BOUNDLESS FLUIDS USING THE LATTICE-BOLTZMANN METHOD

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Kyle Haughey

June 2009

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of one years from the date of submission. I waive reproduction rights after the time span has expired.

____

Signature

____

Date

APPROVAL PAGE

TITLE: Boundless Fluids Using the Lattice-Boltzmann Method

AUTHOR: Kyle Haughey

DATE SUBMITTED: June 2009

Dr. Zoë Wood	
Advisor or Committee Chair	Signature
Dr. Chris Buckalew	
Committee Member	Signature
Dr. David Marshall	
Committee Member	Signature

Abstract

Boundless Fluids Using the Lattice-Boltzmann Method

by

Kyle Haughey

Computer-generated imagery is ubiquitous in today's society, appearing in advertisements, video games, and computer-animated movies among other places. Much of this imagery needs to be as realistic as possible, and animators have turned to techniques such as fluid simulation to create scenes involving substances like smoke, fire, and water. The Lattice-Boltzmann Method (LBM) is one fluid simulation technique that has gained recent popularity due to its relatively simple basic algorithm and the ease with which it can be distributed across multiple processors. Unfortunately, current LBM simulations also suffer from high memory usage and restrict free surface fluids to domains of fixed size. This thesis modifies the LBM to utilize a recursive run-length-encoded (RLE) grid data structure instead of the standard fixed array of grid cells, which reduces the amount of memory required for LBM simulations as well as allowing the domain to grow and shrink as necessary to accomodate a liquid surface. The modified LBM is implemented within the open-source 3D animation package Blender and compared to Blender's current LBM simulator using the metrics of memory usage and time required to complete a given simulation. Results show that, although the RLE-based simulator can take several times longer than the current simulator to complete a given simulation, the memory usage is significantly reduced, making an RLE-based simulation preferable in a few specific circumstances.

Acknowledgements

I would like to express my heartfelt gratitude to the following people who have supported and encouraged me during the creation of this thesis and throughout my academic career:

- To my Creator and King, for creating such a wonderful world to explore and such wonderful people to share it with.
- To my wife Carrie, who has patiently endured many hours of being apart while I wrote code and chapters, and who persistently "encouraged" me to get back to work when I got distracted. Because of you, I was able to keep my promise!
- To my advisor Dr. Wood, for being the best professor ever, and for believing that I could finish this project in only two quarters even when I fell behind schedule.
- To my family and friends, for encouraging my passion for computing.
- To Team TBA, for showing me what teamwork is all about.
- And finally, to the Cal Poly faculty and my fellow students, for filling these last five years with some of the most interesting challenges and joyful experiences of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 2007, the film industry took in $26.72 billion in box office receipts [58], with $9.63 billion coming from the U.S. alone. Examine the credits of almost any recent theatrical release, and you will see a smattering of companies listed who created the computer graphics (CG) effects for the film. CG effects are used in movies to create full or partial scenes that are either too expensive or impossible to create in real life and capture with a real camera. Several recent films such as The Day After Tomorrow [34], Poseidon [80], and Pirates of the Caribbean 3 [80] have used physically-based fluid simulation techniques to create realistic computer-generated scenes involving massive amounts of water.

In addition to the film industry, fluids have also been prevalent in video games for many years; the entertainment software industry generated $7 billion [77] in revenue in 2007. Technology for representing fluids in video games has progressed from simple sprite- and texture-based techniques to height fields and pixel shaders [23], and true fluid simulation is just becoming feasible for use in video games. Although there have not yet been many games featuring physical fluid simulation, several tech demos have been published [72] [62] and at least one upcoming title will be highlighting realistic fluids [10].

These two industries are big business, and have consequently spurred much research into fluid simulation for the purposes of computer graphics for the last

20 years or so.

One particular method for fluid simulation, the Lattice-Boltzmann Method (LBM), has gained popularity due to its relatively simple core algorithm and the ease with which it can enforce physical constraints such as mass conservation [84]. However, due to the way it represents fluid, the LBM suffers from high memory requirements compared to other simulation methods. This thesis applies a run-length encoded (RLE) volume data structure derived from [32] to the LBM in order to reduce the memory required to perform simulations.

One current LBM implementation is contained within the open-source 3D animation application Blender. To take advantage of Blender's advanced rendering capabilities and to provide maximum benefit to the computer graphics community, this thesis implements its modified LBM algorithm within Blender.

In addition to the high memory requirements mentioned above, Blender's LBM implementation also constrains simulations to fixed domains that must be specified by animators before the simulations begin. This can lead to fluid encountering invisible walls (the maximum extent of the simulation domain) during the course of the simulation, which creates unwanted visual artifacts and wastes animators' time (for instance, if an animator has to redo a simulation to remove the visual artifacts). The RLE data structure that this thesis uses for LBM simulation is not inherently confined to a domain of fixed size, and thus the modified LBM simulator also allows fluid to move outside of the original domain.

The results obtained from the RLE LBM simulation show that RLE-based LBM is a mixed bag: the RLE-based simulator takes much longer than the original, array-baesd simulator to complete the given simulations, but uses only a fraction of the memory. These results indicate that most fluid simulations are better off using the original simulator except in a few situations, such as simulations that are so large that the original simulator exhausts the computer's memory and simulations where fluid needs to move outside of the domain.

The primary contributions of this thesis are an implementation of an LBM simulator that uses an RLE grid for cell storage and an analysis of this simulator's strengths and weaknesses.

The following chapter (2) will provide background information on fluid simulation in general (the major algorithms, enhancements, and issues), the LBM in particular, as well as the RLE data structure employed by this thesis (as described by [32]). Subsequent chapters will describe the implementation of the RLE data structure as used in Blender's LBM simulator (3) and experimental results using the modified simulators (4). The final two chapters will explore some opportunities to improve on this work (5) and conclude the paper (6).

Chapter 2

Background

Numerical simulation of fluids has been practical since the late 1960's when computers became powerful enough to handle the vast number of computations required to complete simulations in a reasonable amount of time. Fluid simulation is used in a wide variety of disciplines [66], but here we will examine simulation techniques used primarily in computer graphics.

2.1 User Interaction

In most fluid simulation software, the user interaction pattern is essentially the same.

As input, the user specifies a *domain* in which the simulation will take place, *boundary conditions* that determine what happens to fluid that attempts to flow outside of the domain, physical properties such as fluid viscosities and external forces,and *initial conditions* such as fluid or obstacle regions.

Once the user has specified all of the input parameters, the simulation performs a series of timesteps, outputting various information such as velocity or density fields at each step. Other output information can include isosurface meshes and fluid particle locations for small drops of fluid. Rendering the simulation output is usually considered a separate step because the same simulation data can be rendered in many different ways.

2.2 Simulation Methods

There are three basic methods used for fluid simulation in computer graphics: **Navier-Stokes** (2.4), **Lattice-Boltzmann** (2.5), and **Smooth-Particle Hydrodynamics (SPH)** (2.6). Navier-Stokes and SPH will be briefly discussed below, and the LBM will get a detailed treatment in section 2.5.

2.2.1 Simulation Models

Each fluid simulation method can be classified as an Eulerian or Lagrangian model, where Eulerian models choose fixed points in space and then simulate the fluid quantities (velocity, density) that pass through those fixed points (typically a regular 2D or 3D grid) and Lagrangian models choose particular fluid elements to track and then simulate the elements' trajectories through space (typically implemented as a particle system). In Figure 2.1, the red and blue arrows are velocities at two times at fixed points (Eulerian), and the black line is the trajectory of a particular particle (Lagrangian). Table 2.2.1 shows the model classification for each of the basic simulation methods.

Simulation Method	Model
Navier-Stokes	Eulerian
Lattice-Boltzmann	Eulerian
Smoothed-Particle Hydrodynamics	Lagrangian

Table 2.1: Model Classification for Fluid Simulation Methods

2.2.2 Phase Support

Fluid simulations also differ in the number of different fluids (called called "phases") they can handle at once. Phase support for fluid simulation can be

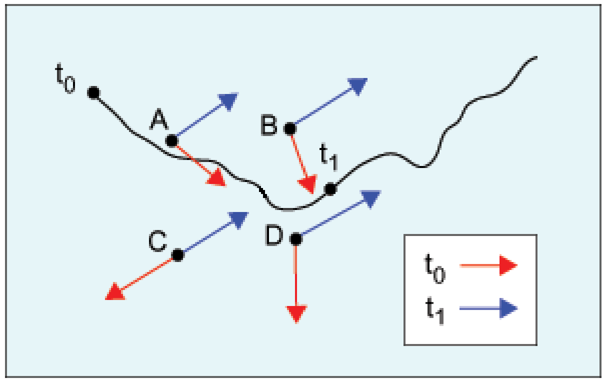


Figure 2.1: Eulerian vs. Lagrangian models (Credit: [74]).

placed into three categories: *single-phase*, *two-phase*, and *multi-phase*. The three categories are described in the following sections.

Single-Phase

Single-phase simulations typically represent the motion of a fluid (gas or liquid) within a domain along with the motion of some density field of particulate matter suspended in the fluid (for example, steam rising from a cup of coffee).

Two-Phase

Two-phase simulations can handle two phases of fluid simultaneously, which usually means a liquid and a gas. An example of a two-phase simulation would be water (liquid) being poured into a glass (which is filled with air before the water gets there). Two-phase simulations are more complex than single-phase simulations because the simulation must track which regions are filled with gas and which regions are filled with liquid. A boundary between a liquid region and a gas region is called an *interface*. In two-phase simulations, the liquid is usually assumed to be much more dense than the gas, and so only the liquid motion is simulated. In these simulations, the liquid/gas interface is called a *free surface* because the liquid surface is free to move anywhere in the domain without being hindered by the gas.

Multi-Phase

Multi-phase simulations are a generalization of the two-phase case. Since there are only two common fluid phases of matter (liquid and gas), multi-phase simulations are most often used in cases where it is necessary to simulate multiple interacting liquids with different densities and viscosities. An example of this would be a container filled with both water and oil (and air). Multi-phase simulations are the most complex class of fluid simulations because the simulation must track not only liquid/gas interfaces for each liquid, but every pair-wise liquid/liquid interface as well. This can be quite tricky, especially when the liquids begin to mix.

2.3 Non-simulated Fluid

Before the rise of true fluid simulation techniques in computer graphics, there were several attempts to animate fluid motion that tried to duplicate observed fluid behavior without respect to the underlying physics. Although these techniques were cutting-edge in their day, they had several limitations that prevented them from being very convincing.

The very first attempts at animating fluids in computer graphics were made in 1986 by Peachey [68] and Fournier and Reeves [21]. These attempts did not use any physically-based model for the water, but used sums of sinusoids and trochoids (a class of functions that generalize cycloids) respectively to model waves breaking on a beach. These methods were limited in that they could only model perturbations in height on a globally flat liquid surface. These papers did, however, start a trend which has continued to this day of using particles to model very small fluid regions.

In 1990, Kass and Miller [35] introduced a technique that they based on a simplified version of the Navier-Stokes equations (the equations describing fluid flow) called the *shallow water equations*. This technique solved the shallow water equations using finite differencing and used a height field to represent the water surface. Chen and Lobo [8] did a similar thing in 1995 when they used finite differencing (a method that uses forward-, back-, or central differences to approximate derivatives) to solve the 2D version of the Navier-Stokes equations and based the liquid height at any given point on the surface on the local pressure computed from the Navier-Stokes equations.

2.4 Navier-Stokes

The first true 3D fluid simulations used for the purposes of computer graphics were performed in 1996 by Foster and Metaxas [20]. To perform their simulation, they used a finite-differencing scheme to obtain a discrete solution to the 3D Navier-Stokes equations on a uniform grid (Eulerian model). They based their work on a classic paper from engineering computational fluid dynamics (CFD) literature [28], but additionally included user control forces and free-surface tracking by means of marker particles (two-phase). This paper was a breakthrough in fluid simulation for computer graphics because it removed the restriction imposed by height fields that fluids could never occupy two heights at the same location on the surface. However, their method still suffered from one of the same problems as engineering CFD simulations (the CFL condition) in that they were severely restricted in the maximum timestep that could be taken without the simulation becoming unstable.

In 1999, Jos Stam proposed a new advection (the process of transporting velocities along themselves) technique that he called the "Semi-Lagrangian Method" [79], which allowed simulations to take much larger timesteps and remain stable. This breakthrough allowed non-trivial simulations to be completed in a reasonable amount of time and made true 3D fluid simulations useful for the first time. However, Stam's new advection method only worked with single-phase simulations, making it useful for smoke simulations but less so for water simulations. Also, the algorithm suffered from *numerical dissipation*, in which the motion of the smoke would be not be as turbulent as it would be in real life.

Fedkiw et al extended Stam's algorithm in 2001 [15] to use a technique called *vorticity confinement* that they took from the CFD literature [81] to re-inject small-scale rotation at locations in the velocity field where the greatest damping occurred.

Foster and Fedkiw again built on Stams algorithm in 2001 [19] when they modified it to handle free surfaces, using a combination of level sets [66] and particles to track the free surface. This new free surface technique has since been extended to handle multi-phase flows [14] [47], but the basic particle/level-set technique still forms the basis for most Navier-Stokes simulation techniques today.

Navier-Stokes simulations have advantages over other simulation methods [84] in that the level-set representation makes it easy to extract a smooth liquid surface and the Semi-Lagrangian advection allows them to take large timesteps. However, Navier-Stokes simulations do require a global pressure-correction step which makes them difficult to parallelize and employ a fixed domain grid [84].

2.5 The Lattice-Boltzmann Method

Another increasingly popular method for simulating fluids is the Lattice-Boltzmann Method (LBM), which was first used for computer graphics in 2003 by Li et al [43] and Wei et al [88]. The LBM is an Eulerian model just like Navier-Stokes simulations, but focuses on fluid interactions at a molecular scale rather than a continuum scale. The molecular interactions handled by the LBM have been shown [22] [51] to closely approximate the Navier-Stokes equations. Instead of directly discretizing the Navier-Stokes equations, the LBM instead maintains a probability distribution at each grid cell of fluid quantities moving in a finite set of velocities. At each timestep, fluid quantities are *streamed* to the neighboring cell in the velocity directions and then *collided* in their new cells, which redistributes the fluid among each of the velocities. The collision step models fluid molecules colliding with each other and changing direction. There are several collision models that can be used, but the most popular one for computer graphics is called BGK [5].

Before Li et al [43] and Wei et al [88] were published, the LBM had been already used for many years in physics applications such as [22], [9], [17], and [52] but was restricted to supercomputers due to its computational complexity [43]. However, Li et al [43] created an LBM implementation that exploited the massively parallel and vector-optimized processors available on commodity graphics processing units (GPUs) that are inexpensive and widely accessible to the public. This allowed LBM simulations to run on commodity hardware and greatly increased its popularity.

The previous techniques, while significant, were all limited to single-phase LBM simulations. It was not until 2004 that LBM simulations were modified to handle two-phase simulations [86] by Thurey and Rude, and not until 2007 that LBM simulations were extended to handle multi-phase flows [92]. Nils Thurey also extended the basic LBM algorithm to use adaptive timestepping to increase simulations' stability [84].

Although the LBM has the advantage over Navier-Stokes simulations of being well-conditioned for parallelization [84], it suffers from high memory requirements as many fluid velocities need to be stored at each grid cell [84]. Additionally, LBM simulations have strict timestep requirements that increase the real-world time it takes to simulate a given interval of animation time [84] and also use a fixed domain grid like Navier-Stokes simulations.

2.6 Smoothed-Particle Hydrodynamics

SPH is a particle-based (Lagrangian) model, and thus does not use a fixed grid of sample points like Navier-Stokes or LBM simulations. SPH can evaluate physical quantities such as velocity or density at any point in space by interpolating the corresponding physical properties of particles within a given spatial distance (called the *smoothing length*) of the point. There are several possible interpolating functions, which are also known in SPH terminology as *kernel functions*.

The first computer graphics fluid simulation algorithm that used SPH was created in 1996 by Desbrun and Gascuel [13]. They based their work off of previous SPH models from astrophysics [24] [56]. However, their algorithm was only geared towards animating highly-deformable bodies rather than fluids in particular, and thus ignored some important fluid properties like force terms from the Navier-Stokes equations or surface tension. Muller et al rectified this in 2003 [59] when they modified Desbrun and Gascuel's algorithm to take these things into account, providing a much more accurate algorithm for simulating fluids. Harada et al also improved the efficiency of SPH calculations in 2007 [27] by performing them in combination of GPU vertex and fragment shaders.

Because SPH does not use a fixed domain grid (although it is possible to use a fixed domain with SPH), fluid particles can move anywhere in space, subject to the precision of the data type used to represent fluid properties. Typically, a float or double data type would be used which is large enough for most simulations. However, the standard method for extracting a surface from the particles, Marching Cubes [45] still requires a grid. This grid can be easily computed, though, based on the bounding box of all particles in the simulation. Recently, Hoetzlein and Hollerer have also developed an alternative surface extraction method to Marching Cubes [31] that is based on wrapping cylinder surfaces around particle streams and shrinking them to fit the stream.

Since SPH uses a particle-based fluid representation, it is not well suited for single-phase flows since it is almost always impractical to completely fill a fixed domain with air particles and attempting to do so with an unrestricted domain will quickly exhaust any system's memory. Thus, SPH is better suited to two-phase [59] and multi-phase [60] [49] flows. In fact, SPH is particularly well-suited to multi-phase flows because simulations can use particles with different color properties. Surface extraction then merely extracts a single surface based on the presence and location of particles, and simply interpolates the surface color between nearby particles.

Although SPH does not require a grid, it does have a few problems [84]. For instance, kernel functions can be quite complex and so smoothing can take a significant amount of time. Also, SPH does have some problems with important physical properties such as mass conservation over the entire simulation domain [84].

2.7 Metrics

Measuring the quality of a given fluid simulation algorithm is a tricky proposition, for a number of reasons. Because fluid simulations are employed in such a wide variety of contexts that each have different requirements and constraints, there is no body of standard metrics used to judge algorithms. Furthermore, the experimental results for each algorithm are generated using different hardware and measurement criteria (for example, some experiments include render time [15] and some don't [20]). Nevertheless, it is possible to extract from the literature a general set of properties that we would like fluid simulation algorithms to possess.

2.7.1 Time and Space Complexity

The time and memory required to perform a fluid simulation over some time interval is very important, because most projects that employ fluid simulations (such as films and games) have budgets and deadlines and run the simulations on real hardware that does not have unlimited computing resources. In fact, video games involve rapid interaction with players and thus have the requirement that any fluid simulation run in real time (30-60 frames per second). Thus, simulation algorithms with lower time and space complexities are highly valued. In addition, merely reducing the constant factor of a simulation algorithm's time or space complexity also has its uses as it can bring a given simulation within the realm of feasibility for a given project.

2.7.2 Accuracy

Another very important metric for measuring the quality of a fluid simulation algorithm is the algorithm's accuracy, which is a measure of how closely the algorithm's results come to duplicating what would happen in the real world given the same physical setup (fluid properties, obstacle properties, gravity, etc.). Accuracy can be measured in a number of ways, and it is important to note that fluid simulation algorithm accuracy for computer graphics is measured very differently than it is in other disciplines such as physics or engineering.

In engineering, an algorithm's accuracy is measured by examining the simulation algorithm and calculating maximum bounds on the error in various quantities such as density or velocity at given points in space. Algorithms are also validated by comparing simulation results to experimental data, such as data provided by wind tunnels [18]. This rigor is necessary for engineering applications because errors have the potential to cause harm to both people and property.

In computer graphics, the accuracy criteria are much less rigorous. Since the end goal of fluid simulations performed in computer graphics is always to produce an image, a simulation's results can be deemed accurate if the final image merely "looks right". That is, if most observers could look at the images produced with a given simulation algorithm and conclude that the fluid behaves correctly, then the algorithm is accurate. Because it is possible to use many different renderers to create an image from simulation data, "looking right" usually refers to the shapes and locations of fluid elements rather than their appearance in the final image.

All of the same error criteria that are used for engineering applications can also be applied to computer graphics applications; indeed, all of the basic algorithms used for fluid simulation in computer graphics today originated from other engineering or science disciplines. However, these error metrics introduce a higher time and space complexity into the algorithms and are unnecessary for a simulation to look right most of the time. As processors get faster and faster and as computers have increasing amounts of memory, we do see more rigorous error metrics being introduced in computer graphics applications.

2.7.3 Time/Space vs. Accuracy Tradeoff

The simultaneous goals of increasing a simulation's accuracy and decreasing the time and memory it requires are in constant tension with each other. One of the easiest ways to increase a simulation's accuracy is to run it using a very high resolution (many samples per spatial unit). However, this can greatly increase the time and memory required to complete the simulation. Conversely, it is possible to reduce the time and space required for a simulation by running the simulation at a lower resolution, but accuracy will suffer. The goal for users then is to choose the correct resolution such that the time and space required will be minimized and the results will still "look right".

2.7.4 Capabilities

Not all fluid simulation algorithms are created equal. The quality of a fluid simulation algorithm is also measured by the range of situations it can handle. For example, some simulations [79] simulate only one type of fluid (such as air) while others can handle two or more fluids. In fact, your choice of algorithm may be dictated by your particular circumstances. Any two algorithms may also support all of the same capabilities while varying in how well they support each capability.

2.8 Algorithmic Enhancements

All of the basic fluid simulation methods described in the previous sections can take copious amounts of time and memory for high-resolution simulations (whether that means very densely packed grid cells or the use of many thousands of particles). Because of this, modest increases in resolution can have a dramatic impact on the time required to perform a simulation, as well as the space required to store all of the grid cells or fluid particles. Conversely, a decrease in time or space complexity can have a profound effect on the maximum resolution that can feasibly used in a simulation for any given project. Thus, much effort has been expended in recent years attempting to reduce both the time and space complexity, even if only by a constant factor, of all of the different fluid simulation methods. The following sections will describe several techniques that various researchers have tried in an attempt to make fluid simulations smaller and faster.

2.8.1 Parallelization

One of the first techniques used when attempting to speed up any algorithm is to try to make it run in parallel on multiple processors, and fluid simulation is no exception. Several papers have been published containing methods for parallelization of all of the basic simulation methods ([30] and [2] for Navier-Stokes simulations, [61], [89], and [41] for LBM simulations, and [83] and [70] for SPH simulations); however, these algorithms are all designed for massively-parallel supercomputers. In computer science, most of the research into fluid simulation parallelization has focused on the GPU because they are both massively parallel (pixels in an image produced by a GPU are usually processed independently) and are optimized for matrix and vector math. Programmable GPU shaders have only been around for the last 7 years or so [65], and so all of the research is fairly recent. Navier-Stokes GPU implementations came first [29], [44], but implementations are also available for the LBM ([42], [57]) and SPH ([39]). The primary challenge in GPU-based simulations is framing the simulation problem in terms of graphics-related concepts such as pixels and texels.

2.8.2 Multigrid and Spatial Partitioning

Multigrid methods are a way to decrease the amount of time required to perform a given fluid simulation at the cost of potentially increased memory usage. The basic idea is to embed a *coarse* grid of lower resolution within a high-resolution *fine* grid and perform simulation on the coarse grid away from "interesting" regions of the fine grid. The "interesting" regions of the fine grid are usually defined to be regions of non-negligible density or regions near the fluid surface [84]. Spatial partitioning methods are similar to multigrid methods in that they both contain fine grids embedded within coarser grids. The difference is that, in spatial partitioning methods, fine grid cells are associated with only one coarse grid cell, and the fine grid cell's volume is completely contained within the coarse cell; this is not generally the case with multigrid methods. Handling fluids at multiple resolutions turns out to be quite challenging because physical properties of the fluids being simulated change at different resolutions, for example the fluid's effective viscosity [84].

Robert Bridson created a subgrid method for Navier-Stokes simulations in 2003 called Sparse Block Grids [6], which was a two-level multigrid method in which fine cells were used near a free surface and coarse cells elsewhere. Losasso et al [46] in 2004 created a Navier-Stokes simulation that used an octree data structure. Zhao et al in 2007 [91] created a subgrid method for the LBM that allocated fine grids in interesting regions and interpolated vector field values between the coarse and fine grids. In 2008, Thurey and Rude [**?**] devised an LBM method that started with a fine grid and created coarsened grids occupying the same volume. Spatial partitioning algorithms are also useful for SPH simulations as well in speeding up the process of finding "close" neighboring particles, such as in [16].

2.8.3 Flexible Domains

In addition to the time and space complexity problem, Navier-Stokes and LBM simulations are restricted to a fixed domain grid that is specified by the user before the simulation is performed. Fluid attempting to exit the domain will either disappear completely or encounter an invisible wall depending on the boundary conditions. This can cause excruciating frustration for users who have to re-run simulations after discovering this. To alleviate this problem, researchers have investigated ways to make simulation domains more flexible. As SPH simulations are particle-based, they don't inherently suffer from this issue. Several solutions have been proposed for Navier-Stokes simulations, including: Shah et al in 2004 [75], which compensated for a mobile domain by adjusting the domain's velocities (Galilean Invariance), Rasmussen et al in 2004 [71], which uses a small simulation domain that can move around in a larger domain, and Houston et al in 2006 [32] that uses a run-length encoded (RLE) volume instead of a 3D array to store grid nodes. These solutions worked well for Navier-Stokes simulations, but they have never been applied to the LBM.

2.9 Related Areas

There are several research areas that are closely related to fluid simulation, but are not considered a part of fluid simulation proper.

One area is coupling fluid simulations with other simulations such as thin-shell simulations ([26]), rigid- and soft-body simulations ([73]), other types of fluid simulations ([48], which couples a Navier-Stokes simulation with an SPH simulation), and even coupling 2D/3D fluid simulations of the same type ([33], [84]). SPH simulation coupling with thin-shell, rigid-body, or soft-body simulations are also generally straightforward to implement because there is extensive research on particle/object collision detection and response ([87], [53], [40], [38]).

Another area is simulation of viscoelastic fluids, which exhibit properties of both liquids and solids (such as lava or pudding). Some viscoelastic simulations have approached the problem from a fluid simulation angle, such as [25] and [11]. Viscoelastic and even solid materials can be treated as fluids that have a very high viscosity [67]. Other simulations have approached the problem from a solid mechanics angle by using spring-based meshes [82] or the Finite Element Method (FEM) [90].

The last related area is animation control, which covers systems that animators can use to make fluid behave how they want. Fluid control systems accomplish this task in a number of ways. Many control systems ([50], [55], [69], [76], [85]) allow users to specify a target shape or particles that attract fluid. Alternatively, there are systems such as [3] that do not attract fluid to a certain shape but instead create forces to make fluid follow user-defined flow trajectories.

2.10 Open Areas

Despite the extensive research that has been performed on simulating fluids for computer graphics, there are still many avenues that remain to be explored.

One such avenue is multi-resolution simulation editing. Although multigrid methods have had some success at handling multiple resolutions within the same simulation, no one has yet devised a system that will cause high-resolution simulation behavior to duplicate low-resolution simulations; the behavior always changes between different resolutions. Kim et al [37] have made some progress on this front by allowing users to synthesize detailed velocity perturbations onto low-resolution simulations, but there is still much work to be done in this area.

GPU-based simulations have also seen quite a bit of success, but new APIs such as CUDA [63], OpenCL [36], and DirectX Compute Shaders [54] are emerging that remove the need to re-frame simulation problems in terms of shaders and textures and promise to make GPU-based simulations much easier to implement. As yet, there have been very few fluid simulations (see [64] as an example) that use these new general-purpose GLU (GPGPU) APIs, but there is ample opportunity to exploit them.

Chapter 3

Algorithms

Because we would like to have a more memory-efficient fluid simulator that does not confine fluid to a fixed domain, we propose modifying the LBM simulator from Blender to use an RLE grid for cell storage.

The following sections describe the algorithms used for RLE-based LBM fluid simulations. Section 3.1 covers in more detail the LBM, which was already implemented within Blender. Section 3.2 covers RLE level sets, which were implemented using [32] as a reference, and Section 3.3 covers the modifications that were made to both the LBM and RLE algorithms in order to integrate them.

3.1 The Lattice-Boltzmann Method in Detail

This section provides a more detailed look at the Lattice-Boltzmann Method, although still at a high level. This description is taken from, and full details can be found in, [84].

As mentioned in the previous section, the LBM works by storing the amount of fluid moving in each of a finite number of velocities at each grid cell. Each discrete velocity is called a distribution function, or DF for short. Each velocity vector is directed at a neighboring grid cell, except for a special zero vector that

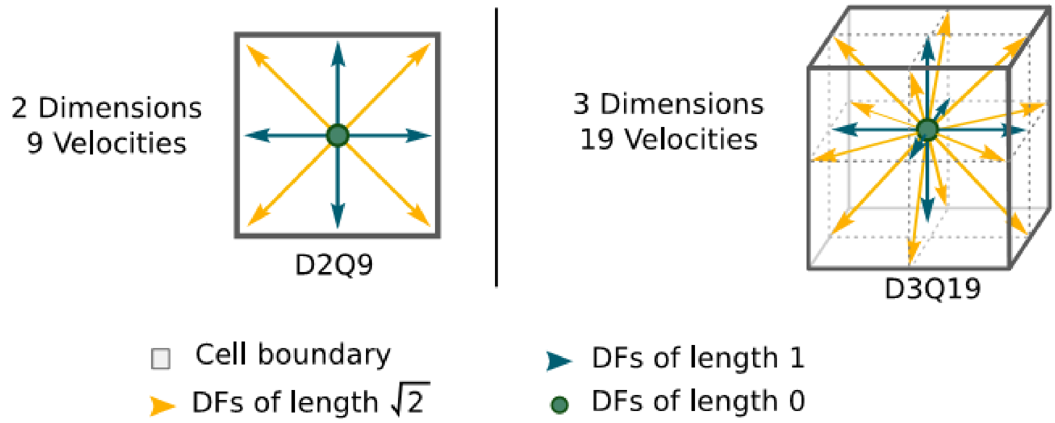


Figure 3.1: Distribution functions (DFs) for the D2Q9 and D3Q19 LBM models (Credit: [84]).

represents non-moving fluid. In memory, the fluid distribution is represented as an array of floating-point values where each element in the array corresponds to a DF and the float value represents the amount of fluid moving in each direction. LBM simulations can be identified by the nomenclature D¡n¿Q¡m¿, where ¡n¿ represents the number of dimensions in the simulation and ¡m¿ represents the number of DFs. Typical values for LBM simulations are D2Q9 for 2D simulations and D3Q19 for 3D simulations. Because a grid cell in three dimensions has 26 neighbors, you would expect 3D simulations to be labeled D3Q27 (26 neighbors plus the zero vector); however, D3Q19 is able to maintain stability while speeding up simulations and requiring less memory [84]. In D3Q19, the DFs point to neighbors in the directions of the cell's faces and edges (assuming a cubic cell), but not the neighbors in the direction of the cell's vertices as shown in Figure 3.1, which also contains an example of the D2Q9 model.

3.1.1 Basic Algorithm

Each LBM timestep has two parts: stream and collide, which can be seen in Figure 3.2. The stream step advects fluid amounts for each velocity to their respective neighboring cells, and once all of the fluid has been advected the collide

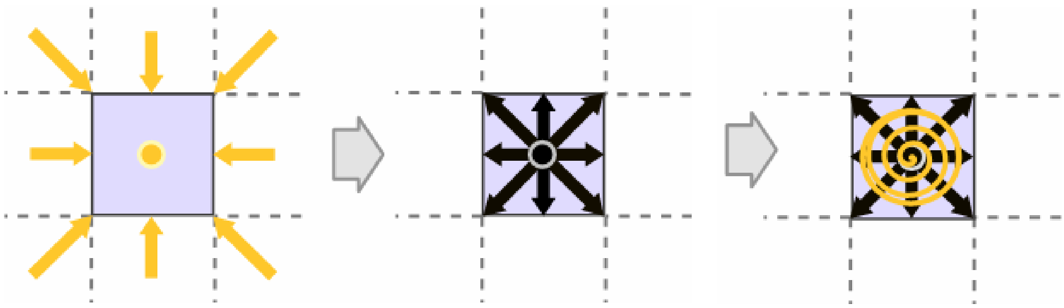


Figure 3.2: LBM stream and collide steps (Credit: [84]).

step redistributes the new velocities at each cell to simulate collisions among the incoming fluid.

The collision model used by [84] is the BGK model described in [4], which modified the fluid distribution by interpolating fluid values with an equilibrium value for each velocity. The equilibrium value for each velocity is the value for which, if all velocities in all cells had their equilibrium values, no fluid values would change.

In practice, two grids are used: one grid contains the cell values at time t and into the other grid are written the values for time t + dt. Also, typically implementations will iterate over the t + dt grid and retrieve incoming fluid from neighboring cells rather than iterating over the t grid and pushing fluid out to neighboring cells. This allows the stream and collide steps to be performed at the same time, since the collide step requires all of the new velocities to be present and the push method would require the collide step to occur in a second pass, taking more time.

The density ρ at each grid cell can be calculated by adding up the fluid amount for each velocity, and the net velocity $\mathbf{u}$ of each grid cell can be calculated by adding up the products of the fluid amounts and the velocity vectors.

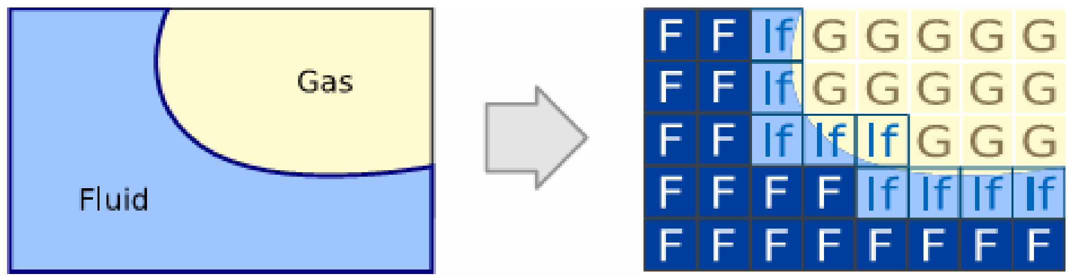


Figure 3.3: Free surface cell flags (Credit: [84]).

3.1.2 Handling Free Surfaces

To handle free-surfaces, [84] flags each cell as one of four types: empty, interface, filled, or obstacle. Empty cells have no fluid in them ($\rho = 0$), filled cells are filled with fluid ($\rho = 1$), and interface cells are partially filled with fluid ($0 <= \rho <= 1$). Figure 3.3 illustrates a free surface and the corresponding cell flags. Obstacle cells are treated specially by the simulation as described in the next paragraph. Also in [84], a layer of obstacle cells is automatically initialized at the domain's boundary to prevent fluid from moving outside the domain.

During the stream and collide steps, the simulation tracks cells that get emptied or filled, and after each time step their flags are re-initialized to ensure that the following properties are maintained:

- Empty cells must have only empty, interface, and obstacle neighbors.
- Fluid cells must have only fluid, interface, and obstacle neighbors.
- Interface cells must have at least one empty neighbor and at least one fluid neighbor.

When iterating over the t + dt grid and the neighboring cell in a particular direction is an obstacle cell, the fluid moving in that direction is reflected according to the obstacle's slip value. For no-slip obstacles, the fluid is reflected to the opposite velocity vector in the same cell. For free-slip obstacles, the fluid is reflected about a plane perpendicular to the obstacle cell's normal and then

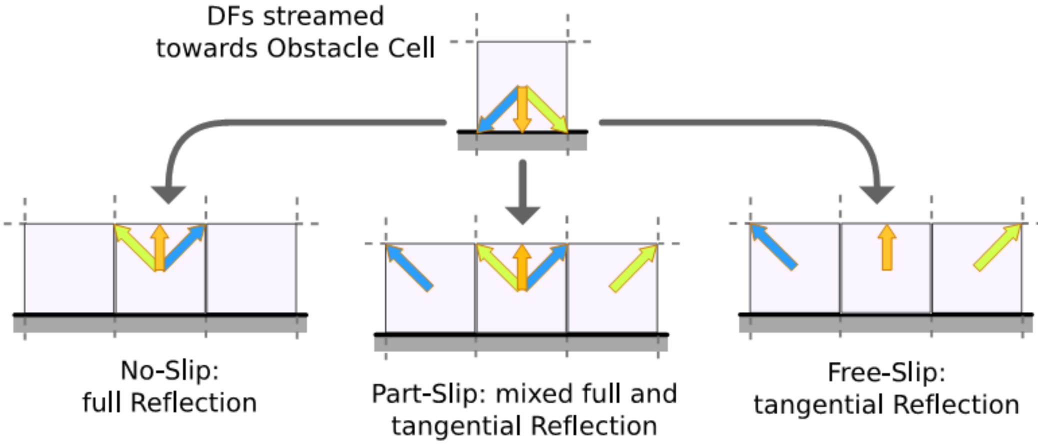


Figure 3.4: Obstacle boundary conditions (Credit: [84]).

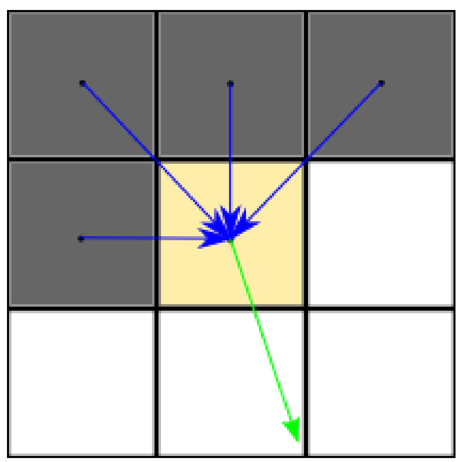


Figure 3.5: Obstacle normal calculation.

advected to an adjacent fluid or interface cell. Each type of reflection can be seen in Figure 3.4.

Obstacle normals can be calculated by summing the vectors from neighboring cells which are also obstacle cells to the current cell as shown in Figure 3.5. In the figure, the cell highlighted in yellow is the current cell, the gray cells are neighboring obstacle cells, the blue arrows are the vectors from the obstacle neighbors to the current cell, and the green arrow is the resultant normal vector.

To prepare the free surface for rendering, [84] uses Marching Cubes [45] to iterate over the domain using the cell densities as isosurface values. The fluid surface is defined to be the locus where the density $\rho = 0.5$.

3.1.3 Stability Concerns

Because fluid moves at maximum one cell width each timestep, there is a maximum fluid velocity that the LBM grid can accommodate. When velocities attempt to grow beyond this threshold, the simulation becomes unstable. To combat this, [84] uses a couple techniques.

First, when velocities become too large, [84] adapts the physical time that a single timestep represents. This involves rescaling all of the equilibrium velocities as well as the physical constants specified by the use such as gravitational force and the fluid viscosity.

In addition, [84] uses a Smagorinksy subgrid model ([78]) to handle high-turbulence flows. This involves calculating the stress tensor at each fluid cell and using it to modify the equilibrium velocities.

3.2 Hierarchical Run-Length Encoded (HRLE) Volumes

Run-length encoding is a simple compression technique that has been used on image data [1] and volume data [12] for quite a while now. Run-length encoding stores sequences of adjacent identical data elements as a single element along with a count of how many elements are in the sequence.

This thesis uses a run-length encoded data structure called a "hierarchical run-length encoded (HRLE) level set", which was created by Houston et al in [32] (full details can be found here). The HRLE grid is hierarchical in the sense of dimension, where information for each dimension is kept separate from the other dimensions and higher dimensions refer to lower ones.

3.2.1 RLE Level Sets

In Houston et al, the RLE level set was used to store level sets, which are an implicit surface representation used in many different applications in computer graphics [66]. Level sets are defined as isocontours of a scalar function in 2D or 3D space, usually the isocontour where $f(\mathbf{x}) = 0$ (the choice of the 0-isocontour will be assumed from here forward). In discrete domains, scalar values are stored at nodes on a regular grid, where the stored value is usually a signed distance to the isocontour. 2D or 3D surface meshes can be generated from level set information by running the Marching Cubes algorithm [45] on it. Houston et al divided space into three types of regions: positive, negative, and defined. Positive regions were regions where $f(\mathbf{x}) > \beta$, where β is a threshold value chosen by the user, and negative regions likewise were regions where $f(\mathbf{x}) < -\beta$. Defined regions were regions where $-\beta <= f(\mathbf{x}) <= \beta$, and the RLE level set allocates memory only for grid nodes within the defined regions.

The RLE level set consists of an array of scalar values, and one or more *RLE blocks*, the collection of which is called an *RLE grid*. An RLE grid contains one RLE block per dimension. If we assign each dimension a number (for example, z=2, y=1, and x=0), we can talk about the block in terms of their position, so the blocks proceed from high to low. The block corresponding to the highest dimension is called the *top block*, and blocks corresponding to lower dimensions are called *lower blocks*.

Each block consists of the following items:

1. Low and high extents for the dimension. An explicit bounding box can be constructed for an RLE grid by using the dimension extents from each RLE block.

2. An array of *runs*, where a run is a *run code* (positive, negative, or defined) combined with an offset. The offset is only significant for defined runs.

3. An array of *run breaks* (of the same length as the run array) that gives the end coordinates of each run. The first run in each *segment* (see next item)

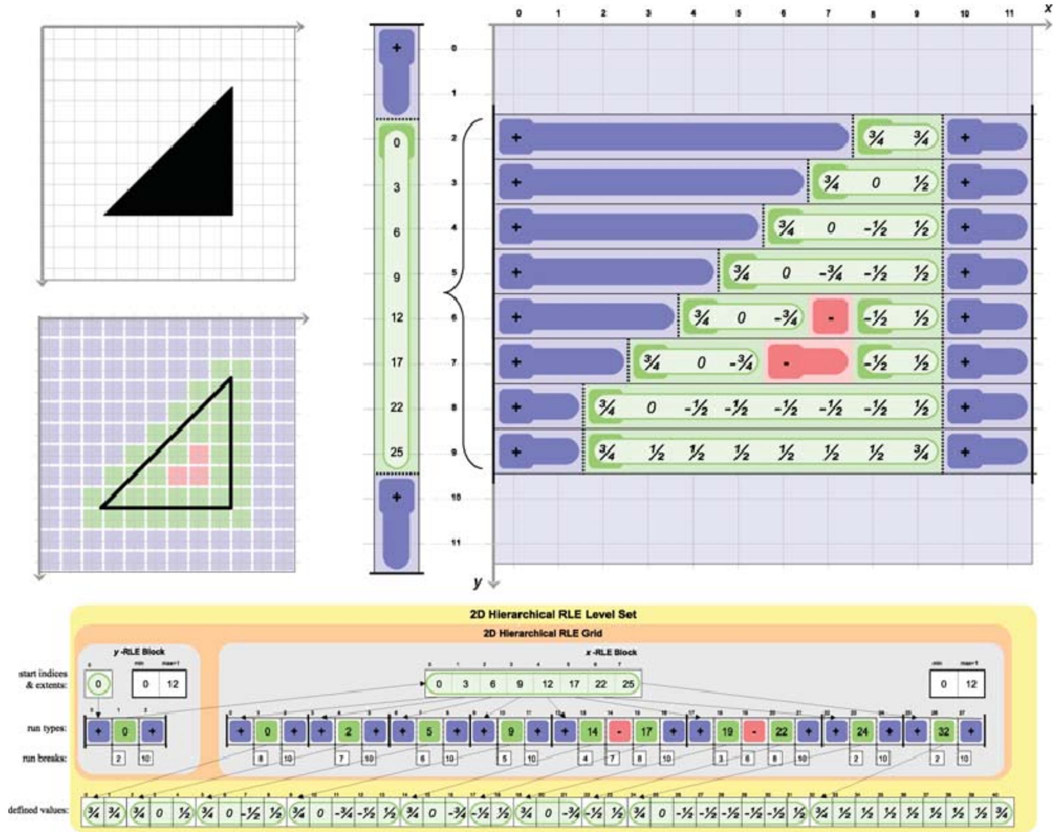


Figure 3.6: 2-dimensional RLE level set (Credit: [32]).

is assumed to have a starting coordinate equal to the low extent, and the last run break in each segment must equal the high extent.

4. An array of *segment start indices* that provides offsets into the run and run break arrays of the first run in each *segment*. A segment is a contiguous (in space) sequence of runs.

Because an RLE block stores all of the runs for its dimension, it will usually have to store multiple segments that correspond to different coordinates in its dimension. In fact, only the top block will have a single segment. An example of a 2D RLE level set can be seen in Figure 3.6.

The meaning of the offsets in defined runs depends on which block is being talked about. For the lowest block, the offsets index into the RLE level set's value array. For higher blocks, the offsets index into the next-lower block's segment start index array.

3.2.2 HRLE Advantages/Disadvantages

The advantage of using an RLE level set instead of a fully-defined array is that you don't have to allocate memory for regions that are far away from the "interesting" region near the isocontour of choice. This makes the storage requirement roughly proportional to the surface area (roughly $O(n^2)$) of the isocontour rather than $O(n^3)$, where n is the number of grid cells along one edge of the domain.

Furthermore, with the trim/dilate algorithm described below, the level set is not confined to a fixed domain, but can adjust itself to follow the surface as necessary.

The disadvantage of RLE level sets compared to arrays is that random access to cell data is much slower, because locating cells in memory requires navigating a data structure instead of simply performing some pointer arithmetic (the random access algorithm is described below in Section 3.2.3.

3.2.3 HRLE Algorithms

There are several algorithms that operate on RLE level sets, but only two are covered here. For details on the other algorithms, consult [32].

Random Access

To access a particular defined cell, it is necessary to start at the top block and navigate downward through the bottom block to the RLE level set's value array. Given a coordinate corresponding to a given block's dimension and starting and ending offsets into the run array, we can perform a binary search to locate the run corresponding to the given coordinate. If the run is not defined (it is positive or negative), then we return NULL. If the run is defined, we can compute the

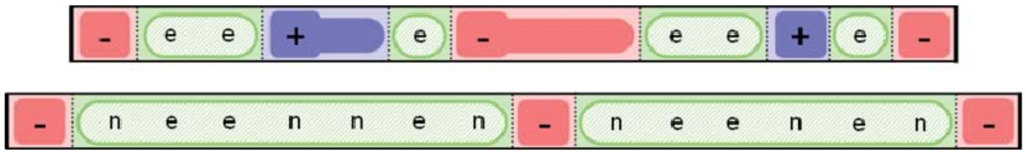


Figure 3.7: An example of 1-dimensional dilation (Credit: [32]).

run's offset and the given coordinate to locate the value corresponding to the coordinate (for the lowest block) or recursively search the next-lower block (for all other blocks). For the top block, the starting and ending offsets are merely 0 and the number of runs in the top block respectively.

Trim/Dilate

The trim/dilate algorithm is used to ensure that the defined region stays centered around the 0-isocontour. If the surface moves (by changing the defined scalar values), then the defined region will also need to move periodically to keep up. To accomplish this, a two-step algorithm was devised in [32] to re-center it. Trim/dilate operates on two RLE level sets, using one as a temporary level set. The trim step uses the original level set as a source and writes to the temporary level set, and the dilate step reads from the temporary level set and writes back to the original level set.

The first step is the trim step, wherein existing defined cells are checked to ensure that they are still "close" to the 0-isocontour . This part of the algorithm iterates over each defined cell in the source set (see [32] for the iteration algorithm) and adds cells whose value is in the range $[-\beta, \beta]$ to the target set.

After the trim step, the dilate step ensures that there is a wide enough band around the surface. Dilation consists of two substeps. First, each segment in each block in the trimmed set is dilated independently as a 1D segment. This is done by creating a $[start - 1, end + 1)$ pair for each defined run in the segment and then taking the union of these pairs, a process which will be described in the next paragraph. Then, starting at the top and handling dilated segments recursively down to the bottom block, we iterate over each coordinate included in the dilated 1D segments. For each coordinate, we create runs in the same-level block of the target set by taking the union of the dilated 1D segment for the current coordinate from the first substep with the dilated 1D segments for each of the current coordinate's immediate neighbors in all higher dimensions. If we are currently at the lowest block, then we must initialize the values in the newly-created defined runs. If a value in the newly-defined runs has the same coordinates as a defined value in the old set, then we can simply copy the value over. If there is no corresponding value from the old set, then we must calculate the signed distance from the grid cell to the suface and initialize the new value with that distance.

To compute unions of $[start, end)$ pairs that may overlap, we use the following algorithm. Create a sorted map from coordinates to `int`s. Iterate over each pair, increment the map count for start, and decrement the map count for end, creating any necessary map entries. Then, create an empty list of pairs that will hold the union runs, initialize a total count variable to 0, and iterate over the map keys. Because the map is sorted, the iteration will proceed from low coordinates to high coordinates. At each coordinate, add the coordinate's map value to the total count. If the total count transitions from 0 to a positive number, then store the coordinate in a temporary variable. If the total count transitions from a positive numebr to 0, push a pair onto the union run list consisting of the previously-saved temporary coordinate and the current coordinate. After iterating through all of the map coordinates, the list will contain pairs that are the union of the input pairs.

3.3 LBM Modifications

Using the RLE set data structure with LBM simulations required several modifications to both the RLE data structure and algorithms as well as the LBM simulation algorithm. You can see an example of an RLE-based grid being used in a 2D LBM simulation in Figure 3.8. In the figure, gray cells are boundary cells,

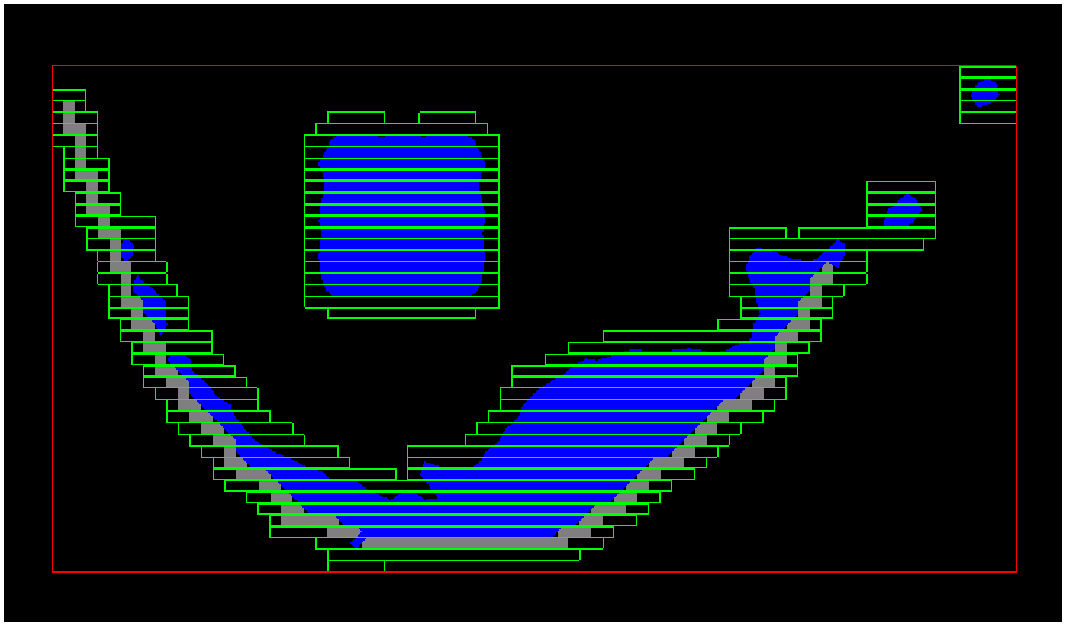


Figure 3.8: An example of a 2D-LBM simulation with no fixed domain.

blue cells are interface or fluid cells, the red box indicates the maximal extent of the domain (which is flexible), and the green boxes represent defined runs within the domain. Regions inside the red box but outside a green box are positive regions that don't have any memory allocated for cells. The changes required to accomplish the integrated LBM simulation are described in the following sections.

3.3.1 Data Structure Modifications

To accommodate a fluid simulation, several modifications were made to the RLE set data structure to accommodate general-purpose data, to improve trim/dilate performance, and to handle arbitrary injection of defined grid cells.

General-Purpose RLE Sets

The RLE data structure used in the modified LBM simulator was changed from being a level set representation that stored only signed distances as `floats` to being a templatized general-purpose data structure that could store an instance of an arbitrary C/C++ datatype at each grid cell.

The modified LBM simulation does not store scalar values, but instead stores `FluidCell` C structs that each contain the flag and DFs for one cell (the previous LBM simulator from [84] stored grid cells in two separate arrays: one for cell flags and one for DFs).

The modified RLE set also only has two types of regions, positive and defined. Positive regions are empty regions that are more than one grid cell away from the fluid surface, and defined regions are regions that are fluid and interface cells along with with empty cells within one grid cell of the fluid surface. The reason for the lack of negative regions is that we cannot ignore fluid cells that are far away from the fluid surface, since doing so would disregard the net fluid velocities contained in these cells. which will impact future timesteps. The reason for including the one-empty-cell layer surrounding fluid and interface regions is that, during the course of an LBM timestep, these empty cells could potentially be re-flagged as interface cells so they must be defined.

It is important to note that modifying the LBM to use the RLE data structure instead of fully-defined arrays changes the memory storage required for a simulation from being proportional to the volume of the domain to being proportional to the current volume of fluid.

Memory Allocation

To prevent the repeated allocation and deallocation of memory during the trim and dilate steps, the modified RLE set also provides a `push()` interface to the trim/dilate code to use when adding or copying values, segment start indices, runs, and run breaks. The RLE set maintains two counters for each of these arrays: a size and a capacity. The size tracks the number of valid entries in each array and the capacity tracks the number of allocated entries for each array.

When an RLE set is first allocated, the capacities are initialized to zero, and the size counters are all initialized to zero before a trim or dilate operation is performed. As array elements are added, the RLE set checks to see if the new size will be greater than the capacity and, if so, reallocates a larger array and copies the old values to the new array before proceeding as normal. The enlargement factor used when an array size reaches its capacity is $(3 * oldCapacity)/2 + 1$.

The astute reader may notice that this behavior is nearly identical to the C++ `vector` class and wonder why `vector` was not employed. It is worthwhile to admit that this was attempted; however, the overhead of `vector`'s `operator[]` caused a significant decrease in simulation performance.

This performance enhancement means that an RLE set at any given timestep may have more memory allocated than is strictly required. This tradeoff however seems worth it in light of the performance increase. This also makes the memory storage requirement proportional to the *maximum* volume of fluid over the simulation's time simulated so far rather than the volume of fluid at each timestep.

Arbitrary Fluid Injection

Any useful fluid simulation needs to at some point inject fluid (at arbitrary locations) into the domain, and the simulation's underlying data structure needs to handle this. The original RLE data structure was initialized from an implicit surface and its trim/dilate steps only handled adding or removing defined grid cells that were relatively close to the fluid surface. To handle injection of defined cells at arbitrary locations, the modified RLE data structure provides clients with an `ensureCell()` method that will define a cell at any client-provided coordinates.

It would be possible to implement `ensureCell()` by performing the trim and dilate operations each time `ensureCell` was called on a previously-undefined cell. However, due to the number of `ensureCell()` calls encountered in practical situations (a cubical drop of fluid occupying a 10x10x10 region could potentially invoke 1000 `ensureCell()` calls), the modified RLE set instead stores previously-undefined cells in a map keyed by the cell's (x, y, z) coordinates.

When accessing a cell, the RLE set first checks to see if the requested coordinates are defined in the map. If they are, then a reference to the map cell is returned. If not, then a reference to the RLE set's cell is returned (assuming of course that the cell at the requested coordinates is defined). To ensure that injected cells stored in a map don't remain there forever, the dilate operation checks both the source set and the source set's injection map when adding values to the target set.

During the dilate operation, injection map values are merely copied to the target set's injection map and before any trim or dilate operation the target set's injection map is cleared.

3.3.2 Algorithm Modifications

Although the basic algorithms for RLE set manipulation and LBM simulation were taken from [32] and [84] respectively, some tweaks were required to get them to interoperate.

LBM Modifications

The original LBM simulator was heavily integrated with its array-based grid storage, and thus had to be heavily modified to accommodate multiple grid storage data structures. The original simulator used pointer arithmetic in several places to speed up cell access, but since pointer arithmetic does not work to locate cells in RLE sets, all cell access was abstracted out into an interface that had one implementation for array-based grids and another for RLE-based grids.

The method for iterating over grid cells also had to be changed. The original simulator simply iterated x, y, and z coordinates over the x, y, and z ranges of the domain. To accommodate multiple grid types, all grid iteration was modified to use iterator objects that were obtained from the individual grid implementations.

The simulation also need to be modified to include an additional "prepare" step to prepare the target grid for the next LBM timestep. The prepare step for array-based grids is a no-op, since all storage is pre-allocated at the beginning of the simulation. For RLE-based grids, however, the prepare step is used to configure the target grid to have all of the defined cells it needs for the next timestep (fluid cells plus the one-empty-cell layer). This configuration consists of trimming the source grid into the target grid, dilating the target grid back into the source grid, and then reinitializing the target grid to have the same defined cells as the source grid. The subsequent LBM timestep is then guaranteed to have all of the defined cells it needs.

The last LBM modification allowed RLE-based grids to include fluid and obstacles outside the domain. The original simulator again iterated over the x, y, z ranges of the domain to determine cells that were filled with fluid or an obstacle. This had to be changed for RLE-based grids to iterate over the bounding box of each object (whether fluid or obstacle) rather than the domain and inject cells that were inside the object.

RLE Set Modifications

During a trim operation, the original RLE set removed cells that were "far away" from the fluid surface. Since we cannot ignore all cells that are "far away" from the fluid surface when simulating fluids, the trim criteria was changed to retain only fluid cells that are not flagged as empty (i.e., cells that are flagged as fluid or interface cells).

During a dilate operation, cells in the target set are initialized either by copying fluid cells that are defined in the source set or the source set's injection map, or setting the cell's flag to empty and zeroing all of the cell's DFs for cells that aren't defined in the source grid or its injection map.

Lastly, because RLE sets can have defined cells that are in the injection map rather than the value array, iteration over the set visits all injected cells before it visits any of the cells in the value array.

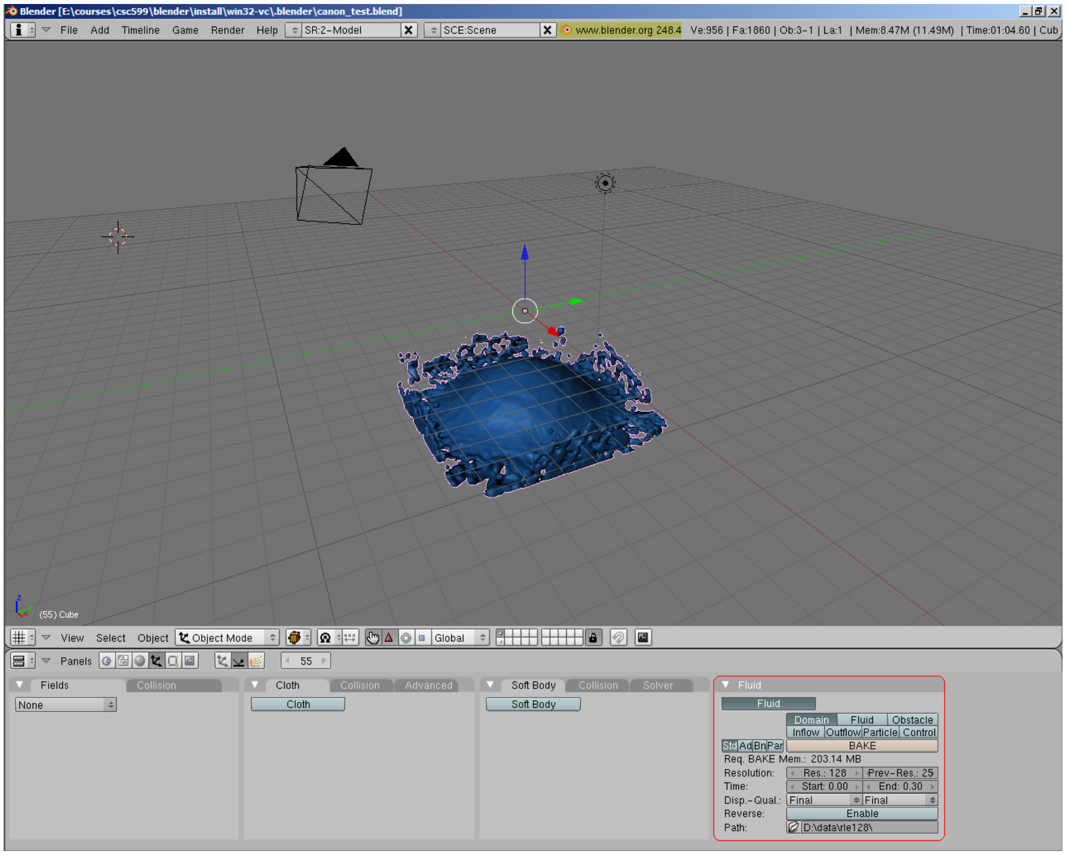


Figure 3.9: Blender interface with fluid panel.

3.4 Blender Changes

The LBM simulation code in Blender is fairly-well isolated from the rest of the code, which made it easy to include the modified simulator code. The only significant changes were interface changes in both Blender and the simulator's API to allow users to choose between array- and RLE-based grids, and to choose for RLE grids whether the fluid would be allowed to escape the domain.

These options were added to the Blender interface in the form of toggle buttons, located in the pre-existing Fluid panel that appears for Blender meshes. The fluid panel in Blender can be seen highlighted in Figure 3.9 with closeups of the new buttons in Figures 3.10 and 3.11. The domain escape button only appears if the user has chosen to use the RLE-based grid.

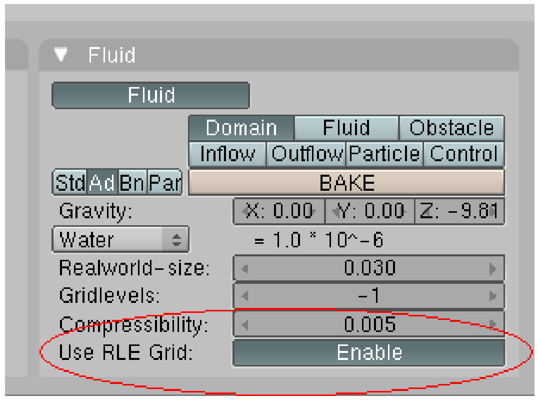


Figure 3.10: Blender interface button toggling array- vs. RLE-based grid storage.

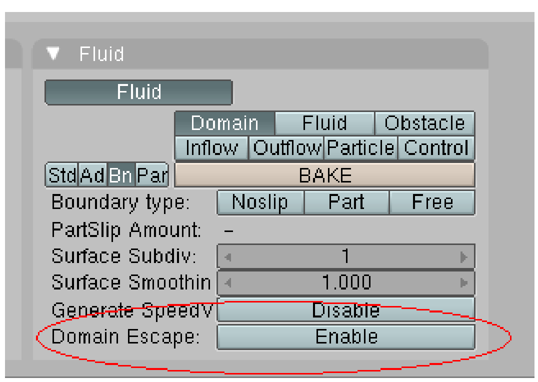


Figure 3.11: Blender interface button allowing RLE-based fluid to escape the original domain.

Chapter 4

Validation

To see how RLE-based simulations compare with array-based simulations, two experiments were performed using both RLE- and array-based grids and data was gathered regarding the time and memory required to perform the simulations, as well as the simulation outputs themselves.

4.1 Experiments

The two experiments that were performed involved a single drop of water starting from rest and falling inside a cubical domain. The domain for both experiments was 10x10x10 Blender units (BU) on each side. The first experiment used a water drop that had a radius r = 2.5 BU, and the second experiment used a drop with r = 1.5 BU. The proportions of the domain volume occupied by the fluid in each experiment are $\frac{\frac{4}{3}\pi(2.5)^3}{10^3} = 6.55\%$ and $\frac{\frac{4}{3}\pi(1.5)^3}{10^3} = 1.41\%$ respectively.

For each experiment, a simulation was run with the given setup at three different resolutions, with n = 32, 64, and 128 grid cells in each dimension. The start-to-finish wall clock time and memory usage were recorded for each experiment at each resolution, and the resulting fluid surface meshes were saved for comparison.An addition run was performed with n = 256, but the array-based simulation crashed Blender because it was trying to allocate too much memory. The RLE-based simulation started successfully, and memory measurements have been included for n = 256. The "domain escape" for the RLE-based simulations was also left disabled to enable a comparison. With the "domain escape" option enabled for RLE-based simulations, the water drops would simply have fallen straight out of the bottom of the domain.

All of the experiments were performed using an Intel Core 2 Duo @ 1.8GHz, with 2 GB of RAM on Windows XP.

4.2 Results and Analysis

When comparing the fluid surface meshes for array- and RLE-based simulations for the same setup and resolution, the meshes were found to be different. Upon further investigation, it was discovered that the meshes differed at small scales while the global behavior of the surface was consistent. This difference can be attributed to different instruction sequences (especially floating-point instructions) generated by the compiler for the different grid types. For instance, merely moving a C++ `class` definition from one header file to another caused similar differences. A comparison between the array- and RLE-based simulation for experiment 1 at three different timesteps with n = 64 can be seen in Figure 4.2. In the figure, the top row is the RLE-based simulation and the bottom row is the array-based simulation. The yellow wireframe cube denotes the boundaries of the domain. Note that, although the small details differ, the global fluid behavior is similar between the two simulation types.

The time and memory measurements for each experiment are detailed in Tables 4.1 and 4.2. The "RLE %" row gives the RLE results as a percentage of the corresponding array result. Figures 4.2 and 4.2 also compare the array- and RLE-based simulation measurements graphically. The results show that, while the RLE-based simulations take several times longer than the corresponding array-based simulations, the memory savings can be significant.

n	32	64	128	256
Mem. Usage (MB)				
Array	5.75	46.00	368.00	N/A
RLE	4.89	24.72	126.12	419.73
RLE %	85.00	53.74	34.27	N/A
Time Taken (s)				
Array	41.95	421	6287	N/A
RLE	182	2183	28072	N/A
RLE %	433.90	518.53	446.51	N/A

Table 4.1: Time and memory results for experiment 1 (r = 2.5 BU).

n	32	64	128	256
Mem. Usage (MB)				
Array	5.75	46.00	368.00	N/A
RLE	4.89	16.63	83.43	281.56
RLE %	85.00	36.16	22.67	N/A
Time Taken (s)				
Array	20.34	204	2766	N/A
RLE	95	944	10499	N/A
RLE %	467.04	462.75	379.57	N/A

Table 4.2: Time and memory results for experiment 2 (r = 1.5 BU).

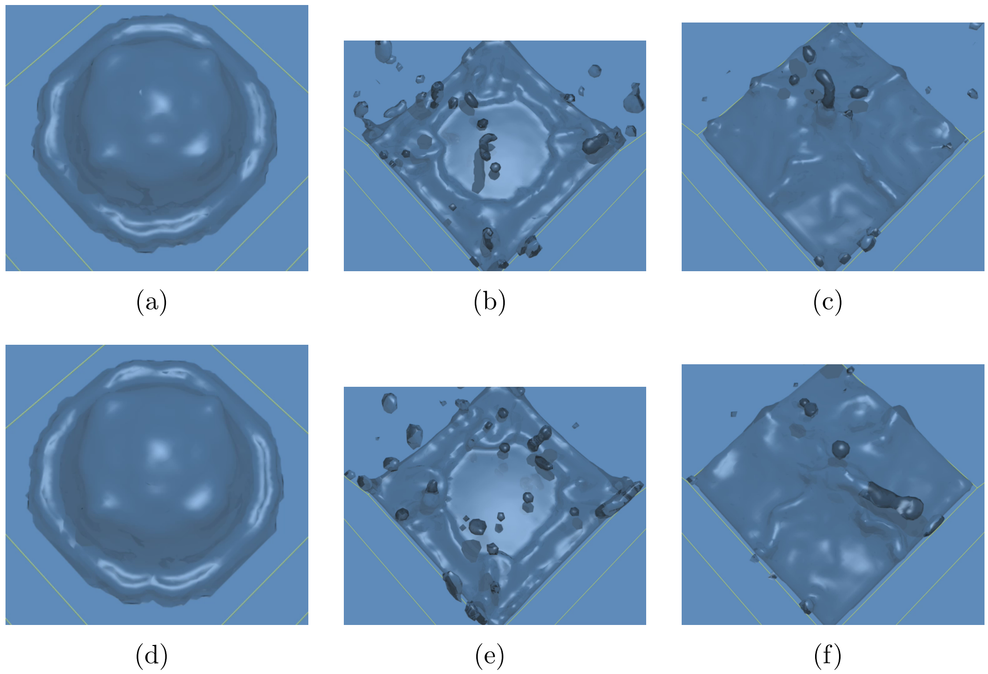
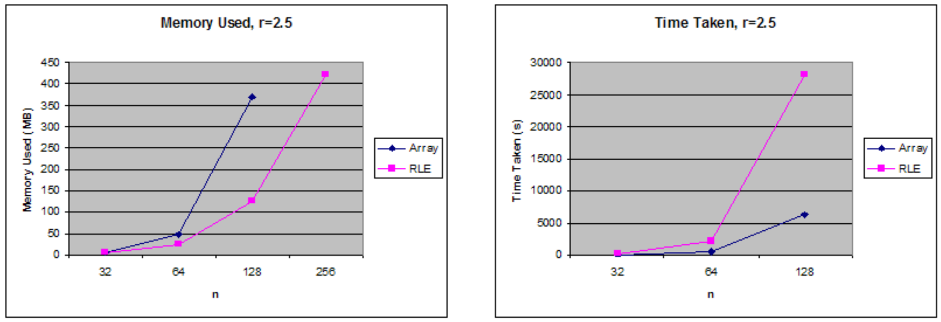


Figure 4.1: Comparison of RLE- vs. array-based fluid simulations. (a) - (c) are RLE-based, and (d) - (f) are array-based.

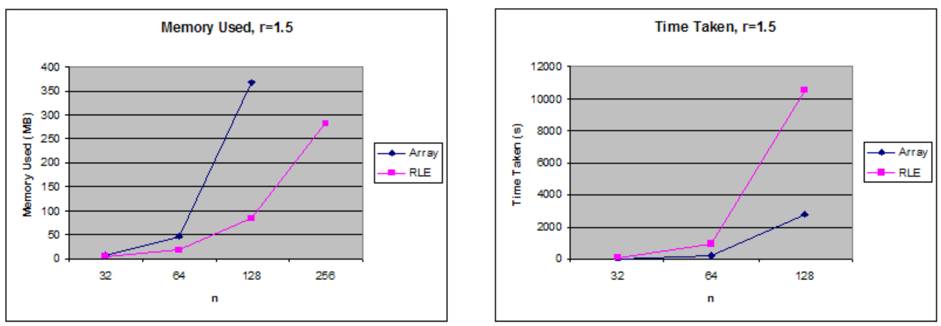
The memory savings achieved by the RLE-based simulation are substantial, ranging between 15% for n = 32 to 66% and 77% for n = 128 in experiments 1 and 2 respectively. Also notice that the percentages of memory used and times taken by the RLE-based simulation at each resolution in experiment 2 (r = 1.5) are less than or equal to those of experiment 1 (r = 2.5), while the memory and times measured for the array-based simulations are constant between experiments. This matches the expected behavior of time and memory requirements being proportional to the volume of the fluid rather than the volume of the domain, and also the expected time and memory reduction when the fluid occupies a smaller proportion of the domain.

In both of these experiments, the RLE-based simulation took roughly five times as long as the array-based simulation. The array-based simulation can access cell contents' in O(1) time, but must iterate over each cell in the grid even though it does not perform any processing for boundary and empty cells. The RLE-based simulation iterates only over defined cells but has O(log(n)) access time. Thus, the tradeoff when using RLE grids is decreased iteration time at



(a) Memory usage (b) Time taken

Figure 4.2: Results for experiment 1 (r = 2.5 BU).



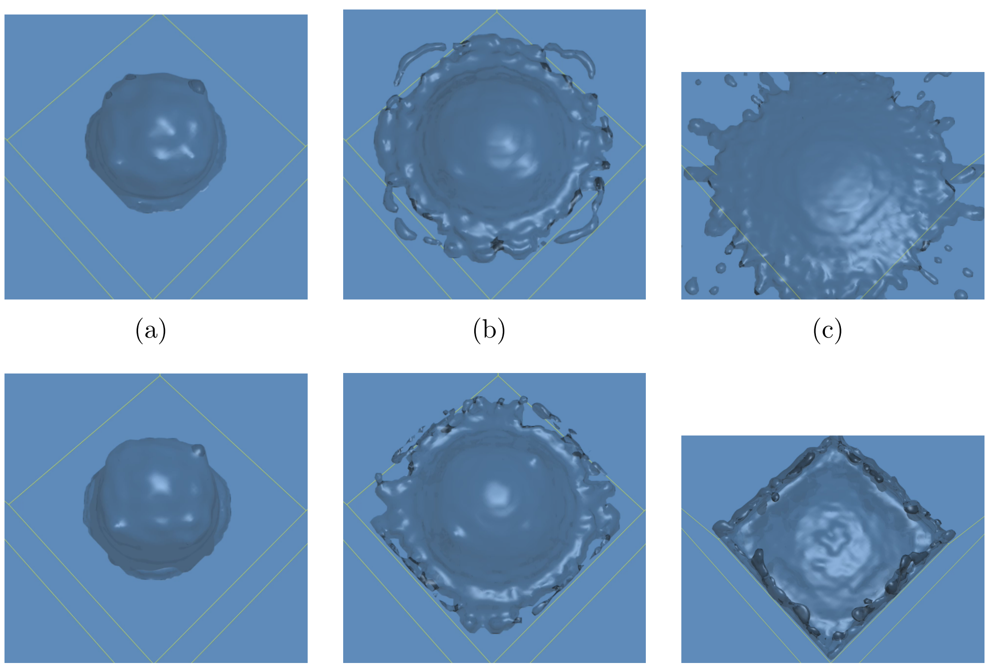
(a) Memory usage (b) Time taken

Figure 4.3: Results for experiment 2 (r = 1.5 BU).

the expense of increased cell access time. Therefore, RLE simulations should take less time than the corresponding array simulations when the time saved during iteration is greater than the increased access time, which will happen when the domain is very large and the volume occupied by the fluid is very small in proportion to the domain. Interestingly, the RLE % for n = 64 actually increased, which is the opposite effect of what one would expect. Perhaps this can be attributed to poor cache alignment for the particular experimental setups. Unfortunately, achieving this seems to require much larger domains than Blender can generate for array-based simulations, which prevents RLE-based simulations from reaching their potential as far as simulation time is concerned. On the other hand, RLE-based simulations are essential for these very large domains because they enable simulations to be performed that were not previously possible.

4.3 RLE Domain Escape

It is interesting to compare the results of having the "domain escape" option enabled vs. disabled for RLE-based simulations. Another experiment was performed using n = 64, with the bottom plane of the domain set to be an obstacle (although the obstacle is not rendered). The results of the experiment are illustrated in Figure 4.3, where the top row depicts a simulation with the "domain escape" option enabled and the bottom row depicts the option disabled. We can see that the fluid in the simulation with the option disabled encounters an invisible wall at the edge of the domain (the yellow wireframe cube) while the simulation with the option enabled allows fluid to move outside of the domain.



(a) (b) (c)

(d) (e) (f)

Figure 4.4: Comparison of RLE-based fluid with the domain escape option enabled and disabled. (a) - (c) have the option enabled, and (d) - (f) have the option disabled.

4.4 Comparison to Houston et al

Because Houston et al [32] used their implementation of an RLE level set to perform Navier-Stokes fluid simulations, it would also be worthwhile to compare the RLE-based fluid simulation presented in this thesis to their results. In [32], Houston et al do not provide any timing measurements for fluid simulations performed using their RLE implementation, but they do achieve memory savings comparable to the results in this thesis. The maximum memory savings they achieved was 77% using a resolution of 136x136x320, which is almost three times as large in terms of voxel count as the n = 128 grid from experiment 1. Their experimental results are not directly comparable with ours due to the different domain sizes and fluid initial conditions, but they are in the same ballpark. Their experiments used much less total memory than the ones presented here, using 20MB for their RLE-based 136x136x320 experiment vs. 126MB for our 128x128x128 experiment), but this can be attributed to the different simulation methods used since LBM stores 19 velocities per grid cell instead of one velocity and one density in Navier-Stokes simulations.

Chapter 5

Future Work

Although RLE-based grids save a lot of memory, there are several improvements that could be made.

First, the original LBM simulation had the ability to use multiple grids, where the additional grids had a much coarser resolution and were used for taking larger timesteps in fluid regions that were far away from the fluid surface [84], thereby speeding up the simulation. Results from the coarse grids then had to be merged back into the finest grid, which required several additional cell flags and passes over the grid. No part of the RLE-based fluid representation explicitly prevents multiple grids from being used, but this algorithmic enhancement was intentionally left unaddressed by this thesis. Enabling multiple grids has the potential to bring simulation times when using RLE-based grids more in line with those of array-based grids.

One of the LBM's advantages over other simulation methods is the ease with which it can be distributed among multiple processors. [84] describes how the OpenMP API [7] can be used on array-based grids to speed up simulation times. This method involved dividing the grid array into two or more partitions of similar size and processing each partition with a different processor. Because there is no immediately obvious way to partition RLE grids into similarly-sized partitions, using an RLE grid prevents parallelization. The original LBM simulation implemented in Blender did not enable parallelized simulations and thus was limited to a single processor for both array- and RLE-based grids, but creating a method to perform parallelized RLE grid simulations could also achieve significant speedups.

Since the greatest disadvantage of RLE grids is the increase in simulation time, it would be worth investigating additional methods for optimizing the RLE data structure. For instance, if the simulation could perform its calculations within the dilate operation when initializing cell values, RLE-based simulations could avoid having to make another pass over the grid cells afterward (in the normal LBM timestep), which could reduce simulation times by roughly 33%. Also, maintaining iterators for not just the current grid cell but also all of its neighbors when iterating over the grid (as described in [32]) would prevent the simulation from having to use the $O(log_n)$ RLE random access method to access neghboring cell values. This would speed simulations up considerably.

Chapter 6

Conclusion

This thesis has demonstrated that RLE-based LBM simulations are not only possible, but can provide significant memory savings at the cost of increased simulation time. Due to the magnitude of the time increase, array-based grids are probably still preferable in most situations that will be encountered in practice, especially in situations where domains will be mostly or completely filled with fluid. However, when the amount of memory used is a concern, then RLE-based grids are the way to go. An example of this type of situation could be simulations performed on embedded devices such as the iPhone or other handheld devices, although these simulations would likely experience further slowdowns due to the lack of computing power available on most mobile platforms. RLE-based simulation *is* necessary in a couple of specific situations where fluid needs to be able to move outside of the original domain or the system performing the simulation does not have enough memory to allocate the array grids.

We believe that RLE-based simulation still has the potential to be faster than array-based simulation in situations where the domain is very large and the percentage of the volume occupied by the fluid is very small, although the evidence presented in this thesis does not support this assertion. We attempted to find a situation where an RLE-based simulation would be faster than an equivalent array-based simulation, which resulted in the waterslide setup seen in Figure 6.1.

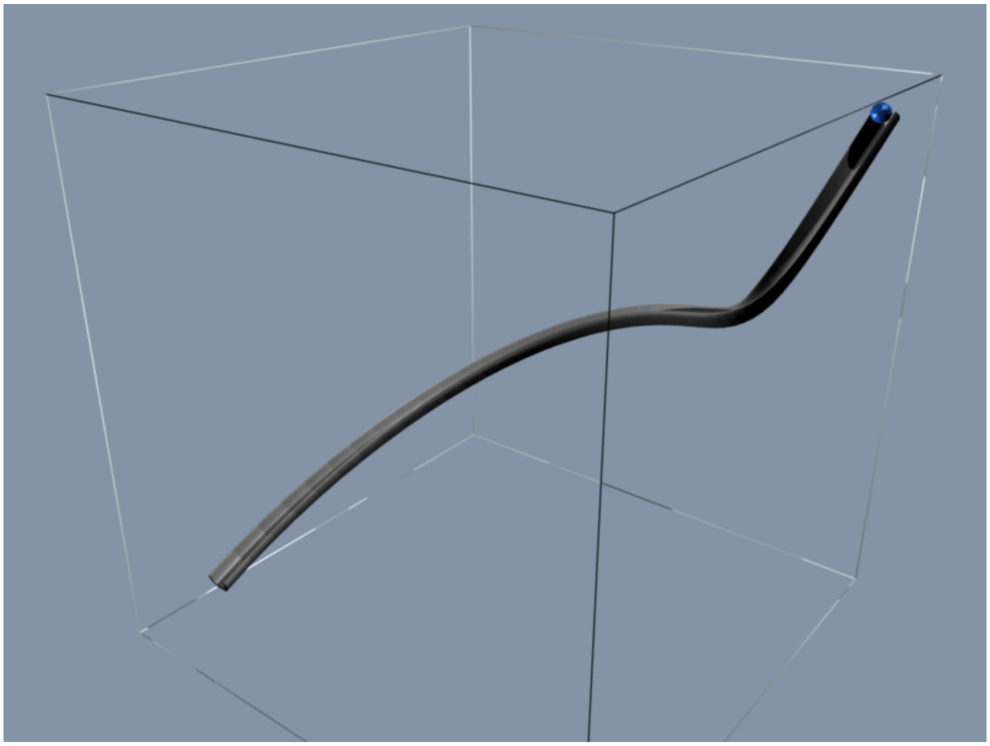


Figure 6.1: Experimental setup with large domain and small proportion of fluid.

This setup contains a waterslide roughly traversing the domain's (rendered in wireframe) diagonal with a fluid source at the top-right. Unfortunately, we could not run the array-based simulation at a high-enough resolution to compare it to the RLE-based simulation due to memory limitations on the system running the experiments. Perhaps future work can confirm or deny this claim by running the experiment on a 64-bit system with more memory.

The primary factor working against the RLE-based simulation is the several-fold increase in the time required to complete a simulation. If this time could be reduced by code optimization and the techniques described in Chapter 5 to only two-fold or even less, then RLE-based simulations would become a much more competitive alternative to the current simulator. As it currently stands, array-based simulations are still king in most situations (at least for the LBM), even though RLE grids do have their own niche in which they excel (memory constrained simulations and fluid outside of the domain).

Bibliography

[1] Utah raster toolkit home page. http://www.cs.utah.edu/gdc/projects/urt/.

[2] A. Averbuch, L. Ioffe, M. Israeli, and L. Vozovoi. Highly scalable two- and Three-Dimensional Navier-Stokes parallel solvers on MIMD multiprocessors. *J. Supercomput.*, 11(1):7–39, 1997.

[3] J. Barbic and J. Popovic. Real-time control of physically based simulations using gentle forces. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–10, Singapore, 2008. ACM.

[4] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *J. Acoust. Soc. Am Phys Rev*, 94:511, 1950.

[5] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral One-Component systems. *Physical Review*, 94(3):511, May 1954. Copyright (C) 2009 The American Physical Society; Please report any problems to prola@aps.org.

[6] R. E. Bridson. *Computational aspects of dynamic surfaces.* PhD thesis, stanford university, 2003.

[7] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP.* Morgan Kaufmann, 2001.

[8] J. X. Chen and N. da Vitoria Lobo. Toward interactive-rate simulation of

fluids with moving obstacles using Navier-Stokes equations. *Graph. Models Image Process.*, 57(2):107–116, 1995.

[9] S. Chen and G. D. Doolen. LATTICE BOLTZMANN METHOD FOR FLUID FLOWS. *Annual Reviews in Fluid Mechanics*, 30(1):329–364, 1998.

[10] M. Chiappetta. Cryostasis to use PhysX for fluid simulation - HotHardware. http://hothardware.com/News/Cryostasis-To-Use-PhysX-For-Fluid-Simulation/, 2008.

[11] S. Clavet, P. Beaudoin, and P. Poulin. Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228, Los Angeles, California, 2005. ACM.

[12] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312. ACM, 1996.

[13] M. Desbrun and M. P. Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *6th Eurographics Workshop on Computer Animation and Simulation96*, page 6176, 1996.

[14] R. Durikovic and K. Numata. Preserving the volume of fluid using multiphase flow approach. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 757–760, 2006.

[15] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 2001.

[16] A. Ferrari, M. Dumbser, E. F. Toro, and A. Armanini. A new 3D parallel SPH scheme for free surface flows. *Computers & Fluids*, In Press, Corrected Proof, 2008.

[17] O. Filippova and D. Hnel. Grid refinement for lattice-BGK models. *J. Comput. Phys.*, 147(1):219–228, 1998.

[18] O. Fischer, T. Kuthada, J. Wiedemann, P. A. B. Dethioux, R. Mann, and B. Duncan. CFD validation study for a sedan scale model in an open jet wind tunnel. 2008.

[19] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30. ACM, 2001.

[20] N. Foster and D. Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, 1996.

[21] A. Fournier and W. T. Reeves. A simple model of ocean waves. *SIGGRAPH Comput. Graph.*, 20(4):75–84, 1986.

[22] Frisch, Hasslacher, and Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56(14):1505–1508, Apr. 1986. PMID: 10032689.

[23] Gametrailers.com. Gametrailers.com - the history of water in video games by facetheglue. http://www.gametrailers.com/player/usermovies/145904.html, 2007.

[24] 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, 1977.

[25] T. G. Goktekin, A. W. Bargteil, and J. F. O'Brien. A method for animating viscoelastic fluids. *ACM Trans. Graph.*, 23(3):463–468, 2004.

[26] E. Guendelman, A. Selle, F. Losasso, and R. Fedkiw. Coupling water and smoke to thin deformable and rigid shells. In *ACM SIGGRAPH 2005 Papers*, pages 973–981, Los Angeles, California, 2005. ACM.

[27] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, page 6370, 2007.

[28] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8:2182, 1965.

[29] M. Harris. Fast fluid dynamics simulation on the GPU. *GPU gems*, 1:637665, 2004.

[30] J. Hauser, M. Spel, J. Muylaert, and R. D. Williams. ParNSS: an efficient parallel Navier-Stokes solver for. *COMPLEX GEOMETRIES, AIAA 94-2263, AIAA 25TH FLUID DYNAMICS CONFERENCE*, pages 94—2263, 1994.

[31] R. Hoetzlein and T. Hollerer. Interactive water streams with sphere scan conversion. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 107–114, Boston, Massachusetts, 2009. ACM.

[32] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.*, 25(1):151–175, 2006.

[33] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *ACM SIGGRAPH 2006 Papers*, pages 805–811, Boston, Massachusetts, 2006. ACM.

[34] J. Iversen and R. Sakaguchi. Growing up with fluid simulation on "The day after tomorrow". In *ACM SIGGRAPH 2004 Sketches*, page 142, Los Angeles, California, 2004. ACM.

[35] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57, Dallas, TX, USA, 1990. ACM.

[36] Khronos Group. OpenCL. http://www.khronos.org/opencl/, 2009.

[37] T. Kim, N. Thurey, D. James, and M. Gross. Wavelet turbulence for fluid simulation. *ACM Trans. Graph.*, 27(3):1–6, 2008.

[38] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122. ACM New York, NY, USA, 2004.

[39] A. Kolb and N. Cuntz. Dynamic particle coupling for GPU-based fluid simulation. In *18th Symposium on Simulation Technique. To appear*, 2005.

[40] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131. ACM New York, NY, USA, 2004.

[41] C. Korner, T. Pohl, U. Rude, N. Thurey, and T. Zeiser. *Parallel Lattice Boltzmann Methods for CFD Applications*, pages 439–466. 2006.

[42] W. Li, Z. Fan, X. Wei, and A. Kaufman. GPU-Based flow simulation with complex boundaries. *GPU Gems*, 2:747764, 2005.

[43] W. Li, X. Wei, and A. Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19(7):444–456, Dec. 2003.

[44] Y. Liu, X. Liu, and E. Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 247–256, 2004.

[45] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM New York, NY, USA, 1987.

[46] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 Papers*, pages 457–462, Los Angeles, California, 2004. ACM.

[47] F. Losasso, T. Shinar, A. Selle, and R. Fedkiw. Multiple interacting liquids. *ACM Trans. Graph.*, 25(3):812–819, 2006.

[48] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-Way coupled SPH and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[49] H. Mao and Y. Yang. Particle-based immiscible fluid-fluid collision. In *Proceedings of Graphics Interface 2006*, pages 49–55, Quebec, Canada, 2006. Canadian Information Processing Society.

[50] A. McNamara, A. Treuille, Z. Popovi?, and J. Stam. Fluid control using the adjoint method. In *ACM SIGGRAPH 2004 Papers*, pages 449–456, Los Angeles, California, 2004. ACM.

[51] G. McNamara and G. Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[52] R. Mei, W. Shyy, D. Yu, and L. Luo. Lattice boltzmann method for 3-D flows with curved boundary. *J. Comput. Phys.*, 161(2):680–699, 2000.

[53] J. Mezger, S. Kimmerle, and O. Etzmu. Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG*, 11(2):322–329, 2003.

[54] Microsoft Corp. Release notes for nov. 2008 DirectX SDK. http://msdn.microsoft.com/en-us/directx/aa937789.aspx, 2009.

[55] V. Mihalef, D. Metaxas, and M. Sussman. Animation and control of breaking waves. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 315–324, Grenoble, France, 2004. Eurographics Association.

[56] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

[57] A. Monitzer. Fluid simulation on the gpu with complex obstacles using the lattice boltzmann method, 7 2008.

[58] MPAA. Entertainment industry market statistics 2007., 2007.

[59] M. Muller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2003.

[60] M. Muller, B. Solenthaler, R. Keiser, and M. Gross. Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 237–244, Los Angeles, California, 2005. ACM.

[61] D. R. Noble, J. G. Georgiadis, and R. O. Buckius. COMPARISON OF ACCURACY AND PERFORMANCE FOR LATTICE BOLTZMANN AND FINITE DIFFERENCE SIMULATIONS OF STEADY VISCOUS FLOW. *International Journal for Numerical Methods in Fluids*, 23(1):1–18, 1996.

[62] NVIDIA Corporation. NVIDIA makes physics a reality for gamers. http://www.nvidia.com/object/io_1218533603421.html, 2008.

[63] NVIDIA Corporation. CUDA zone – the resource for CUDA developers. http://www.nvidia.com/object/cuda_home.html#, 2009.

[64] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html, 2009.

[65] OpenGL ARB. ARB_vertex_program, 2002.

[66] S. Osher and R. P. Fedkiw. *Level set methods and dynamic implicit surfaces.* Springer, 2003.

[67] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares. Particle-based viscoplastic fluid/solid simulation. *Computer-Aided Design*, 2008.

[68] D. R. Peachey. Modeling waves and surf. *SIGGRAPH Comput. Graph.*, 20(4):65–74, 1986.

[69] F. Pighin, J. M. Cohen, and M. Shah. Modeling and editing flows using advected radial basis functions. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 223–232, Grenoble, France, 2004. Eurographics Association.

[70] S. Plimpton, B. Hendrickson, S. Attaway, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society Washington, DC, USA, 1996.

[71] N. Rasmussen, D. Enright, D. Nguyen, S. Marino, N. Sumner, W. Geiger, S. Hoon, and R. Fedkiw. Directable photorealistic liquids. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 193–202, Grenoble, France, 2004. Eurographics Association.

[72] RenderHeads. RenderHeads - portfolio - fluid demo. http://www.renderheads.com/portfolio/fluiddemo/, 2007.

[73] A. Robinson-Mosher, T. Shinar, J. Gretarsson, J. Su, and R. Fedkiw. Two-way coupling of fluids to rigid and deformable solids and shells. *ACM Trans. Graph.*, 27(3):1–9, 2008.

[74] N. P. School. Basic concepts in physical oceanography. http://www.oc.nps.edu/nom/day1/parta.html, 2009.

[75] M. Shah, J. M. Cohen, S. Patel, P. Lee, and F. Pighin. Extended galilean invariance for adaptive fluid simulation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 213–221, Grenoble, France, 2004. Eurographics Association.

[76] L. Shi and Y. Yu. Taming liquids for rapidly changing targets. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 229–236, Los Angeles, California, 2005. ACM.

[77] S. E. Siwek. Video games in the 21st century, 2007.

[78] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91(3):99–164, 1963.

[79] J. Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[80] Stanford University. Computer scientist makes splash with academy award for fluid simulation. http://www.sciencedaily.com/releases/2008/01/080126100827.htm, 2008.

[81] J. Steinhoff and D. Underhill. Modification of the euler equations for vorticity confinement: Application to the computation of interacting vortex rings. *Physics of Fluids*, 6:2738, 1994.

[82] N. Tamura, N. Tsumura, T. Nakaguchi, and Y. Miyake. Spring-bead animation of viscoelastic materials. In *ACM SIGGRAPH 2005 Sketches*, page 64, Los Angeles, California, 2005. ACM.

[83] T. Theuns and M. E. Rathsack. Calculating short range forces on a massively parallel computer: SPH on the connection machine. *Computer physics communications*, 76(2):141–158, 1993.

[84] N. Thuerey. *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method*. PhD thesis, University of Erlangen-Nuremberg, 2007.

[85] N. Thurey, R. Keiser, M. Pauly, and U. Rude. Detail-preserving fluid control. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 7–12, Vienna, Austria, 2006. Eurographics Association.

[86] N. Thurey and U. Rude. Free surface lattice-Boltzmann fluid simulations with and without level sets. In *Workshop on vision, modelling, and visualization (VMV Stanford)*, page 199208, 2004.

[87] B. C. Vemuri, Y. Cao, and L. Chen. Fast collision detection algorithms with applications to particle flow. In *Computer Graphics Forum*, volume 17, pages 121–134. Blackwell Publishing, 1998.

[88] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 75–85, San Diego, California, 2003. Eurographics Association.

[89] J. Wilke, T. Pohl, M. Kowarschik, and U. Rude. *Cache Performance Optimizations for Parallel Lattice Boltzmann Codes*, pages 441–450. 2003.

[90] C. Wojtan and G. Turk. Fast viscoelastic behavior with thin features. In *ACM SIGGRAPH 2008 papers*, pages 1–8, Los Angeles, California, 2008. ACM.

[91] Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman. Flow simulation with locally-refined LBM. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 181–188, Seattle, Washington, 2007. ACM.

[92] H. Zhu, K. Bao, E. Wu, and X. Liu. Stable and efficient miscible liquid-liquid interactions. In *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, pages 55–64, Newport Beach, California, 2007. ACM.