we can use to refer to these constraints.

frame We will use the term frame as it is used in video-processing, which is a single image in an animation. A frame could refer to a completed image, an incomplete image currently being drawn, or an empty image.

render The process of completing the drawing of a frame or sequence of frames. Unless otherwise specified, we assume that to render is to draw a single frame.

frame-rate The number of frames that can be rendered in a second. Typically used as a metric to determine how fast the system is running.

interactive The ability for an someone to be able to noticeably manipulate a system without any noticeable lag. In other words, to obtain an immediate response.

real-time The rendering of a series of frames quickly enough to be interactive (with respect to digital matter), which ideally is about 30 to 60 frames per second.

We treat the fluid simulation like any other interactive video filter. Specifically, at each frame the simulation is called upon to render itself given a series of inputs, such as colour or velocity fields. The output of rendering the fluid simulation are a colour, velocity, and pressure field.

Let $\Delta t \in \mathbb{R}^+$ be the desired change in time, measured in seconds. We assume this value to equal, on average, the amount of time it takes to render a single frame of the fluid simulation. To be interactive, we must be able to run the simulation about 30 to 60 frames a second, so Δt will ideally vary between $\frac{1s}{30}$ and $\frac{1s}{60}$.

Δt need not be constant, since we want the motion to maintain a constant speed regardless of the amount of time it takes to render a frame. Consider an animation of a ball ball that moves across a room in five seconds. If the frame-rate is 60 frames per second, then the ball will move less between each consecutive frame compared to when the frame-rate is 30 frames per second.

Therefore, we compute the Δt between each frame and use that to ensure that our simulation appears to always run roughly at the same speed independent of the time it takes to integrate the equations.

The interactivity requirement implies that we have strict time constraints and can not assume the simulation to be a batch job whose results appear in a few hours.

2.3.3 Numerical Issues

There are multiple ways to discretize the Navier Stokes equations on rectangular grids. We often face situations where accuracy and speed of execution are opposed, and must carefully balance one against the other to ensure that speed constraints are met while navigating away from negative side-effects in the resulting visuals.

There are two issues we wish to bring up in regards to simulating fluids on Eulerian grids.

The first issue arises from the finite differencing of the advection equation where we approximate the equation using a 3×3 kernel for velocities that potentially advect data from outside the window of the kernel.

The second issue relates to the grid cells themselves. Each cell has a fixed area, and the data within the cell need not be located in the middle of the cell. We'll show that by shifting the location of values around, we could increase the accuracy of the simulation.

Finite Differencing Limits

The Courant-Friedrichs-Lewy[4] Condition, also known as the Courant Condition[PTVF07] (henceforth abbreviated as the CFL Condition) with respect to the Navier Stokes equations, is a restriction based upon the timestep and velocity as shown in (2.33)[FF01]. It is the idea that for the equations to converge to a solution, data must travel a distance that is not too great[Weia].

[4] In [FF01] Condition, Lewy is written as Levy.

Let $\Delta\tau \in \mathbb{R}^+$ be the width and height of the individual cells. For simplicity, we assume that $\Delta\tau = 1$.

Let $n \in \mathbb{R}^+$ by the length of the longest vector within the velocity field $\boldsymbol{u}$.

Then we can state the CFL condition with respect to the advection portion of the Navier Stokes equations as[PTVF07]:

$$\Delta t < \frac{\Delta\tau}{\sqrt{2}n} \tag{2.33}$$

Since we assume $\Delta\tau = 1$, (2.33) can be rewritten as:

$$n < \frac{1}{\Delta t\sqrt{2}} \tag{2.34}$$

(2.34) state that the distance travelled of the fluid in a given time-step should be less than $\frac{1}{\Delta t\sqrt{2}}$. Which can be interpreted as preventing the case where fluid attributes would be carried beyond the range of a 3×3 window as obtained through finite differencing.

Let m be the maximum permissible time-step with the CFL condition given a desired time-step of Δt:

$$m = \begin{cases} \frac{1}{\sqrt{2}n}, & \Delta t > \frac{1}{\sqrt{2}n} \\ \Delta t, & \Delta t \leq \frac{1}{\sqrt{2}n} \end{cases} \tag{2.35}$$

Let $k = \lceil \frac{\Delta t}{m} \rceil$ be the number of times we would need to run the advection algorithm to achieve a combined time-step of Δt. That is, we would iterate over the advection algorithm k times using a time-step of $\frac{\Delta t}{k}$.

Apart from iterating over the advection algorithm multiple times to obtain a desired time-step of Δt, we can also restrict the permissible range of u, v and Δt. Restricting the range of these values has the added benefit of setting a maximum amount of time needed to compute a given algorithm, such as advection. Such restrictions can be useful on slower machines that could potentially end up requiring successively more time to compute each frame as the number of iterations increase due to an increase in Δt.

We focus on the practical results of the CFL Condition, however we invite the curious reader to consult [CFL67] for the original paper, [Lax67] to obtain an update of the material, and [PTVF07] for a derivation of the condition in the 1D case.

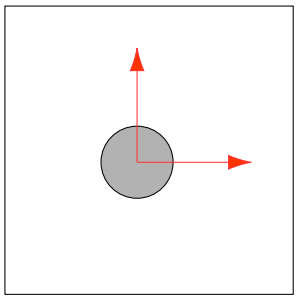

Figure 2.11: Collocated Grids
All the data is concentrated in the middle of the grid cells.

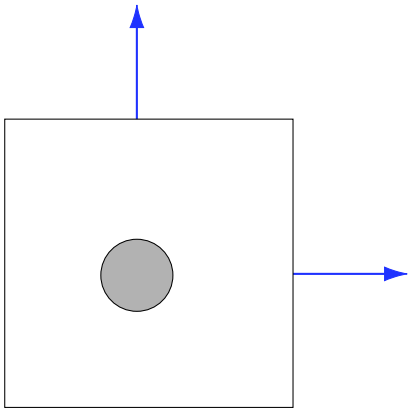

Figure 2.12: Staggered Grids
Scalar fluid attributes are still located in the center of the grid cells, however the velocities are now on the edges of the cells.

Location, Location, Location...

Recall that $\boldsymbol{u}_d$ is an approximation of the continuous function $\boldsymbol{u}$. In order to obtain values anywhere within the grid cells, we can use linear interpolation. Therefore, we could place the base point of the fluid attributes anywhere within a grid cell.

We opted to place all of the fluid attributes in the center of the cells as shown in figure 2.11, known as collocated grids. Collocated grids are typically used for their simplicity[CLT08].

Alternatively, we could have placed the velocities on the edges of the grid cells, known as staggered grids, shown in figure 2.12. Staggered grids were used by[HW65][FSJ01] since they offer increased accuracy[Har04][CLT08].

Staggered grids incur the additional overhead of extrapolating values. Consider the case when a value is on a cell centre but we need to operate on the value on the edge of the cells. In retrospect, we should have implemented staggered grids.

2.3.4 Advection

We begin by looking at the discretization of the advection term, $-(\boldsymbol{u}\cdot\nabla)\boldsymbol{o}$, since we can easily test the results independently of the rest of the fluid solver by fabricating velocity fields and watching the values of $\boldsymbol{o}$ move accordingly. We look at four different ways of advecting fluid data.

First, we begin with finite differencing since that is what we did in our initial implementation.

Second, we look at what is commonly referred to as a semi-Lagrangian advection scheme. This method is based on the method of characteristics and was first applied to fluid simulations in the computer graphics literature by Stam[Sta99]. Stam's semi-Lagrangian is now used in most simulations on Eulerian grids.

Third, we look at methods of increasing the accuracy of Stam's technique by estimating the amount of error incurred through the advection phase.

Finally, we document our implementation which attempts to improve on previous work by pre-computing corrections based on error estimates in the advection phase.

Finite Differencing

Let $\boldsymbol{o} \in \mathbb{R}^k$ represent a collocated vector field that should be advected, such as pressure, colour, or velocity.

First, we break down the advection term as follows:

$$\begin{aligned}
\frac{\partial \boldsymbol{o}}{\partial t} &= -(\boldsymbol{u} \cdot \nabla)\boldsymbol{o} && (2.36)\\
&= -\boldsymbol{u} \cdot \left(\frac{\partial \boldsymbol{o}}{\partial x}, \frac{\partial \boldsymbol{o}}{\partial y}\right) && (2.37)\\
&= -\left(u\frac{\partial \boldsymbol{o}}{\partial x} + v\frac{\partial \boldsymbol{o}}{\partial y}\right) && (2.38)\\
&= -u\frac{\partial \boldsymbol{o}}{\partial x} - v\frac{\partial \boldsymbol{o}}{\partial y} && (2.39)
\end{aligned}$$

Let x and y, when appearing in subscripts, be integers whose values are coordinations referring to the cell centers of the simulation grid.

Let $\Delta\tau$ denote the width and height of the underlying computational grid.

Using finite differencing, we obtain the following discretization of (2.39):

$$\begin{aligned}
\frac{\partial \boldsymbol{o}}{\partial t} &\approx \frac{\boldsymbol{o}_{x,y}^{t+\Delta t} - \boldsymbol{o}_{x,y}}{\Delta t} && (2.40)\\
\frac{\boldsymbol{o}_{x,y}^{t+\Delta t} - \boldsymbol{o}_{x,y}}{\Delta t} &= -u\frac{\boldsymbol{o}_{x+\Delta\tau,y} - \boldsymbol{o}_{x-\Delta\tau,y}}{2\Delta\tau} - v\frac{\boldsymbol{o}_{x,y+\Delta\tau} - \boldsymbol{o}_{x,y-\Delta\tau}}{2\Delta\tau} && (2.41)\\
&= \frac{-u\boldsymbol{o}_{x+\Delta\tau,y} + u\boldsymbol{o}_{x-\Delta\tau,y} - v\boldsymbol{o}_{x,y+\Delta\tau} + v\boldsymbol{o}_{x,y-\Delta\tau}}{2\Delta\tau} && (2.42)
\end{aligned}$$

Since we assume $\Delta\tau = 1$:

$$\frac{\boldsymbol{o}_{x,y}^{t+\Delta t} - \boldsymbol{o}_{x,y}}{\Delta t} = \frac{-u\boldsymbol{o}_{x+1,y} + u\boldsymbol{o}_{x-1,y} - v\boldsymbol{o}_{x,y+1} + v\boldsymbol{o}_{x,y-1}}{2} \quad (2.43)$$

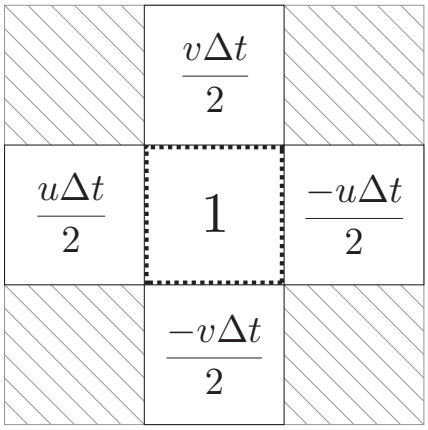

Figure 2.13: Collocated Advection Kernel 3×3 *kernel applied on* $\boldsymbol{o}$ *for the advection of* $\boldsymbol{o}$ *across velocity field* $\boldsymbol{u} = (u, v)$ *using finite differencing.*

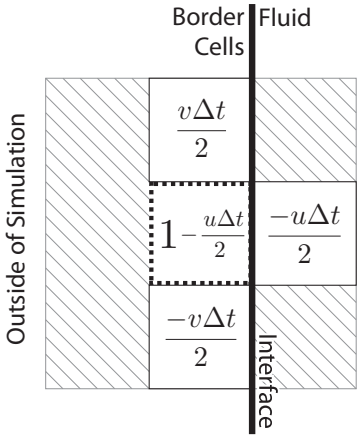

Figure 2.14: Collocated Border Free-Slip Advection Kernel
Velocity on the outer row matches that of the inner row.

The computational kernel for the discretized advection equation (2.43) is shown in figure 2.13. The way we set up our no-slip boundary conditions, the wall that the fluid can not go through are the cells that form the border of the grid. As such, no information can flow into or out of the border.

Figure 2.14 shows the associated free-slip boundary conditions kernel for the left of the grid. We would do nothing in the no-slip case since nothing could flow into the "solid" border.

Figures 2.15 to 2.20 emphasize a problem with finite differencing. Data is assumed to be sufficiently smooth, unlike what is provided by creative genuises[5]. If we blur the image each frame before advecting, then we do not see any artifacts in the advected image, however the result is diffuse rather than sharp. We have yet to try just blurring the rendered result rather than making blurring an integral part of the advection phase.

The Semi-Lagrangian Scheme

The CFL condition combined with the finite-differencing form of the advection equation leads to two undesirable consequences: an increase in the number of iterations as the time-step and velocities increase, or restricting the freedom of the creative individuals who will use the system by giving a specific, pre-defined range to the velocity and time-step.

Recall that we are modeling the movement of particles with the advection term. Would we not avoid the issue of having to do multiple iterations over the advection algorithm if we moved

[5]By 'creative genius', we mean a person who tries things for the aesthetic value without care for the underlying computation. For example, a field of colours is not sufficiently smooth if it contains sharp edges, like those of our example smiley face.

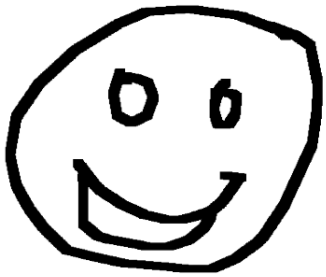

Figure 2.15: Finite Differencing Advection Start
We test using a simple image treated as a colour field. Notice the sharp discontinuous edges of the image.

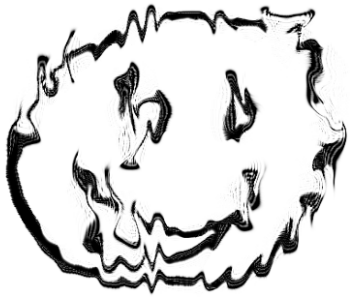

Figure 2.16: Finite Differencing Advection Sample 2
Advection initially starts to look right, except for a few defects in the form of white lines where a solid colour is expected.

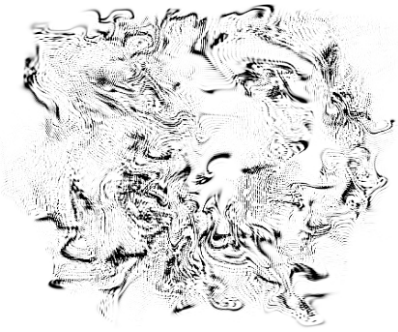

Figure 2.17: Finite Differencing Advection Sample 3
Third frame of the simulation.

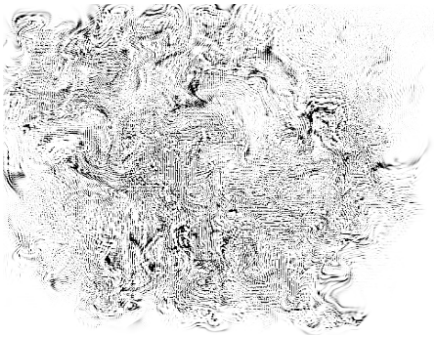

Figure 2.18: Finite Differencing Advection Sample 4
Last frame where the whole is an unrecognizable mass.

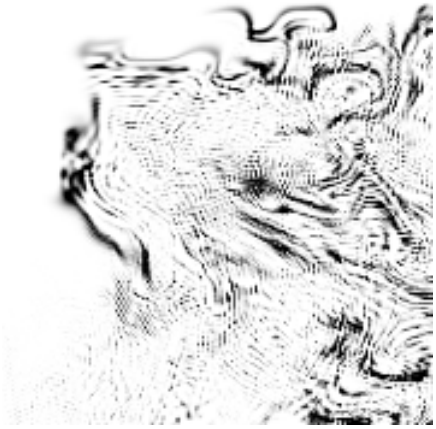

Figure 2.19: Finite Differencing Advection Sample 3 Zoom
Zooming in the top-left corner of sample 3, we see that in many places there is some noise that appears in the simulation.

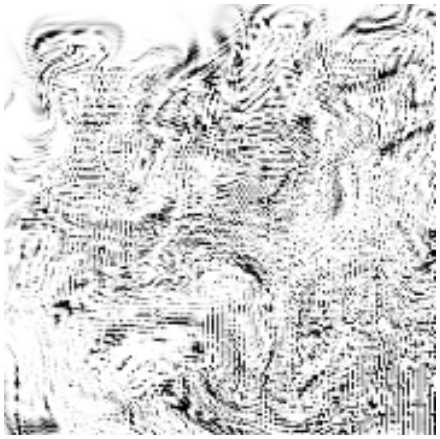

Figure 2.20: Finite Differencing Advection Sample 4 Zoom
Zooming in on the fourth frame, we see the noise amplify compared to the previous sample. The noise comes from oscillating extremes of different colour values.

particles on the underlying grid? We emphasize that the issue the CFL condition raises is that of more data being required than what's covered by a 3×3 kernel.

It is this idea that Stam evokes in [Sta99], referred to as a semi-lagrangian advection scheme in the computer graphics literature.

Let $\boldsymbol{o} \in \mathbb{R}^k$ be a 2D grid that must be advected where $k \in \mathbb{I}^+$.

Consider the cell with colour $\boldsymbol{o}_{x,y}$ whose associated velocity is $\boldsymbol{u}_{x,y}$. If we say the cell itself is a particle, then since the velocity field $\boldsymbol{u}$ is divergence free we can go backwards in the velocity field and follow the flow to what will circulate into the cell[6]. We visually demonstrate this concept in figure 2.21.

For efficiency, it suffices to assume that the direction of the flow does not curve, and that the velocity at $\boldsymbol{u}_{x,y}$ is all we need to determine from where the flow comes from, as shown in figure 2.22. The source location for the advection of fluid flow becomes:

$$\frac{\partial \boldsymbol{o}}{\partial t} = (\boldsymbol{u} \cdot \nabla)\boldsymbol{o} \tag{2.44}$$

$$\approx \boldsymbol{o}_{x-\Delta t u_{x,y}, y-\Delta t v_{x.y}} \tag{2.45}$$

[6]This assumes infinite spatial resolution. Since we assume that the fluid is incompressible, divergent-free implies that whenever a fluid flows out of one space, fluid from elsewhere will flow in to fill the gap. Multiple sources can not flow into a single destination due to the constant density.

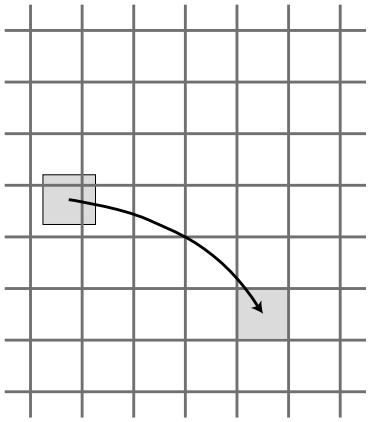

Figure 2.21: Semi-Lagrangian Advection Concept
From the cell whose value we wish to know, we follow the velocity field backwards until we reach the source. The source will most likely be between cells, so we interpolate to get the desired value.

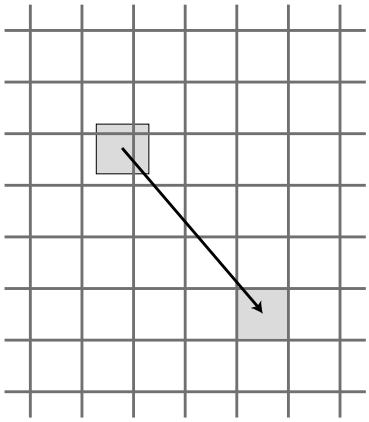

Figure 2.22: Semi-Lagrangian Advection
For convenience, we assume that a given cell's velocity vector is sufficient to approximate the location of the source cell.

Using the semi-lagrangian scheme, we can advect a field in a single pass regardless of the time-step or velocity. Although accuracy suffers as the magnitudes of the values in the velocity field increases, however the visual results are greatly improved compared to the forward advection.

More often than not, the source for a given destination cell will not be found on a cell center. In such cases, the nearby known values are interpolated in order to approximate the source value.

Since semi-lagrangian advection normally ends up interpolating values, over time the colour and velocity field progressively blur. This effect is generally acceptable, and we admit that our simulation suffers from this flaw. The result is that the outputs of our simulation are more diffuse than they should be.

To be generally more accurate, we suggest doing multiple iterations using smaller time-steps as done by [FF01] or using some form of error correction as described in the next section. Enhanced accuracy sometimes trumps visual resolution when fine-scale details are lost due to overly coarse approximations[CLT08].

Error Correction

Consider the following implication of a divergence free velocity field $\boldsymbol{u}$ with infinite resolution. For illustrative purposes, we replace the density field with an infinite resolution sliding picture puzzle where each puzzle piece is unique. The picture puzzle is an accurate analogy in that no

puzzle pieces (density) can be created nor destroyed[7]. However, there is another interesting attribute: if we negate the timestep (reverse the direction of the flow), we would return to the initial state of the puzzle.

Unfortunately, we are not dealing with infinite resolutions and thus have to consider the case of puzzle pieces being approximately moved to their proper destination. Since we can run the advection algorithm "backwards" in time to ideally obtain the original configuration of the puzzle, we can estimate the error by comparing the original configuration of the puzzle with the result of running the algorithm backwards.

Such accuracy issues arise often with semi-langrangian advection schemes since the direction of the flow is approximated as straight lines.

Let $\boldsymbol{\phi} \in \mathbb{R}^n$ be a vector field where $n \in \mathbb{R}$.

Consider a vector field $\boldsymbol{o} \in \mathbb{R}^n$ that we wish to advect along the divergent-free vector field $\boldsymbol{u}$. We will call A an advection operator that takes in a vector field $\boldsymbol{o}^t$ at time t and a change in time Δt and computes an updated vector field $\boldsymbol{o}^{t+\Delta t}$ that flowed along $\boldsymbol{u}$.

Therefore, we can correct a velocity field by:

$$\boldsymbol{\phi}^{t+\Delta t} = A(\boldsymbol{o}, \Delta t) \tag{2.46}$$

$$\boldsymbol{\phi} = A(\boldsymbol{\phi}^{t+\Delta t}, -\Delta t) \tag{2.47}$$

$$\boldsymbol{o}^{t+\Delta t} = \boldsymbol{\phi}^{t+\Delta t} + \frac{1}{2}(\boldsymbol{o} - \boldsymbol{\phi}) \tag{2.48}$$

This application of predictor-corrector is known as the MacCormack method[SFK+08].

[CLT08] clamped the corrected values to those of the nearby locations that Stam's algorithm would obtain for stability.

Extensions to Error Correction

When we implemented the MacCormack method, we realized that we intend to correct for spatial errors in estimating the curve of the flow. As stated, the algorithm varies values based upon an error estimate. We, on the other hand, wish to estimate the error in location.

Let $x \in \mathbb{I}$ and $y \in \mathbb{I}$ be offsets into the grids. Each unique pairing of x and y refers to a different grid cell. The range of x and y is restricted by the size of the underlying grids; which we assume to be of infinite for convenience.

Let $m \in \mathbb{R}$ and $n \in \mathbb{R}$ with an unbounded range be temporary variables to work with.

[7] We do allow temporarily removing pieces to slide the puzzle where vortices exist.

Consider a vector field $\boldsymbol{f} \in \mathbb{R}^2$ where $\forall x \forall y f(x, y) = (x, y)$. From $\boldsymbol{f}$, we can state:

$$\begin{aligned}
\nabla \boldsymbol{f} &= \left(\frac{\partial \boldsymbol{f}}{\partial x}, \frac{\partial \boldsymbol{f}}{\partial y} \right) && (2.49)\\
&= ((1,0),(0,1)) && (2.50)\\
\nabla \boldsymbol{f} \cdot (1,1) &= (1,1) \cdot ((1,0),(0,1)) && (2.51)\\
&= (1,1) && (2.52)\\
\nabla \boldsymbol{f} \cdot (m,n) &= (m,n) \cdot ((1,0),(0,1)) && (2.53)\\
&= (m,0) + (0,n) && (2.54)\\
&= (m,n) && (2.55)
\end{aligned}$$

Let $t \in \mathbb{R}$ be the current time in the simulation and $\Delta t \in \mathbb{R}$ be the size of the time-step.

Recall that the advection equation is:

$$\frac{\partial \boldsymbol{o}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{o} \quad (2.56)$$

For convenience, we work with the equation in a partially discretized form:

$$\begin{aligned}
\boldsymbol{o}^{t+\Delta t} &= A(\boldsymbol{o}^t) && (2.57)\\
&= \boldsymbol{o}^t - \Delta t(\boldsymbol{u} \cdot \nabla)\boldsymbol{o} && (2.58)
\end{aligned}$$

Before we continue, we wish to emphasize the f is a smooth vector field where each entry in a given cell is the index into said cell. Specifically, f is a grid of coordinates that refer to the location of desired values in other fields.

We begin with the prediction part of the MacCormack method:

$$\begin{aligned}
\boldsymbol{\phi}^{t+\Delta t} &= \boldsymbol{f} - \Delta t(\boldsymbol{u} \cdot \nabla)\boldsymbol{f} && (2.59)\\
&= \boldsymbol{f} - \Delta t \boldsymbol{u} \cdot (\nabla \boldsymbol{f}) && (2.60)\\
&= \boldsymbol{f} - \Delta t \boldsymbol{u} && (2.61)
\end{aligned}$$

The prediction phase is dependent on the current velocity field. Now we attempt to compute

the error:

$$\begin{aligned}
\boldsymbol{\phi}^t &= \boldsymbol{\phi}^{t+\Delta t} + \Delta t(\boldsymbol{u}\cdot\nabla)\boldsymbol{\phi}^{t+\Delta t} && (2.62)\\
&= \boldsymbol{f} - \Delta t\boldsymbol{u} + \Delta t(\boldsymbol{u}\cdot\nabla)(\boldsymbol{f} - \Delta t\boldsymbol{u}) && (2.63)\\
&= \boldsymbol{f} - \Delta t\boldsymbol{u} + \Delta t\boldsymbol{u}\cdot(\nabla(\boldsymbol{f} - \Delta t\boldsymbol{u})) && (2.64)\\
&= \boldsymbol{f} - \Delta t\boldsymbol{u} + \Delta t\boldsymbol{u}\cdot(\nabla\boldsymbol{f} - \nabla(\Delta t\boldsymbol{u})) && (2.65)\\
&= \boldsymbol{f} - \Delta t(\boldsymbol{u}\cdot\nabla)(\Delta t\boldsymbol{u}) && (2.66)
\end{aligned}$$

Last, we work out the correction phase:

$$\begin{aligned}
\boldsymbol{f}^{t+\Delta t} &= \boldsymbol{\phi}^{t+\Delta t} + \frac{1}{2}(\boldsymbol{f} - \boldsymbol{\phi}^t) && (2.67)\\
&= \boldsymbol{f} - \Delta t\boldsymbol{u} + \frac{1}{2}(\boldsymbol{f} - \boldsymbol{f} + \Delta t(\boldsymbol{u}\cdot\nabla)(\Delta t\boldsymbol{u})) && (2.68)\\
&= \boldsymbol{f} - \Delta t\boldsymbol{u} + \frac{\Delta t(\boldsymbol{u}\cdot\nabla)(\Delta t\boldsymbol{u})}{2} && (2.69)
\end{aligned}$$

$\boldsymbol{f}^{t+\Delta t}$ contains the corrected coordinates for the source of the advection of the fluid.

With our goal of correcting only for position, we have stumbled across an algorithm that can give us the correction information by adding a single semi-lagrangian advection operation to the system instead of predicting and correcting each field.

2.3.5 Viscosity

Viscosity is a sort of friction between the particles of the liquid. It ensures that different streams of flow affect each other resulting in rotational properties[CM03].

First, we show a discretized form of the viscosity term, $\upsilon\nabla^2\boldsymbol{u}$.

Second, we specify some boundary conditions with their associated implications.

Third, we visualize the computational kernels on 3×3 grids.

Discretization

To ensure that the discretized solution to the viscosity is unconditionally stable, an implicit solution is used. Fortunately, Stam provides an implicit version of the viscosity term [Sta99] derived from the $\upsilon\nabla^2\boldsymbol{u}$ term of (2.30):

$$(\mathbf{I} - \upsilon\partial t\nabla^2)\boldsymbol{u}(\boldsymbol{x}, t+\Delta t) = \boldsymbol{u}(\boldsymbol{x}, t) \qquad (2.70)$$

Using standard finite differencing of (2.70):

$$\text{Let } \boldsymbol{s} = \boldsymbol{u}_{x-1,y} + \boldsymbol{u}_{x+1,y} + \boldsymbol{u}_{x,y-1} + \boldsymbol{u}_{x,y+1} \tag{2.71}$$

$$-\upsilon \boldsymbol{s}\Delta t + \boldsymbol{u}_{x,y} + 4\upsilon \boldsymbol{u}_{x,y}\Delta t = \boldsymbol{u}^t_{x,y} \tag{2.72}$$

$$(1 + 4\upsilon\Delta t)\boldsymbol{u}_{x,y} = \boldsymbol{u}^t_{x,y} + \upsilon \boldsymbol{s}\Delta t \tag{2.73}$$

$$\boldsymbol{u}_{x,y} = \frac{\boldsymbol{u}^t_{x,y} + \upsilon \boldsymbol{s}\Delta t}{1 + 4\upsilon\Delta t} \tag{2.74}$$

Border Conditions

We use $\boldsymbol{u}_{\text{solid}} = -\boldsymbol{u}_{\text{fluid}}$ at the interface since the fluid should be pushed away at the same rate that it is trying to push itself into the solid. There is not much else to do since pressure will ensure that flow only occurs around the boundaries.

Alternatively, we could say $\boldsymbol{u}_{\text{solid}} = \boldsymbol{u}_{\text{fluid}}$, which would result in the fluid flowing through the walls of its container.

Window

A sliding window can be used to show what calculations are done. Since the calculations are repeated for each destination pixel taking source pixels in a given neighborhood, this gives us 3x3 windows as shown in figures 2.23 and 2.24 for the viscosity and a border condition.

2.3.6 Pressure-Projection and Continuity

As we have previously discussed, pressure and the incompressibility condition are related since we focus on incompressible flow[CM03]. The reason is that pressure results from a buildup of fluid matter[FM96]. Since the amount of density is constant, the force resulting from pressure becomes a function of velocity that prevents any compression of fluid flow. Essentially acting like the equation of continuity.

First, we manipulate the equations to put them in a form that is more amenable to discretization.

Second, we discretize the equations.

Third, we comment on the boundary conditions.

Fourth are relevant numerical issues.

Preparing the Equations

We compute pressure the same way as was done by Stam[Sta99], also explained by Harris[Har04], which was taken from Chorin and Marsden's work [CM03].

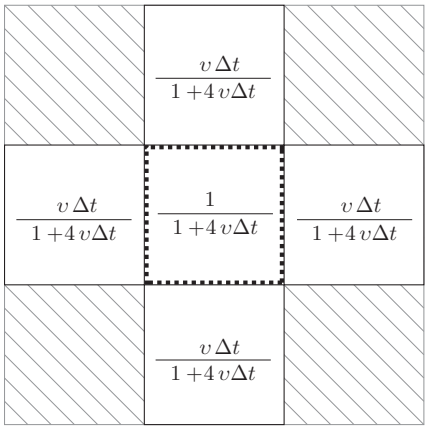

Figure 2.23: Window for Viscosity
The 3x3 kernel can be overlaid anywhere in the source velocity field, $\boldsymbol{u}$, to obtain a new value at the location delimited by the dotted box in the middle. Each of the provided fractions is multiplied to the underlying values in the grid cells and summed. The same calculations are repeated for u and v of $\boldsymbol{u}$.

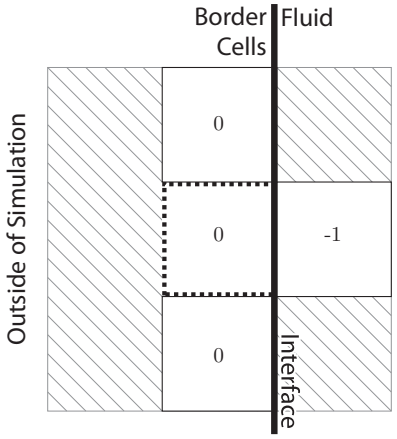

Figure 2.24: Window for Viscosity at Border
We set the cells at the border equal to the negative of the interior cells. The corners of the grid are corner cases which are ignored since experimental results show that the simulation looks good without them.

Before continuing, we will describe two velocity fields, $\boldsymbol{u}$ and $\boldsymbol{w}$. $\boldsymbol{u}$, the velocity field that we have worked with until now, is a divergent-free field that conforms to the continuity equation. $\boldsymbol{w}$ on the other hand is defined identically as $\boldsymbol{u}$ except it is not divergence-free.

The Helmholtz-Hodge decomposition, (2.75), decomposes a vector field $\boldsymbol{w}$ into two components, a divergence-free field $\boldsymbol{u}$ such that $\nabla \cdot \boldsymbol{u} = 0$, and an irrotational field ∇q[TLHD03].

$$\boldsymbol{w} = \boldsymbol{u} + \nabla q \tag{2.75}$$

To obtain Stam's solution, we take the divergence of both sides of (2.75):

$$\begin{aligned} \nabla \cdot \boldsymbol{w} &= \nabla \cdot (\boldsymbol{u} + \nabla q) && (2.76)\\ &= \nabla \cdot \boldsymbol{u} + \nabla^2 q && (2.77)\\ &= \nabla^2 q && (2.78) \end{aligned}$$

For the pressure, it is the divergence-free part of the field that is desired. Therefore the operator $\mathbb{P}$ is defined to transform a divergent vector-field $\boldsymbol{w}$ into a divergence-free vector

field $\boldsymbol{u}$.

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{w} - \nabla q & (2.79)\\ &= \mathbb{P}\boldsymbol{w} & (2.80) \end{aligned}$$

Where $\mathbb{P}$ also has the following properties:

$$\begin{aligned} \mathbb{P}\boldsymbol{w} &= \boldsymbol{w} - \nabla q & (2.81)\\ \mathbb{P}\boldsymbol{u} &= \boldsymbol{u} & (2.82)\\ \mathbb{P}\nabla q &= 0 & (2.83) \end{aligned}$$

(2.81) restates what is known from (2.80).

(2.82) states that a divergence-free field, when projected, doesn't change since it's already divergence-free.

(2.83) says that if the divergent component of the field is projected, then nothing is left. See section 1.3 of [CM03] for a proof as to why this is the case.

If we were to project both sides of the Navier Stokes Equations (2.30):

$$\begin{aligned} \mathbb{P}\left(\frac{\partial \boldsymbol{u}}{\partial t} + \nabla p\right) &= \mathbb{P}\left(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \upsilon\nabla^2\boldsymbol{u} + \boldsymbol{F}_{\text{ext}}\right) & (2.84)\\ \mathbb{P}\left(\frac{\partial \boldsymbol{u}}{\partial t}\right) + \mathbb{P}\left(\nabla p\right) &= \mathbb{P}\left(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \upsilon\nabla^2\boldsymbol{u} + \boldsymbol{F}_{\text{ext}}\right) & (2.85)\\ \frac{\partial \boldsymbol{u}}{\partial t} &= \mathbb{P}\left(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \upsilon\nabla^2\boldsymbol{u} + \boldsymbol{F}_{\text{ext}}\right) & (2.86) \end{aligned}$$

These are the manipulations as found in [CM03][Sta99][Har04], where pressure is cancelled out of the Navier Stokes equations. If pressure is thought of as a means to push particles that are clumped together, to make the vector field divergence free, then the projection takes care of the pressure[8][FM96].

Therefore, as described in [Har04], to solve for the projection operator $\mathbb{P}$, we must solve for q. Fortunately, (2.78), provides a means to compute q.

[8] This only works because the fluid is incompressible!

Discretized Solution

We can rewrite (2.78) as:

$$\nabla \cdot \boldsymbol{w} = \nabla^2 q \tag{2.87}$$

$$\frac{\partial \boldsymbol{w}}{\partial x} + \frac{\partial \boldsymbol{w}}{\partial y} = \frac{\partial^2 q}{\partial^2 x} + \frac{\partial^2 q}{\partial^2 y} \tag{2.88}$$

The left-hand-side of (2.88) can be approximated using finite differencing to obtain[9]:

$$\nabla \cdot \boldsymbol{w} = \frac{\partial \boldsymbol{w}}{\partial x} + \frac{\partial \boldsymbol{w}}{\partial y} \tag{2.89}$$

$$\approx \frac{\boldsymbol{w}^x_{x+\Delta x,y} - \boldsymbol{w}^x_{x-\Delta x,y}}{2\Delta\tau} + \frac{\boldsymbol{w}^y_{x,y+\Delta y} - \boldsymbol{w}^y_{x,y-\Delta y}}{2\Delta\tau} \tag{2.90}$$

$$= \frac{\boldsymbol{w}^x_{x+1,y} - \boldsymbol{w}^x_{x-1,y}}{2} + \frac{\boldsymbol{w}^y_{x,y+1} - \boldsymbol{w}^y_{x,y-1}}{2} \tag{2.91}$$

The super-script in this case denotes the current component of the vector that is being dealt with. The right-hand-side can be solved in a similar manner:

$$\nabla^2 q \approx \frac{\partial^2 q}{\partial^2 x} + \frac{\partial^2 q}{\partial^2 y} \tag{2.92}$$

$$\approx \frac{q_{x+\Delta x,y} - 2q_{x,y} + q_{x-\Delta x,y}}{\Delta\tau^2} + \frac{q_{x,y+\Delta y} - 2q_{x,y} + q_{x,y-\Delta y}}{\Delta\tau^2} \tag{2.93}$$

$$= \frac{q_{x+1,y} - 2q_{x,y} + q_{x-1,y}}{1} + \frac{q_{x,y+1} - 2q_{x,y} + q_{x,y-1}}{1} \tag{2.94}$$

$$= q_{x+1,y} + q_{x-1,y} + q_{x,y+1} + q_{x,y-1} - 4q_{x,y} \tag{2.95}$$

Combining (2.91) and (2.95) the following can be obtained:

$$\nabla \cdot \boldsymbol{w} \approx q_{x+1,y} + q_{x-1,y} + q_{x,y+1} + q_{x,y-1} - 4q_{x,y} \tag{2.96}$$

$$4q_{x,y} = q_{x+1,y} + q_{x-1,y} + q_{x,y+1} + q_{x,y-1} - \nabla \cdot \boldsymbol{w} \tag{2.97}$$

$$q_{x,y} = \frac{q_{x+1,y} + q_{x-1,y} + q_{x,y+1} + q_{x,y-1} - \nabla \cdot \boldsymbol{w}}{4} \tag{2.98}$$

[9]Recall that we assume the width and height of the grid $\Delta\tau$ equals 1.

Boundary Conditions

We consider a no-slip boundary condition. Since the other steps of the simulation ensure that the boundary condition is respected, we simply have to make sure that the pressure does not modify the velocity of 0 at the border. By setting the pressure as equal on both sides of the border, the negative of the gradient will always point away from the border. Therefore fluid will not flow within the border.

Numerical Issues

Our initial implementation used Jacobi's method[PTVF07] to solve the set of equations. The current implementation uses the Gauss-Seidel method[PTVF07] for its improved convergence properties[GS06] with a few modifications for compute speed.

Stam's "Stable Fluids"[Sta99] is known in CFD as a "fractional step method"[ETK+05]. This means that the velocity field is updated assuming the fluid is inviscid[10] and doesn't adhere to the equation of continuity. The Helmhodge-Holtz projection is used to find the closest divergence-free solution. This may not be the exact solution[ETK+05].

2.4 Particles

We have shown the equations that underly our presented fluid simulation. We thought it to be closed-minded of us if we did not include references to alternate means to discretize the fluid flow. Second to grids are particle-based representations, which we give an overview of in this chapter. We forgo an in depth review since the topic is, albeit very interesting, secondary to the produced results.

We begin by providing a brief overview of using particles in lieu of discrete grids.

Second, we give a brief literature review for anyone who wishes to delve deeper into the subject.

2.4.1 Overview

The term *Lagrangian* denotes the idea that the simulation will follow the trajectory of the individual particles. It is named after Joseph Louis, Compte de Lagrange (1736-1813)[KC08]. Combining Eulerian grids and Lagrangian particles will be covered in later sections.

Particle-based methods are varied. Some are hybrids, while others deal solely with particles. Hybrid particle-grid methods include the marker and cell method as introduced by Harlow and Welch[HW65].

[10] little or no viscosity

Purely particle-based methods tend to be based on ideas that come from Monte Carlo methods, known as Smoothed Particle Hydrodynamics (SPH) in the literature[HCB70][Luc77]. There are also other pseudo-physics methods that use particles to deliver interesting visuals that could resemble fluid flow[Ree83].

2.4.2 Previous Work

The literature describing particle-based fluid simulation techniques has at least as long a history as the grid-based simulations. We again attempt to tell the story of fluid-flow, but this time from the point of view of particle-based methods.

1970 Hirt, Cook and Butler present a Lagrangian technique called LINC where they start with a grid, but the corners of the grid follow the flow. This method does not work for flows that overly distort the grid[HCB70].

1977 Lucy describes the basic concepts underlying Smoothed Particle Hydrodynamics, where the Monte Carlo method is used explicitly[Luc77].

1977 Gingold and Monaghan name the Smoothed Particle Hydrodynamics method, their work occurs in parallel to that of Lucy[GM77].

1983 Reeves introduces particle systems to the computer graphics literature. Armed with particles and pseudo-physics, Reeves demonstrated fire effects in Star Trek to generating images of plants produced with particles[Ree83].

1992 Monaghan writes a review detailing the Smoothed Particle Hydrodynamics method, which also serves as a good introduction to the topic[Mon92].

1995 Stam and Fiume describe representing smoke and fire using a method known as diffusion processes[SF95]. They mention that the full Navier Stokes equations is too much for machines of the time to handle and would complicate the life of animators. However the methods they present provide the desired visuals and could be run interactively.

1998 Radovitzky and Ortiz simulate fluids on a mesh, like Hirt, but remesh as the simulation progresses to eliminate the problems that arise from distortions in the mesh[RO98].

2003 Müller, Charypar, and Gross present an interactive fluid simulation based upon SPH[MCG03].

2005 Clavet, Beaudoin, and Poulin describe a method based on SPH to simulate visoelastic fluids by inserting and removing springs between the simulated particles[CBP05].

2005 Selle, Rasmussen, and Fedkiw simulate turbulent flow by storing vorticity information within particles. They opted to not use grids due to the artificial dissipation[SRF05].

2008 Hong, House, and Keyser do away with the SPH method and transfer the particle data to grids to determine change in velocity of the flow while minimizing the number of particles in the system by adding particles where fine details occur and removing particles where there isn't as much detail[HHK08].

2008 Losasso, Talton, Kwatra, and Fedkiw combine a grid based solver to model macroscopic fluid flow and binds it to an SPH based solver for the microscopic fluid effects[LTKF08].

2009 Sin, Bargeil, and Hodgins simulated fluid flow by creating a Voronoi diagram from the particles in order to do the pressure projection step for incompressible flow[SBH09].

2.5 Conclusion

We have shown a method that can be used to simulate fluid flow on rectangular grids by showing how most of the underlying equations can be obtained and how to discretize them. By providing references to related works from decades ago to modern times we have contextualized what we have done.

Due to time constraints, there are other methods to simulating fluid flow that we would have enjoyed spending more time with. These include: the use of adaptive meshes, triangular meshes whose geometry changes over time to accommodate finer details within the flow; and simulating the fluid flow in Fourier space. We leave the exploration of both as future work.

For future work, we would also like to spend more time on SPH methods in order to see how they may be used in interactive simulations.

Chapter 3

Simulation Extensions

Topics within the previous chapter – conservation of matter, viscuous dampening, and advection of matter – compose a minimal set of properties needed to simulate incompressible fluid flow. Our focus now shifts to adding more properties to the simulated flow to obtain a wider range of behaviours.

First: one of the consequences of the artificial diffusion arising from simulating fluid flow on coarse grids and time steps is the dampening of fine swirls characteristic of smoke. What we wish would visually look like billowing smoke, appears to be a dense, stationary blob of smoke. We show a method that neutralizes the artificial dissipation of the fine swirls.

Second: Fluids normally do not exist in their own world, in contrast they interact with other fluids such as air and solids. We describe a simple, yet very innacurate, way to add such interaction to our digital flow.

Third: Denser fluids should sink while lighter fluids float above. We illustrate how this layering behaviour can be simulated despite our incompressible flow.

Fourth: Beyond the small sampling of extensions provided in this chapter, we guide the interested reader to additional behaviours described in the literature.

3.1 Amplifying Swirls

Creeping into each time step of the fluid simulation are omnipresent numerical errors. They lurk in the mapping between unrepresentable numbers to their discrete approximations, concealed between coarse grid cells, thriving on estimates that speed up the simulation.

Victim to these numerical errors is the fine swirl, characteristic of billowing smoke rolling across the horizon. What remains of the swirls is a terrible sight, fine rotational detail blurred out of existence before the slightest visual impact can be made. Like goo, the remains of our attempt at simulating smoke rises into the air.

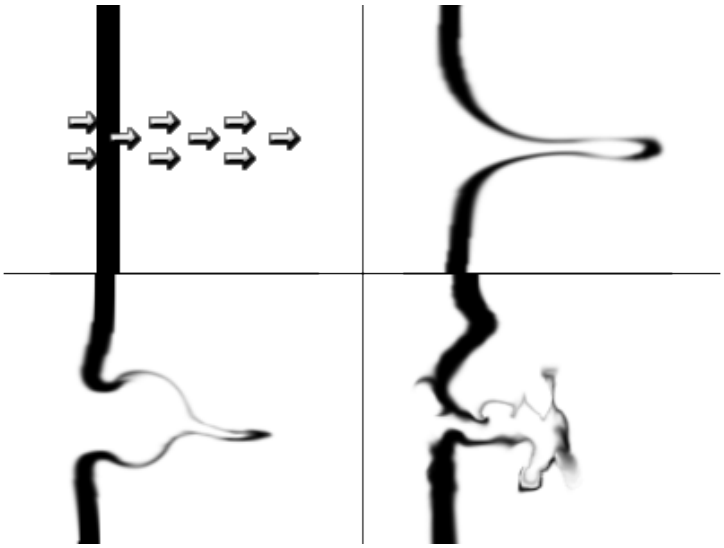

Figure 3.1: Vorticity Confinement Examples
Each example consists of a vertical line of colour drawn down, and a horizontal line of velocities drawn towards the right - as shown in the top-left image. On the top-right there is no vorticity confinement, and the force simply pushes into the line and stretches it. In the bottom-left $\epsilon = 1$ *which restores some of the lost vorticity resulting in a more curved-like result. The last example is* $\epsilon = 10$ *which exaggerates the swirling motion of the fluid.*

Enter Dr. Steinhoff, hailing from the University of Tennessee. Motivated by the difficulty of accurately simulating vortices arising from basic turbulent flow. Clever in his solution: to devise an "additive velocity correction" that undoes the effect of artificial diffusion on fine swirls[SU94].

Let $\boldsymbol{u} \in \mathbb{R}^2$ be a rectangular vector field that describes the direction and speed of the flow of the fluid.

$\epsilon \in \mathbb{R}^+$ is a scalar that describes how much compensation for the dampening of the swirls to apply.

$\Delta\tau \in \mathbb{I}^+, \Delta\tau > 0$ is the size of the grid. We assume that $\Delta\tau = 1$.

$\boldsymbol{\omega} \in \mathbb{R}^3$ is a vector field that describes the rotational attributes of $\boldsymbol{u}$.

$\boldsymbol{\eta} \in \mathbb{R}^3$ is a vector field that points in the direction of increasing magnitudes of rotational behaviour within the digital flow.

$\boldsymbol{N} \in \mathbb{R}^3, |\boldsymbol{N}| = 1$ is identical to $\boldsymbol{\eta}$ except the vectors are normalized.

$\boldsymbol{f_{conf}}$ is the force to apply. Note that the force is applied along ridges of vorticity intensity – the perpendicular to the paddle rotating along the axis and the direction pointing to increases in vorticity.

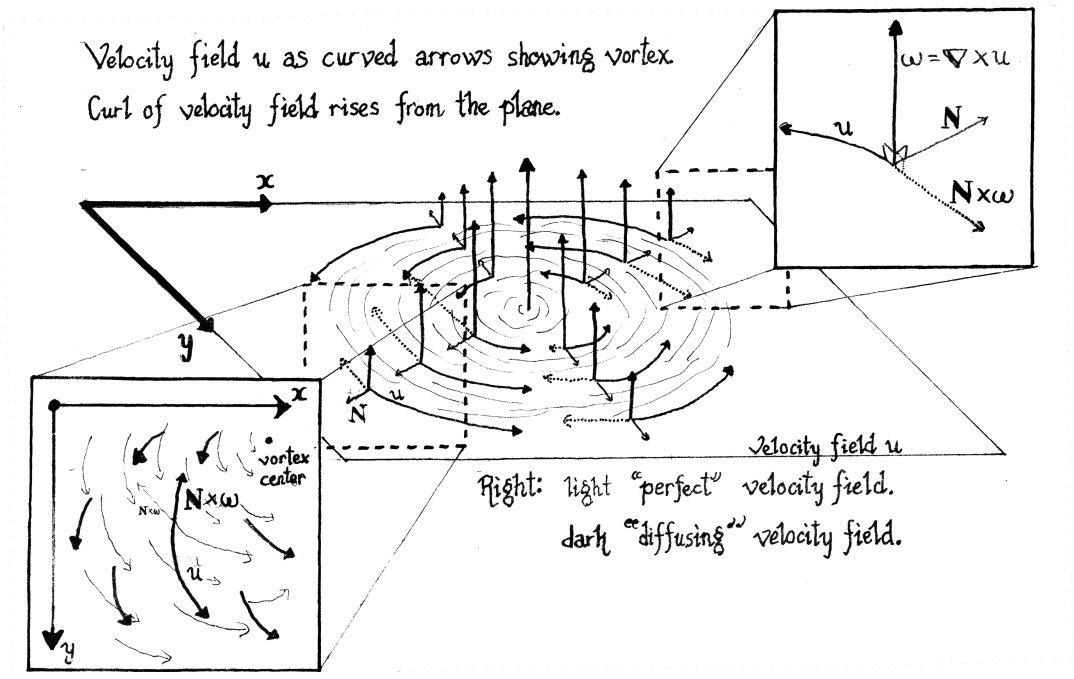

Figure 3.2: Vorticity Confinement Visual
An image depicting what vorticity confinement does.

Vorticity confinement can be written as the following series of equations:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{3.1}$$

$$\boldsymbol{\eta} = \nabla |\boldsymbol{\omega}| \tag{3.2}$$

$$\boldsymbol{N} = \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|} \tag{3.3}$$

$$\boldsymbol{f_{conf}} = \epsilon \Delta\tau \left(\boldsymbol{N} \times \boldsymbol{\omega}\right) \tag{3.4}$$

The curl and the cross-product in (3.1) and (3.4) imply that the presented equations for vorticity confinement operate in 3-space. Fortunately, when dealing with 2-space certain simplifications, as described below, can be done.

The transformation between 3-space and 2-space will be defined as follows:

$$\boldsymbol{u}^{x,y,z} = (\boldsymbol{u}^x, \boldsymbol{u}^y, 0) \text{ where } x, y, z \in \mathbb{R} \tag{3.5}$$

(3.5) implies:

$$\frac{\partial \boldsymbol{u}^n}{\partial z} = 0 \text{ where } n \in \{x, y\} \tag{3.6}$$

$$\frac{\partial \boldsymbol{u}^z}{\partial n} = 0 \text{ where } n \in \{x, y\} \tag{3.7}$$

(3.6) is true since the velocity $\boldsymbol{u}$ is constant for any value on the z axis, as described by (3.5). (3.7) is true since $\boldsymbol{u}^z = 0$, therefore it can't change.

Now, if (3.1) is expanded as below:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u} \tag{3.8}$$

$$= \left(\frac{\partial \boldsymbol{u}^z}{\partial y} - \frac{\partial \boldsymbol{u}^y}{\partial z}\right)\boldsymbol{i} + \left(\frac{\partial \boldsymbol{u}^x}{\partial z} - \frac{\partial \boldsymbol{u}^z}{\partial x}\right)\boldsymbol{j} + \left(\frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right)\boldsymbol{k} \tag{3.9}$$

$$= (0-0)\,\boldsymbol{i} + (0-0)\,\boldsymbol{j} + \left(\frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right)\boldsymbol{k} \tag{3.10}$$

$$= \left(0, 0, \frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right) \tag{3.11}$$

Since the values on the x and y axis become zero only a single grid is needed to store the value of $\boldsymbol{\omega}$.

Looking at (3.11) it can be noted that the z axis plays no part in the result:

$$\boldsymbol{\omega}^{x,y,z} = (0, 0, \boldsymbol{\omega}^z) \tag{3.12}$$

(3.2) can be expanded as follows:

$$\boldsymbol{\eta} = \nabla\,|\boldsymbol{\omega}| \tag{3.13}$$

$$= \left(\frac{\partial|\boldsymbol{\omega}|}{\partial x}, \frac{\partial|\boldsymbol{\omega}|}{\partial y}, \frac{\partial|\boldsymbol{\omega}|}{\partial z}\right) \tag{3.14}$$

$$= \left(\frac{\partial|\boldsymbol{\omega}|}{\partial x}, \frac{\partial|\boldsymbol{\omega}|}{\partial y}, 0\right) \tag{3.15}$$

If we let $r = |\boldsymbol{\eta}|$ then (3.3) becomes:

$$\begin{aligned} \boldsymbol{N} &= \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|} && (3.16)\\ &= \frac{\left(\frac{\partial|\boldsymbol{\omega}|}{\partial x}, \frac{\partial|\boldsymbol{\omega}|}{\partial y}, 0\right)}{r} && (3.17)\\ &= \left(\frac{\partial|\boldsymbol{\omega}|}{r\partial x}, \frac{\partial|\boldsymbol{\omega}|}{r\partial y}, 0\right) && (3.18) \end{aligned}$$

The final cross product found in (3.4) becomes:

$$\begin{aligned} \boldsymbol{f_{conf}} &= \epsilon h\left(\boldsymbol{N} \times \boldsymbol{\omega}\right) && (3.19)\\ &= \epsilon h\left(\left(\frac{\partial|\boldsymbol{\omega}|}{r\partial x}, \frac{\partial|\boldsymbol{\omega}|}{r\partial y}, 0\right) \times \left(0, 0, \frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right)\right) && (3.20)\\ &= \epsilon h\left(\frac{\partial|\boldsymbol{\omega}|}{r\partial y}\left(\frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right), -\frac{\partial|\boldsymbol{\omega}|}{r\partial x}\left(\frac{\partial \boldsymbol{u}^y}{\partial x} - \frac{\partial \boldsymbol{u}^x}{\partial y}\right), 0\right) && (3.21) \end{aligned}$$

After using finite differencing, $\boldsymbol{f_{conf}}$ can easily be added as an external force that restores fine swirls.

3.2 Fluid Boundaries

Consider some coffee bounded by a ceramic mug and air. Drill a hole on the side of the mug. Watch as coffee pours out of the hole, splashing onto the floor.

Capturing the essence of the coffee spill described in the previous paragraph requires the concept of free surfaces. Specifically, we must model how coffee – in general any fluid – interacts with air.

Following the lead of [CLT08], we assume air has a minimal effect on the digital flow. So minimal that we say air is equivalent to empty space which doesn't affect the fluid flow at all.

Two is the number of questions that must be answered when dealing with free surfaces:

I. What happens to the fluid at the interface – the border where fluid and air meet?

II. How do we follow the continually evolving topology of the regions occupied by our simulated fluid matter?

"Free-slip boundary conditions" answers the first question since such boundary conditions allow the fluid to flow out of it's "container", or current region in the case of free surfaces.

"Level set method" answers the second question.

Figure 3.3: Example of Free Surfaces
Screenshot of the fluid simulation dealing with free surfaces. Fluid is white, lack of fluid is black. In this example, fluid was crashing into itself thanks to jumbo-sized velocities being applied to the flow with an unhealthy dose of vorticity confinement.

Imagine the map of an island. At the shore, where land and water meet, we colour the map cyan[1]. The further away from the shore, the purer the blue of the water becomes on the map. Similarly, the green becomes purer the more inland a region is. Quickly looking at such a map allows us to easily determine which points are in the water, which are inland, and we can approximately figure out how far each point is from the shore.

Suppose we replaced colours with numbers. 0 Being cyan, 1 pure green, and -1 pure blue. We would end up with a rectangular scalar field in $\mathbb{R}$ with values within the range $[-1, 1]$ known as the "level set".

In addition, we can advect the level set like any other field along the digital flow's velocity field. This allows us to keep track of the evolving topology of the fluid. To compensate for artificial dissipation, we amplify the values in the level set to make sure that synthetic dampening does not diffuse the fluid boundaries into an unrecognizable mess.

Fields should have valid values outside the regions denoted as containing fluid by the level set. The reason, as we discovered to our detriment, is that the synthetic diffusion makes information spread. When we encode colour into a field to advect and put black where there is no density, black tends to diffuse into the regions which are marked as having density.

There are other ways to keep track of the regions that contain fluid. Harlow and Welch[HW65] used particles to represent densities and also keep track of where the fluid was. Foster and Fedkiw[FF01] coupled particles with the level set in order to obtain more details about the

[1]Colour obtained from mixing blue and green. Very similar to the colour of the sky near the horizon at noon on a clear sunny day.

interface. Enright, Marschner, and Fedkiw[EMF02] used particles near the interface of the flow to correct the level set.

3.3 Buoyancy

Simultaneously pour oil and water into a transparent glass container to obtain two definable layers; the lighter oil on top and heavier water below. A density column.

Let m be the mass of the fluid and ρ the density over a volume V. We have[SJ04]:

$$m = \rho V \tag{3.22}$$

Usually, as temperature decreases the density increases since the fluid takes up more volume for the same mass[2], known as thermal expansion[SJ04].

Continuing down the road of approximations, we use temperature as a driving force to simulate buoyancy.

Let $T \in \mathbb{R}$ be a rectangular scalar field annotating the digital flow with temperature information.

$\boldsymbol{k} \in \mathbb{R}^2, |\boldsymbol{k}| = 1$ is a vector pointing down relative to the direction of gravity.

$\alpha \in \mathbb{R}$ specifies the strength of gravity.

$\rho \in \mathbb{R}$ is a rectangular scalar field specifying distribution of density over the simulation region. Usually $\rho = 1$.

$T_0 \in \mathbb{R}$ denotes the ambient temperature. For convenience, we set this value to 0.

$\beta \in \mathbb{R}$ is how powerful of an effect temperature should apply. Physically, the greater β, the greater the change in density between two fixed temperature values for the digital matter.

Like [FM97b], we model buoyancy with gravity as:

$$\boldsymbol{f}_{\text{buoyancy}} = (\alpha + \beta\rho(T - T_0))\boldsymbol{k} \tag{3.23}$$

3.4 Future Explorations

3.4.1 Multiple Fluids

Even though we treat air as having no effect on the final flow, in reality air does have an effect. Using multiple fluids, we could better model the complex interaction among multiple flows such as liquid and air.

[2] water turning to ice being a notable exception to the rule

3.4.2 Semi-Solids

Solids are not entirely alien to the world of fluids. We could add in a series of springs between the densities of the flow[GBO04]. We could use these springs to freeze digital fluids and to thaw them back into a liquid, for example.

3.4.3 Particles

The use of particles as a core component of the simulation is desirable as a fixed number of particles can be put into the system without fear of a synthetic loss or gain of fluid density.

With an improved pressure solver, this could solve the issue of fluid density being created and destroyed.

Unfortunately, we would potentially lose the aesthetic that results from using a continuous density field.

Chapter 4

Rendering

We define success by the visual impact of our digital flow, not the cleverness of the underlying algorithms. Visually, we present our flow as a rectangular grid of pixels. The transformation of our fluid flow into a pixelated form is this chapter's focus.

First: we show how the fields that make up our digital fluid can be aesthetically translated into a pixelated representation through simple mappings.

Second: we have experimented using cameras to reflect the surrounding world onto a display. Combined with a digital flow, we explain how the reflected world can flow away like goo.

Third: numerical dissipation due to our use of coarse grids diffuses the visual detail in our flow. We describe how particles can help preserve the illusion of fine details.

Finally: we supplement the aforementioned methods of rendering fluid flow by providing references to other methods described in the literature.

4.1 Internal Fields

Imagine a mysterious, transparent, fluid within a transparent spherical container. Squeezing the sides of the container makes the compressed fluid turn a shade of blue. Shaking the container reveals shades of red where the mysterious fluid exhibits the most motion. Our desire is to emulate this peculiar fluid's visual attributes[1].

We go over this method of rendering by first documenting how we transform a field into a colour field. Second, we provide an example describing how to obtain colour from velocity. Third, we describe the results that we have obtained. Last, we communicate a specific mapping we implemented for temperature.

[1] We would like to thank Adrian Freed for giving the idea of mapping velocity magnitude to colour.

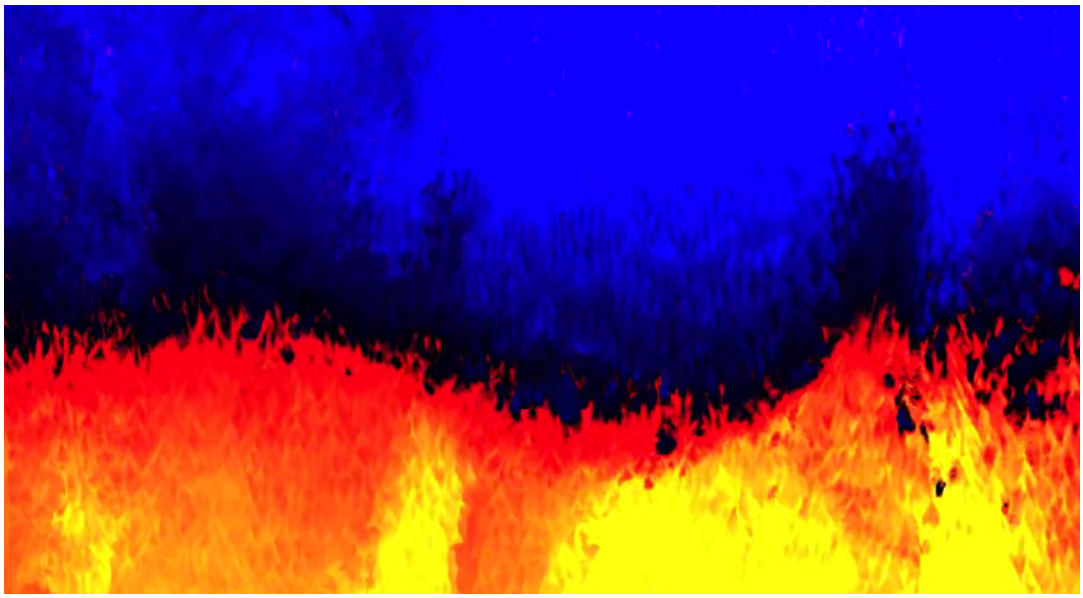

Figure 4.1: Using Pressure to Hint to Colour
In this image, the colour is very bright at the bottom since there is a lot of pressure. When the colour saturates, it goes from red to yellow. We uploaded a video to YouTube, see: `http://www.youtube.com/watch?v=HOdnycIykCc`

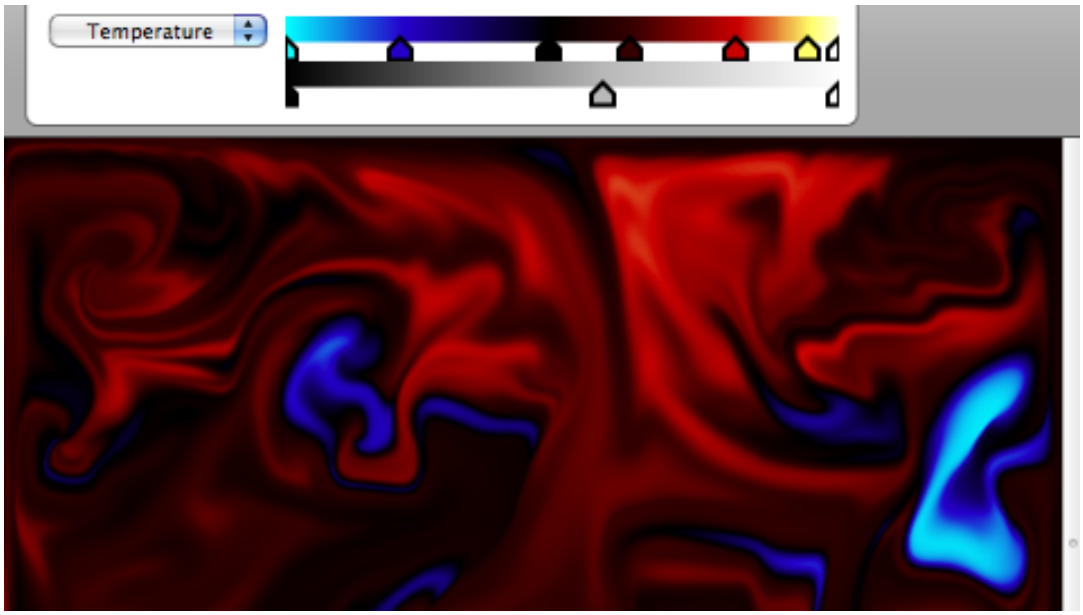

Figure 4.2: Temperature Gradients
Mapping temperature values to various colour gradients allows for visuals based on the black-body spectrum.

Let $\mathbb{D}$ be a set of all vectors in $\mathbb{R}^3$ that represent a RGB (red, green, blue) colour. The range of visible colour values $\mathbb{V} \subset \mathbb{D}$ where $\mathbb{V} := \{(r \in \mathbb{R}, g \in \mathbb{R}, b \in \mathbb{R}) \,|\, 0 \leq r, g, b \leq 1\}$. When displayed to the screen, vectors in $\mathbb{D}$ are mapped to the nearest vector in $\mathbb{V}$.

We define a function called $\boldsymbol{s} \in \mathbb{D}$ that maps a real number x to a colour interpolated between $\boldsymbol{c}_0 \in \mathbb{D}$ and $\boldsymbol{c}_1 \in \mathbb{D}$. We assume that x tends to be in the range $[0, 1]$, however we do not clamp values in order to allow for some exaggeration in the final colour:

$$\boldsymbol{s}(x, \boldsymbol{c}_0, \boldsymbol{c}_1) = \boldsymbol{c}_0 x + \boldsymbol{c_1}(1 - x) \tag{4.1}$$

$$\boldsymbol{s} : (\mathbb{R}, \mathbb{D}, \mathbb{D}) \to \mathbb{D} \tag{4.2}$$

Next in our escapade of defining random mappings, we let $k \in \mathbb{R}$ be a scale factor, that under ideal conditions restricts kx to the range $[0, 1]$. Then, we define $\boldsymbol{t}_c$ as:

$$\boldsymbol{t}_c(x, \boldsymbol{c}_0, \boldsymbol{c}_1, k) = \boldsymbol{s}(kx) \tag{4.3}$$

$$\boldsymbol{t}_c : (\mathbb{R}, \mathbb{D}, \mathbb{D}, \mathbb{R}) \to \mathbb{D} \tag{4.4}$$

$\boldsymbol{t}_c$ is the function we use to transform a scalar value $x \in \mathbb{R}$ into a colour value between $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$.

Imagine the places where the fluid flows faster it takes on colours reminiscent of fire. Specifically, very fast flows would appear white, yellow for slower flows, dark red for even slower flows, and black where the fluid is stationary. Let $\boldsymbol{u}$ be a velocity field where we obtain colours using $\boldsymbol{t}_c\left(|\boldsymbol{u}|, (1.0, 0.6, 0.1), (0.0, 0.0, 0.0), 0.1\right)$. Notice that at 0 velocity, we define a black colour. Then, at a velocity of 10, the colour is $(1.0, 0.6, 0.1)$ which is a slight orange. Since we don't limit the values, as the velocity further increases above 10, we saturate. For example, at a velocity of 20, our colour becomes $(2.0, 1.2, 0.2)$ where the closest visible colour is $(1.0, 1.0, 0.2)$ which appears yellowish. As the velocity increases, the colour becomes white. Similarly, we can obtain colours from pressure and temperature.

The colours we obtained from pressure, temperature, and velocity were composed through additive blending. We allow for negative colours so we may remove colours from the scene. For example, high pressure could subtract colour.

Velocity can be used to emphasize areas of high and low motion. Velocity has an interesting aesthetic since the velocity field flows along itself - therefore waves of quick motion are very apparent.

Pressure can be used to emphasize points where the fluid is pushing against itself. Such as places where high-velocity waves crash into walls or slower parts of the fluid flow. Providing a highlight over places of high velocity can easily be done by colouring pressure.

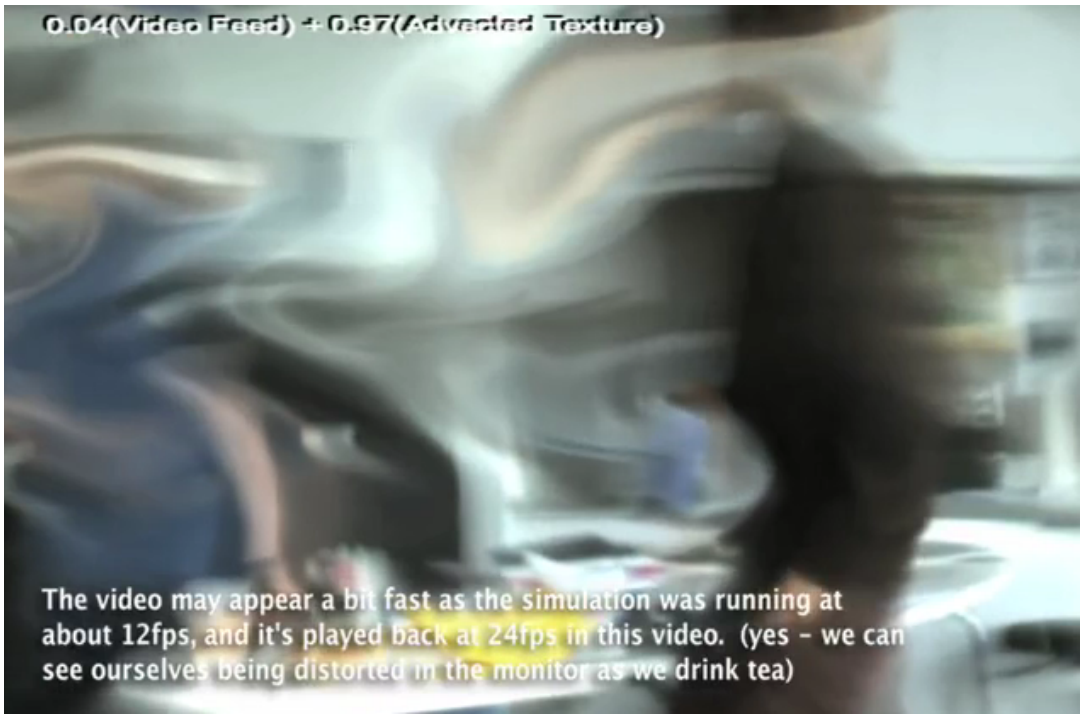

Figure 4.3: Tea-Time Fluid
Here we see ourselves drink tea as our bodies and surroundings become distorted by the fluid flow. Video from ACM Multimedia 2009[SFR09]: `http://vimeo.com/10826801`. *Special thanks to Timothy Sutton and Josée-Anne Drolet who are visible in the picture and helped with the filming.*

Temperature is interesting to colour since it gives a hint as to what forces the system is applying at a given point. It emphasizes the motion of warm flow rising and cool flow lowering.

Last, we bring your attention to figure 4.2. In this case, we used gradients (or piece-wise linear functions) to interpolate among various colours specified at certain key temperatures. By providing a gradient similar to that of the black body spectrum, we can easily, in a non-physical manner, map temperature to colours reminiscent of the black body spectrum.

We have just described a method to visualize the fluid that can be used for aesthetic presentation and debugging purposes. Now we shift our attention to how we may utilize live video.

4.2 Mirrors

Surreal is the concept of mirrors presenting a flowing view of the world, imbued with the properties of a liquid. Swirling the contents of the mirror could invoke rotational behaviour of the reflected world around a central vortex.

Figure 4.4: Traffic Flow
With the velocity field adjusted based upon optical flow, we pointed a camera to the corner of Guy and St. Catherines from the EV building. Vehicles driving through the area would push into the image. Video from ACM Multimedia 2009[SFR09]: `http://vimeo.com/10826801`

We begin with a quick overview of the used equations followed by a description of the results corroborated by a few images.

Let $\boldsymbol{\rho} \in \mathbb{D}$ represent a colour field of a given fluid flow. Also let $\boldsymbol{c} \in \mathbb{D}$ be a field of video from a camera.

Let $h \in \mathbb{R}$ and $i \in \mathbb{R}$ be scaling factors. We define the evolution of our fluid's colour field as:

$$\frac{\partial \boldsymbol{\rho}}{\partial t} = (\nabla \cdot \boldsymbol{u})(i\boldsymbol{c} + h\boldsymbol{\rho}) \tag{4.5}$$

Specifically, we gradually blend in camera data into the colour field. This effect, although very simple, is quite effective. The videos referenced in figures 4.3 and 4.4 show the visual potential.

If $i + h > 1$ then the visuals saturate and potentially fill the screen white since more colour comes in than is discarded each frame. If $i + h < 1$ then the visuals become dark. We assume that $i + h = 1$ since pre-processing and post-processing are more effective methods to brighten and darken the resulting video feed.

If h is small then the world appears to flow. This is since the next frame is mostly based on the previous colour field and very little new information from the camera feed comes in allowing for pixels to travel a long distance before fading out.

Figure 4.5: Flowing Plant
The plant shape is a standard l-system, and the rendering of the plant is done through thousands of particles. The particles flow within the digital fluid, but are attracted to the plant shape. The background colours are derived from velocity and pressure. Video can be found on YouTube: `http://www.youtube.com/watch?v=sTFlgCkuy9Q`

As h increases, the reflected world feels more like a fluid surface, quickly returning to an unmoving state when left unperturbed.

In this section we have described a key part of a fluid mirror. With velocities from optical flow[HS81] added to the velocity field, we can have people interactively warp themselves in a digital mirror.

4.3 Particles

Mentally visualize a large pool filled with no more than 2cm depth of water. For our purposes, this sheet of water behaves identically to a planar fluid flow. Now, imagine a hundred quacking yellow rubber duckies floating in the pool. Waves in the pool give a hint to the direction of the flow, the duckies on the other hand give precise details of the direction and speed of the flow at a given point by tracing characteristic curves of the flow.

[Ree83] provides a good overview of what can be done with particle systems. Our system does not vary that much from any other particle system, but it does have a few original elements that we'll emphasize.

Each of our particles has an initial position, which we can change dynamically. Figure 4.5 shows that we can maintain the shape of an object as it gets carried away by the fluid. We do this by providing a force that pulls particles back to their initial position and we also forcibly

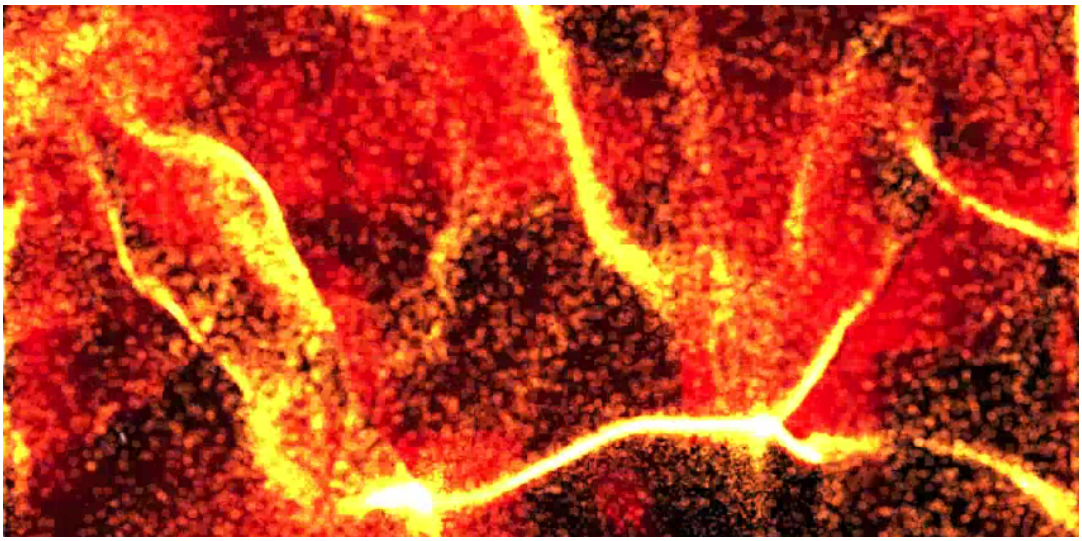

Figure 4.6: Particles in Fluid
The red particles become yellow and white when they overlay and saturate due to additive blending. This helps give the illusion of a 'hot' material. Video can be found on YouTube: `http://www.youtube.com/watch?v=Rmmw1vOAd8c`

Figure 4.7: Particles in Coloured Fluid
Red pressure and cyan velocities are augmented by bright particles at the edges of the waves. Video can be found on YouTube: `http://www.youtube.com/watch?v=Rmmw1vOAd8c`

Figure 4.8: Particles in Video
Live video warped by the fluid simulation augmented by particles. The video's brightness depends upon velocity and pressure. Video can be found on YouTube:
`http://www.youtube.com/watch?v=2s2Pis9631g`

move particle positions towards their initial positions. The latter was required in flows with chaotic velocity fields that would render any image unrecognizable.

We also allow for the particles to flow away and then reconstruct a shape in real time. We also allow for the shape to be re-specified so that the particles would flow, in a convincing way, to the new shape.

Our particles have random masses and speeds based upon user-specified ranges in order to provide variety to the displayed particles. The linked videos demonstrate the aesthetic potential.

4.4 Previous Work

We briefly list a few references that we believe to be very interesting and complement the text of this chapter.

1965 Harlow and Welch[HW65] used multitudes of massless marker particles to visualize fluid flow on a coarse grid.

1983 Reeves[Ree83] introduces particle systems to the computer graphics literature. Many methods currently used by particle systems are described in this paper.

1990 Kass and Miller[KM90] describe simulating and rendering of a height-field by simplifying the Navier Stokes equations to the 2D wave equation.

1996 Foster and Metaxas attempted three ways to rasterize the fluid data[FM96]. First they used massless marker particles, second they only used particles at the free surfaces, and third was to use height fields. The data from the visuals was used to fill in the density field for a 3D simulation of the fluid.

1997 Foster and Metaxas focused on marker particles flowing along the fluid[FM97b]. The particles were used to create a "density map" that was used for ray casting. For added details, secondary rays were sent to the light sources to get self-shadowing.

2001 Foster and Fedkiw rendered isocontours[FF01].

2006 Treuille, Lewis, and Popović used solid objects within the fluid to denote the flow. Like leaves in the air[TLP06]. For continuous fields, they did semi-lagrangian advection for that field.

Chapter 5

Execution Environment

Previously, we focused on equations describing fluid flow broken down into bite-sized kernels to be executed by some generic computational hardware. Driven by our insatiable desire for ever more detail in our interactive digital flow, we find ourselves trying to reorganize our code to better suit the execution environment. Often we found ourselves "optimizing code" to the detriment of legibility.

First: we introduce key ideas behind reorganizing code for faster execution speed through generalizations and metaphorical widget factories.

Second: we show how gaming the way the CPU executes code and processes data can lead to increased execution speed.

Third: we describe how I.B.M.'s Cell Broadband Engine processor differs from the CPU. We explain how these differences can lead to different strategies to accelerate program execution.

Fourth: we show how to tap into the currently fashionable source of considerable computational power, the GPU. We outline where the added performance comes from and the resulting pitfalls.

Last: to guide the implementation we look at the performance characteristics of the discretized Navier Stokes equations.

5.1 Preliminaries

Instead of jumping directly into the specifics about the various processors, we will first explain a few shared concepts. We use two fictitious widget producing factories, which we will refer to as Factory A and Factory B, as metaphors for describing what the hardware does.

5.1.1 Terminology

Before we describe a few ideas common to the examined processors, we introduce some terminology that relate to future discussions.

Big-O notation Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ be functions. Let $C \in \mathbb{R}$ and $k \in \mathbb{R}$ be constants. We say $f(x)$ is big O of $g(x)$ if $|f(x)| = C|g(x)|$ where $x > k$[Ros02], in other words $f(x)$ approximates the growth of $g(x)$. A common shorthand for "$f(x)$ is big O of $g(x)$" is to write "$f(x) = O(g(x))$". For our purposes, g typically maps the amount of work to the needed effort. Big-O provides us a way to compare how much effort is needed as the amount of input grows for different algorithms.

Hot-spots A common rule of thumb, similar to the Pareto Principle, states that 90% of execution time of a computer program is spent in 10% of the code[Chi01]. This 10% is what we refer to as hot spots. Hot-spots are where we focus our effort when we want to optimize code for speed.

Latency The amount of time elapsed from the time of request to completion of a given task[Int09]. Within the context of [Int09], time is in cycles and a task is an instruction.

Processor A mindless technological device, to which most of this chapter is dedicated, that blindly executes streams of instructions.

Flop "Floating-point operations per second"[App07]. A metric used to measure how fast an application is running and the potential speed of a processor.

Throughput The amount of time after one tasks starts that another identical task may begin[Int09][1] the number of tasks executed in parallel without affecting response time[NL06], or "the amount of material or items passing through a system or process"[App07]. Combining the above definitions, throughput is the number of tasks that can be completed in a given amount of time.

Optimization Within the context of software, the act of altering code, manually or automatically, with a specific goal in mind. For example, we could optimize code as to reduce the number of instructions used. Another metric, which is the focus of this chapter, is to optimize for reduced execution times.

Stride The distance between two consecutive elements along a given dimension in an n-dimensional array, usually measured in bytes. Consider a two dimensional array of bytes with width $w \in \mathbb{I}$ and height $h \in \mathbb{I}$. If the array is stored in a list then we could define the index in the list as $x + y * w$ for some arbitrary (x, y) coordinate such that $0 \leq x \in \mathbb{I} < w$ and $0 \leq y \in \mathbb{I} < h$. Along the x-axis, the stride is 1 byte, whereas along the y-axis the stride is w bytes. Another example is an element within a structure, where the stride equals the size of the structure.

[1] In context of [Int09], time is in cycles, and tasks are instructions.

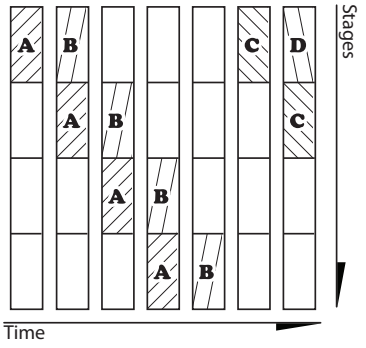

Figure 5.1: Instruction Pipeline
As time progresses, instructions go through the various stages of the pipeline. Since instruction C *depends on the results of instruction* B*,* C *may not start until* B *finishes, thus discarding potential throughput.*

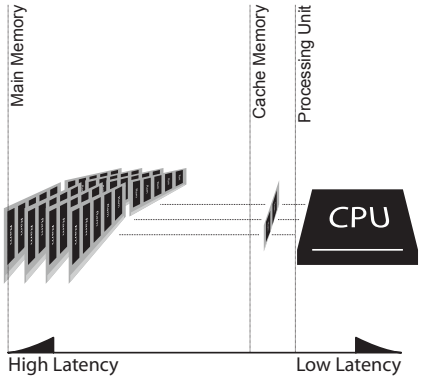

Figure 5.2: Memory Hierarchy
Cache memory is smaller, closer to the processor, and faster than the abundant main memory.

Packed When the stride equals the size of the given element. For example, a list of bytes with stride of 1 byte is packed.

Alignment We say an item is aligned to a given number of bytes n if the memory address m that the item is located at is divisible by n. Imagine the code fragment '`int k = 0;`', then `k` is aligned to a 16 byte boundary if `&k` $mod\ 16 \equiv 0$ where `&k` is the address of variable `k`.

KiB A kibibyte. n KiB $= 2^{10}n$ bytes[IEC10].

MiB A mebibyte. n MiB $= 2^{20}n$ bytes[IEC10].

Hertz The number of cycles (or samples) per second. Abbreviated as Hz.

5.1.2 Instruction Pipeline

Having looked at a few definitions, we return to our example widget factories. For our analogies to make sense, our imaginary factories only produce widgets when they receive an order.

Raw materials flow to the various stages of Factory B's assembly line. Each worker takes an incomplete widget and brings it closer to completion. Each worker depends on another worker as they sequentially work in parallel. We say sequentially since the construction of widgets is strictly a sequential task. Parallelism arises from all of the workers working in parallel.

Likewise, computational devices transform raw instructions into processed, or executed, instructions through a series of stages such as fetch, decode, and execute[NL06]. This series of stages is known as the instruction pipeline.

In an attempt to maximize instruction throughput, using a technique known as instruction pipelining, the processor will not wait for an instruction to complete before beginning execution of the next instruction. After an instruction has entered and completed the first stage of the pipeline, the next instruction can start if all dependencies have been satisfied, as illustrated in Figure 5.1.

This parallelism requires a conscious effort to see sequential computer code as being executed in parallel. In other words, we wish to minimize the data dependencies among nearby instructions. Carefully taking advantage of this hardware feature can yield great improvements in execution time.

Consider the following function that sums the digits from 1 to n. 1.20 seconds are required to run the function when $n = 1000000000$ on a PowerPC G5:

```
int sumOfDigits(int n)
{
  int s = 0;
  for (int x=1; x<=n; x++)
    s += x;

  return s;
}
```

Each iteration within the loop depends on the previous iteration. Rewriting the code to take advantage of the processor's pipeline reduces the execution time to 0.50 seconds on the same hardware[2]:

```
int sumOfDigits(int n)
{
  int a=0, b=0, c=0, d=0;
  for (int x=1; x<=n; x+=4)
  {
    a += x;
    b += x+1;
    c += x+2;
    d += x+3;
  }

  return a+b+c+d;
}
```

[2]Strictly speaking, this test is misleading given the G5 executes code out of order. Despite this fallacy, we believe that the results are still relevant to the current discussion. We cover out-of-order execution within the next section.

Although we have just increased the speed of execution of the function by over a factor of two, we should always ensure that our chosen algorithm is optimal for the given task. Consider the following which will always outperform our previous attempts:

```
int sumOfDigits(int n)
{
  return (n*n + n)/2;
}
```

We would like to emphasize how much easier it is to work with the last, shorter, function compared to the "optimized version". We shudder when we imagine the potential for the creation of spaghetti code for the sake of increased execution speed in more complex functions.

5.1.3 Data Access

If we assume that the factories only do work when a widget is requested, we can ask how much of the production time is spent obtaining raw materials.

To simplify the task of obtaining raw materials, Factory B has built two warehouses. A large warehouse far away where land and taxes are unbelievably cheap and a small warehouse attached to their widget factory.

In a logistics nightmare, trucks continually transport containers filled with raw materials from the large warehouse to the small warehouse and completed widgets back to the large warehouse. In the worst case scenario, a raw material is shipped from the large warehouse only upon realizing that it has run out at the factory and attached small warehouse.

Similarly, processors are like factories that transform data. Main memory is, relatively, far away from the processor. Fetching data from main memory takes a significant amount of time, so we have cache memory that is faster and typically located on the processor. Relating to our example, main memory is like the large warehouse and cache memory like the small warehouse.

Main memory can be conceptually divided into equal-sized groups known as cache lines. Cache lines are loaded into cache memory and evicted based on the changing memory access patterns of the software.

If software accesses memory that's within the cache, a cache hit occurs. In the case of a cache miss, the processor stalls waiting for the requested data to be fetched from main memory.

Gaming the cache is possible by knowing the rules governing what remains in cache and what gets evicted. In a synthetic case involving simulating the 2D wave equation with the main goal to emphasize the impact of cache misses, we have measured a greater increase in execution speed by working with the cache compared to throwing an additional core at the same problem.

5.1.4 Throughput and Latency

Factory A aggressively advertises how they can produce a widget within the hour. For Factory A, the latency to produce a widget is one hour and the throughput is a single widget per hour.

Factory B is quite silent in this regard since it takes them 24 hours to produce a widget. However, thanks to a modern production pipeline, they can begin working on a subsequent widget after the first half-hour of production. Due to this parallelism, throughput and latency are not functions of each other. They have a latency of 24 hours with a peak throughput of 2 widgets per hour.

If Factory A and Factory B were asked to produce 47 widgets, they would both produce 47 widgets in the same time-frame. As the number of desired widgets increases beyond 47, Factory B's greater throughput makes it faster despite Factory A's reduced latency.

Such situations involving throughput versus latency arise often in software. As an example, consider our synthetic example on page 58 where we split operations into four streams that could be executed in parallel. This idea could be expanded so that we would repeat the same computation four times; and be faster than if we invoked our initial function four times. For further details specific to the PowerPC G5, but generalizable, we refer the reader to [App08b]. We will provide some related information to the Core 2 processor in section 5.2.

5.1.5 Conclusion

Imagine the case where a computer program takes advantage of the processor's pipeline to the detriment of the cache. Cache misses would negate any gains in execution speed. Therefore execution speed can only be increased through a careful balance of keeping data local to the processor and maximizing instruction throughput.

Optimized code for a given algorithm is a solution to a puzzle whose pieces are the code bordered by the underlying hardware and software. This chapter focuses on the pieces of the puzzle.

5.2 The CPU

The CPU is the most familiar of the examined processors providing comparatively the most expressive freedom. Despite its familiarity, the CPU also has many interesting peculiarities. We are now tailoring the properties of a generic processor described in the previous section to better represent Intel's Core 2 Duo processor[3].

First: the instruction pipeline of Intel's processors is not as simple as the general one we described. We explain the relevant details.

[3]Unless otherwise specified, many of our discussions in this section are applicable to CPUs in general. Specific numbers apply to an Intel Core 2 Duo clocked at 3.06GHz with 6MiB of cache.

Second: we attempt to outline how accessing data on Intel's processors compares to our general case in the previous section.

Third: data-level parallelism can be explicitly specified in order to gain some additional speed.

Fourth: using multiple cores requires some form of synchronization. Understanding the primitives used to synchronize the threads can help in our search for additional speed.

Fifth: since most of our calculations are done on floating point numbers, we provide a general overview of how they are represented within the CPU.

Last; we briefly look at ideas surrounding profiling of code on Intel's Core 2 Duo.

Before we enter the main subject matter, we must confess that most of the optimizations we describe here should, under ideal conditions, be done by the compiler. Unfortunately, at the time of writing, the machine is unable to read the mind of the human ordering it around and therefore may require some assistance in the form of manual re-organization, or optimization, of code.

5.2.1 Instruction Execution

Consider the first n elements of an infinitely long sequential to-do list with tasks potentially exhibiting dependencies on previous tasks. Assuming we'd want to work on the tasks described within this list, we could do each task one by one in sequential order. Alternatively, we could be pro-active and attempt to do parts of tasks in parallel and be smarter about how the work gets done as long as the end results are the same.

Through what is known as out-of-order execution Intel's Core 2 processors have a similar pro-active desire to get work done faster. Like a team consisting of multiple specialized workers, the Core 2 contains multiple execution units. Each unit has a specific purpose, such as floating-point arithmetic. Some units run in a single cycle, some units are pipelined and there can have multiple of each type of unit[Int09].

The execution units run instruction derived micro-operations, μops. μops are stored in what is known as a reservation station until their inputs are available[Lev]. In addition, the reservation station schedules the μops for execution[Int09].

μops are obtained from machine instructions through processes known as micro and macro fusion. Micro-fusion splits an instruction into one or more μops to further expose parallelism. Macro-fusion fuses multiple instructions into a single μop for greater throughput.

Once μops pass through their respective execution units, they are stored in the reorder buffer and removed in the order that they were stored in the reservation station[Lev]. The reorder buffer is responsible for the architectural state and ordering of exceptions[Int09].

This out-of-order execution is great for sequential lists of instructions exhibiting few dependencies among adjacent instructions. Conditionally branching instructions, not being predictable,

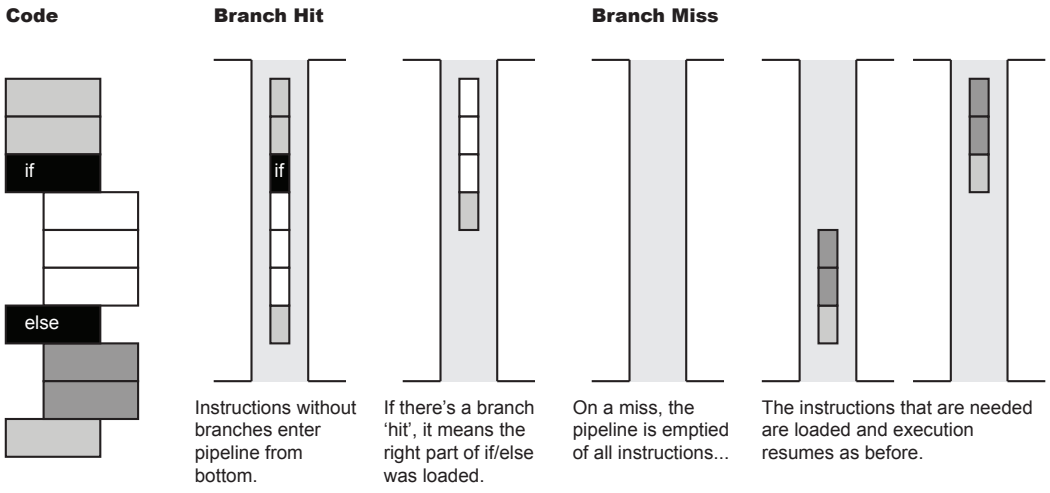

Figure 5.3: Branch Hit And Miss
The different shades of gray represent code outside the `if` *statement, within the* `if` *statement, and in the* `else` *statement.*

make executing code out of order more interesting due to the uncertainty about the next set of instructions to load.

Based on previous encounters with a specific branch instruction in the code, the CPU can guess whether a branch will be taken (if a jump occurs) or not using what is known as branch prediction. Lacking previous information on a given conditional branch, the Core 2 assumes that branches are not taken[Int09]. In addition, software can provide hints to whether the next branch will be taken or not.

Ambitiously, the Core 2 will accumulate results after the guessed branch into the reorder buffer. If the guess is not correct, the results and instruction pipeline are flushed and the CPU starts loading the correct stream of instructions and resumes processing. See figure 5.3 for a simplified visual that only shows the effect on the instruction pipeline.

Not related to the Core 2, certain CPUs are capable of what is known as Simultaneous Multi-Threading (SMT) – Hyper Threading (HT) on Intel processors. The idea is to maximize the use of the execution units by having two threads share the same core. If one thread pauses due to a cache miss, then the other thread can use the otherwise unused processing resources.

We gave a sampling of what the CPU does. However, there are many more things that could happen than what was mentioned here, such as loop caching. We can only suggest using a profiler to determine where to optimize and to ensure that the optimization worked as expected and that the execution time has actually been reduced.

5.2.2 Data Access

As we have previously seen, the Intel Core 2 has been designed to run as many instructions as quickly as possible. The multiple execution units and out of order execution allow the processor to quickly execute instructions, making data access even more important. Our advice is to remain predictable with regards to memory accesses. In this section, we will elaborate on this idea of predictability.

Recall that memory is copied into the cache in groups known as cache lines. The cache lines are 64 bytes long on Intel Core 2 processors. The Core 2 that we used has two caches, one L1 per core that is 32KiB and has 3 cycles of latency and one cycle of throughput. The slower L2 is shared among both cores and has a size of 6MiB, with a minimum latency of 15 cycles and throughput of 3 cycles.

On 512×512 simulation grid, using 4 byte floats, we require 4 MiB just for the velocity vectors. We require two velocity fields in the advection phase, much more than the available cache...

We can try providing hints to the CPU regarding which addresses of data in main memory we will want to access in the future. These hints do have an impact on execution speed and they can be ignored if the CPU is busy fetching other data due to cache misses. We would like to note that a PowerPC processor equipped with the Altivec extensions allows us to specify ranges of memory that will be sequentially accessed using a single instruction.

Knowing that the oldest untouched cache line gets evicted from cache might serve as a motivating factor to maximize the spatial locality of data accesses. For optimal speed, we could also restrict ourselves to a dataset that we know fits in the fastest cache, the 32KiB L1. Or the 6 MiB L2 if that is not possible. Intel suggests that half of the cache size divided by the number of logical threads using that cache be thought of as usable[Int09].

If restricting ourselves to data sets that fit in cache is not possible, we should take advantage of the automatic data prefetching available on the processor. We can trigger prefetching with two cache misses in the last level of cache, signaling to the hardware that it should start preemptively loading data in subsequent cache lines[Int09].

16 forward and 4 backward automatic prefetch streams are supported[Int09] – more than enough for most programs. This allows us to process data sets much larger than would fit in cache in a more efficient way.

Accessing memory addresses sequentially helps the CPU manage the cache. Maximizing the use of the accessed data before the chances that it gets evicted from cache are high helps prevent cache misses.

If simply accessing data was the only problem, then we'd be done. We were caught by surprise once we discovered the amount of overhead associated with writing data back out to main memory. Upon consulting the Intel Optimization Reference Manual[Int09], an indispensable

Function	Time (Intel Core 2)
localityOutOfOrder	30.33s
localityInOrder	3.41s
localitySimpleHit	3.33s
localityHybridHit	3.31s

Table 5.1: Performance lost from cache misses

resource, we realized that the system is optimized to write out 64 consecutive aligned bytes of data. Other cases lead to writing data back to main memory byte by byte.

Data Access Tests

Our curiousity led us to set up a few synthetic tests to measure the effects of accessing data on an Intel Core 2.

The tests takes place on a rectangular grid of width 4096 and height 4096, which we call $c \in \mathbb{R}$. Assume, that the computational kernel doing an arbitrary calculation on the grid is:

$$c_{x,y} \quad \approx \quad c_{x+1,y} - c_{x-1,y} + c_{x,y+1} - c_{x,y-1} \tag{5.1}$$

Our goal was to simulate the memory accesses of a Gauss-Seidel iterative solver, so we didn't create a test that actually converges to a solution. We decided to ignore boundary conditions for simplicity to focus on what we wanted to measure: the impact of accessing data.

We stored the data in a sequential list and we project the 2D (x, y) coordinates to a single integer:

$$f(x,y) \in \mathbb{I} \quad = \quad x + y * w \tag{5.2}$$

(5.2) can be generalized to convert a coordinate in an n dimensional array into an index of a list. If the n dimensional coordinate within c is $\boldsymbol{x} \in \mathbb{I} = \{x_0, x_1, x_2, ..., x_{n-1}\}$ and the length of each dimension of the array is $\boldsymbol{w} \in \mathbb{I} = \{w_0, w_1, w_2, ..., w_{n-1}\}$:

$$i = \sum_{j=0}^{n-1} x_j \prod_{k=0}^{j} w_k \tag{5.3}$$

All tests were compiled with optimizations similar to those that we would expect of a release application (`-O3` in GCC). We wanted results that reflected gains that would be noticed in an actual application given the effort the compiler already does in optimizing code.

We begin with the pathological case where we access data in the worst possible way. Our test machine had 6MiB of L2 cache. Each float was 4 bytes long. 4096 elements requires 16KiB, much larger than the 4KiB page size[4]. Therefore, for the first iteration along the y axis we would have a cache miss on each memory access. Since the total data size is 64MiB, cache hits would not occur on subsequent iterations of the outer-most loop. To ensure we hit the worst-case, we write back to non-contiguous memory addresses.

Unsurprisingly, our first test is the slowest:

```
void localityOutOfOrderTest(float *in_data)
{
  int x,y;
  int itr;
  for (itr=0; itr<40; itr++)     //Repeat 40 times
  {
    for (x=1; x<4095; x++)       //For each x...
    {
      for (y=1; y<4095; y++)     //For each y...
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }
}
```

Simply switching the inner-most loop with the middle loop ensures data is read and written to sequentially. Consulting table 5.1, we almost see a 10× gain in speed.

```
void localityInOrderTest(float *in_data)
{
  int x,y;
  int itr;
  for (itr=0; itr<40; itr++)
  {
    for (y=1; y<4095; y++)
    {
      for (x=1; x<4095; x++)     //x has smaller 'stride'
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
```

[4]Automatic stride detection for prefetching data does not work across pages

localityOutOfOrderTest			30.33s	localityInOrderTest			3.41s
4097			cache miss (128)	4097			cache miss (128)
8193	16376	store (4)	cache miss (128)	4098	4		
12289	16376	store (4)	cache miss (128)	4099	4		
16385	16376	store (4)	cache miss (128)	4100	4		
20481	16376	store (4)	cache miss (128)	4101	4		
24577	16376	store (4)	cache miss (128)	4102	4		
28673	16376	store (4)	cache miss (128)	4103	4		
32769	16376	store (4)	cache miss (128)	4104	4		
36865	16376	store (4)	cache miss (128)	4105	4		
40961	16376	store (4)	cache miss (128)	4106	4	store (40)*	
	stride (bytes)				stride (bytes)		

*** Optimal store for Intel processors is exactly 64 bytes, else it does several partial transactions.**
*** Each cache miss fetches 128 bytes of 128-byte aligned data from main memory on Intel processors.**

Table 5.2: First ten array access indices for sample functions.
For an Intel processor, each cache miss results in 128 bytes being fetched from main memory into the CPU.

```
    }
  }
}
```

Consider that in our second test, if we had just completed the loop where y is 10 and itr is 0, we could have computed y at 9 and i at 1 without changing the output value. Since there are more chances that y at 9 is still in cache, we get further improvements in performance. This method is known as a cache blocking technique[Int09].

Over multiple tests, the following program was consistently faster, although not by much:

```
void localitySimpleHitTest(float *in_data)
{
  int x,y;
  int itr;
  int cy;
  for (cy=1; cy<=40; cy++)      //For rows 1...40
  {
    for (y=cy; y>0; y--)        //Compute iterations
    {
      for (x=1; x<4095; x++)
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
```

```
      }
    }
  }

  for (cy=41; cy<4095; cy++)     //For rows 41...4095
  {
    for (itr=0; itr<40; itr++)  //Compute 40 iterations
    {
      y = cy-itr;                //Pointing y back so data
      for (x=1; x<4095; x++)     //dependencies respected
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }

  for (cy=1; cy<40; cy++)        //For rows 1...40
  {
    for (itr=cy; itr<40; itr++) //Compute missed iterations
    {
      y = 4095-cy-itr;
      for (x=1; x<4095; x++)
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }
}
```

On the last of our tests, we questioned if we could further minimize the number of cache misses. The answer was "yes"!

What we did was repeat the previous algorithm doing 10 iterations 4 times instead of all the 40 iterations at once. The avid reader will notice that on our test machine we simply reduced the number of cache misses on the L2 cache.

The method is simply pushing the idea of blocking even further.

```
void localityHybridHitTest(float *in_data)
{
  int x,y;
  int itr;
```

```
int cy;
int i2;
for (i2=0; i2<4; i2++)        //Repeat the previous 4 times
{
  for (cy=1; cy<=10; cy++)  //And do 10 iterations at a time
  {
    for (y=cy; y>0; y--)
    {
      for (x=1; x<4095; x++)
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }

  for (cy=11; cy<4095; cy++)
  {
    for (itr=0; itr<10; itr++)
    {
      y = cy-itr;
      for (x=1; x<4095; x++)
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }

  for (cy=1; cy<10; cy++)
  {
    for (itr=cy; itr<10; itr++)
    {
      y =4095- cy-itr;
      for (x=1; x<4095; x++)
      {
        in_data[x+y*4096] =
          (in_data[x+1+y*4096] - in_data[x-1+y*4096] +
          in_data[x+(y+1)*4096] - in_data[x+(y-1)*4096]);
      }
    }
  }
```

```
  }
}
```

As we will later show, Stam's[Sta99] solver, that we're basing our simulation on, is embarrassingly data-bound. We wrote these tests to inform the data access patterns of the implementation.

5.2.3 Data-Level Parallelism

So far, we have encountered implicit parallelism extracted from sequential streams of instructions executed by the CPU. One way to be explicit about the parallelism found in sequential code, supported by Intel's Core 2, is known as data level parallelism.

Data level parallelism refers to a single machine instruction operating on multiple data in parallel. These instructions are appropriately called single instruction multiple data (SIMD) instructions or vector instructions.

SIMD instructions operate on what's known, within context, as vectors. A vector is an n-byte long packed array of a primitive data type aligned to an n-byte boundary. n usually is 16, but can vary depending upon the processor. For example, the Multimedia Extensions (MMX) found in Intel's Pentium processors had 8 byte long vectors.

Vectors tend to be the input for SIMD instructions and these instructions usually output a vector. The desired operation is repeated for each respective element of the input vectors. For example, if we multiply vector (a, b, c, d) with (e, f, g, h) we obtain the vector (ae, bf, cg, dh).

Vectorization is the act of transforming program to use SIMD instructions. Since vectors have strict storage requirements, vectorizing code usually involves a rethinking of how data is stored in memory and operated on.

We make this clear using an example based on one from [I.B08] revolving around particles.

We can describe a particle using the following structure:

```
typedef struct {
  float x, y, z;
} point;

struct {
  point velocity;
  point position;
} particles[1024];
```

Opposed to the previous example consisting of an array of structures, we could also write the above as a structure of arrays:

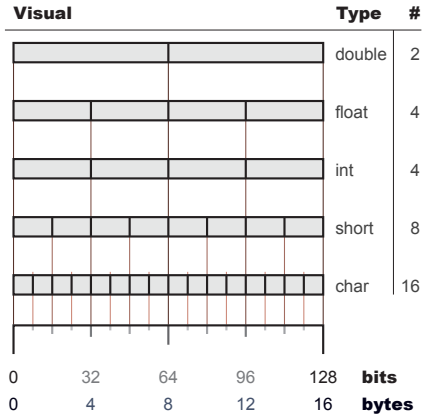

Figure 5.4: SIMD Data
SIMD data types are typically 128 bits wide, consisting of a '#' of smaller data types within. Each SIMD data type can be thought of as 128-bit long arrays of basic data types.

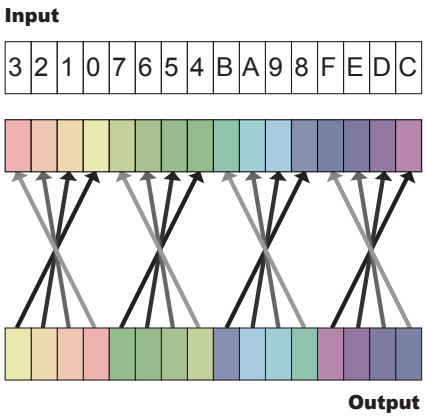

Figure 5.5: SIMD Permutation
Permutation, or swizzling, allows for the swapping of components of the vector. The hexadecimal digits on top specify which colored byte from the source should be picked for each destination byte.

```
typedef struct {
  float x[1024];
  float y[1024];
  float z[1024];
} point;

struct {
  point velocity;
  point position;
} particles;
```

Assume that we wish to add the velocity to the position each frame – same as Euler integration with a timestep of 1. For our example, we assume that a float is 4 bytes long, and a vector is 16 bytes.

With the structure of arrays, we would need to pad the `point` structure to 16 bytes if we wanted to use SIMD instructions to add `point` structures together.

Arrays of structures would not need the added padding to use vector instructions as components could be added individually. For example, `particles.position.x += particles.velocity.x` could add the first four elements of the x-velocities to the first four elements of the x-positions using a single SIMD instruction.

When reorganizing data to use SIMD instructions, we should question whether the time needed to translate between internal representations exceeds the potential gain in speed.

Another issue involves conditional branches where different elements in a vector would need to go through a different branch.

Consider a vector where different elements in the vector must undergo different operations. We could use sequential code, or we could compute all the operations for each element and use masks to combine the results. For small segments, computing all branches and merging the results using masks is preferable since there is no potential loss of speed due to a poorly predicted branch.

Moving away from the theory to the actual hardware, SSE (Streaming SIMD Extensions) and Altivec (also called Velocity Engine) are the names given to the extensions providing SIMD operations on Intel Core and PowerPC processors respectively. GCC also provides a processor independent way to access certain vector operations. Most compilers can attempt to automatically generate SIMD code from sequential code, however be sure to check the generated code as your mileage may vary.

SIMD operations can greatly increase the execution speed of an application. However, this increase in speed may not materialize due to moving data around. As well, reorganizing data structures may be problematic, as such having SIMD operations on the mind at a start of a project can have benefits in the long run.

5.2.4 Synchronization Primitives

Observing a team of antisocial workaholics, we notice an obscene lack of communication, rampant duplication of effort, and consistent clashes among workers. Similarly, multiple processing cores working on the same problem without any inter-communication is bound to fail.

One of the simplest of inter-processor communication mechanisms is known as 'Compare and Swap' (CAS). Compare and swap takes as input an address, an assumed existing value, and a value to set. If the data at the given address equals the assumed value as seen by all caches, then the data pointed to by the input address is set to the new value atomically. The function can return failure if it could not atomically set the value or if the expected value was not found at the given address.

An abstraction of compare and swap is known as a 'spinlock'. Spinlocks have two methods, which we'll call `lock` and `unlock`. Each `lock` must be followed by a single `unlock` and `lock` can not be nested. `lock` loops indefinitely until the test for 0 to set to 1 succeeds. `unlock` tests for 1 and sets to 0 and should be guaranteed to succeed as other processors are spinning in the `lock` function. Any failure in `unlock` indicates that `lock`s and `unlock`s are not properly paired.

We refer to a mutex as an abstraction above spin locks. Instead of indefinitely spinning within the `lock` method, a mutex will give up and go in the OS's wait queue as to avoid monopolizing CPU resources [App06].

Spinlocks, typically found within the kernel, can be faster than a mutex if locks are used sparingly and do not lock resources for a long time. Of course, testing will reveal the actual results.

5.2.5 Floating-point arithmetic

Our application heavily relies on the floating-point capabilities of the CPU. Intel's Core 2 can do 8 floating point operations per cycle per core. To achieve such throughput, SIMD operations are required[Int09].

Internal representation of the numbers also plays a role on performance. Floating point numbers are composed of a sign, exponent, and mantissa. The sign is a single bit, 0 for positive and 1 for negative. The exponent is represented as a value 2^{n-v} where v is a constant that varies depending upon the representation and n is the value that is stored. The mantissa is a value of the form $1 + \frac{1}{2^n}$ where n is the value that is stored. Obviously, not all numbers can be accurately represented using floating point values; however they can store very large and very small values.

To accommodate a larger range of numbers, the IEEE standard states that when the exponent is $0 = n - v$ special processing needs to be done since the representation of the floating-point value changes. These values that require further processing are called denormals. This processing is not always done in hardware, and can slow down the execution of an application. Fortunately, denormals can be turned off.

Denormals can be flushed to 0 by enabling what is known as DAZ (denormals are zero) mode by setting a bit within the MMXCR (multimedia extensions control register). A piece of sample code taken from[App08a] can clarify the operation:

```
int oldMXCSR = _mm_getcsr();
int newMXCSR = oldMXCSR | 0x8040;    //Denormals are zero + flush to zero
_mm_setcsr( newMXCSR );
```

Intel provides a good reference of the individual bits in the MMX control register [Int08].

5.2.6 Profiling

Our discussion of the CPU has thus far brought to light many pieces of the puzzle that must be arranged to increase the performance of our applications. Profiling is a means of ensuring that the we properly assembled the puzzle pieces and to hint to other interesting pieces.

Profiling is the act of measuring various aspects of the execution of a program. This could range from timing segments of code to measuring the number of cache misses.

The execution properties of code are modified to allow profiling such as modifying the program itself and/or running a separate process to gather statistics. This instrumentation has an

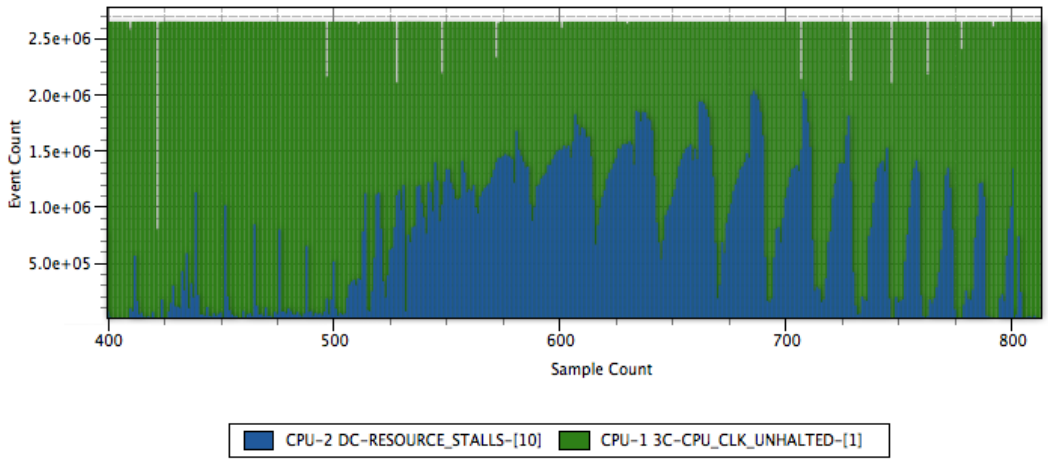

Figure 5.6: Example output from Shark
CPU_CLK_UNHALTED *counts the number of instructions, and* RESOURCE_STALLS-[10] *counts the number of branch misses. Overlaid on top of each other gives a good idea of how much performance is lost from branch misses. In this example, there is a spike of branch misses that severely affected application performance. The output was altered with Photoshop for legibility reasons.*

impact on the performance of the code. It can even hide bugs, for example adding a few statements to write to the console takes time and implicitly synchronizes among other threads that output potentially hiding synchronization problems in the code[PE09].

Apart from timers, we can listen in on counters on the CPU. Tools, such as Apple's Shark, attempt to associate the values within the counters with actual lines of code within a program. This provides a nifty way to see where most of the program time is spent, which parts of code exhibit the most cache misses, etc. On PowerPC G5s, Shark also does some static analysis showing any stalls due to dependencies among instructions.

Figure 5.6 gives an example of using Shark to obtain performance metrics. We have found Shark to be an invaluable tool during the development of our applications.

5.3 Cell Broadband Engine

Each core of Intel's Core 2 processor is capable of performing the same operations, therefore the cores are equally complex. Is it necessary that all cores have the same inate capacity to process tasks? I.B.M.'s Cell Broadband Engine processor, henceforth referred to as the Cell, forgoes few equally complex cores for many simple cores directed by a single complex core.

We speak of 'elements' rather than 'cores' within context of the Cell. Each element is a self contained combination of a processing unit, memory, and communication unit. The two types

of elements are called the Power Processing Element (PPE) and the Synergistic Processing Element (SPE)[PE09].

5.3.1 The Power Processing Element

The single Power Processing Element located on the Cell contains a simultaneous multi-threaded PowerPC core with two virtual threads. We shall not further elaborate on its execution characteristics since they are sufficiently similar to those of the Core 2. Another reason to avoid details regarding optimizing code for the Power Processing Element is that it is relatively weak.

5.3.2 The Synergistic Processing Elements

The Power Processing Element ideally delegates compute intensive operations to the Synergistic Processing Elements. Synergistic Processing Elements can be thought of as allocatable computational resources. We believe it to be easier to consider the Synergistic Processing Elements as separate machines, where code is explicitly uploaded and data explicitly transferred among the computational units. An application, using `spe_context_create`, can request a Synergistic Processing Element – and be guaranteed to always run on the same physical element within the system.

Each of the eight Synergistic Processing Elements contain some memory known as the Local Store (LS), a communication unit referred to as the Memory Flow Controller (MFC), and a processing unit called the Synergistic Processing Unit (SPU).

256KiB is the size of the Local Store. The Synergistic Processing Unit is only able to address data located within the local store. The implication is that the entire program and current data set must be at most 256KiB in size[I.B09].

The Memory Flow Controller adds DMA and mailbox capabilities. Using the Memory Flow Controller, programs executing on the Synergistic Processing Unit tend to DMA results of previous computations back to main memory and DMA data needed for future computations into the Local Store. All the while the Synergistic Processing Unit operates on data currently located within the Local Store. Embarrassingly data-bound applications will quickly hit data transfer limits between the Cell and main memory. We have found that compressing data transferred to and from main memory can benefit data-bound problems.

If compression is not feasible, then DMA operations can also occur between cores by moving data along the Element Interconnect Bus. Since the Element Interconnect Bus (EIB) is like a ring, elements that are physically closer to each other have reduced communication latencies. Testing, at application startup, the time needed to communicate among cores can be used as a metric to determine which tasks will be assigned to which elements[PE09].

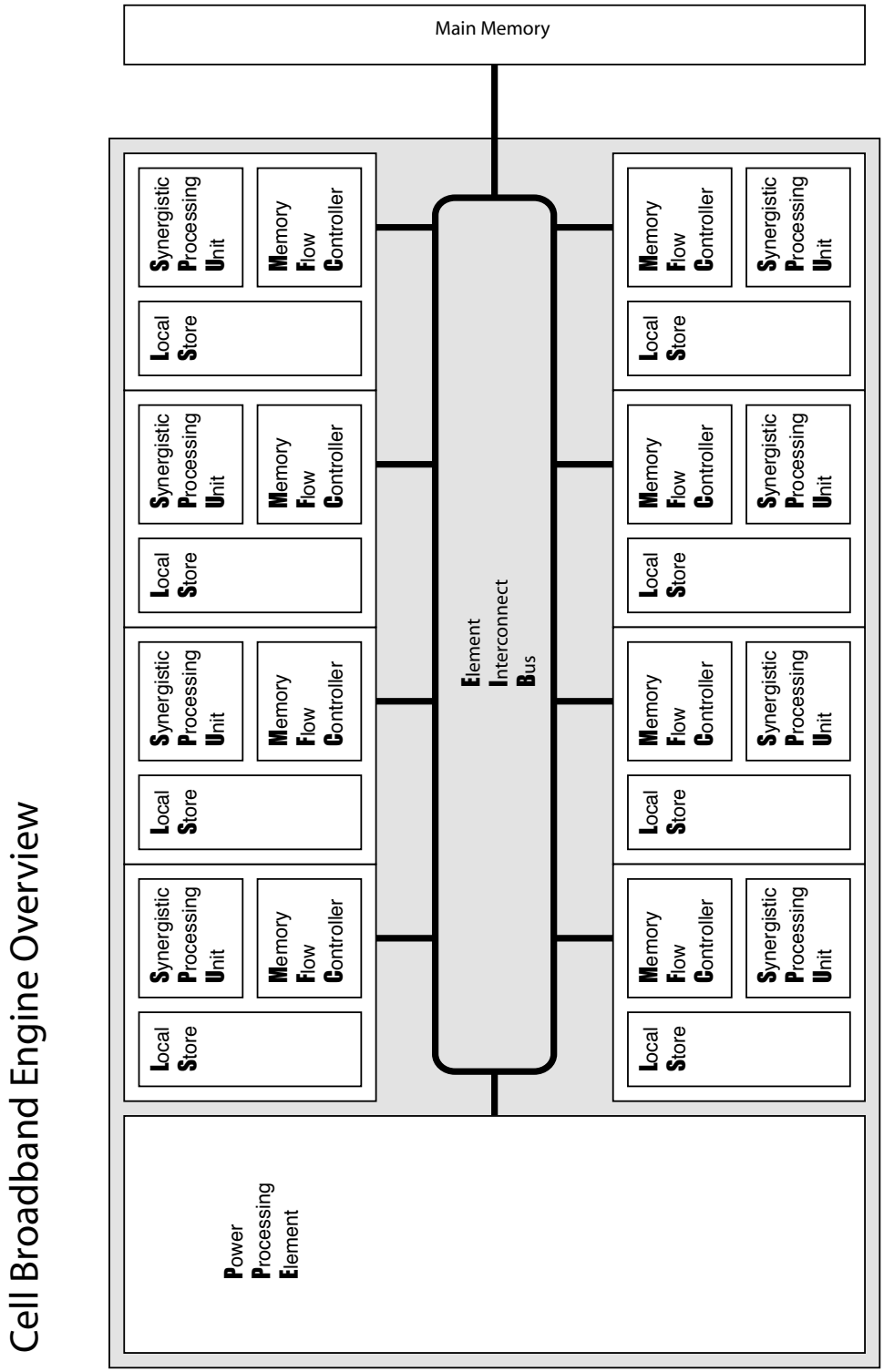

Figure 5.7: Layout of Cell Broadband Engine
A visual to accompany the text-based representation.

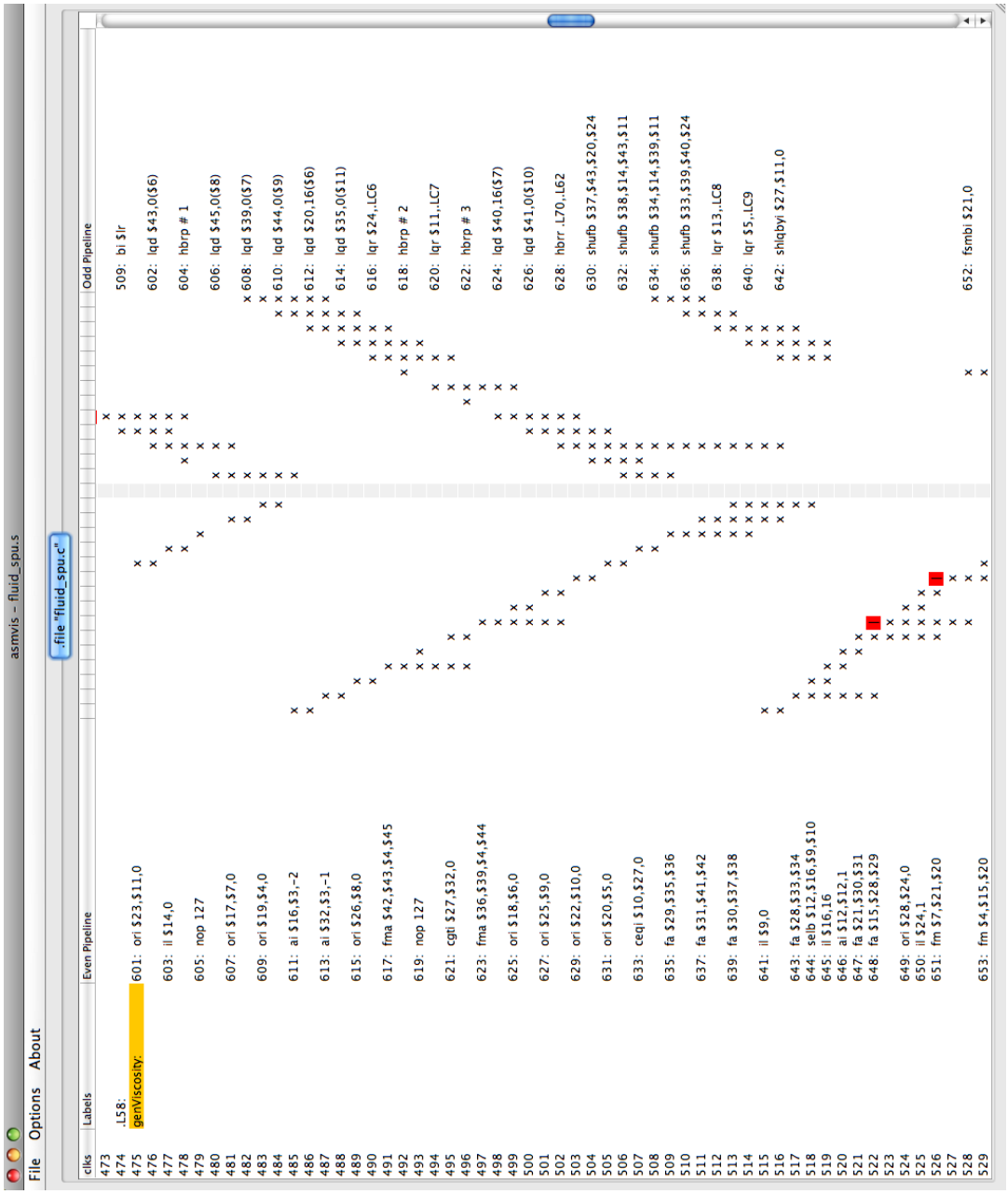

Figure 5.8: I.B.M.'s Assembly Visualizer

Screenshot of I.B.M.'s Assembly Visualizer static analysis tool. On the left we have the even pipeline, and the odd pipeline on the right. The instructions are listed in sequence that they will be executed with data-dependency stalls highlighted in red. The `X`*s going down indicate, from a given instruction, how many cycles it will take to execute. Ideally, there is no red and both pipelines are busy working. The tool can be downloaded from I.B.M. at* `http://www.alphaworks.ibm.com/tech/asmvis`

There is no operating system running on the Synergistic Processing Units. Compensating for the large size of the local store, context switches are also less frequent, if they even occur at all.

The Synergistic Processing Units have an 'odd' and an 'even' pipeline. In general, the even pipeline tends to do all of the arithmetic, rotates, and shifts where the odd pipeline handles shuffles, mask, loads and stores from LS and main memory, and branches[I.B09]. Maximizing the use of both pipelines can greatly increase the instruction throughput of an application.

The SPUs are in-order cores, as we described in section 5.1.2 on page 57. To reduce stalls within the SPU, data dependencies among consecutive operations should be avoided. Verifying the usage of the pipelines can be done using a static analysis tool, such as I.B.M.'s Assembly Visualizer, shown in figure Figure 5.8.

The SPUs are highly simplified. They only support SIMD-style instructions on 16-byte aligned vectors. Scalar operations are translated into SIMD-style instructions by the compiler. To emphasize the use of SIMD, the 128 registers within the Synergistic Processing Units are each 16 bytes long[I.B09].

The Cell, from the onset, may appear to be difficult to develop for. However, we believe the added control over the SPUs greatly simplifies the process of optimizing code. Recall, hotspots tend to compose 10% of the code and 90% of the execution time. On the Core 2 we would have implicitly gamed the cache, on the Cell we explicitly manage data transfers to get the desired speed for the slowest 10% of the code.

5.4 GPU

Imagine a teapot hovering in space. The teapot is made of millions of planar triangular glass shards painstakingly glued together. We will now consider two operations: moving the teapot and taking pictures of the teapot.

Assume that the positions of the vertices of the triangles making up the teapot are known, documented on innumerable sheets of paper. We wish to update the documented positions to reflect shifting the vertices to the right by 5 meters. Numerically, if the positive x axis points to the right, this means taking each coordinate and adding 5 meters to the x component. Reducing the amount of time needed to complete the task is trivial; we split the work among multiple people. Each person can work independently computing updated positions.

We consider taking pictures of the teapot as reducing the dimensionality from 3 (4 if time is included) to 2. Assume that we have already figured out which triangle, all of them are opaque, is visible at each point. Then, our problem becomes one of determining the colour at each point in the resulting two dimensional image. The colour at each point can be obtained by repeating the same computation. Given the base colour, normal to the triangle, location of light sources, and colours of light sources we can estimate the resulting visible colour. More

complicated techniques can be used to simulate refraction, reflection, and shadows – however we wish to emphasize the inherent parallelism of the operation.

Mapping an image to the face of the teapot requires knowledge of how the image is mapped to the individual triangles that make up the teapot. The colour from the image gets added to the equation determining the final visible colour. This process is also known as texturing.

Whereas the CPU and Cell primarily operate on sequential lists of tasks, GPUs are designed to repeat the same task over a range of data. Be it a list of vertices that need to be transformed or a series of points in an image whose colour needs to be determined.

The parallelism we described with our example teapot is reflected within the GPU's hardware and programming interfaces. Similarly, problems must map well to the parallelism exposed in the hardware to benefit from the GPU.

There are many APIs that allow us to take advantage of the GPU hardware. We experimented with the OpenGL Shader Language (GLSL) and the Open Computing Language (OpenCL) programming interfaces to access the GPU.

GLSL programs are referred to as shaders. There are three types of shaders available in GLSL, each extends the default behaviour of the graphics pipeline with custom code: Geometry shaders allow for the generation of vertices to dynamically generate meshes. Vertex shaders transform and project vertices. Fragment shaders compute the final colour of individual pixels.

OpenCL programs are known as OpenCL kernels[Khr08]. Each kernel is repeated for a set number of operations with explicitly specified inputs and outputs.

For each type of GPU program, we can assume that there is a set of number of tasks to run. On CUDA-based GPUs, like the geForce 8800M GTS that we used for development, each task is assigned a hardware thread[NVI09b]. Each task uses its unique ID to determine what work it has to do. On our test GPU all threads running in parallel must execute the same kernel.

Threads are divided into groups of 32 known as warps, named after the "first parallel thread technology"[NVI09b].

Each warp is executed on a Single Instruction Multiple Thread (SIMT) unit. Each thread within a warp must execute the same instruction even though branching instructions can diverge. Therefore, for optimal throughput, programs within a warp should branch the same way.

As we have noticed with Intel's Core 2 Duo processor and I.B.M.'s Cell processor, data plays a very important role in the performance of the application. There are three types of memory that we will look at, each with different purposes and memory access rules. We will present some rudimentary information regarding memory accesses which we believe to be sufficient for most cases.

Shared memory, known in OpenCL as local memory[NVI09a], can be thought of as a work area. It is on-chip memory which is as fast as a register in the ideal case. Each consecutive memory segment belongs to a different memory bank[NVI09a]. On our examined hardware, there are 16 shared memory banks[NVI09b]. Ideally, each thread accesses a different memory bank within shared memory.

Global memory, also known as global memory in OpenCL[NVI09a], is akin to main memory on the other processors we have seen. It is slow except under certain conditions. Optimal memory access occurs when the threads in a half-warp, the first 16 or last 16 threads in a warp, access the nth element of a basic data type within global memory in the nth thread of a half-warp[NVI09b]. This restriction was relaxed in newer hardware.

Texture memory is very different than what we have previously encountered on the other processors. It also has one benefit that makes it desirable to use for simulating fluid flow: cache misses are aware of the 2D nature of the image and load spatially local values[NVI09b]. Even though the image extensions are optional within OpenCL, they were supported on our target hardware.

Apart from memory, data types are also different. On the GPU we have access to very fast non-IEEE compliant floating-point arithmetic operations and native support for half-floats. These are very desirable features for our fluid simulation since we can halve the required memory bandwidth.

The GPU is a highly parallel processor by nature. It requires work to be submitted as parallel operations, where failure to do so results in disappointingly slow execution speeds.

5.5 Algorithm Analysis

Algorithm and hardware are meant to be wed together by the end of this document. Hoping for decent final performance by implementing and tirelessly tuning the implementation may not be a wise strategy. In this section, we return to the discretized form of the Navier Stokes equations to estimate how well they will run on the target hardware.

The presented method to obtain estimates was already applied to Stam's "Stable Fluids"[Sta99] solver by Kim[Kim08]. We have attempted other means of estimating the performance, however Kim's method to estimate the potential performance has given us results that best match what we observed from our implementation.

An overview of the method will first be presented. Within which we will describe how we obtained our estimates, what we assumed, and how we differ from Kim.

Then we shall show the results of applying the method.

5.5.1 Overview

Two heuristics derived from the discretized form of the Navier Stokes equations will inform our estimated execution times. Compute speed estimated as the number of floating point operations per second and the memory bandwidth as the number of data elements needed to be accessed.

Assuming optimal use of the hardware (specifically there are no stalls due to dependencies among instructions), then the optimal speed is the slowest of either the number of instructions processed (compute bound) or the time waiting for memory (memory bound).

To translate our count of floating point operations or memory accesses into speed requires a bit of research into the performance numbers underlying the examined hardware. Our opinion is that for this project, determining the required statistics for the target hardware to achieve a desired execution speed for a given amount of detail is more practical.

For Intel's Core 2 Duo (specifically the X9100), the peak memory bandwidth is 6.4GiB per second and 24.48 gigaflops[Int][Int06][5]. The geForce 8800 GTS has a memory bandwidth of 51.2GiB per second and can do 240 gigaflops[bib11]. The Cell processor has a memory bandwidth of 25.6GiB per second and can do 76.8 gigaflops[Kim08][6].

For accounting purposes we will treat each part of advection, each iteration of the viscosity solver, each iteration of the pressure-projection solver, etc. as different steps.

Each step requires the use of a series of fields – which must be downloaded from main memory. This is reasonable since we assume that data will remain in cache until it is no longer needed for a given step.

Each floating-point value is assumed to be 4 bytes long; if $w \in \mathbb{I}$ is the width of the field and $h \in \mathbb{I}$ is the height of the field therefore each field is $4wh$ bytes. We treat the x-velocity to be a different field than the y-velocity.

We do not assume, like Kim, that multiply-add instructions are fused into a single instruction. The final conclusion remains the same since fluid solvers based on Stam's "Stable Fluids"[Sta99] are embarrassingly data-bound.

5.5.2 Analysis

We begin with the viscosity, since it is the easiest to deal with. Recall from equation (2.71):

$$\text{Let } \boldsymbol{s} \quad = \quad \boldsymbol{u}_{x-1,y} + \boldsymbol{u}_{x+1,y} + \boldsymbol{u}_{x,y-1} + \boldsymbol{u}_{x,y+1} \tag{5.4}$$

[5] [Int] gives gigaflops and DDR2 memory bandwidth at 800Mhz, with [Int06] specifies that such memory can do 6.4GiB per second

[6] Kim provided the number of gigaflops for a dual-processor Cell system. We halved the number of flops he obtained, and since 6 of 8 SPUs are available on the PS3 we multiplied the result by $\frac{6}{8}$ to obtain 76.8. We can also say that at 3.2 Ghz with 4 floats of maximum throughput per cycle, we get 12.8 gigaflops per SPU. For 6 Synergistic Processing Units the peak is 76.8 gigaflops.

$$\boldsymbol{u}_{x,y} = \frac{\boldsymbol{u}^t_{x,y} + \upsilon\Delta t(\boldsymbol{s})}{1 + 4\upsilon\Delta t} \tag{5.5}$$

There are 6 flops to compute $\boldsymbol{s}$, 3 for the summing x-velocity and 3 for summing the y-velocity. Multiplying $\Delta t\upsilon$ to $\boldsymbol{s}$ requires another 2 flops. Adding $\boldsymbol{u}^t_{x,y}$ needs 2 more flops. If $1 + 4\upsilon\Delta t$ is put into a constant then dividing by $1 + 4\upsilon\Delta t$ uses another 2 flops. That brings the total to 12 flops for each iteration for each grid point.

If we do 20 iterations of the viscosity solver, then we need 240 flops for each grid cell.

Memory-wise, we download both the x-velocity and y-velocity from main memory and send them back. That is $16wh$ bytes transferred per iteration, for a total of $320wh$ bytes.

Moving on to pressure, recall (2.91) and (2.98):

$$\nabla \cdot \boldsymbol{w} = \frac{\boldsymbol{w}^x_{x+1,y} - \boldsymbol{w}^x_{x-1,y}}{2} + \frac{\boldsymbol{w}^y_{x,y+1} - \boldsymbol{w}^y_{x,y-1}}{2} \tag{5.6}$$

$$q_{x,y} \approx \frac{q_{x+1,y} + q_{x-1,y} + q_{x,y+1} + q_{x,y-1} - \nabla \cdot \boldsymbol{w}}{4} \tag{5.7}$$

Again, there are $320wh$ bytes transferred assuming 20 iterations of the pressure solver. That is 4 fields, downloading the x-velocity, y-velocity, pressure and uploading the updated pressure values to main memory.

We have $9wh$ flops per iteration for $180wh$ flops.

Once we have computed the pressure using an iterative solver, applying the pressure requires $24wh$ bytes transferred. As well, we need $6wh$ flops.

Our next target advection. Before we can do any estimates we require a clearer picture of the number of computations required. Recall that we pre-computed the locations from which to advect in (2.69):

$$\boldsymbol{f}^{t+\Delta t} = \boldsymbol{f} - \Delta t\boldsymbol{u} + \frac{\Delta t(\boldsymbol{u} \cdot \nabla)(\Delta t\boldsymbol{u})}{2} \tag{5.8}$$

For any grid cell indexed at $x, y \in \mathbb{R}$ where the x component of $\boldsymbol{u}$ is u and the y component is v we estimate the $\frac{\Delta t(\boldsymbol{u}\cdot\nabla)(\Delta t\boldsymbol{u})}{2}$ term using semi-lagrangian advection:

$$\boldsymbol{f}^{t+\Delta t} = \boldsymbol{f} - \Delta t\boldsymbol{u} + \frac{\Delta t(\boldsymbol{u}_{x-\Delta tu,y-\Delta tv})}{2} \tag{5.9}$$

The coordinates $(x - \Delta tu, y - \Delta tv)$ may not exist on cell centres. If a cell centre's coordinates

are integer values, then we can estimate the top-left cell of the 2-cell high 2-cell wide region from which we may pick values as $(\lfloor x - \Delta t u \rfloor, \lfloor y - \Delta t v \rfloor)$.

The weighting constants $m \in \mathbb{R}, 0 \leq m \leq 1$ and $n \in \mathbb{R}, 0 \leq m \leq 1$ are both 0 if the desired value is found in the top-left cell. We compute m and n as:

$$m = x - \Delta t u - \lfloor x - \Delta t u \rfloor \tag{5.10}$$

$$n = y - \Delta t v - \lfloor y - \Delta t v \rfloor \tag{5.11}$$

Last, we must interpolate the grid cells in the 2×2 window using m and n. If $i = \lfloor x - \Delta t u \rfloor$ and $j = \lfloor y - \Delta t v \rfloor$:

$$\boldsymbol{u}_{x-\Delta t u, y-\Delta t v} \approx (1-m)((1-n)\boldsymbol{u}_{i,j} + n\boldsymbol{u}_{i,j+1}) + m((1-n)\boldsymbol{u}_{i+1,j} + n\boldsymbol{u}_{i+1,j+1})$$

To compute $\boldsymbol{f}^{t+\Delta t}$ we require 2 fields, and output 2 fields – the 2 components of $\boldsymbol{f}$. That is $16wh$ bytes transferred.

The actual calculations include $3wh$ flops for $x - \Delta t u$, $3wh$ flops for $y - \Delta t v$, $2wh$ flops for m, $2wh$ flops for n, $24wh$ flops for $\boldsymbol{u}_{x-\Delta t u, y-\Delta t v}$, and $8wh$ flops to compute the remainder of $\boldsymbol{f}^{t+\Delta t}$ if Δt is premultiplied to $\frac{1}{2}$. This gives us a total of $42wh$ flops.

We already know the target cells, and simply need to compute the weighting factor. For moving the velocity field we input $16wh$ bytes and output $8wh$ bytes with $28wh$ flops. The colour field requires $24wh$ bytes as input and $16wh$ bytes as output since we store RGBA values using single-precision 32-bit floats. We also use up $56wh$ flops.

That brings the total for the entire advection phase, including the velocity field and a colour field, to $80wh$ bytes being transferred and $126wh$ flops.

For the entire basic simulation, our totals are $744wh$ bytes being transferred and $552wh$ flops. More bytes are transferred and flops required if we include external forces and adding in information to the velocity field. However, these numbers reflect what we wish to tell about the fluid simulation.

Figures 5.9 and 5.10 visually show the results. Notice that before we hit the compute limits on any of the processors, we hit the bandwidth limit. This is why during the implementation we put in substantial amounts of effort to minimize the use of memory bandwidth.

5.6 Conclusion

Ultimately, the algorithm, underlying hardware, and other running processes determine the possible minimum amount of time that a given computer program will have to execute. The

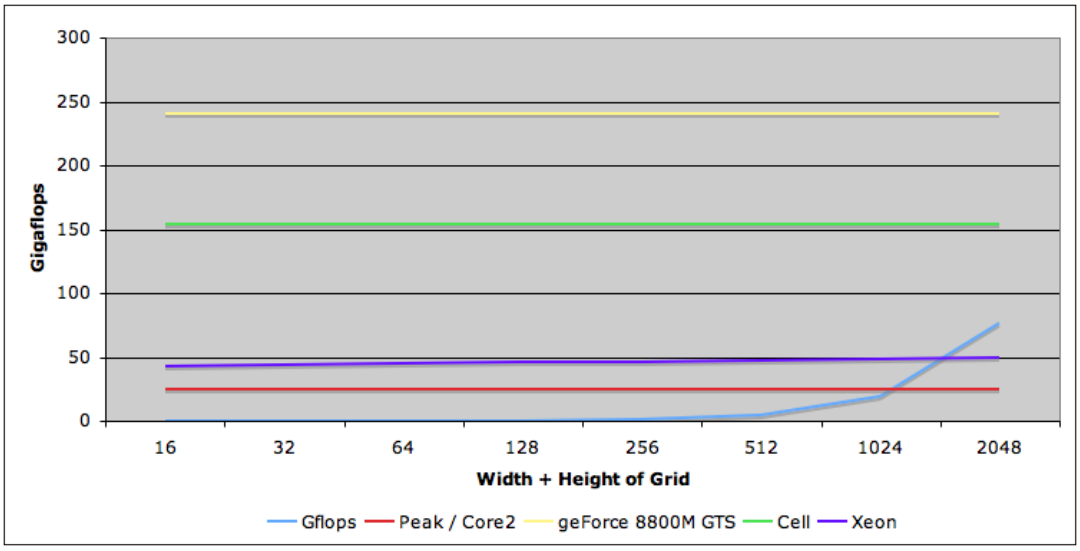

Figure 5.9: Gigaflops Required vs. Available
For square grids we graph the available computational power on various hardware compared to the required computational power for a given grid size.

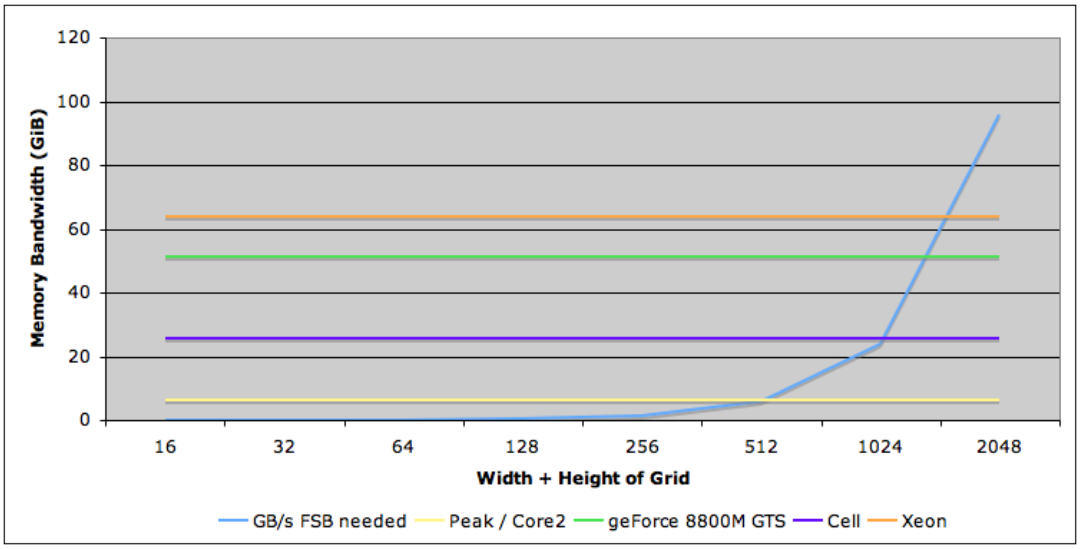

Figure 5.10: Bandwidth Required vs. Available
For square grids we graph the available memory bandwidth on various hardware compared to the required memory bandwidth for a given grid size.

hardware sets the stage by determining how quickly a series of instructions will execute and how quickly data can be accessed. Our goal is to work within the given limits to efficiently use the hardware.

Chapter 6

Task Scheduler

Consider a team of people working on the same project. If the team is small, say 3 people, then communication and coordination among team members without any central leader is feasible. If the team were increased in size to hundreds of people, then to avoid communication taking over all effort a manager should be appointed.

Likewise, managing a few threads is relatively simple. As the number of threads and rules for interactions among threads increase, the implementation becomes more challenging. When we map threading to actual hardware, further complexities arise as we start battling against the overhead of maintaining and communicating amongst the threads.

First; we look at the motivation that lead us to create a library to manage the parallel portions of our code.

Second; the literature is filled with different ways of managing parallelism within code. We list references that have shaped our thinking.

Third; we focus on the data structures and associated implementation strategy behind our task scheduler.

Fourth; we analyze the tests that we did during development of the thread scheduler. Noted are the successes and failures of our testing methodology.

Fifth; we could have done much more than what we did. We note future work that may yield very interesting results.

6.1 Motivation

Our initial goal was to reduce the memory requirements of our fluid solver. Intuition led us to see if we could parallelize an in-place Gauss-Seidel solver for the pressure and viscosity solvers

thus halving the memory requirement during those phases. This led us to creating a system that knew about data dependencies among the various steps, or tasks, of our fluid solver.

Recall that all of the data that makes up our fluid simulation is stored within grids. If an iteration within the pressure solver (among others) is a task, then we will say that computing the values for a given row is a sub-task.

Notice that each of these sub-tasks can potentially be run in parallel, upon condition the data they use is not being written to by any other sub-task and that any dependent sub-tasks have been completed.

A library that schedules a series of potentially parallel tasks (or sub-tasks in our case) is known as a task scheduler. Tasks schedulers attempt to simplify code by providing an abstraction on top of threads.

Task schedulers tend to be either static or dynamic. Dynamic task schedulers determine at run-time which tasks to run. Static task schedulers use a plan that was devised off-line to determine which tasks to run.

Our reason for using a dynamic task scheduler is that the run-time for individual tasks may not be deterministic for all situations. For example, advection is prone to cache misses that take up a non-deterministic amount of time.

Unfortunately, the overhead of dynamic task schedulers must be sufficiently small as to not outweigh any performance benefits[KA99].

Next we outline what has already been done.

6.2 Previous Work

The previous work that we describe below can be divided into two groups. There are articles that describe task scheduling, and articles that describe how to parallelize a Gauss-Seidel iterative solver. These reflect the ideas that we were working with when developing our task scheduler.

1961 Scheduling tasks can use the same logic as scheduling workers on an assembly line. An example is Hu's algorithm[Hu61]. Assuming that all tasks run in equal time and that there is no time required in switching tasks once a previous has completed, Hu's algorithm takes a tree describing the dependencies and prunes n of the furthest nodes from the final node where n is the number of workers (processors). These pruned nodes (tasks) are executed, and the algorithm repeats itself until no more tasks are present.

1981 Burton and Sleep introduce the ideas underlying task stealing. They introduce a parallel machine where when one processor starts to idle, it can steal a pending task from another processor. Functional languages were used to emphasize the concurrency within the program.

1995 Blumofe, Joerg, Kuszmaul, Leiserson, Randall, and Zhou introduce Cilk[BJK+95], a dynamic task scheduler where tasks can spawn many other tasks and form a sort of dependency tree. Each processor has a queue of tasks assigned to it, and sub-tasks are added to the same queue. When a queue becomes empty, tasks are stolen from another queue.

1996 Philbin, Edler, Anshus, Douglas, and Li described a thread scheduler that took in addresses of the data of the threads used when running[PEA+96]. They would re-order the execution of the threads by hashing similar addresses of data to the same buckets, and executing threads based upon which bucket they were in. They aimed to use threads to reschedule data accesses on single processor machines in order to maximize the use of cache memory.

1997 Stals, Rüde, Weiß, and Hellwagner showed that being more careful about what was in cache could be used to accelerate the multigrid method[SRWH97].

1998 Weissman used performance counters to help a scheduler optimally assign threads to processors[Wei98]. Specifically, Weissman looked at the number of cache misses between context switches combined with user annotations on data sharing among threads to estimate which thread to give CPU time to.

1999 Efficient use of cache memory was the means that Douglas, Hu, Kowarschik, Rude, and Weiss used to optimize the Gauss-Seidel algorithm with red-black ordering on grids[DHK+99]. They reported being able to increase the speed of execution by at most 5 times.

1999 Kwok focused on static task schedulers that use a directed acyclic graph (DAG), where the run-time is deterministic. Therefore, it becomes possible for them to know how long a given task will take ahead of time to create an ideal order of execution[KA99].

2000 Kowarschik, Rüde, Weiß, and Karl[KRWK00] did further work on a cache-optimized multigrid solver with results relating to Poison's equation in two dimensions. Here, the method used by [SRWH97] and [DHK+99] to maximize the use of the L1 cache was called the windshield wiper technique.

2000 Upper and lower bounds on the number of cache misses with task stealing was explored by Acar, Bellock, and Blumofe[ABB00].

2002 Seinstra, Koelma, and Geusebroek created a minimalist library of parallelized image processing functions so that users of the library did not need to worry about the underlying parallel architecture[SKG02].

2007 Kumar, Hughes, and Nguyen presented a hardware accelerated task-scheduler for chip multiprocessors[KHN07]. They had a global queue for all of the tasks, and multiple smaller queues for each core.

2007 Using a multigrid solver tuned for the underlying hardware can provide decent performance with image processing compared to FFT methods was shown by Stürmer, Köstler, and Rüde[SKR07].

Existing task schedulers notably include Apple's "Grand Central Dispatch" released with their latest operating system (Mac OS 10.6 and iOS 4.0) and Intel's "Intel Thread Building Blocks".

Other means of simplifying threading and concurrency in applications include OpenMP for machine-local speedups through annotating potentially parallel portions of sequential code and MPI (Message Passing Interface) for network-distributed software.

6.3 Implementation Details

Consider a single grid where each cell is updated based upon the values of neighbouring cells. Furthermore, imagine there is depth to the grid which represents future iterations, as seen in figure 6.1. If we update the first row in the first iteration, we can not update the second row in parallel. Likewise, each iteration must be run sequentially.

Imagine that we focus on individual rows and iterations. The first row on the second iteration only depends upon the first and second row of the first iteration to be completed. This is how we extract parallelism out of sub-tasks.

The final result is a blocking technique[Ker08] that applies to each step of the solver. For example, once we complete computing the viscosity for the first two rows of the grid, there is no reason to not start processing pressure. In addition, the first two rows of velocity data are already most likely in cache, thus reducing the required memory bandwidth.

A problem that continually arose during the development of the scheduler was that it needed to run fast enough to mask the overhead needed to run it. We ended up designing a minimalistic data-structure that is visually represented in figure 6.2.

As implemented, each task has a series of subtasks. Each subtask is assumed to work on individual chunks of memory. And each proceeding and preceding dataset of a previous task's subtasks depend on the dataset of the current task's subtasks. For example, subtask 1 of the preceding task is assumed to generate the output for subtask 1 of the current task.

Within 6.2, only the numbers stored in the array called "Work Data" is part of our data-structure. These are:

Completed: The number of subtasks that have been completed for the given task. It is a 16-bit integer that is atomically incremented whenever a subtask completes.

Working: Either equal to "Completed" or one greater than "Completed". Only when "Working" is equal to "Completed" is a subtask delegated to a thread. "Working" a 16-bit integer concatenated to "Completed" forming a 32-bit integer.

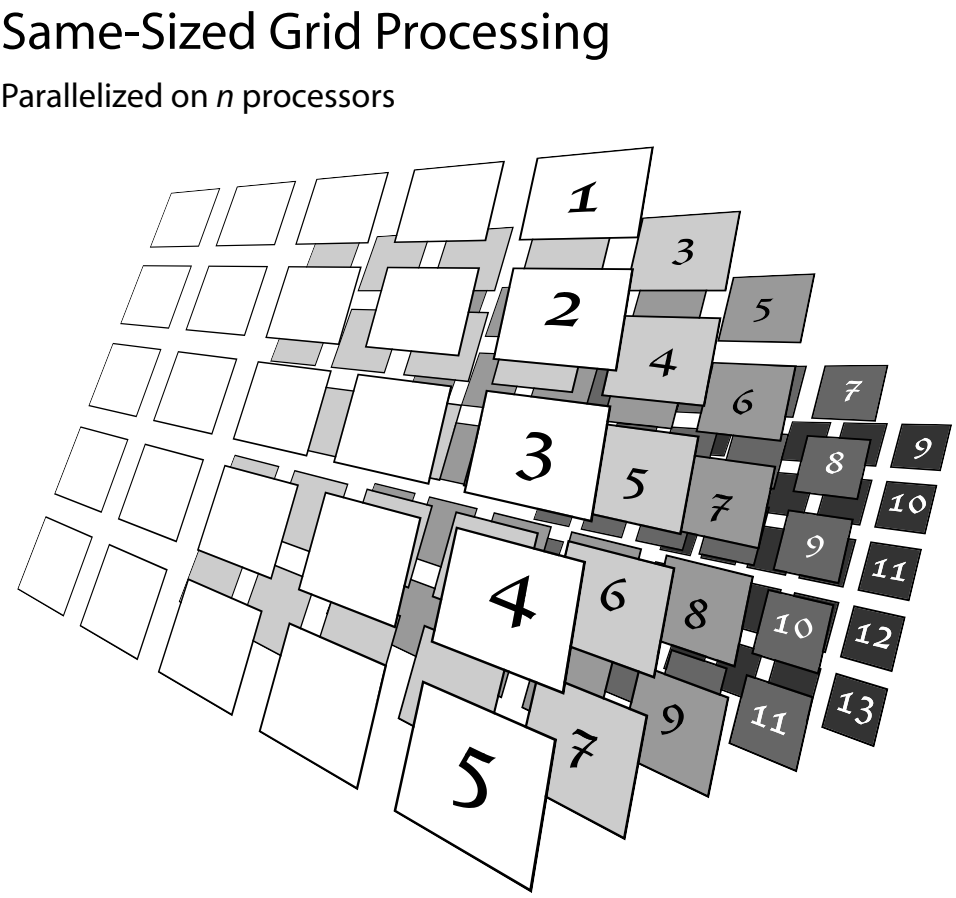

Figure 6.1: Visual Representation of Parallelization Strategy
A single grid, where depth represents future iterations. Rows with the same numbers can be safely executed in parallel. Each set of numbered rows must be computed in sequential order, and barriers are normally used to ensure a set is completed before the next starts. Rows are executed in sequential order.

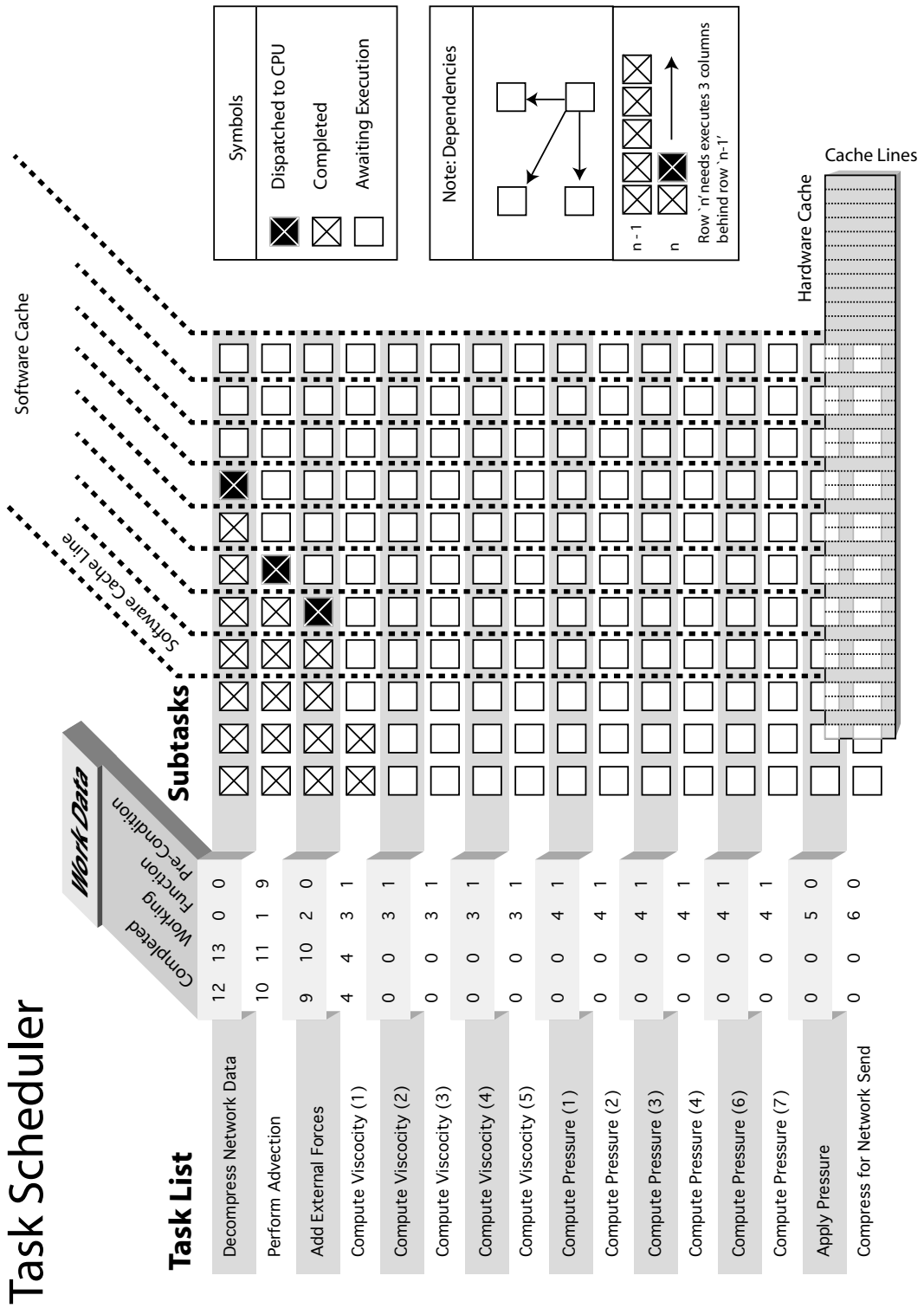

Figure 6.2: The Task Scheduler

At it's core, our task scheduler has a set of tasks that each transform a type of data. In relief is the actual data used to drive the task scheduler. Behind the portion in relief is the conceptual view of what is happening.

Method	Trial 1	trial 2	Trial 3	Trial 4	Average
In Order	67.40	67.38	67.31	67.33	67.36
Cache 1	67.56	67.67	67.65	67.61	67.62
Cache 2	67.29	67.14	67.32	67.34	67.27
Cache 4	67.15	67.11	67.15	67.15	67.14
Cache 8	67.02	66.94	66.98	66.99	66.98
Cache 16	66.82	66.94	66.89	66.94	66.90
Cache 32	66.86	66.82	66.82	66.85	66.84
Cache 64	66.81	66.68	66.83	66.77	66.77
Cache 128	66.80	66.82	66.77	66.91	66.82
Cache 256	66.89	66.88	66.89	66.90	66.89
Cache 512	67.61	67.61	67.63	67.55	67.60
Cache 1024	68.04	67.95	67.99	68.01	68.00

Table 6.1: Single Core Task Scheduler Performance
"In Order" refers to the "localityInOrderTest" function on page 65. "Cache" is the size of the software cache. Notice that the overhead of the task scheduler masked when the software cache is set to a value between 2 and 256 inclusively.

FN: The function that should be executed for each subtask. It is an 8-bit integer. "FN" shares a 32-bit integer with other values describing the dependencies – one telling if the subtasks must be completed in sequential order, and the other is the pre-condition.

Pre-Condition: Describes how many subtasks of the previous task must be completed before a subtask of the current task may be started.

In addition to providing a means to schedule tasks, we also attempt to estimate what useful data is currently within cache. We work on the assumption that the most recent values remain within cache, which is perfectly valid for desktop processors[GBST06].

We game this behaviour by assuming that if one sub-task depends upon multiple sub-tasks, then it implies that there is shared data among the sub-tasks. We mark a range of tasks as having their dependent data stored in cache memory. We call this range the "software cache" in figure Figure 6.2. Only 2 integers are needed to keep track of the "Software Cache". One for the start, which is atomically incremented whenever needed, and one for the length of the cache which remains constant therefore no synchronization is needed when obtaining its value.

6.4 Results

Our scheduler, under ideal circumstances, will consistently outperform a naïve implementation (without a thread scheduler) while providing a simpler interface to parallelizing our code. The results of figures 6.3 and 6.4 with associated tables 6.1 and 6.2 were obtained by testing the

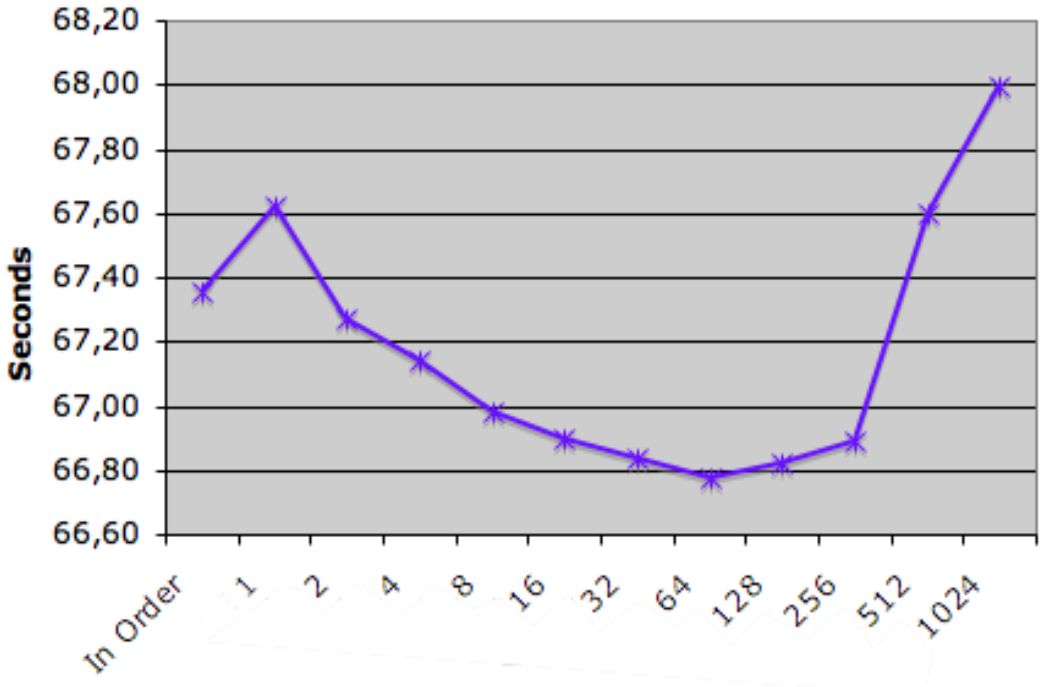

Figure 6.3: Task scheduler performance (1 core)
Number of seconds to complete calculations as described in previous chapter on a 64MB data set using 800 iterations for various sizes for the software cache. Notice how performance rapidly detoriates past a software cache size of 256. At that size, 4MiB of data are used per row. With a software cache of size 512, there is more software cache than hardware cache.

Method	Trial 1	Trial 2	Trial 3	Average
Cache 1	69.23	69.24	67.97	68.82
Cache 2	68.05	68.11	67.12	67.76
Cache 4	64.86	65.17	64.17	64.73
Cache 8	55.78	57.70	55.30	56.26
Cache 16	45.76	46.62	47.25	46.54
Cache 32	38.18	38.98	38.07	38.41
Cache 64	35.03	35.33	35.77	35.38
Cache 128	34.66	34.33	34.16	34.38
Cache 256	34.11	34.04	33.94	34.03
Cache 512	34.52	34.48	34.51	34.50
Cache 1024	34.99	34.90	34.87	34.92

Table 6.2: Dual Core Task Scheduler Performance
Note that additional cores incur additional synchronization overhead for small software caches.

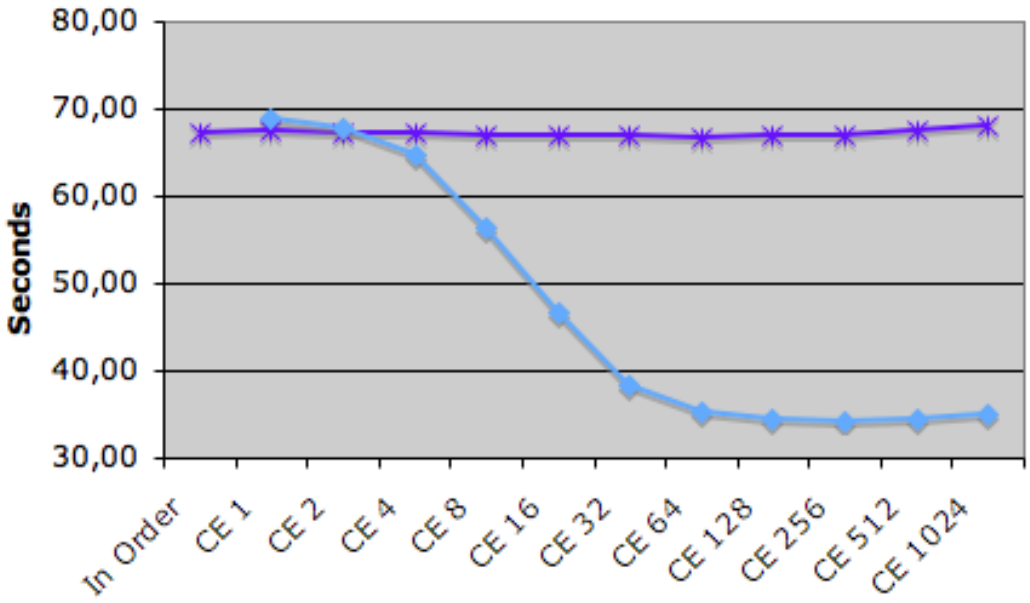

Figure 6.4: Task scheduler performance (2 cores)
Same test as in figure 6.3 except 2 cores are used. Of interest is that the ideal size for the software cache is larger.

`localityInOrderTest` method found on page 65. We attribute performance gains to our ambitious implementation, the simplicity of the source problem, and that we actually did hit the cache more often.

Our implementation is ambitious since we did not use any mutexes and rather built the system on top of 'compare and swap' and other atomic operations. Doing so, we have allowed multiple threads to run through the scheduling code at the same time without needing to block too much. On the down-side, adding a feature or debugging the task-scheduling code can take days as threading issues tend to be very obscure.

The simplicity of the source problem – that we are running the same kernel 800 times to obtain a result – returns questionable results. Especially since we were intending to compose multiple different kernels. It is an unfortunate fallacy with our tests.

Using a profiler, such as Apple's Shark[1], it is possible to determine that the number of L2 cache misses and memory bandwidth have been halved compared to the naïve implementation on an Intel Core 2 Duo in the uniprocessor case.

[1] Apple CHUD/performance tools: `http://developer.apple.com/tools/performance/overview.html`

6.5 Further Investigations

Even though the task scheduler works; there are many potential improvements to explore. Such changes include a ready queue, improved multiprocessing, self-tuning, and using a standard task scheduler package in the back end.

The reason to look at a ready queue is that the processor spins over a range of rows where a next task can potentially be found. If it doesn't find one it gets pushed into a queue for waiting threads. If there were a list that could be accessed using non-blocking means that identified the next item to process, then the scheduler could potentially run faster. If managing the data structure is too complex; then it may actually slow down the thread scheduler.

Only a single processor can work on a task at a time. This was to support a Gauss-Seidel iterative solver which has better convergence properties than a Jacobian iterative solver. However, for problems such as advection all the processors could potentially work in parallel. The advection might even be taken out of the task scheduler.

The task scheduler is very sensitive to the underlying hardware upon which it is run. If the hardware changes, it needs to know its ideal software cache size. Ideally, the scheduler would vary this value at run-time using statics gathered through slight variations of the software cache size.

Given there are existing task schedulers such as the one from Intel (Intel Thread Building Blocks) and from Apple (Grand Central Dispatch) – these may form a suitable base to build the current scheduler. Our logic for building our own task scheduler was to minimize the overhead of managing the potentially thousands of tasks as the grid-size increased and minimizing the memory used by the data-structures. We should compare the execution speed if we minimize the sharing of cache lines among cores.

6.6 Conclusion

We have developed a task scheduler that simplifies the implementation of our fluid simulation while attempting to maximize the use of the processor's cache. It is sufficient for this project, even though there are many more improvements that warrant further investigation.

Chapter 7

Compression

Sending image information across the network can be challenging. A gigabit ethernet connection can handle 1000 megabits of data per second, equal to 125MiB of data per second. On an interactive system at 30 frames per second $\frac{125\text{MiB}}{30} = 4\text{MiB}$ of data can be transferred per frame.

With a 512×512 grid composed of 32-bit single precision floating point values requires 1MiB of data. The fluid simulation has 2 such grids for velocities and 4 such grids for colour data. This leads to a total 6MiB per frame being sent and received. As the resolution of the simulation grid increases, compression becomes essential if we wish to share the velocity and colour grids with other machines on the network.

We have looked at multiple ways to compress data. As of writing, we have not found a method that is suitably fast.

First: we will look at the data that we have to send. Specifically, the colour data can be represented as bytes. The velocity field should remain as floating-point though.

Second: knowing that the data will be continuous, we play with a simple mathematical model that allows us to quickly compress and decompress data.

Third: to obtain better results we read a few papers on compression and attempted to implement an algorithm called Fast and Efficient Image Compression System (FELICS).

Fourth: we look at methods of compression that require less bit shifting. Specifically, we become interested in the discrete cosine transform.

Fifth: half-floats are sufficient for our simulation. We look at the IEEE standard show a means of converting between floats and half-floats.

Sixth: we look at different ways that we could compress data without too great of a performance penalty.

7.1 Data

We have two types of data within our fluid flow. The information dictating how fluid flows, and the visual representation of the fluid.

The information that dictates how fluid flows, velocity and pressure, is bestrepresented using floating point data since values can fluctuate from very small to very large.

Data that deals with the visual representation of the fluid can be reduced to bytes. Consequently, we can use standard methods to encode image data.

Within this chapter, we describe our attempts to reduce the size of these two data-sets while retaining enough information to reconstruct the data with little noticeable visual degradation.

7.2 Continuous Run-Length Encoding

Run-length encoding is typically used in images that have a limited colour palette. Rather than encode a series of a colour values, a colour is followed by a number indicating how many pixels it spans[Weid][1]. Such an encoding is very good for cartoons, diagrams, etc.

Since a lot of the output images we produce have a blurred aesthetic, we thought of storing the image as gradients. Our data-structure would be identical to that of run-length-encoding, except we would interpolate between the start and end colours to obtain a gradient.

We look at curvature since it describes the change of the change in colour. If the change of the change in colour is near 0, then we can safely guess that the underlying data can be estimated as a linear interpolation between two colours.

Let r be a list of data from a single row of the grid to encode.

Therefore we can write the rate of change and curvature respectively as:

$$\frac{dr}{dx} \tag{7.1}$$

$$\frac{d^2r}{dx^2} \tag{7.2}$$

The total curvature for subset k of list r is defined as:

$$c = \int_k \frac{d^2r}{dx^2} \tag{7.3}$$

First order change can be easily represented using two points and linear interpolation. Therefore, the data in r is broken in l subsets of data whose total curvature does not exceed c.

[1] [Weid] deals with the general case of a set of numbers.

Since compression is done on a per-row basis, visual artifacts such as lines may appear in the final image. We attempted to reduce the artifacts by compressing the differences between lines (a trick to reduce entropy used in PNG compression) – however we suffered from insufficiently accurate reconstruction of the image since differences between lines were too different from the source image.

Our simple compression scheme was suitably fast but resulted in overly noticeable visual artifacts.

7.3 FELICS

FELICS is a lossless compression algorithm created by Howard and Vitter[HV93]. Every bit is counted within their compression algorithm.

The algorithm assumes that for a given pixel, its colour can be determined by near-by pixels. Specifically, the difference between the value of the left and top pixels are used to determine the potential range. If the current pixel falls within the range, then it is encoded with exactly the number of bits required for a number that size. Else, Golumb[Gol66] codes are used prefixed with a bit to say if the value is less than or greater than the average of the nearby pixels.

Upon decompression, a bit is read to determine whether a Golumb code or colour differences are used.

Unfortunately, the algorithm was too slow for our purposes despite having excellent results. When it ran, it took more time than what we allocated to render a single frame of our fluid simulation. We did manage to vectorize parts of the algorithm, but for little gain.

7.4 Half Floats

Floating-point values are composed of a sign bit, mantissa (exponent), and significand (fractional part). The latter 2 are integers, the mantissa is of the form 2^x and the mantissa of the form $\frac{1}{x}$.

Converting to a 16-bit float is the process of taking the 5 least significant bits of the mantissa and the 10 most significant bits of the significand combined with the sign bit.

If the mantissa is too small, we round to 0. We do not wish to trouble ourselves with denormal values. If the mantissa is too large where the value exceeds 2^{16}, we say the value is infinity.

Below we provide a working implementation that converts between 32-bit and 16-bit floats:

```
typedef union half_bits half_bits;
union half_bits
```

```
{
  float f;
  unsigned int i;
  float16 s[2];
  unsigned char c[4];
};

//Macros for portability...
#if __LITTLE_ENDIAN__
  #define BP_LEFT   1
  #define BP_RIGHT  0
#else
  #define BP_LEFT   0
  #define BP_RIGHT  1
#endif

//Convert a float to a half-float  (scalar)
float16 float2half(float in_float)
{
  half_bits hb;
  hb.f = in_float;

  short mantissa = (hb.s[BP_LEFT] >> 7) & 0x00FF;
  mantissa += 16-127;
  if (mantissa < 0)
    return 0;

  if (mantissa > 32)
    return (hb.s[BP_LEFT] & 0x8000) | (0xAFFF);

  return     (hb.s[BP_LEFT] & 0x8000)       //Sign bit     (1 bits)
      | ((mantissa << 10) & 0x7C00)      //Mantissa     (5 bits)
      | ((hb.s[BP_LEFT] << 3) & 0x03F8)    //Significand (7 bits)
      | ((hb.s[BP_RIGHT] >> 13) & 0x0007);  //Significand (3 bits)
}

//Convert a half-float to a float  (scalar)
float half2float(float16 in_half)
{
  short mantissa = (in_half & 0x7C00) >> 10;

  if (mantissa == 0)
    return 0;
```

```
  if (mantissa == 0x001F)
  {
    if (in_half & 0x8000)
      return -INFINITY;
    return INFINITY;
  }

  mantissa += 127 - 16;

  half_bits hb;
  hb.s[BP_LEFT] =   (in_half & 0x8000)          //Sign bit     (1 bits)
        | ((mantissa << 7) & 0x7F80)        //Mantissa     (8 bits)
        | ((in_half >> 3) & 0x007F);        //Significand (7 bits)
  hb.s[BP_RIGHT] =    (in_half << 13) & 0xFFFF; //Significand (16 bits)

  return hb.f;
}
```

The listing outlines one of the issues that we had to continually deal with whenever we tried to do any fancy bit manipulations: the endianness of the machines varied. In addition to the clutter from vectorizing code, we often had to deal with writing different branches depending upon byte ordering.

On the upside, the conversion to and from 16-bit floats is a series of simple bit manipulations which could be written using SIMD operations.

7.5 Future Work

Powering the JPEG file format, the Discrete Cosine Transform provides a lossy way to compress image data. We believe that this can deliver the performance that we seek and would also be suitable to represent our velocity field. Tests will be needed to see if this is the case.

We have not exhausted what we could do with what we call continuous run-length encoding. If we could augment the algorithm to be aware of colour differences spatially rather than confined to the line results would be much better.

For FELICS, we should take a second look at the literature to see existing methods to more quickly do the required bit manipulations.

Unfortunately, we are not optimistic since most compression algorithms assume that compression can take more time compared to the decompression. For us, we want a very fast compressor and decompressor.

7.6 Conclusion

We are still searching for a suitable method to transfer data over the network given our strict time constraints. Meanwhile, requirements changed as machines got faster and we can now run our fluid simulation on the same machine that uses the resulting data.

It is still our desire to find a compression algorithm that would enable us to send video at the compression and decompression speeds that we desire.

Chapter 8

Implementation

After having looked at multiple aspects related to interactive fluid simulations, we finally put the pieces together to form a whole.

First: we look at how the system was initially structured. It was an elegant design. Over time, we were forced to deviate from this idealized structure to better suit the underlying hardware.

Second: we provide details on the CPU version of our fluid solver.

Third: we give further details on the GPU implementation.

Fourth: we explore the Cell implementation.

8.1 Original Design

Initially, our goal was to take advantage of the common preference for cache-local data of the various processors, especially for the CPU and Cell where data transfer has a greater impact on execution speed than compute time with respect to integrating the Navier Stokes equations on rectangular grids[Kim08].

Since I.B.M.'s PowerPC G5 CPU's Altivec extensions looked and behaved very similarly to the instructions available on the Synergistic Processing Units of the Cell, we decided to first implement and optimize our simulation for the PowerPC G5. The tools were much easier to use and lead to an easier initial implementation.

The PowerPC G5 version would include reusable components such as the code for network connectivity. We chose the protocol used by the "jit.net.send" object in the Max/MSP/Jitter, the programming environment used in the Topological Media Lab. The PowerPC G5 version would also include the scheduler, described in the previous section, that we hoped would guide data accesses on the various architectures.

In addition, we would use opaque data structures to hide implementation details from the host application. Then, running optimized parts of the solver would be a matter of compiling the appropriate files. In theory that would have been a good solution.

8.2 CPU

Our initial target platform for our fluid simulation was Apple's Mac OS X. We had several challenges due to our choice of OS. Besides the OS we also had to make a few adaptions for the out-of-order execution on the CPUs. Then we had to deal with atomic instructions which are more challenging to use than mutexes. Afterwards we built the networking code. We also added some code to check on the performance of the system at run-time. Finally, we integrated our fluid simulation within the Max/MSP/Jitter programming environment.

During development, Apple was in the midst of switching from PowerPC architecture to Intel architecture. This had, among others, two very noticeable effects:

SIMD Any vectorized code we had written for the PowerPC (given some constraints) would easily port to the Cell's Synergistic Processing Elements – allowing us to debug the computational kernels before they were run on the Cell. Unfortunately, Intel machines were unable to run our optimized code which lead us to rewriting our algorithms specifically for the Intel Core 2 Duo machines. After reading some code from the Bullet physics library[1], we followed their lead and worked with scalar code operating on aligned structures containing 4 floating-point values. For trivial operations, the compiler generated the code we wanted in a portable, cross-platform way.

Endianness The PowerPC machines had the same endianness as the Cell. Looking forward, in our minds this would make communication between the Cell and PowerPC machines easier. With Apple's transition to Intel machines well underway, we found ourselves having to adapt our networking code and any code that took advantage of the byte ordering within the system.

Both the Intel and PowerPC CPUs that our Fluid Simulation targeted were out-of-order execution units. Gauss-Seidel, unless we use red-black ordering, has very strict data-dependencies. We processed groups of 16 consecutive floating-point values the same way a Jacobian iterative solver would. This exposed enough parallelism to maximize the use of the processor's execution units. Our accuracy definitely did suffer – but we did not care since it looked right.

Working solely with atomic instructions complicated the implementation of our task scheduler. This complication, namely the ABA problem[DPS10], lead to the reason why much of the members driving the scheduler are in unions. Unions are used so that the whole 32-bit value

[1] see `http://bulletphysics.org`

is unique as it mutates to reflect the state of the system. We found this to be an essential trick when a lock is being avoided and a value can both be incremented and decremented.

The networking code was written to run on its own thread. It would, once in a while, push a grid as input to the simulation and take as output the grid to send back to the host requesting remote computations. We organized the interface so that the application would not need to worry about synchronization issues by using four buffers: one contains the data currently being sent, one is free to receive data, one contains new data to send, and another filled with the latest received data.

Since the programming environment of the Topological Media Lab is Max/MSP/Jitter, we implemented the protocol found in the "jit.net.send" object. In addition to being able to send uncompressed frames of video, the protocol also called for the ability to send messages consisting of string, integer, and floating-point values. The messages were then used to control various parameters of the fluid simulation, such as where to send data in addition to the amount of viscosity, vorticity confinement, gravity direction, and magnitude, etc.

The CPU version of our simulation had a logger that was a thin layer on top of the task scheduler allowing us to query how much time individual tasks took. This told us how much time we spent dealing with the OS, how much time our scheduler took to run, etc. The output can be seen in Figure 8.1 for a 512×512 grid on a Core 2 Duo. Notice that the simulation is running at almost 30 frames per second, near what was predicted using our estimates. This time does not include overhead from the networking system, which has a substantial impact on execution speed.

As technology marched forward and CPUs become faster, we no longer needed a remote machine to run our fluid simulation. At that point we adapted our scheduler to use Apple's Grand Central Dispatch for the dynamic thread management and made our fluid simulation a Max/MSP/Jitter external (in other environments, the term plugin would be used in lieu of external). We used Grand Central Dispatch since it can dynamically adjust the number of threads based upon the system load. Also, Max/MSP/Jitter is not very good at using multiple cores so we would take the given input, return the output of the previous computation, start processing the current input and tell Max/MSP/Jitter we are done with the current frame. CPU usage was greatly improved.

When it came time to implement particles, we wrote a separate plugin for Max/MSP/Jitter that would take in the velocity field and the set of particles and output the updated particle positions for each frame. This plugin also worked a frame-behind like the fluid plugin. We used the existing OpenGL libraries within Max/MSP/Jitter to render the particles.

In the end, the CPU version of our solver was the most complete and functional of the group. It had the most development time and effort behind it. All of the pictures showing results of the fluid simulation in this document are from the CPU solver.

Figure 8.1: CPU Application Results

The performance measurements of the application. The scheduler runs very quickly, the sum of the time spent on all cores by the scheduler is 2.5 milliseconds per frame. Advection runs in about the same amount of time as pressure and viscosity since it continually does cache misses. Overall time is the time needed to run the simulation plus any overhead from the OS and associated libraries. Simulation time is the amount of time needed to run the simulation in the absence of rendering, user input, networking, etc. These measurements are gathered in real-time while the application is run with minimal overhead.

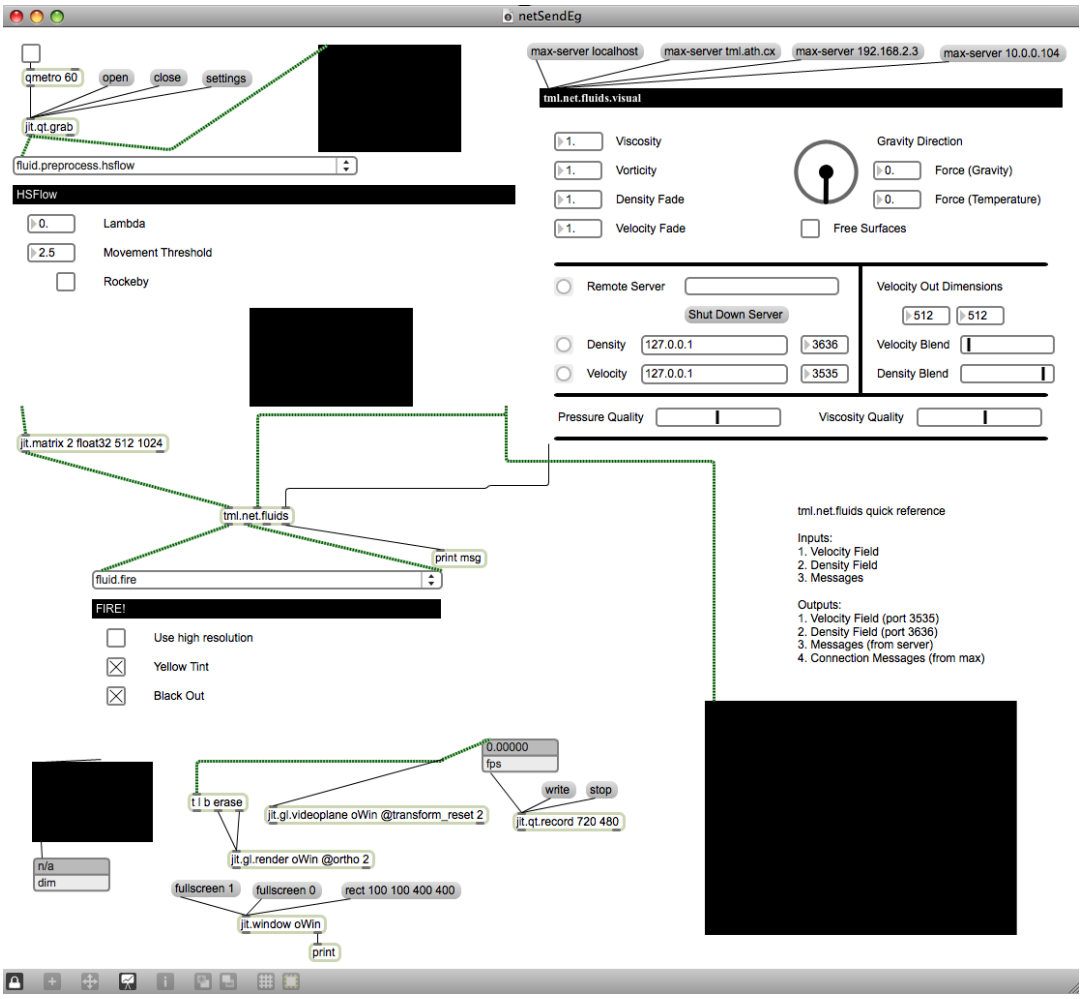

Figure 8.2: Max/MSP/Jitter Patch

A Max/MSP/Jitter application used to control the remote-server version of our fluid application. Colouring of the fluid was only available in the Max/MSP/Jitter plugin.

8.3 GPU

The GPU version failed to meet our expectations. We will look at what worked, then at why Max/MSP/Jitter integration failed.

The GPU is very different from the CPU and the Cell. Since there was support for GLSL shaders within Max/MSP/Jitter, we opted to write shaders in a test application to make sure it functioned as expected and port our shaders to the Max/MSP/Jitter environment afterwards. Our test implementation only worked on Mac OS 10.5 since the drivers in Mac OS 10.4 did not support the needed OpenGL extensions for the target hardware of the time.

As we started to write code for Max/MSP/Jitter, we noticed that the OpenGL shader support was robust for 8-bit images. However, our simulation – despite many attempts – required floating-point support. 16-bit half-floats were supported by the hardware but did not work on Max/MSP/Jitter unless parts of the program were deleted and re-inserted each time we opened our program's patches (source files).

Later, we noticed that support for 32-bit floats seemed to function. We wrote many Max/MSP/Jitter abstractions to mask the OpenGL faults with some success. However, developing using our abstractions was challenging since closing and re-opening the Max/MSP/Jitter programming environment would sometimes make our program run differently. The reason, we believe, is that when an object (like a texture) is disconnected as the source to another object (like a slab – rendering a rectangle with a shader to a texture) then the disconnection is not taken into account until a new source object is provided.

We also considered writing an OpenCL version, however our tests showed that GLSL allowed half-floats whereas in OpenCL we were forced to use 32-bit floats. We tried using the Quartz Composer environment to build the shaders, but the environment kept on freezing the system. In the end, we concluded that the older GLSL was a more stable platform.

We did not attempt to add in the network code to the GPU solver since it should run locally given the machines in the Topological Media Lab had very fast video-cards which could handle the computations locally. It is now possible to share OpenGL buffers between applications under OS X, but we have not had time to explore that possibility yet.

Our GPU code was left in an imperfect state. The first improvement would be to remove many of the needless branches within the code. The second improvement would be the use texture border conditions rather than if statements within the code.

8.4 Cell

Our strategy for the Cell processor back-fired. We will look at what worked, what didn't, and how we worked around these issues.

The optimized kernels from the CPU version were easily ported to the Cell as was all the networking code. If we did not have to worry about the Synergistic Processing Units then everything would have compiled without any modifications. All of the kernels fit in a single program that was uploaded to each Synergistic Processing Unit. The program took as input a task, a series of rows to upload, a series of rows to download, and information about time-steps. Essentially, all that was left was to hook it up to the task scheduler.

Unfortunately, we did not anticipate the overhead associated with starting execution of a task on the Synergistic Processing Units. To our dismay, we found our simulation was spending more time requesting that the Synergistic Processing Units do work rather than actually doing any work. We confirmed this by uploading a program on the Synergistic Processing Units that did nothing.

Therefore, we had to rethink how our simulation was going to work on the Cell. We could have the Cell act like the GPU, computing a single kernel at a time on a series of grid-points. Compositing all the steps together (advection, pressure, etc.) would lead to a working fluid simulation. That would have been the safe and boring way forward. Also, it would guarantee that we hit the memory bandwidth wall faster.

During our previous misadventures, we learned that the Cell's Local Store could hold the program and 60 rows of 1024-floating-point element lists. Each of those 60 rows could hold velocity, pressure, sources, etc. information.

Our conclusion was that ideally we should have each Synergistic Processing Unit run the fluid simulation on a small region. Within that small region, we would use a blocking technique to maximize the use of data as it gets loaded in and out of the SPU. We opted for a finite-difference based advection scheme since it was easier to implement.

To understand why, consider the following: Assume we have 4 input and output types of data. Minimally, we have to bring into the Local Store the velocity fields, the pressure field, and the source of velocity. Of the 60 rows of data we store in the Local Store, 15 are for the x-velocity, 15 for the y-velocity, 15 for the pressure, and 15 for the sources.

Of those 15 rows dedicated to each type of data that we bring in and out of memory: 1 row should be reserved for DMA input, 1 row for DMA output. Of the 13 remaining rows, we simply have to load the first 3 rows of the subset of the fluid simulation that we wish to compute. Then we can process the first iteration of the second row. While that happens, the 4th row is being DMA'd into the Local Store so that the first iteration of the third row may continue. Actually, the same pattern as found in figure 6.1 on page 89 arises.

Once all portions of the computation are completed, minimally including pressure, advection, and viscosity, we start sending data back to main memory. By doing this, we have maximized the use of the Local Store and are a step closer to being compute-bound rather than data-bound.

The down-side is that at the moment we are stuck with 12 iterations to run the pressure, viscosity, and advection solvers unless if we want to do additional data transfers.

Unfortunately, as tests have revealed, we are still data-bound. Using a minimal version of the simulation we have noticed very little improvement using 6 Synergistic Processing Units compared to 5 Synergistic Processing Units.

As future work, we will store in main memory data as 16-bit half-floats and only once data is within the Local Store will we decode the half-floats in floats. We could push the idea further by storing visual information as bytes and decompressing it into floats as needed. Given we are data-bound, these optimizations should greatly improve the performance (and functionality) of our Cell-based fluid simulation.

Next on the list would be to reduce the width of the tiles to allow more iterations of the solvers. This would mean adjusting the inherited CPU code to only apply the left and right border conditions when they are desired.

Last, we would like to vary where the computations are split among frames. This would help mask the tiling of the computations which are now, in the worst case, clearly visible. In the best case the tiling is invisible to the user.

8.5 Conclusion

We have implemented a fluid simulation on 3 different varied hardware platforms. Each provided sufficiently different challenges such that each implementation of the solver is unique to match the hardware that it runs on.

The CPU version is the most widely deployed and used version of our fluid solver since it is the easiest to use. A plugin that bypasses the need to deal with any networking issues and provides excellent frame-rates at high resolutions made it practical.

There are now efficient ways to transfer OpenGL texture data among applications. This may make putting effort into reworking the GPU version of our application worth-while.

For the Cell version, once the network issues are worked out so that we can send full-frame video at high frame-rates then it will most likely be the fastest machine. Our initial tests reveal that dealing with 1024×1024 grids at around 30 frames per second may be feasible[2].

[2]We should note that the functionality was reduced to meet constraints that we set for memory bandwidth usage. Different algorithms make straight comparisons challenging.

Chapter 9

Conclusion

We have created multiple versions of our fluid solver during the course of this project. We continually kept on increasing the speed of execution while adding features and have not stopped thinking of other promising improvements.

First: we review what we did. We detail what software we created and what we learned in the process.

Second: we describe potential and existing uses of our developed fluid simulation. The gamut runs from games on portable devices to large-scale installations.

Third: we list publications that we co-authored during the development of our software.

Fourth: if we had not put a stop to our development, we would have continued improving our fluid simulation. We detail future improvements that we are very tempted to explore.

9.1 What was done

We have written a fluid simulation that we have ported, with various degrees of success, to various computing platforms. In our hunt for greater speed combined with the desire for increased complexity, we found ourselves taking advantage of hardware-specific features. Often, the use of these features forced us to substantially rework our solver for each piece of hardware.

From this process, we learned how to solve a discretized form of the incompressible Navier Stokes equations including parts of the derivation. We studied and observed the importance of keeping data local to the cache, various methods to compress data, and the peculiarities of various parallel processing architectures through experimentation.

Even as we wrote this document, the technology kept on changing. For example, the introduction of task scheduling within the OS or simply changing processor as we attempted to write while attempting keeping our simulation up to date.

Figure 9.1: Skylight at CCA, Nuit Blanche 2009
Picture of skylight, project by the Topological Media Lab / Morgan Sutherland / Michael Fortin that uses our fluid simulation. Image used with permission from the CCA.

For the fun of it, we have recompiled a version of our fluid simulation for Apple's iPod and iPad. This was trivial since we kept the original not-optimized C code for reference when things inevitably broke during development. Compared to Autodesks's Fluid FX application (released in 2010), our simulation is much slower. Admittedly, our simulation ran completely on the CPU while FluidFX most likely used the GPU to do the numerical integration.

9.2 Applications

Our fluid simulation tends to be used for responsive installations, environments augmented with media. Environments that sense the world and respond in creative ways.

For example, it has been used in an installation by Morgan Sutherland at the Canadian Centre for Architecture called Skylight for the 2009 Nuit Blanche.

Our simulation has been used in a 2010 installation entitled 'Il Y A'. For this installation,

Figure 9.2: Il Y A Test
Image of Il Y A from the summer of 2009 in the Topological Media Lab while visual effects were being tested.

under the guidance of Sha Xin Wei, Jean-Sébastien Rousseau ran two instances of the fluid simulation so that two people on opposite sides of a screen could play with a digital translucent mirror. Depending on the system's mood, fluidly distorted the results. The simulation's ability to emulate a range of special effects, such as fire and smoke, was also shown off[1].

9.3 Publications

With Sha and Rousseau[SFR09], we describe parts of the fluid simulation as they were functional in 2009. Of interest within this paper is the comparison of the visual result of our advection scheme versus the McCormack advection scheme. Since then we added a task scheduler to accelerate our simulation, worked in vectorized versions of the individual components, added in vorticity confinement, and additional colour controls.

9.4 Future work

The computational kernels that we dealt with were relatively simple. Much of our time was spent hand-tuning individual kernels, redundant work that should have been handed off to a compiler. If we would have invested time in a subset of an OpenCL-like language we could

[1] Jean-Sébastien Rousseau has posted videos from the testing to his Vimeo page. We refer the curious reader to `http://vimeo.com/14291581`, `http://vimeo.com/14291643`, and `http://vimeo.com/14291640` for footage from the system running in real-time

have separated optimizations from the computational kernels which would have provided a more maintainable code base.

We would also like to explore more types of fluid behaviour. For example, state changing from liquid to smoke would be very interesting. We are also interested in fluid-solid hybrids.

Finally, we would like to explore simulations that use particles as an intrinsic part of the simulation rather than as an afterthought to provide visual flair.

Bibliography

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[ABB+03] Mark Adler, Thomas Boutell, John Bowler, Christian Brunschen, Adam M. Costello, Lee Daniel Crocker, Andreas Dilger, Oliver Fromme, Jean-loup Gailly, Chris Herborth, Alex Jakulin, Neal Kettler, Tom Lane, Alexander Lehmann, Chris Lilley, Dave Martindale, Owen Mortensen, Keith S. Pickens, Robert P. Poole, Glenn Randers-Pehrson, Greg Roelofs, Willem van Schaik, Guy Schalnat, Paul Schmidt, Michael Stokes, Tim Wegner, and Jeremy Wohl. Portable Network Graphics (PNG) Specification (Second Edition). W3C Recommendation 10 November 2003, World Wide Web Consortium (W3C), 2003.

[App06] Apple Inc. Kernel Programming Guide: Synchronization Primitives. 2006. URL: <http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/synchronization/synchronization.html%23//apple_ref/doc/uid/TP30000905-CH218-TPXREF110>.

[App07] Apple Inc. *Dictionary.app*, chapter New Oxford American Dictionary, 2nd Edition. Apple Inc., 2007.

[App08a] Apple Inc. Hardware - SSE Performance Programming. 2008. URL: <http://developer.apple.com/hardwaredrivers/ve/sse.html>.

[App08b] Apple Inc. Hardware - Throughput and Latency in Vector Programming. 2008. URL: <http://developer.apple.com/hardwaredrivers/ve/software_pipelining.html>.

[bib11] nVidia GeForce 8800M GTS Video Card, 01 2011. URL: <http://www.gpureview.com/geforce-8800m-gts-card-570.html>.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime

system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[CBP05] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In *Symposium on Computer Animation 2005*, pages 219–228, 2005.

[CFL67] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. Res. Dev.*, 11(2):215–234, 1967.

[Chi01] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *SIGPLAN Not.*, 36:191–202, May 2001. URL: <`http://doi.acm.org/10.1145/381694.378840`>.

[CLT08] Keenan Crane, Ignacio Llamas, and Sarah Tariq. *GPU Gems 3*, chapter Real-Time Simulation and Rendering of 3D Fluids, pages 633–675. Addison-Wesley, Upper Saddle River, NJ, 1 edition, 2008.

[CM03] Alexandre Joel Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer-Verlag, New York, 3 edition, 2003.

[DHK+99] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, Ulrich R Ude, and Christian Weiß. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal*, 10:21–40, 1999.

[DPS10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10, pages 185–192, Washington, DC, USA, 2010. IEEE Computer Society.

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21(3):736–744, 2002.

[ETK+05] S. Elcott, Y. Tong, E. Kanso, P. Schröder, and M. Desbrun. Discrete, vorticity-preserving, and stable simplicial fluids. In *ACM SIGGRAPH 2005 Courses*, page 9. ACM, ACM, 2005.

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, New York, NY, USA, 2001. ACM.

[FM96] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471 – 483, 1996.

[FM97a] N. Foster and D. Metaxas. Controlling fluid animation. In *Computer Graphics International*, volume 97, pages 178–188, 1997.

[FM97b] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM New York, NY, USA, ACM.

[GBO04] Tolga G. Goktekin, Adam W. Bargteil, and James F. O'Brien. A method for animating viscoelastic fluids. *ACM Trans. Graph.*, 23(3):463–468, 2004.

[GBST06] Richard Gerber, Aart Johannes Casimir Bik, Kevin B. Smith, and Xinmin Tian. *The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms*. Intel Press, Hillsboro, Ore., 2nd edition, 2006.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, November 1977.

[Gol66] S. W. Golumb. Run-length encodings. *IEEE Transactions on Information Theory*, 12:399–401, July 1966.

[GS06] D. Vaughan Griffiths and Ian Moffat Smith. *Numerical Methods for Engineers*. Chapman & Hall/CRC, Boca Raton, Fla, 2 edition, Jun 2006.

[Har04] Mark J. Harris. *GPU Gems - Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter Fast Fluid Dynamics Simulation on the GPU, pages 637–665. Addison-Wesley, 2004.

[HBSL03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[HCB70] C. W. Hirt, J. L. Cook, and T. D. Butler. A Lagrangian Method for Calculating the Dynamics of an Incompressible Fluid with Free Surface. *Journal of Computational Physics*, 5:103–+, February 1970.

[HHK08] Woosuck Hong, Donald H. House, and John Keyser. Adaptive particles for incompressible fluid simulation. *The Visual Computer*, 24(7):535–543, 07 2008.
URL: <`http://dx.doi.org/10.1007/s00371-008-0234-z`>.

[HS81] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *ARTIFICAL INTELLIGENCE*, 17:185–203, 1981.

[Hu61] TC Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[HV93] Paul G. Howard and Jeffrey Scott Vitter. Fast and efficient lossless image compression. In *in Proc. 1993 Data Compression Conference, (Snowbird)*, pages 351–360, 1993.

[HW65] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[I.B08] I.B.M. *Programming Tutorial*. I.B.M., 3.1 edition, 2008.

[I.B09] I.B.M. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*. I.B.M., version 1.12 edition, April 2009.

[IEC10] IEC. Prefixes for binary multiples. 2010.
URL: <http://www.iec.ch/zone/si/si_bytes.htm>.

[Int] Intel Corporation. *Intel® microprocessor export compliance metrics*.
URL: <http://www.intel.com/support/processors/sb/cs-023143.htm>.

[Int06] Intel Corporation. *Intel® 965 Express Chipset Family Memory Technology and Configuration Guide*. Intel, 2006.

[Int08] Intel Corporation. x87 and SSE Floating Point Assists in IA-32: Flush-To-Zero (FTZ) and Denormals-Are-Zero (DAZ). 2008.
URL: <http://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz/>.

[Int09] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 020 edition, November 2009.
URL: <http://www.intel.com/products/processor/manuals/>.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[KC08] Pijush K Kundu and Ira M Cohen. *Fluid Mechanics*. Academic Press, Amsterdam; Boston, 4th edition edition, 2008.

[Ker08] Phil Kerly. Cache blocking technique on hyper-threading technology enabled processors. *Intel Software Network*, 2008.
URL: <http://software.intel.com/en-us/articles/cache-blocking-technique-on-hyper-threading-technology-enabled-processors/>.

[KHN07] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
URL: <`http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`>.

[Kim08] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 99–106. ACM, 2008.

[KM90] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57, New York, NY, USA, 1990. ACM.

[KRWK00] Markus Kowarschik, Ulrich Rüde, Christian Weiß, and Wolfgang Karl. Cache-aware multigrid methods for solving poisson's equation in two dimensions. *Computing*, 64:381–399, 2000.
URL: <`http://dx.doi.org/10.1007/s006070070032`>.

[Lax67] Peter D. Lax. Hyperbolic difference equations: a review of the courant-friedrichs-lewy paper in the light of recent developments. *IBM J. Res. Dev.*, 11(2):235–238, 1967.

[Lev] David Levinthal. Cycle Accounting Analysis on Intel Core 2 Processors.
URL: <`http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-white-papers/`>.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 457–462, New York, NY, USA, 2004. ACM.

[LTKF08] Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. Two-Way Coupled SPH and Particle Level Set Fluid Simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[Luc77] L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.

[LZF10] Michael Lentine, Wen Zheng, and Ronald Fedkiw. A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Trans. Graph.*, 29(4), 2010.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, page 159. Eurographics Association, 2003.

[Mon92] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

[NL06] L. Null and J. Lobur. *The essentials of computer organization and architecture*. Jones & Bartlett Publishers, 2nd edition, 2006.

[NVI09a] NVIDIA Corporation. *NVIDIA OpenCL Best Practices Guide*, 2009.

[NVI09b] NVIDIA Corporation. *OpenCL Programming Guide for the CUDA Architecture*, 2009.

[PE09] Michael Perrone and Robert Enenkel. I.B.M. Cell Broadband Engine Tutorial. Tutorial, 2009.

[PEA+96] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 60–71, New York, NY, USA, 1996. ACM.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK; New York, 3 edition, September 2007.

[Ree83] W. T. Reeves. Particle Systems – a Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[RO98] R. Radovitzky and M. Ortiz. Lagrangian finite element analysis of newtonian fluid flows. *International Journal for Numerical Methods in Engineering*, 43(4):607–619, 1998.
URL: <http://dx.doi.org/10.1002/(SICI)1097-0207(19981030)43:4<607::AID-NME399>3.0.CO;2-N>.

[Ros02] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill Higher Education, 2002.

[SBH09] Funshing Sin, Adam W. Bargteil, and Jessica K. Hodgins. A point-based method for animating incompressible flow. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Aug 2009.

[SF95] J. Stam and E. Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 129–136. ACM New York, NY, USA, 1995.

[SFK+08] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An Unconditionally Stable MacCormack Method. *Journal of Scientific Computing*, 35(2-3):350–371, 2008.

[SFR09] Xin Wei Sha, Michael Fortin, and Jean-Sébastien Rousseau. Calligraphic video: a phenomenological approach to dense visual interaction. In *MM '09: Proceedings of the seventeenth ACM international conference on Multimedia*, pages 1091–1100, New York, NY, USA, 2009. ACM.

[SJ04] Raymond A. Serway and John W. Jewett. *Physics for scientists and engineers.* Thomson-Brooks/Cole, Belmont, CA, 6th edition, 2004.

[SKG02] F. J. Seinstra, D. Koelma, and J. M. Geusebroek. A software architecture for user transparent parallel image processing. *Parallel Computing*, 28(7-8):967 – 993, 2002. URL: <http://www.sciencedirect.com/science/article/B6V12-46FG8KF-1/2/ae1c9784b528918e6005a247b6199ced>.

[SKR07] M. Stürmer, H. Köstler, and U. Rüde. A fast full multigrid solver for applications in image processing. *Numerical Linear Algebra with Applications*, 15(2-3):187–200, December 2007.

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 910–914, New York, NY, USA, 2005. ACM.

[SRWH97] Linda Stals, Ulrich Rüde, Christian Weiß, and Hermann Hellwagner. Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations. In *Proceedings of the The Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Str08] Gilbert Strang. Computational Science and Engineering I, 2008. URL: <http://ocw.mit.edu/courses/mathematics/18-085-computational-science-and-engineering-i-fall-2008/>.

[SU94] John Steinhoff and David Underhill. Modification of the Euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluids*, 6:2738–2744, August 1994.

[TLHD03] Yiying Tong, Santiago Lombeyda, Anil N. Hirani, and Mathieu Desbrun. Discrete multiscale vector field decomposition. *ACM Trans. Graph.*, 22(3):445–452, 2003.

[TLP06] A. Treuille, A. Lewis, and Z. Popović. Model reduction for real-time fluids. *ACM Transactions on Graphics (TOG)*, 25(3):834, 2006.

[TMPS03] A. Treuille, A. McNamara, Z. Popović, and J. Stam. Keyframe control of smoke simulations. In *ACM SIGGRAPH 2003 Papers*, page 723. ACM, 2003.

[Weia] Eric W. Weisstein. Courant-Friedrichs-Lewy Condition. *Wolfram MathWorld – A Wolfram Web Resource.*
URL: <http://mathworld.wolfram.com/Courant-Friedrichs-LewyCondition.html>.

[Weib] Eric W. Weisstein. Divergence theorem. *Wolfram MathWorld – A Wolfram Web Resource.*
URL: <http://mathworld.wolfram.com/DivergenceTheorem.html>.

[Weic] Eric W. Weisstein. Euler, Leonhard (1707-1783). *Eric Weisstein's World of Scientific Biography.*

[Weid] Eric W. Weisstein. Run-length encoding. *MathWorld – A Wolfram Web Resource.*
URL: <http://mathworld.wolfram.com/Run-LengthEncoding.html>.

[Wei98] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 127–138, New York, NY, USA, 1998. ACM.

[WST09] Martin Wicke, Matt Stanton, and Adrien Treuille. Modular bases for fluid dynamics. *ACM Transactions on Graphics (TOG)*, 28(3), August 2009.

Appendix A

Math

Unable to put aside our curiosity regarding the incompressible Navier Stokes equations and their potential discretizations, we found ourselves doing a brief review of certain topics normally found in a numerical methods course.

First: having seen so many different styles to express mathematical equations in the literature, we decided to describe our notation.

Second: we review some basic concepts underlying grid-based computations, mainly finite differencing methods.

A.1 Notation

The notation used within this document is quite standard[1] with the exception of a few notational oddities. Therefore the mathematical notation we used is documented here for completeness. Individual symbols with physical meaning are documented within the next appendix starting on page 127.

Vectors are in a bold font: $\boldsymbol{u}$.

Scalars are in italics: s.

Fields have no differentiating notation, and are either pointed out in the text or implied by the computation at hand. For example, a vector field and a vector scalar would both appear in an identical bold font.

However, subscripts on vector or scalar fields can indicate specific elements. For example, in the 2D case, $\boldsymbol{u}_{x,y}$ (equivalent to $\boldsymbol{u}(x, y)$) indicates the value of $\boldsymbol{u}$ at location x, y.

When dealing with staggered grids we have to deal with cases where data is not on grid-cell centres. For example, assuming a distance of 1 between grid cells, $\boldsymbol{u}_{x+0.5,y}$ can be read as the

[1] In the limit in which that is possible...

value between $\boldsymbol{u}_{x,y}$ and $\boldsymbol{u}_{x+1,y}$. It is safe to assume that linear interpolation is used to get the desired value. Depending upon the circumstances, different interpolation schemes may be used. On a staggered grid, since values are picked off-center of the grid, the values of $u_{x,y}$ and $v_{x,y}$ are determined as follows[HW65]:

$$u_{x,y} \equiv \frac{1}{2}\left(u_{x-\frac{1}{2},j} + u_{x+\frac{1}{2},j}\right) \tag{A.1}$$

$$v_{x,y} \equiv \frac{1}{2}\left(v_{x,y-\frac{1}{2}} + v_{x,y+\frac{1}{2}}\right) \tag{A.2}$$

$$u_{x+\frac{1}{2},y+\frac{1}{2}} \equiv \frac{1}{2}\left(u_{x+\frac{1}{2},y} + u_{x+\frac{1}{2},y+1}\right) \tag{A.3}$$

$$v_{x+\frac{1}{2},y+\frac{1}{2}} \equiv \frac{1}{2}\left(v_{x,y+\frac{1}{2}} + v_{x+1,y+\frac{1}{2}}\right) \tag{A.4}$$

At times it is necessary to only refer to specific component of a vector. That information is put in the super-script using variables x, y, and z for the first, second, and third component such as: $\boldsymbol{x} = \{\boldsymbol{x}^x, \boldsymbol{x}^y, \boldsymbol{x}^z\}$. If ever the letter t appears in the superscript, then it refers to time. If multiple dimensions and/or time need to be specified, then they'll be separated by commas, for example: $\boldsymbol{x}^t = \{\boldsymbol{x}^{x,t}, \boldsymbol{x}^{y,t}, \boldsymbol{x}^{z,t}\} = \{\boldsymbol{x}^{x,y,z,t}\}$. The reason for this non-standard notation is to avoid confusion between subscripts indicating position in a vector field with the data indicating which component of a vector to pick. We will be verbose within the text once powers and components become indistinguishable.

Certain sources represent the Laplacian as $\hat{}$, others as Δ, some as $\nabla \cdot \nabla$, to the extreme of $\nabla^T \nabla$, and then there's the preferred notation used in this document; ∇^2.

Dimension refers to the size of vectors, and not the space spanned by a field. For example, $\boldsymbol{x} \in \mathbb{R}^3$ means that $\boldsymbol{x}$ has 3 components. If $\boldsymbol{x}$ is a field, then it is a plane unless otherwise specified.

A.2 Grid-Based Computations

We discretize the incompressible Navier Stokes equations on rectangular grids using finite differencing equations. We give an overview of pertinent topics within numerical methods below.

First: we provide an overview of finite differencing equations with information about their accuracy.

Second: we introduce solving PDEs as a simultaneous solution of a system of equations.

Finite Differencing

The derivative of $f(x)$, written as either $f'(x)$ or $\frac{df(x)}{dx}$ can be expressed as:

$$\frac{df(x)}{dx} = lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \tag{A.5}$$

Finite differencing drops the limit to obtain an approximation known as the forward difference:

$$\frac{df(x)}{dx} \approx \frac{f(x+h)-f(x)}{h} \tag{A.6}$$

h is the distance between samples. The smaller h becomes, the more accurate the solution. (A.6) is first-order accurate; as shown through a Taylor Series expansion around h[Str08]:

$$\frac{df(x)}{dx} \approx \frac{f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+...-f(x)}{h} \tag{A.7}$$

$$\approx f'(x)+\frac{1}{2}hf''(x)+\frac{1}{6}h^2f'''(x)+... \tag{A.8}$$

$$0 \approx \frac{1}{2}hf''(x)+\frac{1}{6}h^2f'''(x)+... \tag{A.9}$$

As visible in (A.9), errors start to creep in at the second derivative. $f'(x) \equiv \frac{df(x)}{dx}$ is subtracted from both sides of the equation leaving the error term behind.

The finite difference equation used in this document to approximate the first order derivative is different than (A.6) which we will show to be better. First, we shall look at an equivalent representation of the derivative known as the backward difference:

$$\frac{df(x)}{dx} \approx \frac{f(x)-f(x-h)}{h} \tag{A.10}$$

If the forward and backward differences are summed, we obtain the central difference:

$$\frac{2df(x)}{dx} \approx \frac{f(x+h)-f(x-h)}{h} \tag{A.11}$$

$$\frac{df(x)}{dx} \approx \frac{f(x+h)-f(x-h)}{2h} \tag{A.12}$$

The forward and backward differences are of first order accuracy. The central difference is of

second order accuracy, as confirmed by the following Taylor Series expansion[Str08][PTVF07]:

$$
\begin{aligned}
f'(x) &= \frac{df(x)}{dx} && \text{(A.13)}\\
&= \frac{f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+...-(f(x)-hf'(x)+\frac{1}{2}h^2f''(x)-\frac{1}{6}h^3f'''(x)+...)}{2h}\\
&= \frac{2hf'(x)+\frac{2}{6}h^3f'''(x)+...}{2h} && \text{(A.14)}\\
&= f'(x)+\frac{1}{6}h^3f'''(x)+... && \text{(A.15)}\\
0 &= \frac{1}{6}h^3f'''(x)+... && \text{(A.16)}
\end{aligned}
$$

We can see that the second-order error that appeared in (A.9) is cancelled out in (A.16).

The finite second derivative is derived from both the forward and backward difference. The reason is to reduce the number of sample points thus simplifying boundary conditions.

$$
\begin{aligned}
\frac{d^2f(x)}{d^2x} &= \frac{d}{dx}\left(\frac{df(x)}{dx}\right) && \text{(A.17)}\\
&\approx \frac{d}{dx}\left(\frac{f(x+h)-f(x)}{h}\right) && \text{(A.18)}\\
&\approx \frac{f(x+h)-f(x)-f(x+h-h)+f(x-h)}{h^2} && \text{(A.19)}\\
&= \frac{f(x+h)-2f(x)+f(x-h)}{h^2} && \text{(A.20)}
\end{aligned}
$$

Using a Taylor Series expansion, it is possible to show that the finite difference approximation is accurate to the third order:

$$
\begin{aligned}
f''(x) &= \frac{d^2f(x)}{d^2x} && \text{(A.21)}\\
&= \frac{f(x)+hf'(x)+\frac{h^2f''(x)}{2}+\frac{h^3f'''(x)}{6}+...-2f(x)+f(x)-hf'(x)+\frac{h^2f''(x)}{2}-\frac{h^3f'''(x)}{6}+...}{h^2}\\
&= f''(x)+\frac{1}{12}h^4f''''(x)+... && \text{(A.22)}\\
0 &= \frac{1}{6}h^3f'''(x)+... && \text{(A.23)}
\end{aligned}
$$

Therefore, errors start to creep in when the function that the grid approximates is fourth order or higher. Visually, this can be seen by the truncation error, what is left after elements have been cancelled out, in (A.23)[PTVF07].

Using the central difference and second-order difference equations, the usual vector calculus

operations can be defined as[Har04]:

$$\begin{aligned}
\nabla f(x,y) &= \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right) \\
&\approx \left(\frac{f_{x+\Delta x,y} - f_{x-\Delta x,y}}{2\Delta x}, \frac{f_{x,y+\Delta y} - f_{x,y-\Delta y}}{2\Delta y}\right) \\
\nabla \cdot f(x,y) &= \frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y} \\
&\approx \frac{f_{x+\Delta x,y} - f_{x-\Delta x,y}}{2\Delta x} + \frac{f_{x,y+\Delta y} - f_{x,y-\Delta y}}{2\Delta y} \\
\nabla^2 f(x,y) &= \nabla \cdot \nabla f(x,y) \\
&= \frac{\partial^2 f(x,y)}{\partial^2 x} + \frac{\partial^2 f(x,y)}{\partial^2 y} \\
&\approx \frac{f_{x+\Delta x,y} - 2f_{x,y} + f_{x-\Delta x,y}}{(\Delta x)^2} + \frac{f_{x,y+\Delta y} - 2f_{x,y} + f_{x,y-\Delta y}}{(\Delta y)^2}
\end{aligned}$$

Within the text, we use $\Delta\tau$ in the place of Δx and Δy since we assume that the individual discrete cells are square.

The finite difference equations shown within this section are at the heart of our fluid simulation. Next, we show how they are solved over rectangular grids.

Matrices

Let $\boldsymbol{u}_{x,y} : \mathbb{R}^2 \to \mathbb{R}^2$ be a function whose result is a vector describing the velocity at position (x, y). For convenience, we see each (x, y) as a bounded lookup on a discrete grid that approximates an underlying continuous function. For convenience, we say $x, y \in \mathbb{I}$.

When solving a finite difference equation; each grid cell within the rectangular grid can be seen as an equation that must be solved with respect to adjacent cells. We are looking for a solution that will simultaneously solve an equation per grid cell.

For example, we'll look at $\frac{\partial f}{\partial t} = \nabla \cdot f(x, y)$ which could be approximated as the following:

$$\frac{f^{t+\Delta t} - f^t}{\Delta t} = \frac{f_{x+1,y} - f_{x-1,y} + f_{x,y+1} - f_{x,y-1}}{2} \tag{A.24}$$

$$f^{t+\Delta t} = \Delta t \left(\frac{f_{x+1,y} - f_{x-1,y} + f_{x,y+1} - f_{x,y-1}}{2}\right) - f^t \tag{A.25}$$

Which can be rewritten as a series of equations over a discrete grid:

$$f_{1,1} = -\Delta t\frac{1}{2}f_{1,0} - \Delta t\frac{1}{2}f_{0,1} + \Delta t\frac{1}{2}f_{2,1} + \Delta t\frac{1}{2}f_{1,2} - f_{1,1} \quad \text{(A.26)}$$

$$f_{2,1} = -\Delta t\frac{1}{2}f_{2,0} - \Delta t\frac{1}{2}f_{1,1} + \Delta t\frac{1}{2}f_{3,1} + \Delta t\frac{1}{2}f_{2,2} - f_{2,1} \quad \text{(A.27)}$$

$$f_{1,2} = -\Delta t\frac{1}{2}f_{1,1} - \Delta t\frac{1}{2}f_{0,2} + \Delta t\frac{1}{2}f_{2,2} + \Delta t\frac{1}{2}f_{1,3} - f_{1,2} \quad \text{(A.28)}$$

$$f_{2,2} = -\Delta t\frac{1}{2}f_{2,1} - \Delta t\frac{1}{2}f_{1,2} + \Delta t\frac{1}{2}f_{3,2} + \Delta t\frac{1}{2}f_{2,3} - f_{2,2} \quad \text{(A.29)}$$

$$\ldots \quad \text{(A.30)}$$

Each of these equations can be described by the following kernel. For a change in time of 2 (for clarity), wherever the kernel is overlaid on the source grid we obtain the appropriate multipliers for the underlying values:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & -1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{(A.31)}$$

This series of n equations with n unknowns can be written in the form of a sparse matrix known as a tridiagonal matrix with fringes[PTVF07] – not shown as it is quite big. Each entry on the main diagonal corresponds to the middle entry in the 3x3 window.

Solving the system of equations (using an iterative scheme, for example) solves $\nabla f(x, y)$. Unfortunately, the fully explicit scheme that we describe here is unstable for large time-steps.

Appendix B

Physics

We do not claim to be physicists, however since we deal with physical phenomena physics is an unavoidable topic. There are two items we wish to elaborate on.

First: Several symbols we have consistently used to refer to specific physical properties. Below we list these symbols and how we use them.

Second: we give a short overview of the motion of particles. Our overview should answer any pending questions about massless marker particles.

B.1 Symbols

For completeness, the symbols used within this document can be found below. We included them in the appendix since most of them are standard and are liberally used within the literature. Within the text we will be explicit when we recycle a symbol for a new purpose.

Δt	Change in time, usually used to represent the amount of time passed within a single step of the simulation.
t	A variable representing absolute time. $t + \Delta t$ is used to refer to the next iteration in the simulation. t in the superscript of a variable refers to the value of that variable at time t. $\boldsymbol{x}^t$ refers to the value of $\boldsymbol{x}$ at time t.
$\boldsymbol{u}$	A vector representing the velocity. $\boldsymbol{u} = \frac{d\boldsymbol{x}}{dt}$ where $\boldsymbol{x}$ is the position of a potentially moving body. u and v refer to the respective x and y components of $\boldsymbol{u}$.
$\boldsymbol{a}$	Acceleration, defined as $\boldsymbol{a} = \frac{d\boldsymbol{u}}{dt}$
p	Pressure. A repulsive force arising from particles pushing against each other.
T	Temperature.

$\boldsymbol{x}$	Denotes a position. Typically the position of any particle currently under consideration.
ρ	Density. The more particles that are bunched together, the higher the density.
$\Delta\tau$	When discretizing an equation onto a fixed grid, $\Delta\tau$ is the size of the width and height of the grid.
$\hat{\boldsymbol{n}}$	Normal vector to the surface of a region. Given a function $\boldsymbol{f}$ describing the shape of the region, the normal is computed by $\frac{\nabla \boldsymbol{f}}{\lvert\nabla \boldsymbol{f}\rvert}$.
m	Mass. The amount of matter within a body[App07].
P	The set of all particles in the system. The position of an individual particle is denoted as $\boldsymbol{x}_i, i \in P$ and the particle's velocity as $\boldsymbol{u}_i$.

B.2 Motion of Particles

Several times we have alluded to the movement of particles. We provide a quick introduction to massless particles to complement the main text. Implementation details are left as an exercise for the reader.

We begin with a simple particle. Let's say it exists in $n \in \mathbb{I}^+$ dimensional space and has position $\boldsymbol{x} \in \mathbb{R}^n$.

If the particle were alone in space within its own universe, then nothing would affect it. That is, if it were moving at a certain speed, it would keep on moving. This constant motion is velocity. If $\boldsymbol{v} \in \mathbb{R}^n$ is velocity and t time, then $\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}$.

Let us suppose that the particle started to curve towards a planet, attracted by the planet's gravitational force. Then we say the particle is accelerating towards the planet. Acceleration is the change of velocity, denoted as $\boldsymbol{a} \in \mathbb{R}^n$, $\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$.

An accelerating force is something that either slows down or accelerates, in this case, the particle. It is our expectation that more force is required to push something that is heavier.

Let us say that we are applying a force $\boldsymbol{f} \in \mathbb{R}^n$ where $\boldsymbol{f}$ is a function of time. Then, $\boldsymbol{f} = m\boldsymbol{a}$, or equivalently $\boldsymbol{a} = \boldsymbol{f}/m$.

Massless marker particles have instantaneous velocity. That is, their velocity is determined by an external object – like flowing liquid, for example. In this case $\boldsymbol{f} = \boldsymbol{u}$ where $\boldsymbol{f}$ is the force from the fluid.

For example, assume we have a velocity field $\boldsymbol{u}$ containing a particle at position $\boldsymbol{x}$ at a given time t. Then the velocity of the particle is $\boldsymbol{u}_{\boldsymbol{x}^x, \boldsymbol{x}^y}$.

That is how we dealt with massless marker particles. For added variation, we later added some random mass within a given range.

Appendix C

Terminology

Depending upon the field that a person specializes in, they will potentially refer to the same thing using different words compared to another person in a different field. We attempt to clarify some of our vocabulary below.

accuracy The closer a simulation mimics what it's simulating, the more accurate the simulation.

compute unit In the context of OpenCL, something that can do calculations. For example, a processor such as a CPU or GPU.

fluid attribute Properties that follow the particles that logically make up a fluid. These properties can evolve, such as velocity, temperature, etc.

kernel In mathematics, kernel is used to denote the operation done on a small region of a field.

memory When used in the context of a computational unit, refers to the off-chip memory that the computational unit uses to store information. This is main memory for the CPU. 'Memory' with respect to a GPU refers either to main memory for shared memory systems or the GPU's own separate memory.

stability The higher the chances that a simulation will feed back into itself values that constantly get further away from normal values, the less stable the simulation is. An unconditionally stable simulation will compensate for outlandish values externally input into the system.

speed Can refer to velocity, or more abstractly to refer to the time needed to do something.

stencil In numerical methods, this can denote a small region of a field used to do computations. In computer graphics, stencil can refer to a buffer that is typically used as a mask.

window In pattern recognition, denotes a small area of a large image that is being examined at a given time.