# A HYBRID FLUID SIMULATION ON THE GRAPHICS PROCESSING UNIT (GPU)

A Thesis

by

REBECCA LYNN FLANNERY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2008

Major Subject: Visualization Sciences

# A HYBRID FLUID SIMULATION ON THE GRAPHICS PROCESSING UNIT (GPU)

A Thesis

by

REBECCA LYNN FLANNERY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

| | |
|---|---|
| Chair of Committee, | Donald H. House |
| Committee Members, | Ergun Akleman |
| | John Keyser |
| Head of Department, | Tim McLaughlin |

May 2008

Major Subject: Visualization Sciences

## ABSTRACT

A Hybrid Fluid Simulation
on the Graphics Processing Unit (GPU). (May 2008)
Rebecca Lynn Flannery, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Donald H. House

This thesis presents a method to implement a hybrid particle/grid fluid simulation on graphics hardware. The goal is to speed up the simulation by exploiting the parallelism of the graphics processing unit, or GPU. The Fluid Implicit Particle method is adapted to the programming style of the GPU. The methods were implemented on a current generation graphics card. The GPU based program exhibited a small speedup over its CPU based counterpart.

# TABLE OF CONTENTS

# LIST OF FIGURES

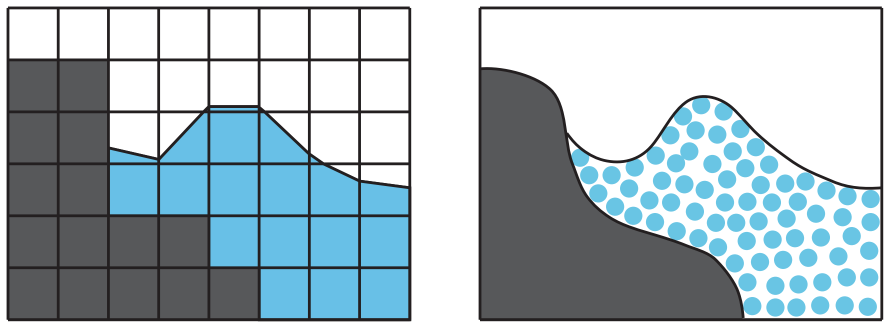
# CHAPTER I

# INTRODUCTION

Because use of liquid simulations is becoming increasingly common in the computer graphics industry, research on fluid simulation methods has become an important research direction in the field. The requirements for liquid simulations in computer graphics are slightly different from those in other fields. What is important is not physical accuracy but physical plausibility and appearance. The tradeoff between making a fast simulation and an accurate simulation is constantly an issue. Ideally a simulation should also be directable, giving the animator some degree of control over how the fluid will behave. This control could be something as simple as specifying a viscosity constant for the fluid, or it could be something more complicated, like specifying the exact frame in which a wave will begin to break.

Methods for simulating fluids fall into the two main categories indicated in Figure 1. Eulerian methods, Figure 1(a), use a grid to track where the fluid is. Fluid flows through the individual grid cells. Lagrangian methods, Figure 1(b), break the fluid up into discrete particles. Each particle holds a fraction of the fluid volume, and as the particle moves throughout the space, the fluid travels with it. Both methods have their drawbacks. Grid-based methods are usually faster, since they can be run on relatively course grids. However, the courser the grid, the more detail of the liquid surface is lost. Also it is hard to conserve momentum, and mass dissipation can be a problem. Solid obstacles must be approximated by sampling the geometry to the grid. Particle-based methods preserve detail of the liquid surface and guarantee mass conservation, but a lot of particles are needed because the particles must be present

The journal model is *IEEE Transactions on Visualization and Computer Graphics.*



(a) Grid-based (b) Particle-based

Fig. 1. Two types of fluid simulation.

everywhere throughout the fluid. Also, particle-based methods can run into stability problems when trying to enforce incompressibility. It is possible to use exact representation of obstacles for collision detection, although some methods approximate the geometry using immobile particles. More recently some hybrid methods combining grid and particle approaches have also emerged. Grid-based and particle-based methods tend to have complementary strengths and weaknesses. Hybrid methods try to exploit this characteristic and draw on the strengths of both.

Researchers are constantly looking for ways to make these fluid simulations faster. One fairly recent development has been the addition of programmability to graphics processing units, or GPUs. Within the past five years advances in the programmability of GPUs have made them suitable for accelerating not only graphics applications but also general purpose applications. The key to the GPU's speed is parallelism. Instead of operating on each piece of data separately, it performs the same operations on each piece of data simultaneously.

Given the GPU's specialized architecture, some algorithms are more compatible with its computational style than others. The two key factors that determine algorithm suitability are parallelism and independence. For parallelism, the same operations must be performed on every data element. Consider the case where a program can take one of two branches based on the value of the data element. If most of the elements take one branch but a few take the other branch, the GPU will end up running both branches for all the data elements. Thus a program that does not exhibit parallelism will not gain any speed from being transferred to the GPU. The other requirement is independence. Each data element must be relatively independent of other data elements. If other data elements must be used to do computations on the current elements, those data accesses must be predictable. For example, it is all right if every data element accesses the data elements on its right and left. But it is not acceptable if the data element has to access others at random. Grid-based computations are especially well suited to being run on the GPU. A rectangular grid is easily represented as a texture, with each grid cell corresponding to a single texture element, or *texel*. The same operation can be applied to each cell independently of the others.

The focus of this thesis is mapping a hybrid fluid simulation method onto programmable graphics hardware. I attempt to exploit the GPU's parallelism to make the fluid simulation run at more interactive frame rates. This speedup would be of use to animators who could preview the simulation and tweak the parameters to get it just right, instead of waiting a long time for the simulation to run every time.

# CHAPTER II

# BACKGROUND

## A. The Navier-Stokes Equations

The Navier-Stokes equations are a set of non-linear equations that describe the motion of a fluid at any point within the flow. The full equations can be manipulated into different forms based on the characteristics of the fluid. For a viscous, incompressible fluid the equations are the momentum equation

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2.1}$$

And the mass conservation equation

$$\nabla \cdot \mathbf{u} = 0 \tag{2.2}$$

Here the equations are given in vector form, and they are equally valid for both two and three dimensions. The fluid is described by two continuous fields, a velocity field $\mathbf{u}$ and a pressure field $p$. The equations describe how these two fields change over time.

The momentum equation (Equation 2.1) determines how the fluid accelerates due to the forces acting on it. Each term of the equation can be thought of as an acceleration due to a force. The easiest term to understand is the last one, $\mathbf{f}$. This term accounts for acceleration due to all external forces acting on the fluid. Generally, gravity is the only external force specified; however, external forces can also include things like wind blowing on a particular spot of the fluid surface or user specified forces. The second term, $-\frac{1}{\rho}\nabla p$, gives us the acceleration due to the gradient of the pressure. The term $\rho$ is fluid density, mass per unit volume. If the pressure at a particular point is equal on all sides, the net force will be zero. If, on the other hand, the pressure is higher on one side than on the other, a force will push from the higher pressure towards the lower pressure. The third term, $\nu\nabla^2\mathbf{u}$, accounts for acceleration due to viscosity. Intuitively, this can be thought of as each point in the fluid trying to move at the same velocity as nearby surrounding points. If the fluid on one side of a point is flowing faster, and the fluid on the other side of the point is flowing slower, the point will try to adjust its velocity towards the average of the velocities. The viscosity constant $\nu$ determines how strongly the point favors the surrounding velocities over its own velocity.

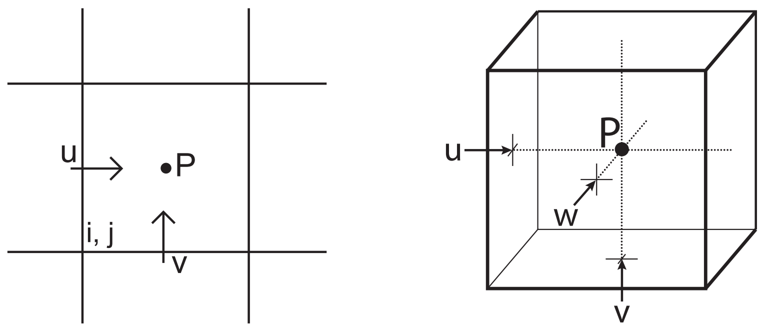
The first term of the momentum equation, $-(\mathbf{u} \cdot \nabla)\mathbf{u}$, is the hardest to understand. It accounts for the advection of the fluid. Advection is how any property of the fluid gets moved along according to the flow of the fluid. As an example, think of a point in space that has a temperature, $T$, associated with it. The temperature at that point could change in one of two ways. It could change simply because the point itself is heating up or cooling down, or it could change because the surrounding fluid flow causes hot air to be replaced by cold air. The latter case describes advection. In this particular term, it is the fluid momentum which is being advected.

The mass conservation equation (Equation 2.2) is responsible for enforcing incompressibility of the fluid. It states that the divergence of the fluid must be zero everywhere in the fluid. One way to think about this is to visualize a small chunk of fluid volume. Because the density of the fluid is constant, the total mass within the volume must also remain constant. When the fluid is sloshing around, some fluid will be entering and leaving the chunk. Thus, in order for the mass to remain constant, the rate of fluid entering the chunk must be exactly equal to the rate of the fluid leaving. Also, since the fluid is incompressible everywhere, this equation holds for all individual chunks of fluid and thus everywhere in the fluid.

## B. Eulerian Fluid Simulation Methods

The basis for most of the Eulerian methods currently used in computer graphics is the *marker and cell* method [1]. It was originally developed for the computational fluid dynamics (CFD) community and has since been adapted for use in computer graphics. In this method, a set of massless marker particles is used to track the position of the fluid. The particles have no effect on the motion of the fluid but are used only to determine whether a given cell contains any fluid. This is important to determine which cells are on the surface of the fluid. Cells are marked as solid, fluid, surface, or empty. A cell with a solid object in it, like part of a wall, is a solid cell. Cells with no particles are considered empty. A cell containing particles and next to at least one empty cell is a surface cell. All remaining cells are marked as fluid cells. The marker and cell method uses a *staggered-grid arrangement* for storing the velocities and pressures for each cell. For each grid cell the velocity components, $u$ in the $x$ direction and $v$ in the $y$ direction, are defined at the centers of each face and the pressure is defined at the cell center. This arrangement extends easily into three dimensions as shown in Figure 2.

One of the first attempts to use the Navier-Stokes equations for liquid simulations in the computer graphics field was made by Kass and Miller [2]. Rather than solving the full Navier-Stokes equations throughout the liquid volume, they used a simplified form of the equations called the shallow water equations. The shallow water equations are defined by three assumptions. First, the surface of the liquid is represented as a height field. This assumption puts the most serious limitations on the method, as there can be no free splashing of liquid or breaking of waves. Second, the vertical component of velocity can be ignored. Third, the horizontal component of velocity within a single vertical column of liquid is fairly constant from top to bottom. This



(a) 2D (b) 3D

Fig. 2. A staggered-grid arrangement. For a cell $(i,\ j)$ in two dimensions or a cell $(i,\ j,\ k)$ in three dimensions the pressure is defined at the cell center, and velocity components are defined at the center of each cell face.

assumption rules out situations where there is turbulence within the fluid. Kass and Miller solved the differential equations using an implicit method for stability. The method occasionally errs on the side of making the fluid gain volume. To correct this gain, they calculate the volume of fluid for the new time step, compare it to the volume for the previous time step, and subtract any added fluid, distributing the difference evenly over all the liquid samples. This method was implemented in both 2D and 3D.

Chen and da Vitoria Lobo [3] created a 3D liquid simulation by solving the 2D incompressible Navier-Stokes equations and using the computed pressures to get a height field. The height is a scaled value of the pressure, so a higher pressure yields a higher surface. Just like the previous method, there is no splashing or wave breaking. Solving the full 2D equations did introduce some advantages. The method took into account viscosity and could handle surface turbulence. In addition to having a pressure field at the surface, there was also a velocity field. They used this to move floating objects around with the current. They solved the equations explicitly instead of implicitly, which occasionally lead to instability problems. They set boundary conditions by setting the velocities on the boundary to zero for fixed objects and to the speed of the object for moving objects.

The first fully 3D physically-based fluid simulation for the graphics community was developed by Foster and Metaxas [4]. Their method is based largely on the work of Harlow and Welch [1]. In this method, the incompressible Navier-Stokes equations are solved over a regular rectangular mesh in three dimensions. They use the staggered-grid approach to store the velocities and pressures. The use of marker particles to track surface position is extended to include the addition and subtraction of particles at inflow and outflow boundaries. The Navier-Stokes equations are integrated explicitly. Because of this explicit integration the method is simple to code, but it is unstable for large time steps. If the fluid velocity is too large anywhere in the simulation domain, the simulation blows up. They compensate for this limitation by using a larger grid cell size, which allows the time step to be bigger but leads to a lack of detail in the fluid.

A detailed description of how to handle boundary conditions and the fluid surface is given. For boundary conditions, both the velocity and the pressure must be set to appropriate values. The component of velocity normal to an obstacle face is always zero because fluid cannot flow into or out of a solid obstacle. In Figure 3, $u$ must be set to zero since it is the velocity component normal to the obstacle face. The pressure in a solid cell is set equal to the pressure in the adjacent fluid cell. This ensures that there will not be an acceleration across the solid/fluid boundary due to a pressure gradient. The tangential velocity at the boundary must also be set, according to the desired conditions. For a no-slip boundary, the tangential velocity at the boundary is zero. In order to meet this condition, the tangential cell face velocity inside the obstacle cell is set equal and opposite to that of the fluid outside

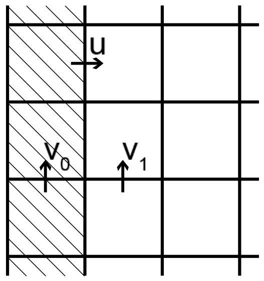


Fig. 3. Setting velocities at obstacle boundaries. The velocity component on an obstacle cell face must be zero ($u = 0$). For a free-slip boundary ($v_0 = v_1$). For a no-slip boundary ($v_0 = -v_1$).

($v_0 = -v_1$). For a free-slip boundary, the tangential boundary is set equal to that of the outside fluid ($v_0 = v_1$). Surface cells are handled differently. The velocity is set such that the divergence of fluid in the cell is zero. The pressure on surface cells is set to atmospheric pressure.

Stam [5] solved the problem of instability by switching from an explicit Eulerian solver of the Navier-Stokes equations to one that uses several approaches to dealing with the various components of the equations. This method was designed specifically for the use of the computer animation community, where the look of the liquid and the speed of the simulation are more important than numerical accuracy. The main advantage of this method it that it remains stable even at arbitrarily large time steps. The downside is that is suffers from too much numerical dissipation, even at relatively small time steps. The fluid flow loses vorticity and dampens too quickly. In extreme cases the fluid can lose mass.

Stam breaks the solution to the Navier-Stokes equations up into four incremental steps: add external force, advect, diffuse, and project. The results of each step are used as input to the next step. The addition of external forces is computed using a straightforward explicit Euler integration. To solve for the advection step, Stam introduces a technique called backtracing. In this technique, a virtual particle is introduced at the point where velocity should be updated. Then the virtual particle is traced backwards along a partial streamline, determining where it would have been at some time $t$ ago. The velocity at the original point is then updated to the velocity at the backtraced position. The diffusion step solves for the effect of viscosity. Stam chose to solve this step using an implicit form of the equation to ensure stability when the viscosity is large. Any of the first three steps may cause the velocity field to cease being divergence free. The final projection step returns the velocity field to a divergence free state by adjusting the pressure field. It is also solved using a Poisson solver, rather than the relaxation technique used by Foster and Metaxas.

The problem involving loss of vorticity was addressed by Fedkiw et al. in [6]. A technique called vorticity confinement is used to inject the energy lost due to numerical dissipation back into the fluid. At each time step they determine which areas have less vorticity than surrounding areas and inject a force to introduce vorticity back into that area. This technique was designed for simulating smoke, not liquid, and uses the inviscid form of the Navier-Stokes equations.

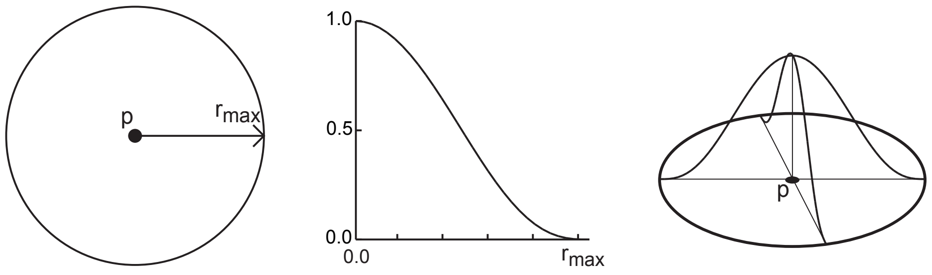
Foster and Fedkiw [7] introduced the particle level set method for surface tracking. The level set method was originally developed for the CFD community [8]. Rather than generate the surface every time step from a grid or marker particles, the level set method starts with a signed distance function that describes the surface and then evolves that surface over time. Since the method concentrates on the evolution of the surface, it does not take into account mass, and so is prone to volume loss. The particle level set method helps to correct the mass dissipation by using inertialess particles to help trace a moving implicitly defined surface. Particles that break free from the liquid surface are treated as small drops until they return to the main body of fluid. In the original level set, these small splashes would dissappear if they were smaller than the resolution of the underlying grid. The addition of the particles also allowed for more surface detail, since the resolution of the particles is finer than that of the grid.

## C. Lagrangian Fluid Simulation Methods

The first particle system for computer graphics was introduced by Reeves [9]. Particle systems use a cloud of primitive particles to define the volume of an object. The appearance and motion of the particles are controlled by procedural equations. The particles can then move and change appearance individually, affecting the appearance of the object as a whole. Generation of particles is just as important as the particle dynamics. Each particle is initialized with attributes like position, velocity, color, and lifespan. These initial attributes can be set to produce varying kind of effects. Later, Sims introduced several methods for particle choreography [10]. He described operations to model motions such as vortices, spirals, damping due to air resistance, and bouncing. Using these techniques, he created a simulation of a waterfall. The system worked well for this particular situation, but could not handle more general fluid simulations because the particles did not interact with each other.

Miller and Pearce used a particle system for animating viscous fluids [11]. This is the first known Lagrangian fluid simulation for computer graphics. In their system, particles interacted with each other. The equations used were more like spring equations and were not based on the Navier-Stokes equations.

The first two papers on 3D simulation of liquids using Lagrangian particle-based methods together with the Navier-Stokes equations both appeared in the same year. The first of these papers was a fairly fast method that could be used with interactive applications [12]. It is based on the Smoothed-Particle Hydrodynamics (SPH) method. In the SPH method each particle has a position, velocity, and pressure associated with it. As illustrated in Figure 4, each particle also has an area of influence that is weighted by a kernel. The area of influence goes from the particle center



(a) Particle and area of influence (b) Kernel profile (c) Weighted effect

Fig. 4. The shape of a weighted kernel. The area of influence and the shape of the kernel combine to create the weighted effect of a particle. Kernel weights determine the degree to which one particle affects another.

out to some maximum radius. The kernel function is used to determine how much influence the particle has on other nearby particles. A particle that is very close to another will be weighted more strongly than a particle farther away. Operations such as gradient and Laplacian are computed numerically using a weighted combination of particles within the kernel. Using these operations, forces on each particle due to pressure and viscosity can be computed. Unfortunately this method does not enforce incompressibility, and as a result the simulated liquid looks "springy."

The other paper [13] was based on the Moving Particle Semi-implicit (MPS) method [14]. This method is similar to SPH but is designed for simulating incompressible fluids. As in SPH, a weighted kernel is used as an area of influence for each particle. The forces on each particle are calculated, and the particles are moved to new temporary positions. At this point the density of particles is not the same throughout the fluid, which it should be in an incompressible fluid. A new step is added to correct for the particle density, and the particles are moved to their final locations.

## D. Hybrid Fluid Simulation Methods

One of the earliest numerical fluid simulation methods combining the Eulerian and Lagrangian viewpoints was the Particle-In-Cell (PIC) method [15]. This method was designed to simulate compressible fluids. The simulation area is represented by a fixed grid of equally sized cells, and the fluid is represented by particles. Fluid velocities and temperatures are stored on the grid, while fluid mass is stored in the particles. At the beginning of a time step, the density in each cell is calculated from the number and mass of particles in the cell. A state equation for the fluid is used to determine the pressure in a cell based on its density and temperature. Next, the acceleration due to the pressure gradient is computed, and this value is used to update the grid velocity to tentative values. The individual particles are then moved according to a velocity that is linearly interpolated from the nearest stored grid values. Then, the final velocity and temperature values for each cell are calculated. If no new particles have entered the cell, the tentative values are accepted as the final values. Otherwise, a weighted average of the values from the cell and the values from surrounding cells from which the new particles have come is computed. The PIC method is successful in modeling highly distorted fluid flow, but some numerical viscosity is caused by transferring the momentum back and forth from the grid to the particles.

The Fluid Implicit Particle (FLIP) method extends PIC to use an adaptive grid and attempts to reduce some of its numerical dissipation [16]. More information is stored in the particles, namely mass, velocity and internal energy. The algorithm proceeds similarly to PIC. At each time step, the particle quantities are transferred to a grid. The momentum equation is solved on the grid resulting in a new velocity field. These new velocities are subtracted from the starting grid velocities to get the change in velocity for the time step. Velocity difference is then transferred from the grid back to the particles and added to the velocity stored in each particle. Finally the particles are moved to new positions according to the grid velocities. FLIP was rediscovered and introduced to the computer graphics community by Zhu and Bridson [17]. Though the original FLIP was designed for compressible fluids, the authors here used a version in an unpublished manuscript from 1992 to adapt it for incompressible fluids.

## E. GPU Simulations

Harris et al. [18] solved the Navier-Stokes equations on the GPU as part of an interactive cloud simulation. Thermodynamics and the rates of evaporation and condensation were also taken into account. The fluid dynamics equations are solved using Stam's approach [5]. The simulation proceeds on a 3-dimensional staggered grid. Obstacles are limited to a single row of boundary cell surrounding the simulation volume. The fluid fills the whole volume, so there is no interface between fluid and air. For an introduction to fluid simulation on the GPU, the reader is referred to [19], which is based largely on the aforementioned work. It concentrates on the Navier-Stokes equations and includes implementation details for solving them using fragment programs. For simplification purposes the author uses a 2-dimensional simulation and cell-centered velocities.

Crane et al. [20] added arbitrary dynamic obstacles and a free surface between fluid and air. They use cell-centered velocities on a 3-dimensional grid. A MacCormack scheme is used for advection, rather than backtracing, to alleviate some of the numerical smoothing. To represent smoke, a simple color field is used. For liquids, which have a more defined surface, the level set method is used. Rendering of the fluid is done using a ray-marching fragment shader.

Particle simulations for the GPU were implemented by Kolb et al. [21] and Kipfer et al. [22]. The methods presented in both are similar. Both store particle data in textures, keep all data on the GPU, and use fragment programs to perform each step of the simulation. Kolb et al. represent objects implicitly using several 2-dimensional signed-distance depth maps. Particles can collide with these objects, but there is no inter-particle collision detection. Kipfer et al. handle inter-particle collisions. Scene objects are represented as a height field.

A fully particle-based fluid simulation on the GPU was described by Kolb and Cuntz [23]. The underlying method for their simulation was SPH. Rather than sorting particles to compute neighborhoods for interacting forces, they draw point sprites. The point sprite encodes how much influence the particle has at a given position P. Thus for a given position it is possible to add the contribution from surrounding particles without explicitly computing which particles are nearby.

# CHAPTER III

# METHODOLOGY

The goal of this work is to exploit the characteristics of the GPU to improve the performance of a fluid simulation. Therefore some details on how the GPU works are presented. The first section covers the graphics pipeline and which parts of it are programmable. The next section discusses the differences between programming for the CPU and programming for the GPU. Finally mapping the FLIP algorithm to the GPU is covered.

## A. The Programmable GPU

Data moves through the GPU in what is called the graphics pipeline. An overview is shown in Figure 5. Raw vertex data enters the pipeline. In the first stage vertices are transformed from object space into clip space. Then they are assembled into primitives like lines and triangles. These primitives are rasterized producing fragments. In the fragment processing stage these fragments undergo interpolation, texturing, and coloring to determine their final color. Finally fragment operations are performed. These operations are, in order, scissor test, alpha test, stencil test, depth test, blending, dithering, and logical operations. The final output of the graphics pipeline is a collection of pixels which are then written out to a buffer. With the addition of

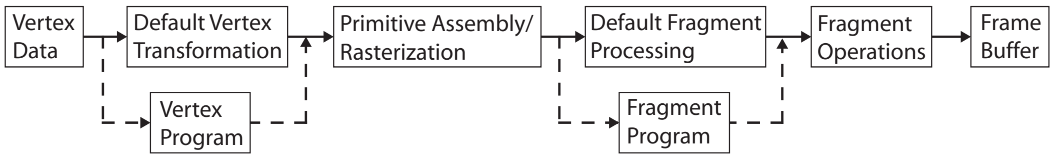

Vertex Data
Default Vertex Transformation
Vertex Program
Primitive Assembly/ Rasterization
Default Fragment Processing
Fragment Program
Fragment Operations
Frame Buffer

Fig. 5. The programmable graphics pipeline.

programmability to GPUs, the programmer can write her own custom vertex or fragment programs. These programs replace the corresponding default portions of the pipeline.

The vertex data can come from either the CPU or the GPU. It used to be that in order to use a texture as input to a fragment program, you had to copy it to the CPU and then back to the GPU. Now it is possible to take the output of the graphics pipeline and pipe it directly into the fragment processor. You can also do texture reads in vertex programs. You can also keep vertex information on the GPU without having to go through the CPU. Transferring data to and from the CPU is slow compared to keeping it on the GPU, so it is advantageous to keep data on the GPU whenever possible.

## B. Computation on the GPU

Mapping a program from the CPU to the GPU is not always a straightforward process. Computations on the GPU can be fast due to the highly parallel nature of the GPU. However, this parallelism also imparts certain restrictions. First it is important to understand how data can be stored and manipulated on the GPU.

Data is represented on the GPU as textures. In a typical program, you might have some numerical data stored in an array. If you wanted to manipulate the data on the CPU, you would probably cycle through the data using a for-loop and do calculations on each array entry individually. To perform the same operation on the GPU, you would convert the array into a texture. Then you would pass this texture to the GPU and perform your calculations on each texture element in parallel. In addition to the speed gained from performing calculations concurrently, another advantage is that texture elements can hold up to four values. For textures, these are usually thought of as the red, green, blue, and alpha components. For general purpose computing, however, they can be thought of as x, y, z, and w, or any other values the user would like. GPUs are optimized to perform operations on these four-vectors. For example, in a scalar multiplication, the hardware can multiply all four of the values by the scalar in a single instruction.

A texture cannot be read from and written to at the same time on the GPU. Thus in order to update a set of data, the current data must be read from one texture, the appropriate operations performed, and then the updated data must be written to a second texture. To update the data again, one must read from the second texture, and write back to the first texture. This process of shuffling data back and forth between two buffers is known as *ping-ponging*.

Compared to the speed at which data flows through the GPU, the rate at which data is transferred on and off of the GPU is relatively slow. Thus, when running programs on the GPU it is important to minimize copying of data in and out of graphics memory. Ideally any data that is going to be reused should be kept on the GPU. One technique commonly used to implement this goal is *render-to-texture*. In this technique, the output of a fragment program is rendered to a texture in graphics memory rather than the default frame buffer. This texture can then be used immediately as the input to another fragment program. The data does not have to be copied off the GPU at any point.

One important limitation of GPUs is their inability to perform scatter operations. The CPU equivalent of a scatter operation would be $a[i] = x$. In this case $i$ is a computed address which could be anywhere in the array $a$. On the GPU the output address is fixed. One way to get around this problem is to convert the scatter operation into a gather operation. The equivalent of a gather operation on the CPU is $a = x[i]$. In this case the output destination is fixed, but it can read its data from anywhere in the array $x$. One example of converting a scatter to a gather operation can be found in this FLIP program, when the particle velocities are transferred to the grid. Each particle has an area of influence that covers several grid points. In other words, each particle is trying to scatter it's velocity to several grid points. Another way to see this problem is from the point of view of the grid points. Each grid point is affected by multiple nearby particles and wants to gather the velocities from those particles.

## C. Implementing FLIP on the GPU

In this thesis the FLIP method has been adapted to run on the GPU. An outline of the algorithm is shown in Figure 6 for reference in the discussion below. Particles are used to represent the fluid. Though much of the work is performed on a grid, the particles hold the information about the fluid's position and velocity persistently. The grid used during a single time step is discarded at the end of the time step, which means adaptable grids can be used. A static grid is used in this work for ease of programming. Each step of the process is implemented using one or more fragment programs. The following sections explain how each step is implemented on the GPU.

### 1. Particles to Grid

The first step of the FLIP algorithm is to transfer velocities from the particles to the grid. This is done by computing a weighted average of nearby particle velocities for each velocity component on the grid. The main challenges involved in implementing this step are determining which particles influence each grid point, determining the weight of a particle's velocity on a particular grid point, and combining the contribu-

1. Transfer particle velocities to the grid
2. On the grid:
   - Add external forces
   - Diffuse velocities based on the fluid's viscosity
   - Project the velocity field onto a divergence free field
   - Set air/surface boundary velocity values to satisfy $\nabla \cdot \mathbf{u} = 0$
3. Subtract the new grid velocities from the saved grid velocities
4. Transfer the difference in velocity from the grid to the particles
5. Move the particles according to the new grid velocity field

Fig. 6. The FLIP algorithm.

tion of all the particles on a particular grid point.

First we must determine which particles influence each grid point. This turns out to be as easy as drawing points centered on each particle. If the points are rendered to an output buffer the size of the velocity texture, then each fragment produced during rasterization will correspond to a single grid velocity. The size of a point should be large enough to cover all grid points in its area of influence.

Point sprites can be used to determine the weight of a particle's velocity on a given grid point. The distance from the particle center to a grid point determines how much influence the particle has on that point. If the particles are rendered as points, there is no way to calculate that distance. Point sprites are similar to points but with texture coordinates added. A point sprite can be thought of as a quad whose upper-left corner has $(s, t)$ texture coordinates of $(0, 0)$ and whose lower-right corner has texture coordinates of $(1, 1)$. Figure 7 shows a point sprite centered at position $P$. When the point sprite is rasterized, multiple fragments are generated, including the one centered at $f$. The texture coordinate assigned to this fragment and passed to the fragment program reflects where the point sprite hits the point $f$. In the figure, the generated texture coordinate would be approximately $(0.7, 0.1)$.

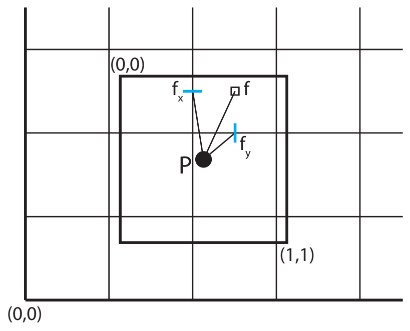


Fig. 7. Transferring velocity from a particle to the grid using point sprites. The texture coordinate of $f$ is passed to the fragment program. For staggered-grid velocities, the texture coordinates of $f_x$ and $f_y$ must be computed. Note that point sprites have texture coordinates of $(0, 0)$ in the upper left.

Knowing that the center of the particle is, by definition, located at texture coordinate $(0.5, 0.5)$ the distance from the particle center to the cell center can be computed and the weight determined. Since a staggered grid is being used, we actually want the distance from the particle center to $f_x$ and $f_y$. The texture coordinates at $f_x$ and $f_y$ can be computed knowing the point sprite size, $pSize$. If the texture coordinate at $f$ is $(s, t)$, then the texture coordinate at $f_x$ will be $(s - \frac{0.5}{pSize}, t)$ and the texture coordinate at $f_y$ will be $(s, t + \frac{0.5}{pSize})$.

At each grid point, the weighted velocities must be combined. Several particles may contribute to the velocity at each grid point. In the typical graphics pipeline, fragments that map to the same pixel cannot be combined. The value in the one that is closest to the camera is kept, and all of the rest are discarded. In our 2D particle simulation, where all the particles are at the same depth, the last particle to be drawn is the only one that shows up. But for this function, we need to sum the contribution from the various particles at each grid point. OpenGL provides a mechanism for this called blending. Using blending, instead of overwriting the value in the destination buffer, the value of any incoming fragments can be combined with the value already stored in the destination buffer. The user can choose exactly how the two values get combined. The general formula is $F_s S_{val} + F_d D_{val}$ , where $S_{val}$ and $D_{val}$ are the values stored in the source fragment and destination buffer, $F_s$ is the scale factor applied to the source fragment, and $F_d$ is the scale factor applied to the value in the destination buffer. The scale factor can have values like zero, one, alpha, or one minus alpha, where alpha is the value stored in the alpha component of the fragment. In this case a scale factor of one is used for both the source and destination, which results in a simple adding of the incoming value to the value already in the destination buffer. In the fragment program the weighted velocity component is stored in one channel of the output, and the weight is stored in another channel. After the contribution from all the particles is summed up, one additional fragment program pass is needed to normalize the accumulated weighted velocities by dividing by the summed weights.

### 2. External Forces

The next step of the process is to add external forces, such as gravity and user defined forces. Let $\mathbf{u_n}$ be the velocity field obtained after the transfer from particles to grid and $\mathbf{u_n^*}$ be the velocity field that will result from adding external forces for a time step $\Delta t$. Then this step of the algorithm can be expressed as

$$\mathbf{u_n^*} = \mathbf{u_n} + \Delta t \mathbf{f}. \tag{3.1}$$

The grid velocity obtained after transferring from the particles is already contained in one texture. Another texture of the same size contains the external forces. The values in the force texture are multiplied by the time step and the result is added to the values in the velocity texture. External forces should not be added to velocity components on Empty/Empty cell edges or edges bordering at least one Solid cell.

In this simulation there are no external forces other than gravity. Since the value of gravity is the same throughout the field, there is no need to use a texture to represent that field. A gravity constant is defined in the fragment program and added like any other external force.

### 3. Diffusion

Viscosity is a measure of a fluid's resistance to flow. In this step velocity diffusion due to viscosity is computed. The implicit form of the equation,

$$(\mathbf{I} - \nu \Delta t \nabla^2)\mathbf{u}_\mathbf{n}^{**} = \mathbf{u}_\mathbf{n}^{*}, \tag{3.2}$$

is used for stability. This equation may not seem to make sense considering that the Laplacian operator $\nabla^2$ operates on scalar fields, and $\mathbf{u}_\mathbf{n}^{**}$ is a vector field. However it is simply solving two scalar fields, $u_{n\ x}^{**}$ and $u_{n\ y}^{**}$, at the same time.

*Jacobi iteration* can be used to solve the equation. It is used to solve systems of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a matrix that has only nonzero diagonal elements. Using this method, an approximate solution is found for each diagonal matrix entry and these values are plugged into the next iteration. The iteration continues until the solution converges. This method is especially suited to coding on the GPU since each equation is treated independently. The Jacobi iteration for an $n$-by-$n$ system is defined as:

for $i = 1$ to $n$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \tag{3.3}$$

the derivation of which is shown in [24]. Note that to solve for an element of the $\mathbf{x}$ vector for iteration $k + 1$, entries from the previous iteration $k$ are used.

Equation 3.2 can be expressed in matrix form as

$$\left(\mathbf{I} - \frac{\nu\Delta t}{\Delta x^2}\begin{bmatrix} \ddots & & \ddots & \ddots & \ddots & & \ddots & \\ & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 \\ & & \ddots & & \ddots & \ddots & \ddots & & \ddots \end{bmatrix}\right)\begin{bmatrix} \vdots \\ u_{i-1,j}^{(k+1)} \\ \vdots \\ u_{i,j-1}^{(k+1)} \\ u_{i,j}^{(k+1)} \\ u_{i,j+1}^{(k+1)} \\ \vdots \\ u_{i+1,j}^{(k+1)} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ u_{i-1,j}^{(k)} \\ \vdots \\ u_{i,j-1}^{(k)} \\ u_{i,j}^{(k)} \\ u_{i,j+1}^{(k)} \\ \vdots \\ u_{i+1,j}^{(k)} \\ \vdots \end{bmatrix}. \quad (3.4)$$

Some rearranging yields

$$\begin{bmatrix} \ddots & & \ddots & & \ddots & & \ddots & & \ddots & \\ & 1 & \cdots & 1 & & -4 - \frac{\Delta x^2}{\nu\Delta t} & & 1 & \cdots & 1 \\ & & \ddots & & \ddots & & \ddots & & \ddots & & \ddots \end{bmatrix}\begin{bmatrix} \vdots \\ u_{i-1,j}^{(k+1)} \\ \vdots \\ u_{i,j-1}^{(k+1)} \\ u_{i,j}^{(k+1)} \\ u_{i,j+1}^{(k+1)} \\ \vdots \\ u_{i+1,j}^{(k+1)} \\ \vdots \end{bmatrix} = -\frac{\Delta x^2}{\nu\Delta t}\begin{bmatrix} \vdots \\ u_{i-1,j}^{(k)} \\ \vdots \\ u_{i,j-1}^{(k)} \\ u_{i,j}^{(k)} \\ u_{i,j+1}^{(k)} \\ \vdots \\ u_{i+1,j}^{(k)} \\ \vdots \end{bmatrix}. \quad (3.5)$$

Using the formula from Equation 3.3 on a row of the matrix results in the following generic equation for a single element:

$$u_{i,j}^{(k+1)} = \frac{1}{4\nu\Delta t + \Delta x^2}\left[\nu\Delta t\left(u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + u_{i+1,j}^{(k)}\right) + \Delta x^2 u_{i,j}^{(k)}\right]. \quad (3.6)$$

This form is well suited to a fragment program, as the data accesses are predictable.

It accesses the left, right, top, bottom, and center cells. The same formula can be used for both the $x$ and $y$ velocity components and thus can be performed concurrently in the fragment program.

Free-slip boundaries are used in this simulation. Velocity components inside walls are set to the parallel velocity component in the adjacent Fluid cell.

## 4. Projection

The last step to perform on the grid is projection onto a divergence-free field. The final grid velocity must be divergence-free in order to satisfy the mass conservation equation, Equation 2.2. At this point we have an intermediate velocity $\mathbf{u_n^{**}}$ and are trying to obtain the final grid velocity $\mathbf{u_{n+1}}$. Looking at the last remaining term of the momentum equation, Equation 2.1, yields

$$\mathbf{u_{n+1}} = \mathbf{u_n^{**}} - \frac{\Delta t}{\rho}\nabla p. \tag{3.7}$$

To obtain the final velocity, simply subtract the gradient of the pressure, $p$, from the intermediate velocity. Unfortunately, we do not yet know the values of $p$. To simplify the equation a bit let $\phi = \frac{\Delta t}{\rho}p$. Then because $\frac{\Delta t}{\rho}$ is a constant $\nabla\phi = \frac{\Delta t}{\rho}\nabla p$. Substituting this into Equation 3.7 gives

$$\mathbf{u_{n+1}} = \mathbf{u_n^{**}} - \nabla\phi. \tag{3.8}$$

Taking the divergence of both sides of Equation 3.8 yields

$$\nabla \cdot \mathbf{u_{n+1}} = \nabla \cdot \mathbf{u_n^{**}} - \nabla^2\phi. \tag{3.9}$$

From the mass conservation equation (2.2) we know that the final velocity must be divergence free, and so $\nabla \cdot \mathbf{u_{n+1}} = 0$. Thus Equation 3.9 can be simplified to

$$\nabla \cdot \mathbf{u_n^{**}} = \nabla^2 \phi. \tag{3.10}$$

This equation can then be solved in a manner similar to the diffusion equation. The matrix form of the equation is

$$\frac{1}{\Delta x^2} \begin{bmatrix} & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ \ddots & & \ddots & \ddots & \ddots & & \ddots & & \\ & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \\ & & \ddots & & \ddots & \ddots & \ddots & & \ddots \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \end{bmatrix} \begin{bmatrix} \vdots \\ \phi_{i-1,j} \\ \vdots \\ \phi_{i,j-1} \\ \phi_{i,j} \\ \phi_{i,j+1} \\ \vdots \\ \phi_{i+1,j} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \nabla \cdot \mathbf{u}_{i-1,j} \\ \vdots \\ \nabla \cdot \mathbf{u}_{i,j-1} \\ \nabla \cdot \mathbf{u}_{i,j} \\ \nabla \cdot \mathbf{u}_{i,j+1} \\ \vdots \\ \nabla \cdot \mathbf{u}_{i+1,j} \\ \vdots \end{bmatrix}. \tag{3.11}$$

Using the formula from Equation 3.3 on a row of the matrix results in the following generic equation for one iteration of a single element:

$$\phi_{i,j}^{(k+1)} = \frac{1}{4}\left[\phi_{i-1,j}^{(k)} + \phi_{i,j-1}^{(k)} + \phi_{i,j+1}^{(k)} + \phi_{i+1,j}^{(k)} - \Delta x^2 \nabla \cdot \mathbf{u}_{i,j}\right]. \tag{3.12}$$

The pressure equation is solved for Fluid cells only. The pressures of adjacent Surface and Solid cells will still be needed in the calculation though. The pressure in all surface cells should equal the atmospheric pressure. This can be set once before the pressure calculation begins. Solid boundary conditions must be enforced. Since the velocity component on a Solid/Fluid edge should always be zero, the gradient of pressure across the edge should also be zero. To ensure this, the pressure of a Solid cell adjacent to a Fluid cell is set equal to the pressure of that Fluid cell.

It is important to use enough iterations for the pressure to propagate throughout the field. Otherwise the divergence-free condition will not be met and the volume of the fluid may change. This is most noticeable in the steady state, where the fluid will slowly sink. Finally, subtract the gradient of $\phi$ from $\mathbf{u}_\mathbf{n}^{**}$ to get the divergence-free velocity field. Note that since the value of $p$ is not needed anywhere else in the algorithm, there is no need to convert back from $\phi$.

### 5. Air/Surface Velocities

All grid cells containing fluid, including Surface cells, must be divergence-free. The projection step operates only on Fluid cells. Afterwards the Surface cells may still need to be adjusted. Here this is done by adjusting the values of velocity components on Empty/Surface cell edges. A Surface cell may have from one to four of these edges. In the case where there is only one Empty/Surface edge, there is only one possible value to make the cell divergence-free. If there are two Empty/Surface edges, and they are adjacent, the values from the opposite sides of the cell are used. In other words, the adjustable x-velocity component copies the value from the fixed x-velocity component, and the y-velocity component is treated similarly. The other case where there are two Empty/Surface edges is when they are opposite each other. Let $v_{f1}$ and $v_{f2}$ be the two fixed velocity components. Then assign $(v_{f1} - v_{f2})/2$ to one of the adjustable velocity components and $(v_{f2} - v_{f1})/2$ to the other such that the divergence is zero. If there are three Empty/Surface edges, only one of them will be opposite a fixed velocity component. Set the velocity component on that edge to the fixed velocity component. For the other two components, take their average and set that as the new value for both. The final case is where all four velocity components are on Empty/Surface edges. Then set each component equal to the average of itself and the component opposite it.

## 6. Grid to Particles

The next step in the FLIP algorithm is to update the particle velocities. The final grid velocity is subtracted from the initial grid velocity to get the change in velocity. This change is interpolated to the particles and added to the current velocities stored in the particles.

## 7. Particle Advection

The final step is to move the particles. It is important to move the particles according to the newly calculated grid velocities and not according to the newly calculated particle velocities. When the particles are moved, their newly calculated velocities stay with them. In this way, the velocity is advected. The particles are moved using first-order Euler integration. First the grid velocity is interpolated at the particle position. Then that interpolated velocity is multiplied by the time step and added to the particle's current position to get the new position.

# CHAPTER IV

# IMPLEMENTATION

This chapter covers implementation details that are idependent of the FLIP algorithm. It includes design decisions related to the specific hardware used as well as optimizations. For this work the fluid simulation was implemented in OpenGL 2.0. The graphics card used was an NVIDIA GeForce 7600.

## A. Framebuffer Objects

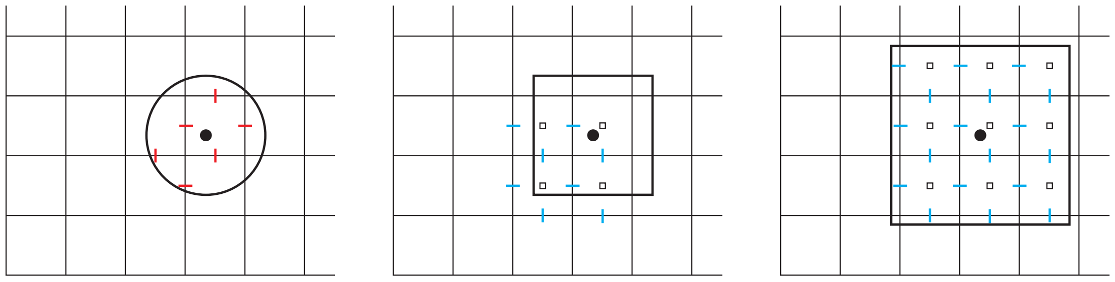
Framebuffer objects are used extensively throughout the simulation. They are primarily used to support render-to-texture. Images can be attached to framebuffer objects. When a fragment program renders its output to one of these attachments, the resulting image can then be used directly as a texture without having to be copied to another place in memory. Multiple textures can be assigned to a single framebuffer object, with certain restrictions. All textures attached to the same framebuffer object must have the same size, bit-depth, and internal format.

## B. Equality Testing in Shaders

In the GeForce 7 series of GPUs, there is no native support for integers. Everything is stored as floating point. Therefore, when testing for equality, one must test if they are equivalent within some small tolerance, $\epsilon$. For example, the test $if(x == 1)$ would have to be written $if(abs(x-1) < \epsilon))$.

## C. Velocity Splatting

Point sprites are used to "splat" the particle velocity onto the grid, as described in the Methodology chapter. However, this is harder to do with staggered velocity than with cell-centered velocity. In this simulation the diameter of a given point splat is 2. In other words, the influence of the particle drops to zero at a distance of 1.0 from the center. But because the velocities are stored at the cell edges, the point sprites must be created with glPointSize set to 3. Otherwise, certain fragments will be missed. A fragment is only processed if its center is covered by the point splat. Figure 8 shows a point centered at $(3.4, 2.4)$. Part of it would cover the y-velocity component located at grid position $(3.5, 3.0)$. But this velocity is stored in fragment $(3, 3)$, which has it's center at $(3.5, 3.5)$. And a point with a size of 2 would only reach up to 3.4 (not 3.5), so the fragment would not get counted.



(a) Velocities influenced by the particle

(b) Velocities covered by a point of size 2

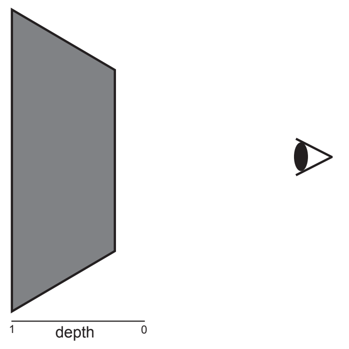
(c) Velocities covered by a point of size 3

Fig. 8. Velocity splatting. (a) shows the velocity components that should be covered by an area of influence with diameter of twice the cell size. Using a point size of 2 misses two velocity components, as shown in (b). All the velocities are covered with a point size of 3, as in (c).

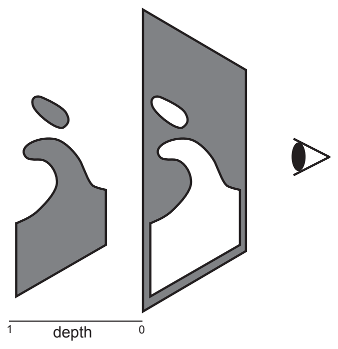
## D. Early-Z

There are times when one does not want to run an expensive fragment program on every fragment in the simulation. For example, the Poisson equation for the pressure only needs to be solved for the Fluid cells. Running the fragment program on the other cells is a waste of computation, compounded by the fact that the equation is solved by many iterations of the fragment program. We can avoid processing unnecessary fragments by using a method called early-z. This is done in two passes. In the first pass, we mark all of the cells that we do not want processed by changing their depth values. Then in the processing pass, the fragments representing non-fluid cells are rejected based on their depth values before they ever get processed. A conceptual view of how early-z works in shown in Figure 9.

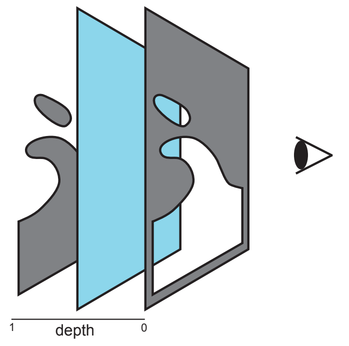
First, writing to the depth buffer is enabled. All values in the depth buffer are cleared to 1. These values are clamped to the range $[0, 1]$ where 1 is the farthest from the camera. The culling pass creates a sort of mask for determining which fragments will be processed later. It is performed by drawing a buffer-sized quad at depth 0 and running a fragment program that discards those fragments that are wanted for later processing. The depth of any fragments that are not discarded gets set to 0, while the depth of those fragments that are discarded remains at 1. After the culling pass, writing to the depth buffer must be disabled and depth testing enabled. Now any number of processing passes can be performed without recomputing the depth buffer. To perform a processing pass, a buffer-sized quad is drawn anywhere between depth 0 and 1. Since the depth test has been enabled, only those fragments with a depth less than that stored in the depth buffer will be processed. Remember that the depth buffer has a value of 1 for fluid fragments. So if the quad is drawn at depth 0.5, then those fragments will be drawn and processed because 0.5 is less than 1.



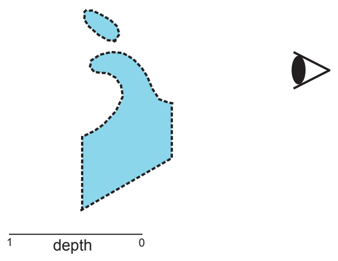
(a) Clear depth buffer to depth 1.



(b) Culling pass.



(c) Draw quad between depth 0 and depth 1.



(d) Processed fragments.

Fig. 9. Early-z.

Implementing early-z can be tricky. There are a few criteria that must be met to ensure that it is not disabled, and these criteria vary from one graphics card to another. First, writing to the depth buffer must be disabled during the processing pass. If the depth value is going to change, the program will not know ahead of time whether it will pass the depth test or not. Also, the direction of the depth test must not change. If it is set to GL_LESS during the culling pass, then it must be set to GL_LESS or GL_ALWAYS during the processing pass. Changing it to GL_GREATER will disable early-z.

# CHAPTER V

# RESULTS AND CONCLUSION

## A. Results

The results of the simulation looked mostly correct. Figure 10 shows a dam of water breaking. This is a traditional setup to test fluid simulations. Figure 11 shows the fluid interacting with an obstacle. There were a few problems with fluid getting stuck on the ceiling, especially in the corners. This is due to the manner in which obstacles are handled. By definition fluid cannot flow into or out of an obstacle. Thus the condition is enforced that the velocity component on a solid boundary must equal zero. While this condition does keep the fluid from sinking through the floor of the simulation, it also introduces an artificial viscosity on the ceiling. The fluid is free to move horizontally but has restricted vertical velocity. When the fluid reaches the corner, it will generally curve and flow down the side wall. If the corner cell becomes isolated, with no other fluid or surface cells touching it, any particles still in that cell stop moving. This quirk is due to the way in which zero divergence is enforced on surface cells.

A small number of tests were done to determine how much of an effect an increase in the number of particles has on the performance of the simulation. The results are shown in Figure 12. The number of frames per second drops rapidly at first but then flattens out. Changing the resolution of the grid does not have as much of an effect as changing the number of particles. The speed of the simulation run on a 25x25 grid was only slightly better than one run on a 45x45 grid. It would, seem, then that in order to increase the detail in the simulation, it would be better to increase the grid resolution. However if the grid resolution is increased, the number of particles must be increased, too. Otherwise the particle density in the cells can get too low. When that happens, some cells temporarily lose all their particles and are counted as Empty, even if they are in what should be a solid block of Fluid cells. Also, as the grid resolution increases, the number of pressure equation iterations must be increased. There must be enough iterations for the pressure to propagate out to all of the cells.

The biggest bottleneck in the simulation seemed to be the particles. The culprit may be the blending operation used while transferring the velocities from the particles to the grid. This operation is known to be slower than normal fragment processing. Another culprit may be the point sprites. Each point sprite generated covers several fragments, many of which are ultimately not needed. Using cell-centered velocities would eliminate these wasted fragments but at the cost of reduced accuracy of the simulation.

## B. Future Work

The most obvious area for an addition to this work is to move the simulation from two dimensions to three dimensions. Many of the fragment programs were set up with this future expansion in mind. Changing the fragment programs to accomodate three dimensions is often as easy as changing from sampling four adjacent values to sampling six. One challenge is how to represent the data in three dimensions. Depending on the hardware being used, the simulation may be able to use a 3D texture, or it may have to use a 2D texture with the various slices of the 3D texture laid out in an array. Also, generating a 3D surface from particles is an additional challenge.

Incorporating moving obstacles into the simulation would make it more dynamic. The current simulation supports arbitrary but static obstacles. Allowing inflow and outflow of fluid would also keep the simulation from becoming static so quickly.

GPU technology is constantly evolving. Already, newer generations of hardware promise native support for integers and scalar textures instead of only four-component textures. This means fewer arithmetic calculations and smaller memory requirements.

## C. Conclusion

The goal of this thesis was to implement a hybrid particle/grid fluid simulation on the GPU. The ultimate goal was to speed up the simulation by exploiting the parallelism of the GPU. In this respect it was modestly successful. There was a speedup of 2 to 4 times that of a comparable simulation run on the CPU. The expectation is that deeper analysis of the algorithm could identify bottlenecks and allow for additional speed enhancement.

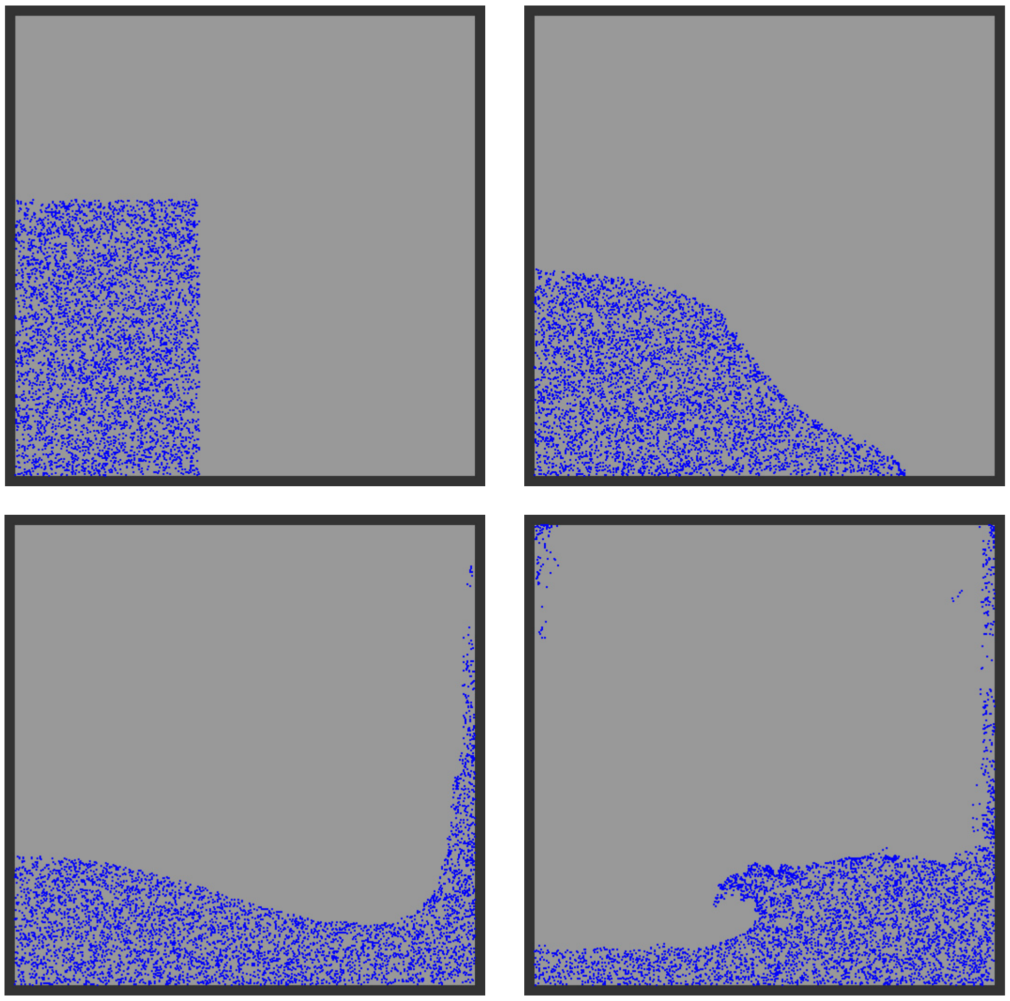


Fig. 10. A water dam breaking. This simulation was run at 45x45 grid resolution with 5000 particles at 20 frames per second.

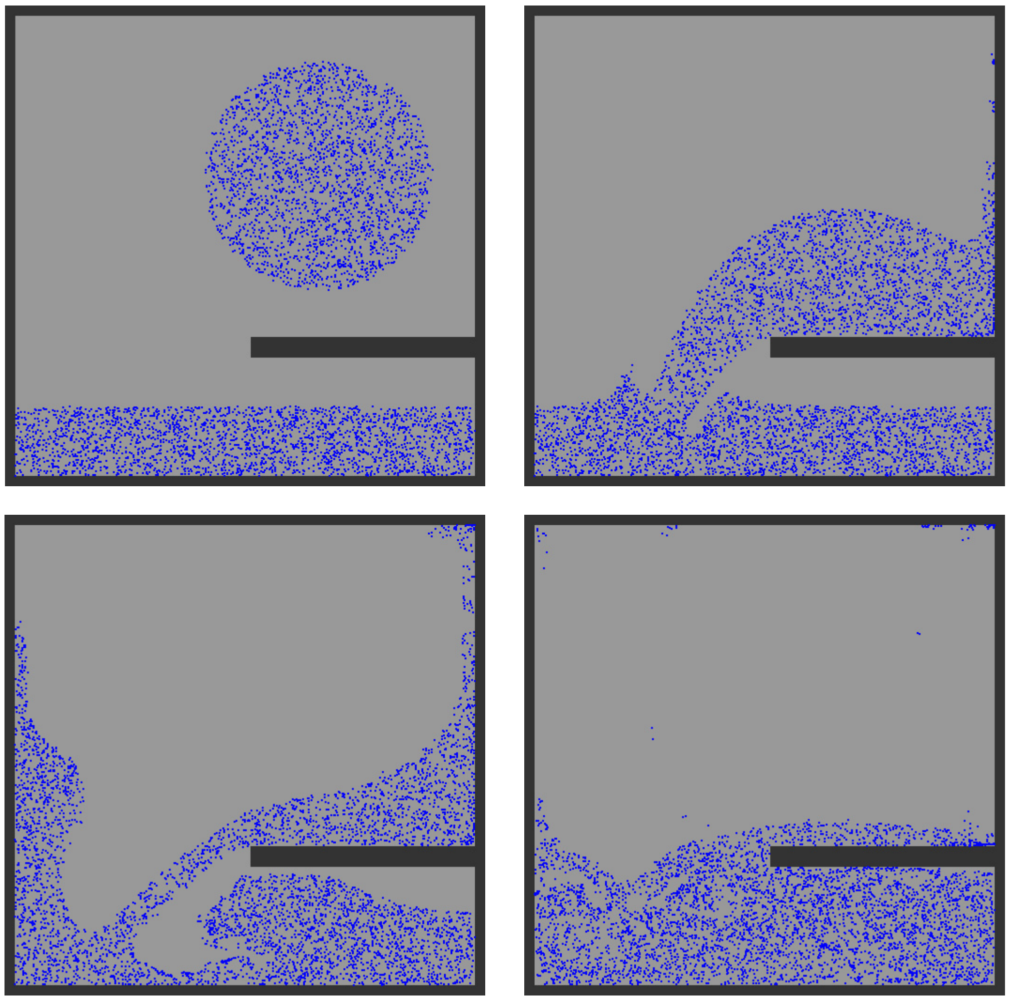


Fig. 11. Fluid interacting with obstacles. This simulation was run at 45x45 grid resolution with 5000 particles at 20 frames per second.

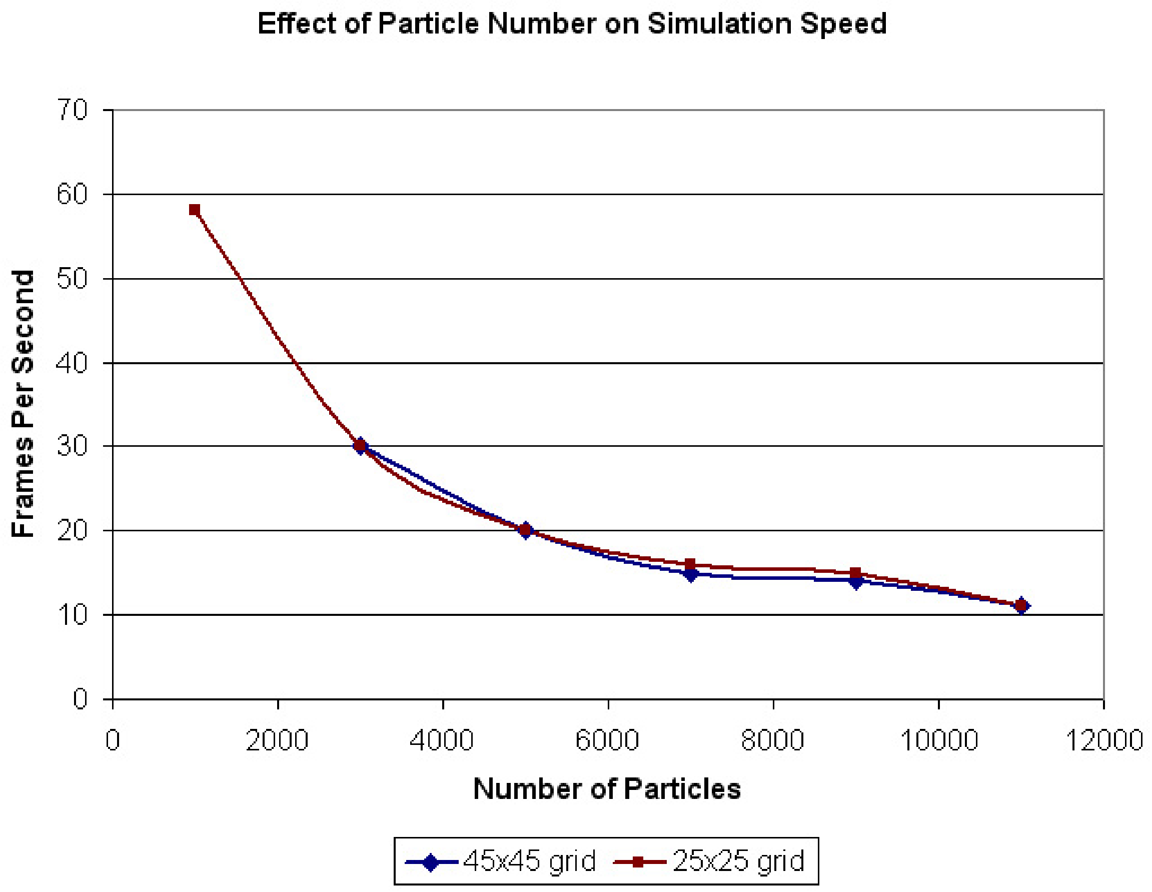


Fig. 12. Performance analysis. As the number of particles increases, the frames per second decreases. Increasing grid resolution decreases performance slightly.

## REFERENCES

[1] F. H. Harlow and J. E. Welch, “Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface,” *The Physics of Fluids*, vol. 8, pp. 2182–2189, December 1965.

[2] M. Kass and G. Miller, “Rapid, stable fluid dynamics for computer graphics,” in *SIGGRAPH ’90: Proc. 17th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 1990, pp. 49–57.

[3] J. X. Chen and N. da Vitoria Lobo, “Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations,” *Graphical Models and Image Processing*, vol. 57, no. 2, pp. 107–116, 1995.

[4] N. Foster and D. Metaxas, “Realistic animation of liquids,” *Graphical Models and Image Processing*, vol. 58, no. 5, pp. 471–483, 1996.

[5] J. Stam, “Stable fluids,” in *SIGGRAPH ’99: Proc. 26th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128.

[6] R. Fedkiw, J. Stam, and H. W. Jensen, “Visual simulation of smoke,” in *SIGGRAPH ’01: Proc. 28th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 2001, pp. 15–22.

[7] N. Foster and R. Fedkiw, “Practical animation of liquids,” in *SIGGRAPH ’01: Proc. 28th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 2001, pp. 23–30.

[8] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations," *J. Comput. Phys.*, vol. 79, no. 1, pp. 12–49, 1988.

[9] W. T. Reeves, "Particle systems-a technique for modeling a class of fuzzy objects," in *SIGGRAPH '83: Proc. 10th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 1983, pp. 359–375.

[10] K. Sims, "Particle animation and rendering using data parallel computation," in *SIGGRAPH '90: Proc. 17th annual conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: ACM, 1990, pp. 405–413.

[11] G. Miller and A. Pearce, "Globular dynamics: A connected particle system for animating viscous fluids," *Computer Graphics*, vol. 13, no. 3, pp. 305–309, 1989.

[12] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *SCA '03: Proc. the 2003 ACM SIGGRAPH/Eurographics symposium on Computer Animation.* Aire-la-Ville, Switzerland: Eurographics Association, 2003, pp. 154–159.

[13] S. Premoze, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker, "Particle-based simulation of fluids," *Computer Graphics Forum*, vol. 22, no. 3, pp. 401–410, September 2003.

[14] S. Koshizuka, H. Tamako, and Y. Oka, "A particle method for incompressible viscous flow with fluid fragmentation," *Computational Fluid Dynamics Journal*, vol. 4, pp. 29–46, 1995.

[15] F. H. Harlow, "Hydrodynamic problems involving large fluid distortions," *J. ACM*, vol. 4, no. 2, pp. 137–142, 1957.

[16] J. U. Brackbill and H. M. Ruppel, “Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions,” *J. Comput. Phys.*, vol. 65, no. 2, pp. 314–343, 1986.

[17] Y. Zhu and R. Bridson, “Animating sand as a fluid,” in *SIGGRAPH ’05: ACM SIGGRAPH 2005 Papers.* New York, NY, USA: ACM, 2005, pp. 965–972.

[18] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra, “Simulation of cloud dynamics on graphics hardware,” in *HWWS ’03: Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* Aire-la-Ville, Switzerland: Eurographics Association, 2003, pp. 92–101.

[19] M. J. Harris, “Fast fluid dynamics simulation on the gpu,” in *GPU Gems*, R. Fernando, Ed. Boston, MA: Addison-Wesley, 2004, ch. 38, pp. 637–665.

[20] K. Crane, I. Llamas, and S. Tariq, “Real-time simulation and rendering of 3d fluids,” in *GPU Gems 3*, H. Nguyen, Ed. Upper Saddle River, NJ: Addison-Wesley, 2007, ch. 30, pp. 633–675.

[21] A. Kolb, L. Latta, and C. Rezk-Salama, “Hardware-based simulation and collision detection for large particle systems,” in *HWWS ’04: Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* New York, NY, USA: ACM, 2004, pp. 123–131.

[22] P. Kipfer, M. Segal, and R. Westermann, “Uberflow: a gpu-based particle engine,” in *HWWS ’04: Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* New York, NY, USA: ACM, 2004, pp. 115–122.

[23] A. Kolb and N. Cuntz, “Dynamic particle coupling for gpu-based fluid simulation,” in *Proc. 18th Symposium on Simulation Technique*, 2005, pp. 722–727.

[24] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: The Johns Hopkins University Press, 1996.

# VITA

|  |  |
|---|---|
| Name: | Rebecca Lynn Flannery |
| Address: | Visualization Sciences<br>Texas A&M University<br>C418 Langford Center<br>3137 TAMU<br>College Station, TX 77843-3137 |
| Email Address: | flannery@viz.tamu.edu |
| Education: | B.S., Computer Science, Texas A&M, 2003<br>M.S., Visualization Sciences, Texas A&M University, 2008 |