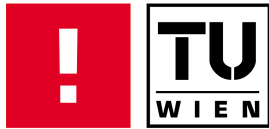
Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).



FAKULTÄT FÜR !NFORMATIK

Fluid Simulation on the GPU with Complex Obstacles Using the Lattice Boltzmann Method

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik und Digitale Bildverarbeitung

ausgeführt von

Andreas Monitzer
Matrikelnummer 0225165

am:
Institut für Computergraphik und Algorithmen und
VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH

Betreuung:
Betreuer: *Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Eduard Gröller*
Mitwirkung: *Dipl.-Inf. Raphael Fuchs*

Wien, 17.07.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Fluid Simulation on the GPU with Complex Obstacles Using the Lattice Boltzmann Method

Andreas Monitzer

mailto:andreas@monitzer.com
http://www.monitzer.com

Abstract

[English]

Real-time computer graphics and simulation has advanced to a level of realism that was regarded as unthinkable a few decades ago. However, fluid simulations are still in an infant state for applications that require interactivity. Recent developments in programmability of graphics processing units on current graphics cards have enabled researchers to treat these cards as stream co-processors. This class of processors are designed for parallelizable algorithms that do not make heavy use of branching. Algorithms having these properties can be accelerated significantly compared to implementations on current central processing units. Since grid-based fluid simulations fit perfectly into this scheme, this has become a hot topic in research. Various approaches will be presented in order to determine a combination of algorithms that can easily be parallelized and allow integrating rigid objects with complex boundaries into a fluid simulation at interactive rates. Additionally, the usage of fluid simulations in computer games will be discussed. An underwater pinball game will be introduced as a practical example, highlighting the considerations that have to be taken into account when adding this game element that was previously impossible to use.

[German]

Die Echtzeit-Computergrafik und -Simulation haben sich zu einem Niveau entwickelt, das vor wenigen Jahrzehnten noch als undenkbar galt. Ungeachtet dessen stecken Flüssigkeitssimulationen für interaktive Anwendungen immer noch in den Kinderschuhen. Die progressive Weiterentwicklung der Graphikkarten-prozessoren hat in letzter Zeit dazu geführt, dass Forscher die Karten als Stream-Koprozessoren behandeln können. Diese Klasse von Prozessoren wurde entwickelt, um parallelisierbare Algorithmen mit wenigen Sprungbefehlen zu verarbeiten. Die Laufzeit von Algorithmen, die diese Eigenschaften besitzen, kann auf einen Bruchteil der auf herkömmlichen Prozessoren benötigten verkürzt werden. Da rasterbasierte Flüssigkeitssimulation perfekt in dieses Schema passt, ist es zu einem sehr aktiven Forschungsgebiet geworden. Es werden einige Ansätze präsentiert werden, um eine gute Algorithmen-Kombination zu finden, die einfach parallelisiert werden kann und gleichzeitig Objekte mit komplexen Oberflächen erlauben. Zusätzlich wird die Verwendung von Flüssigkeitssimulationen in Computerspielen betrachtet. Ein Unterwasser-Flipper wird als praktisches Beispiel vorgestellt, um die Überlegungen hervorzuheben, die notwendig sind, um eine solche früher nicht verwendbare Simulation effektiv einzusetzen.

Contents

1 Nomenclature **iii**

2 Introduction **1**
2.1 Previous Work . . . 3
2.2 The Navier-Stokes Equations . . . 4
2.2.1 Euler's Simplified Equations . . . 5
2.2.2 Boundary Conditions . . . 6
2.2.3 Implementations . . . 6

3 Fluid Simulation Using the Lattice Boltzmann Method **8**
3.1 The Lattice Gas Cell Automata . . . 8
3.2 The Lattice Boltzmann Method . . . 9
3.2.1 Lattice Geometry . . . 9
3.2.2 The Macroscopic Properties of a Lattice Cell . . . 10
3.3 Gravity . . . 13
3.4 Initial Conditions . . . 13
3.5 Boundary Conditions . . . 13
3.6 Physical Correspondency . . . 14

4 General Purpose-Programming on Graphics Hardware **16**
4.1 Bitonic Sort . . . 16
4.2 Shader Programming . . . 20
4.3 CUDA . . . 22
4.3.1 Architecture . . . 22
4.3.2 CUDA's Programming Language . . . 25
4.3.3 Optimization Strategies . . . 27
4.4 Adapting CFD to the GPU Using CUDA . . . 30
4.4.1 Visualizing the Flow . . . 31

5 Complex Obstacles in Fluid Simulations **35**
5.1 Voxelization on the GPU . . . 35
5.2 Integrating a Physics Engine into a Fluid Simulation . . . 39
5.2.1 Rigid Body Simulation using the Bullet Physics Library . . . 39

5.3 Solid-Fluid Coupling 40
5.3.1 Mei et al.'s Extrapolation Method 40
5.3.2 Noble's Method for Partially Saturated Cells 42
5.4 Fluid-Solid Coupling 45
5.5 Two-Way Coupling 46

6 Game Design with Fluid Simulations and Their Implementation 48
6.1 The Fluid Pinball 49
6.1.1 The Pinball Game 49
6.1.2 Adapting the LBM Implementation to a Game 51
6.1.3 Ogre 52
6.1.4 Pinball Game Elements 53
6.1.5 Implementation 55

7 Results 67
7.1 The Basic Application 67
7.2 The Game-like Application 68
7.3 Analysis of the Performance Measurement Results 68

8 Conclusion 71

Chapter 1

Nomenclature

i	index over the lattice directions
$f_i(x,t)$	particle distribution at position x and time t in direction i
f_i^{eq}	equilibrium distribution function in direction i
τ	fluid viscousity
ρ	fluid density
p	fluid pressure
u	fluid velocity
u_s	solid velocity
ρ_s	solid density
Ω_i	fluid collision operator in direction i
Ω_i^s	solid collision operator in direction i
Δx	grid spacing on the x-axis
Δy	grid spacing on the y-axis
Δz	grid spacing on the z-axis
Δt	the amount of time requred for calculating one iteration of the simulation (might vary on every iteration)

Chapter 2

Introduction

Real-time computer graphics have advanced to a level of realism that was regarded as impossible a few decades ago. However, fluid simulations are still in an infant state for applications that require interactivity, due to performance issues.

Graphics processing units (GPUs) on current graphics cards can be treated as stream co-processors. This class of processors is designed for parallelizable algorithms that do not make heavy use of branching. Algorithms having these properties can be accelerated significantly compared to implementations on current central processing units (CPUs). Additionally, GPUs do not suffer from caching issues, since they have very closely defined input and output streams and are optimized at the hardware level for this configuration.

Grid-based fluid simulations are an obvious choice for GPU-based calculation, since operating on the cells of a grid is easily parallelizable. However, care has to be taken to avoid slowdowns caused by making inefficient use of the card's features.

In previous work, a fluid animation using LBM took 244 seconds per frame to calculate [Thürey *et al.*, 2006]. In this thesis, we introduce a method for bringing similar animations to interactive rates, including complex objects immersed in the fluid. As a test case, a computer game was implemented, which uses fluid behavior as the primary game element in a pinball-like simulation (see Figure 2.1). Even though the animation has to be believable by the user, it does not have to be physically correct. In addition, a visualization of the fluid is required, which does not have to be physics-based, but allows the player to see motion in the game world.

Since today's computational power is thought to be inadequate for real-time fluid animations, realistic 3 dimensional fluids are not used in current games on the market, missing an important element that can enhance the immersion into the virtual world. In the future, many types of games can benefit from adding these capabilities, like puzzle, role-playing or action games or in the edutainment sector.

A complete implementation for a game requires the consideration of multiple aspects with competing solutions, not all equal for the task at hand. Section 2.2 outlines the classic approach to basic fluid simulation using the Navier-Stokes equations, while Chapter 3 describes the Lattice Boltzmann approach,

Figure 2.1: A real-world implementation of the fluid pinball concept from TOMY, 1977, a tank filled with water. There are two pumps at the bottom, each to be controlled by a player by pressing the big orange buttons. A point is scored when the ball is sent through the opponent's goal. When the orange button is pressed, the water pressure also elevates the footballer's foot at the bottom and the goalkeeper, allowing the player to parry an incoming ball.

better suited for the concrete programming environment, which is explained in Chapter 4. Since a game requires interactive interaction between solids and fluids, multiple solutions for immersing complex obstacles are explained in Chapter 5. All these ideas are then collected and used in a complete framework for games, as described in Chapter 6, which is evaluated in Chapter 7. Chapter 8 concludes this thesis.

2.1 Previous Work

In 1965, Intel's co-founder Gordon E. Moore formulated an empirical observation, which is well-known today as Moore's law. Its prediction was remarkably accurate. However, in recent years, physical barriers have slowed down progress. Current advancement in computational speed is aimed at parallelization, but unlike faster processors, programs do not support parallelization automatically, it has to be implemented by the programmer, or even new algorithms have to be developed.

In the field of computer graphics (CG), parallelization was always a well-known technique for speeding up offline and online rendering. For example, companies like Pixar^{TM} or DreamWorks^{TM} use server clusters to create fully CG-based animation movies. In the area of interactive rendering for personal computers, the companies ATI^{TM} (now part of AMD^{TM}) and NVidia^{TM} developed graphics cards that use multiple cores for rendering three dimensional scenes at interactive rates[1].

Since these graphics cards became more and more powerful, both companies added more and more rendering features to them, until a general programming language for GPUs was introduced, which was very similar to assembler. To simplify the development process, C-based languages were specified, which allowed easy creation of programs to run directly on the GPU. The most important ones are the High-Level Shading Language implemented in Direct3D, the OpenGL Shading Language used in OpenGL, and NVidia^{TM}'s Cg, which was implemented for both graphics programming interfaces. All three are similar to each other.

Another consequence of the shading languages were the possibility to use the GPU for general purpose calculations (GPGPU). The cards support a two-dimensional storage format called texture, which can be used as both input and output of calculations (but not the same texture for both operations at the same time). A shader can use a gathering-based algorithm to write arbitrary values to a given location in the output texture. Since the result of a calculation can not interfere with other calculations in any way in a single pass, the architecture can be used as a streaming processor.

This approach requires knowledge of the OpenGL- or Direct3D-API, which is outside the scope of most scientists who want to use parallel streaming processors. Thus, NVidia^{TM} developed an extension to the C++ programming language called CUDA, allowing a more direct approach to programming the GPU. It is explained in more detail in section 4.3.

Not all tasks can be accomplished by a stream architecture, but one of those fields that have that potential is computational fluid dynamics (CFD), which is the main focus of this thesis.

[1]Interactive rates are defined as being fast enough to create the impression of motion in the observer. When considering humans, this is a rate of 25 to 60 full frames per second.

2.2 The Navier-Stokes Equations

The first attempts at simulating fluids in computer graphics were using wave-based approximations that do not allow interactivity with the fluid [Peachey, 1986], but give a very realistic impression used in many computer games, as shown in Figure 2.2. Wejchert and Haumann [Wejchert and Haumann, 1991] implemented more complex two dimensional flows by assembling them from well-known primitives like vortices and sinks.

Figure 2.2: A screenshot from the game Crysis, demonstrating the wave-based fluid simulation used in most of the current games.

Chen and da Vitoria Lobo [Chen and da Vitoria Lobo, 1995] introduced the Navier-Stokes equations (NS) to the graphics community, which are directly derived from Newton's second law, as a method for simulating flows even at interactive rates. They allow calculating the fluid movements at arbitrary detail, and are suitable for describing many different phenomena, like water, clouds, smoke, foams, and even motion of stars inside a galaxy.

The basic formulation of the NS for incompressible fluids are [Chen and da Vitoria Lobo, 1995; Stam, 1999]:

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

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

where $\cdot$ denotes a dot product between vectors, ∇ is the vector of spatial partial derivatives (see Table 2.1), u and p are the velocity and pressure field of the fluid respectively, ρ is the density and ν is the kinematic viscosity. f is a vector representing external forces.

Operator	Definition
Gradient	$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}\right)$
Divergence	$\nabla \cdot u = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$
Directional Derivative	$u \cdot \nabla = u_x \frac{\partial}{\partial x} + u_y \frac{\partial}{\partial y} + u_z \frac{\partial}{\partial z}$
Laplacian	$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$

Table 2.1: Explanation of the ∇ operator used in the NS for three dimensions.

Following the approach of Harris [Harris, 2004], (2.2) is called the *continuity equation*. It means that fluids conserve mass [Stam, 1999]. The right-hand side of (2.1) consists of four parts:

The advection term $-(u \cdot \nabla)u$ represents the force that the surrounding fluid particles exert on a particle and causes it to transport itself along the velocity field.

The pressure term $-\frac{1}{\rho}\nabla p$ causes regions with a higher pressure to accelerate the molecules away from that area, since the fluid is incompressible. This element is an important characteristic of the current state of the fluid.

The diffusion term $\nu\nabla^2 u$ represents the force caused by the viscosity of the given fluid. The viscosity is the factor that differentiates thicker fluids like oil and syrup from thinner ones like water and alcohol.

External forces f can be global ones like gravity, acting equally on all cells of the simulation, or local ones like magnetic fields, chemical reactions or temperature differences modelled as local forces, which affects only a part of the domain, optionally in a non-uniform way.

2.2.1 Euler's Simplified Equations

A simplified version of the NS are known as the incompressible Euler equations:

$$\nabla \cdot u = 0 \tag{2.3}$$

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \nabla p + f \tag{2.4}$$

When comparing them to (2.2) and (2.1), it can be seen that the Euler equations are directly deducible by assuming that the density ρ equals 1, and removing the viscosity term. This is an appropriate approximation for gases, since diffusion is negligible in such a domain. Implementations for simulating smoke are demonstrated in Fedkiw *et al.*; Losasso *et al.* [Fedkiw *et al.*, 2001; Losasso *et al.*, 2006].

2.2.2 Boundary Conditions

Since the fluid domain has to be finite, special considerations have to be taken for the borders. Three different concepts are possible:

1. Closed boundaries: The fluid is enclosed by walls which can not be passed.
2. Free-flow boundaries: The fluid is not enclosed in any way, but molecules exiting the domain are discarded.
3. Periodic boundary: Molecules exiting the domain on one side enter the domain on the opposing side. For two dimensions, this can be thought of like wrapping the fluid around a three dimensional torus. This concept does not exist in nature, but it can be helpful for programatically generating tiling textures [Stam, 1999].

In addition to border boundary conditions, borders inside the fluid can represent obstacles. Solutions for these are a major focus of this thesis (see also Chapter 5).

2.2.3 Implementations

The NS equations are inherently dimensionless. In practice, this means that both two dimensional and three dimensional solutions are possible. Since calculating the equations for three dimensions is computationally more expensive, Chen and da Vitoria Lobo [Chen and da Vitoria Lobo, 1995] solved them for two dimensions only, and then used the pressure field p as a height map (higher pressure results on more displacement of the mesh from the ground at a given point). The justification given is that higher pressure at the base of a fluid results in taller columns of the surface above, due to the incompressibility of the fluid. Krüger and Westermann [Krüger and Westermann, 2005] propose using multiple layers of two dimensional fluids and interpolate between them to get a more realistic look.

The difficulty imposed by (2.1) is the part on the left side of the equal sign, $\frac{\partial u}{\partial t}$. This is not easy to solve, because it is non-linear. Various solutions have been proposed, which will be outlined below.

Chen and da Vitoria Lobo [Chen and da Vitoria Lobo, 1995] used a finite-difference solution to create an iterative solver for the NS equations.

Stam [Stam, 1999] emphasizes the importance of stable calculations. When the time steps used for calculating the NS are too large (also limited by other factors like the size of the domain or the viscosity), the simulation "blows up". This effect is non-linear and causes small errors in the simulation to amplify due to numerical reasons. To avoid this, Stam used a method called *method of characteristics* using a semi-Lagrangian solver, which is unconditionally stable. However, the simulation suffers from too much numerical dissipation [Scheidegger *et al.*, 2004], which means that this method is only suitable for situations where the fluid simulation is only used as a visual effect or where comparability to real-life fluids is not important. One application of this method was created by Steve Taylor in his game "Plasma Pong" shown in Figure 2.3.

Liu *et al.* [Liu *et al.*, 2004] implemented the solver introduced by Stam [Stam, 1999] on the GPU in three

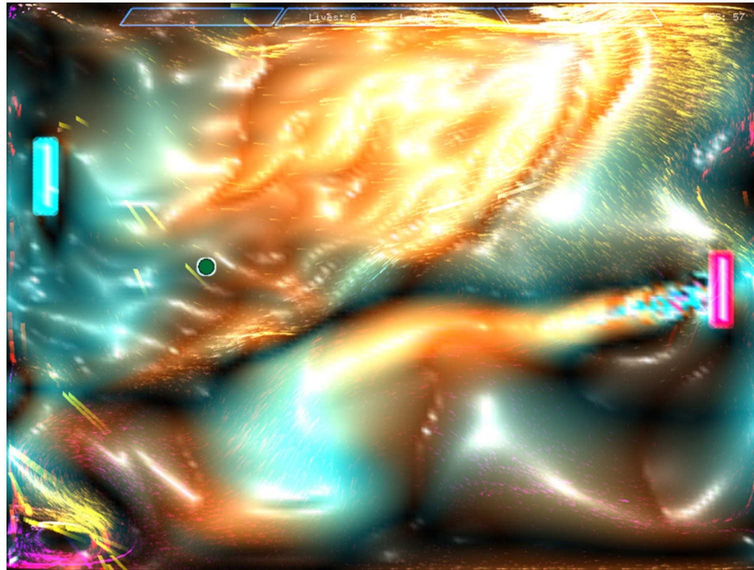


Figure 2.3: An implementation of the Pong game with the combination of a fluid solver based on the work of Stam [Stam, 1999] called “Plasma Pong” by Steve Taylor.

dimensions. It slices the third dimension of the domain into multiple planes, which are then tiled into a two dimensional texture. Scheidegger *et al.* [Scheidegger *et al.*, 2004] used a different method called “Simplified Marker and Cell”, which uses an explicit solver, that means it is subject to certain time step limitations to maintain a stable simulation.

An overview of the methods mentioned here is shown in Figure 2.4.

	CPU-based	GPU-based
Finite-difference NS	[Chen and da Vitoria Lobo, 1995]	
Semi-Langrangian NS	[Stam, 1999]	[Liu *et al.*, 2004]
Lattice-Boltzmann	[Thürey, 2003]	[Wei *et al.*, 2004]
Simplified Marker and Cell		[Scheidegger *et al.*, 2004]
Euler	[Fedkiw *et al.*, 2001; Losasso *et al.*, 2006]	

Figure 2.4: A breakdown of different fluid computation techniques.

Chapter 3

Fluid Simulation Using the Lattice Boltzmann Method

In 1872, the Austrian physicist Ludwig Boltzmann developed the Boltzmann equation, which is a mathematical model to describe the dynamics of an ideal gas at microscopic scale (Note that the NS equations are a simplification of the Lattice Boltzmann equations, which assume that the particles are dense).

If this equation would be applied directly, every single molecule of the gas would have to be stored and simulated (using its position and direction). Calculating these would be unrealistic today, due to the limitation posed by processors and memory. Thus, simplifications were developed.

3.1 The Lattice Gas Cell Automata

The lattice gas cell automata (LGCA) uses a equally-spaced grid ("lattice"), where every cell stores multiple boolean flags, each with its own movement vector e_i, that denote whether a molecule exists at this location moving in this direction. At every time step, every molecule is copied along its movement vector to the cell at that border, this is called the "streaming phase". Whenever two molecules arrive from opposing directions in a single cell, they collide and get redirected using partially non-deterministic boolean operations (for more in-detail information, the reader is referred to Wei *et al.* [Wei *et al.*, 2004]), which is called the "colliding phase".

This method is a direct quantization of the Boltzmann equations in space and time. It is simple, but suffers from statistical noise. This is caused directly by the boolean operations, and could be reduced by averaging in space and time, limiting the resolution [Wei *et al.*, 2004; Thürey, 2003].

3.2 The Lattice Boltzmann Method

A more sophisticated, but essentially similar, model that does not exhibit the noise problem called the *Lattice Boltzmann Method* (LBM) was developed based on the LGCA. The boolean flags are replaced by a distribution function:

A cell in the LBM lattice stores a number of scalars f_i ($i = 0 \dots m$, where m depends on the geometry, as outlined in the next section), counting the number of molecules moving into the direction e_i (the number of molecules existing in a single cell is thus the sum of all f_i belonging to that cell).

The streaming operation is essentially unchanged, and the collision operation only needs a different approach to handle distributions instead of boolean values as input and output, which will be explained later.

3.2.1 Lattice Geometry

The historically important LGCA geometry introduced by Frisch *et al.* [Frisch *et al.*, 1986] uses a two dimensional hexagonal lattice. However, a LBM lattice has to be symmetrical to satisfy the isotropic requirement of fluid properties [Wei *et al.*, 2004], which means that it has to be an equally-spaced grid.

Since the selection of the geometry depends on the application and dimension, a nomenclature has been developed for easy identification. The format is "DnQm", where n is the number of dimensions (usually 2 or 3), and m is the number of distinct lattice velocities. A common two dimensional geometry used is D2Q9, shown in Figure 3.1.

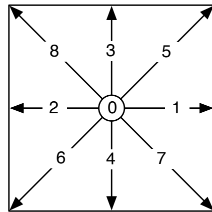


Figure 3.1: The D2Q9 LBM geometry, including a suggested ordering for the index i. The zero-velocity vector is visualized by a small circle in the center.

In three dimensions, three options are widely used [Wei *et al.*, 2004], as can be seen in Figure 3.2. They are:

1. D3Q15: Zero velocity ($i = 0$), faces ($i = 1 \dots 6$), corners ($i = 7 \dots 14$)
2. D3Q19: Zero velocity ($i = 0$), faces ($i = 1 \dots 6$), edges ($i = 7 \dots 18$)
3. D3Q27: Zero velocity ($i = 0$), faces ($i = 1 \dots 6$), edges ($i = 7 \dots 18$), corners ($i = 19 \dots 26$)

In order to explain the effect these velocity vectors have on the streaming phase, the neighbor cells that are affected by that phase are shown in Figure 3.3 (subject to the limitations of visualizing a three dimensional construct in two dimensions).

The number of velocity vectors has a direct impact on the performance of the simulation. D3Q15 is prone to numerical instability and other artifacts visible in fluid visualizations, while D3Q27 requires 27 copy operations per streaming phase, which is expensive. D3Q19 is a good tradeoff between those two extremes, and thus is a widely applied approach [Wei *et al.*, 2004; Li, 2004; Thürey, 2003; Thürey *et al.*, 2006].

Non-Cartesian Grids

Since the lattice is regular, it suffers from a common issue well known from other fields like shadow mapping. Certain points of interest do not have enough adjacent cells for the effects that are exposed by an equivalent real-world test, while other areas are close to the equilibrium distribution, but receive the same calculation time in the algorithm. Filippova and Hänel [Filippova and Hänel, 1998] demonstrate that it is possible to refine the grid, while still retaining the stability properties. However, this enhancement has not yet been demonstrated to lend itself well to a GPU-based implementation.

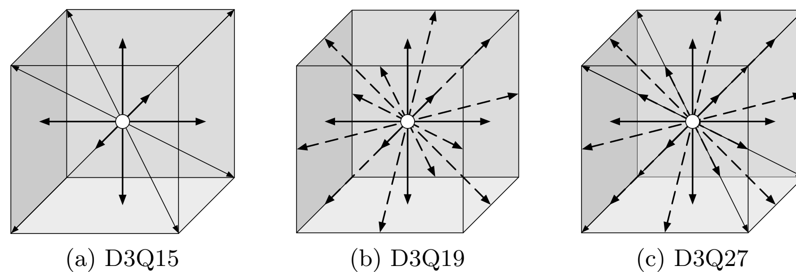
3.2.2 The Macroscopic Properties of a Lattice Cell

The velocity and density values of a cell as known from the NS can be calculated according to Li [Li, 2004] by using the equations

$$\rho = \sum_i f_i \qquad u = \frac{1}{\rho} \sum_i f_i e_i \tag{3.1}$$

After discretizing the Boltzmann equation in both space and time, it can be simplified as follows:

$$f_i(x, t + \Delta t) - f_i(x, t) = \Omega_i \tag{3.2}$$



(a) D3Q15 (b) D3Q19 (c) D3Q27

Figure 3.2: The directional vectors e_i in different LBM geometries.

where Ω_i is a fluid collision operator modelling the collision between fluid molecules in a cell. It is not possible to compute an exact solution for this operation, but in 1992, a simple collision operator called the *Bhatnagar-Gross-Krook approximation* (BGK) was introduced to the LBM [Thürey, 2003]. It uses a single relaxation time approximation to reduce the operator to operations suitable for computers. It is based on the idea that the main effect of the collision operator is to bring the molecule distribution closer to the equilibrium distribution, which is defined as

$$f_i^{eq} = \omega_i \rho \left(1 - \frac{3}{2} u^2 + 3(e_i \cdot u) + \frac{9}{2}(e_i \cdot u)^2 \right) \tag{3.3}$$

where ω_i is a constant that depends on the lattice geometry (see Table 3.1). The collision operator itself is defined as

$$\Omega_i = -\frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(\rho, u)) \tag{3.4}$$

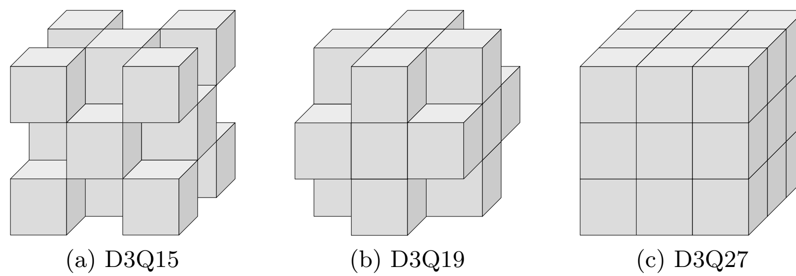
where τ is a constant that represents the viscosity of the fluid, given by $\tau = \frac{1}{2}(1 + 6\nu)$ [Wei *et al.*, 2004].

Combining the streaming operation with equation 3.2 and 3.4 leads to a full description of the method in a single, easy to understand formula:

$$f_i(x + e_i, t + \Delta t) = f_i(x,t) - \frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(\rho, u)) \tag{3.5}$$

The steps required to calculate this equation for the D2Q9 lattice are outlined visually in Figure 3.4.

Unlike most methods based on the NS, the LBM is unconditionally stable, while still demonstrating fluid behavior. The only limitation is that information in the grid cannot travel faster than one cell distance per streaming phase (usually called c_s, speed of sound or "Mach number").



(a) D3Q15 (b) D3Q19 (c) D3Q27

Figure 3.3: Cells affected by the streaming phase of the center cell.

Geometry	i	$\rightarrow\omega_i$	Geometry	i	$\rightarrow\omega_i$
D2Q9	$i = 0$	$\rightarrow\omega_i = \frac{4}{9}$	D3Q15	$i = 0$	$\rightarrow\omega_i = \frac{2}{9}$
	$1 \leq i \leq 4$	$\rightarrow\omega_i = \frac{1}{9}$		$1 \leq i \leq 6$	$\rightarrow\omega_i = \frac{1}{9}$
	$5 \leq i$	$\rightarrow\omega_i = \frac{1}{36}$		$7 \leq i$	$\rightarrow\omega_i = \frac{1}{72}$
D3Q27	$i = 0$	$\rightarrow\omega_i = \frac{8}{27}$	D3Q19	$i = 0$	$\rightarrow\omega_i = \frac{1}{3}$
	$1 \leq i \leq 6$	$\rightarrow\omega_i = \frac{2}{27}$		$1 \leq i \leq 6$	$\rightarrow\omega_i = \frac{1}{18}$
	$7 \leq i \leq 14$	$\rightarrow\omega_i = \frac{1}{216}$		$7 \leq i$	$\rightarrow\omega_i = \frac{1}{36}$
	$15 \leq i$	$\rightarrow\omega_i = \frac{1}{54}$			

Table 3.1: The weight ω_i for various lattice geometries, where i is the index parameter of the directional vector e_i.

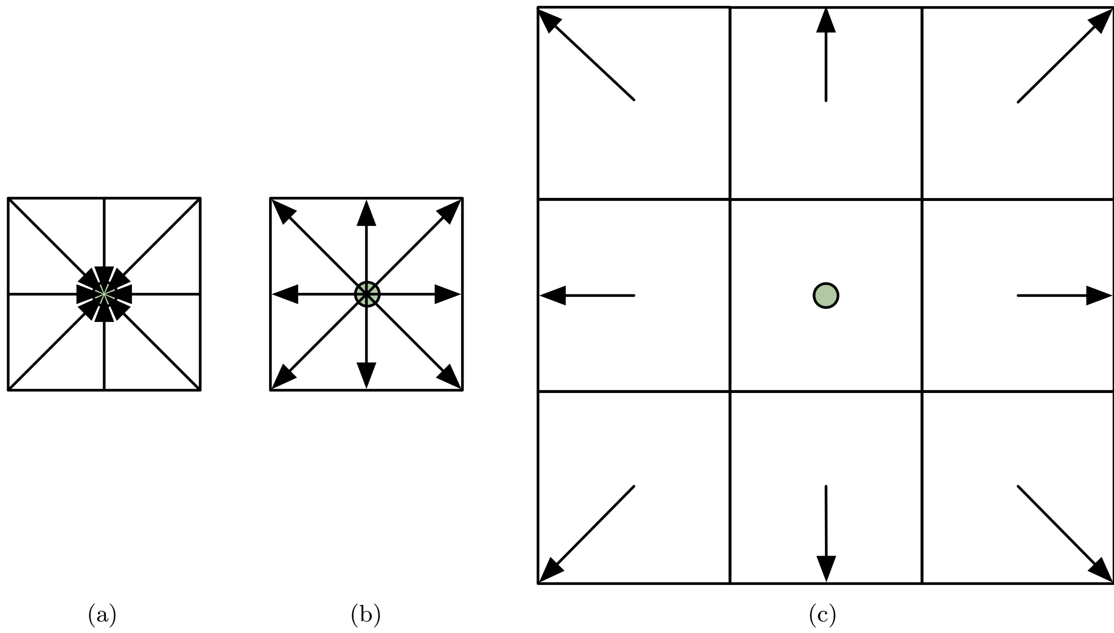


Figure 3.4: The basic steps of the LBM operation in a D2Q9 lattice: (a) Iteration start, (b) after the collision step and (c) after the streaming step.

3.3 Gravity

Gravity is an important concept in physics. Since the goal of this thesis includes combining a traditional rigid body simulation with a fluid simulation and the rigid bodies have to exhibit a behavior as if being subjected to gravity for a pinball machine, the LBM implementation should also exhibit the same gravity to keep a consistent simulation.

In its original form, the LBM does not account for external forces like gravity acting on the fluid. Buick and Greated [Buick and Greated, 2000] outline several methods of varying complexity and compare them using test cases.

The most accurate extends the BGK collision operator by another factor:

$$\Omega_i = -\frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) + \left(1 - \frac{1}{2\tau}\right)\frac{3}{\omega_i}F \cdot e_i \tag{3.6}$$

where F is the force to be applied. The operation $F \cdot e_i$ describes the force to be applied in direction e_i, while the factor $\frac{3}{\omega_i}$ accounts for the fact that the travel distances between cells in the lattice differ depending on the direction. The factor $\left(1 - \frac{1}{2\tau}\right)$ causes slower-moving fluids to be affected to a greater extend by gravity.

3.4 Initial Conditions

The LBM equation is a *recursive* equation. Although such a function is directly implementable on a computer, the initial conditions have to be known in order to start such an algorithm.

Since the fluid simulation tends towards the equilibrium distribution, the rest position can be determined by using f_i^{eq} with any ρ and u, which can serve as the initial conditions. A good value for u would be the zero-vector, while ρ can be determined by using physically-based values from the fluid to be simulated (see Section 3.6).

Note however, that in a system using gravity, this configuration will not result in a rest configuration, since this force causes the rest configuration to have an uneven density distribution. One possibility for working around this problem is to run the simulation until it comes to rest before introducing any other forces and presenting the interactive display to the user. The initial pressure for each cell can also be approximated based on the depth with respect to the gravity direction. In the simulation described in this thesis, this value can not be exactly determined without a full fluid simulation, since it also depends on objects immersed in the fluid.

3.5 Boundary Conditions

The concepts outlined in Section 2.2.2 are directly applicable to the LBM:

1. Closed boundary: On edges where streaming would result in copying the information to a non-existing cell, the information is written to the current or the adjacent cell to the part of the distribution pointing in the opposite direction. This operation is called *bounce-back*.

   The cell used for the operation depends on the surface material properties of the boundary, as outlined in Figure 3.5. Material properties between these two extremes can be simulated by using a linear combination of both methods.

2. Free-flow boundary: The flow streaming to a non-existing cell is discarded. Into the cell's opposite flow direction, the average of the fluid lost due to this operation in the whole grid has to be inserted in order to avoid violating the mass conservation rule.

3. Periodic boundary: The flow is inserted into the same direction on the other side of the fluid. This could be achieved by using a modulo operation for calculating the adjacent cell for the streaming operation.

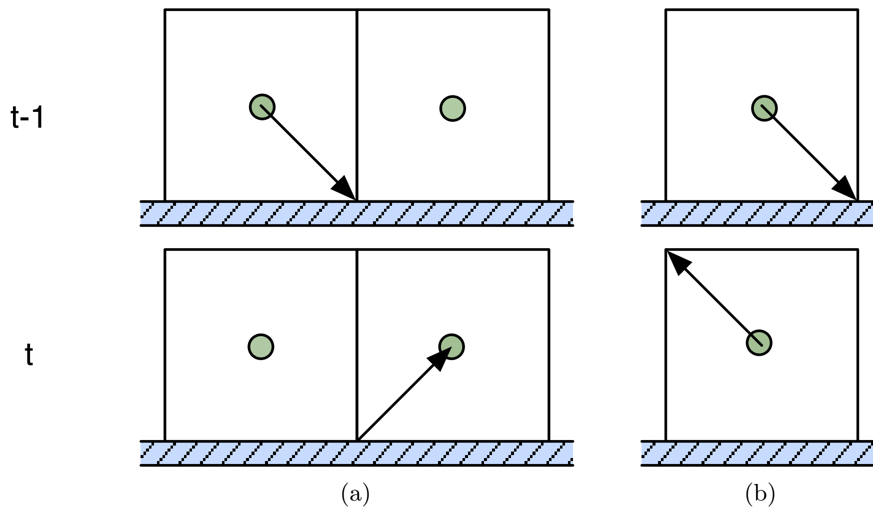


Figure 3.5: The two bounce back methods: (a) demonstrating free-slip and (b) demonstrating no-slip, each before and after the streaming operation.

3.6 Physical Correspondency

Since coupling the fluid simulation with a rigid body simulation is desired (see chapter 5.2.1), a closer look has to be taken at the units and scales used in the Lattice Boltzmann method, based on the Newtonian physics model.

The si-unit for the fluid density is $\frac{kg}{m^3}$, which corresponds to the mass of the fluid per cubic meter. Common values are for example $998.2071\frac{kg}{m^3}$ for water and $1.204\frac{kg}{m^3}$ for air, both at $20°C$. ρ is the fluid density, but with respect to the volume of a grid cell $V = \Delta x \cdot \Delta y \cdot \Delta z$. The mass scale can be defined

arbitrarily, as long as it's used consistently in the whole simulation. Thus, the density of a cell is $\frac{\rho}{V}\frac{kg}{m^3}$ when using the kg unit for mass.

As outlined in equation 3.1, the fluid distributions f_i are just fractions of ρ, and the collision operator Ω_i can be added to a fluid density, so they all share their unit.

The relaxation time τ is measured in seconds and defines the amount of time required until the fluid comes to rest. Usually, a dimensionless relaxation value $\tau^* = \frac{\tau}{\Delta t}$ is used, which allows the simulation to operate at arbitrary time steps[1].

The si-unit for speed is $\frac{m}{s}$. The velocity vector u is given in grid spacing per time step, thus $\begin{bmatrix} \frac{\Delta x}{\Delta t} \\ \frac{\Delta y}{\Delta t} \\ \frac{\Delta z}{\Delta t} \end{bmatrix}$.

Based on the above information and equation 3.1, the si-unit for e_i can be derived:

$$\begin{aligned} u &= \frac{1}{\rho}\sum_i f_i e_i \\ \left[\frac{m}{s}\right] &= \frac{1}{\left[\frac{kg}{m^3}\right]}\sum\left[\frac{kg}{m^3}\right] e_i \\ \left[\frac{m}{s}\right] &= \left[\frac{m^3}{kg}\right]\cdot\left[\frac{kg}{m^3}\right] e_i \\ \left[\frac{m}{s}\right] &= e_i \end{aligned}$$

That is, the si-unit of e_i is meters per second.

[1] To avoid confusion, τ^* is usually expanded in equations in this thesis.

Chapter 4

General Purpose-Programming on Graphics Hardware

Programming a parallel streaming processor requires a vastly different approach to solving problems than in common single-threaded multi-purpose processing. For example, while the heapsort and quicksort algorithms are considered to be very efficient, they do not allow parallel processing. When it comes to implementing sorting on the GPU, the NVidia^{TM} developers recommend the bitonic sort or radix sort. The bitonic sort will be outlined in more detail in order to explain why it is a good choice for GPU processing, demonstrating the general properties of applicable algorithms.

4.1 Bitonic Sort

Batcher [Batcher, 1968] outlines the sorting algorithm originally designed for hardware implementation using logic gates. A basic premise is the *bitonic sequence*.

A sequence is called bitonic if it contains a juxtaposition of a monotonically non-decreasing and a monotonically non-increasing sequence. Given a bitonic sequence $a = \langle a_n \rangle$ with $2n$ numbers, we form two sequences:

$$min(a_1, a_{n+1}), min(a_2, a_{n+2}), \ldots, min(a_n, a_{2n}) \tag{4.1}$$

and

$$max(a_1, a_{n+1}), max(a_2, a_{n+2}), \ldots, max(a_n, a_{2n}) \tag{4.2}$$

Batcher [Batcher, 1968] proves that these two sequences are themselves bitonic, and in addition, no number of 4.1 is greater than any number of 4.2. This operation is usually called "merge". In practice, the operation in 4.1 and 4.2 is done in-place by using a compare-and-exchange operation:

```
for (i ← 0; i < n; i++) {
   if (get(i)>get(i+n)) exchange(i,i+n);
}
```

Where get(i) is a function that returns the ith element of the sequence. Note that n is assumed to be a power of two, in order to be able to divide the sequence recursively into two halves at each step. The functions used by this algorithm can be written as follows (based on a tutorial by Thomas W. Christopher):

```
void sortup(int m, int n) { // from m to m+n
     if (n=1) return;
     sortup(m,n/2);
     sortdown(m+n/2,n/2);
     mergeup(m,n/2);
}
void sortdown(int m, int n) {// from m to m+n
     if (n=1) return;
     sortup(m,n/2);
     sortdown(m+n/2,n/2);
     mergedown(m,n/2);
}
void mergeup(int m, int n) {
     if (n=0) return;
     int i;
     for (i ← 0; i < n; i++) {
          if (get(m+i)>get(m+i+n)) exchange(m+i,m+i+n);
     }
     mergeup(m,n/2);
     mergeup(m+n,n/2);
}
void mergedown(int m, int n) {
     if (n=0) return;
     int i;
     for (i ← 0; i < n; i++) {
          if (get(m+i)<get(m+i+n)) exchange(m+i,m+i+n);
     }
     mergedown(m,n/2);
     mergedown(m+n,n/2);
}
```

A one-item sequence is always bitonic, which can be used as the starting point of the recursion, making sortup(0,N) the initial function call (N being the number of elements in the sequence). An example can illustrate this algorithm. Given the sequence $5, 8, 2, 4, 9, 3, 7, 1$, the following operations take place:

First, the sort functions are recursively executed, which divide the sequence into equal parts. Each of these iterations runs a merge operation afterwards, which does the compare-and-exchange operation explained above. After this loop, the merge function calls itself twice, passing one half of the sequence respectively. The complete sorting operation is visualized in Figure 4.1.

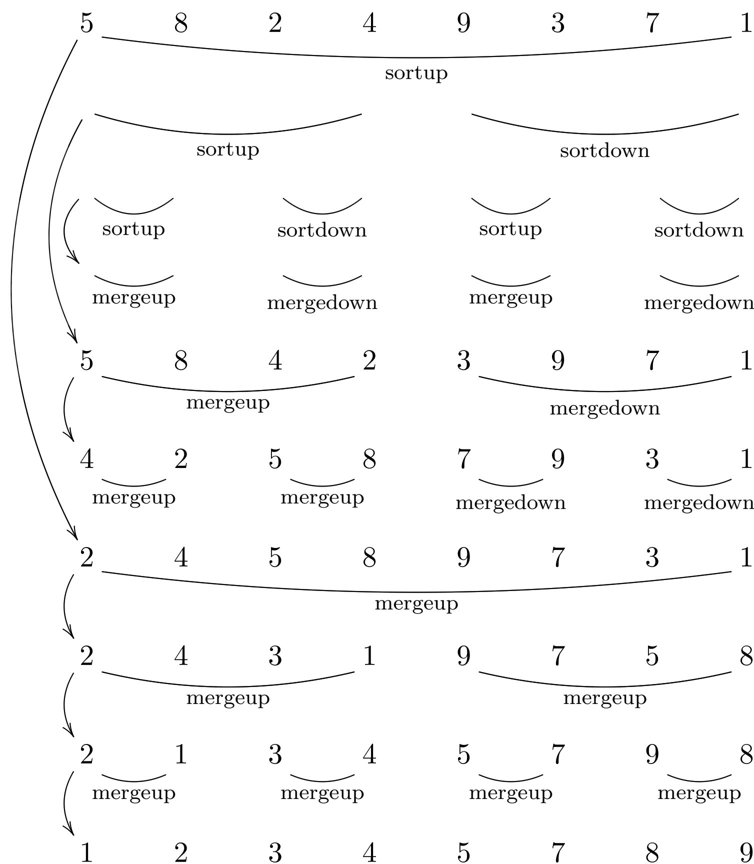


Figure 4.1: The iterative approach to the bitonic sorting algorithm. The arrows on the left indicate the operations the originating line spawns.

As already mentioned, this example implements the algorithm recursively. However, this approach is not suited for execution on current GPUs, since the compiler produces stackless machine code (all functions are inline).

When the bitonic sort should be optimized for a parallel machine, it has to be implemented using an iterative way. The first step is to define two operations, each taking two parameters, ascending exchange and descending exchange (they are part of mergeup and mergedown in the recursive implementation). These operations compare the two values and return them in ascending and descending order, respectively. Then, a simple rule has to be applied (treating the index as a binary value):

```
for k in (every bit in the index ascending from 1)
```

```
    for j in (every bit in the index descending from k−1)
      for i in (every element in a) in parallel
        ixj ← i xor 2^j
        if ixj > i
          if i and 2^k = 0
            ascending exchange i and ixj
          else
            descending exchange i and ixj
```

This algorithm is visualized in Figure 4.2, where the rectangles group the performance-critical innermost loop (where all operations are independent of each other's outcome). This loop is parallelizable, which reduces the run time from $O(n \cdot log^2 n)$ on a single-threaded processor to $O(log^2 n)$ on a multiprocessor running n operations in parallel.

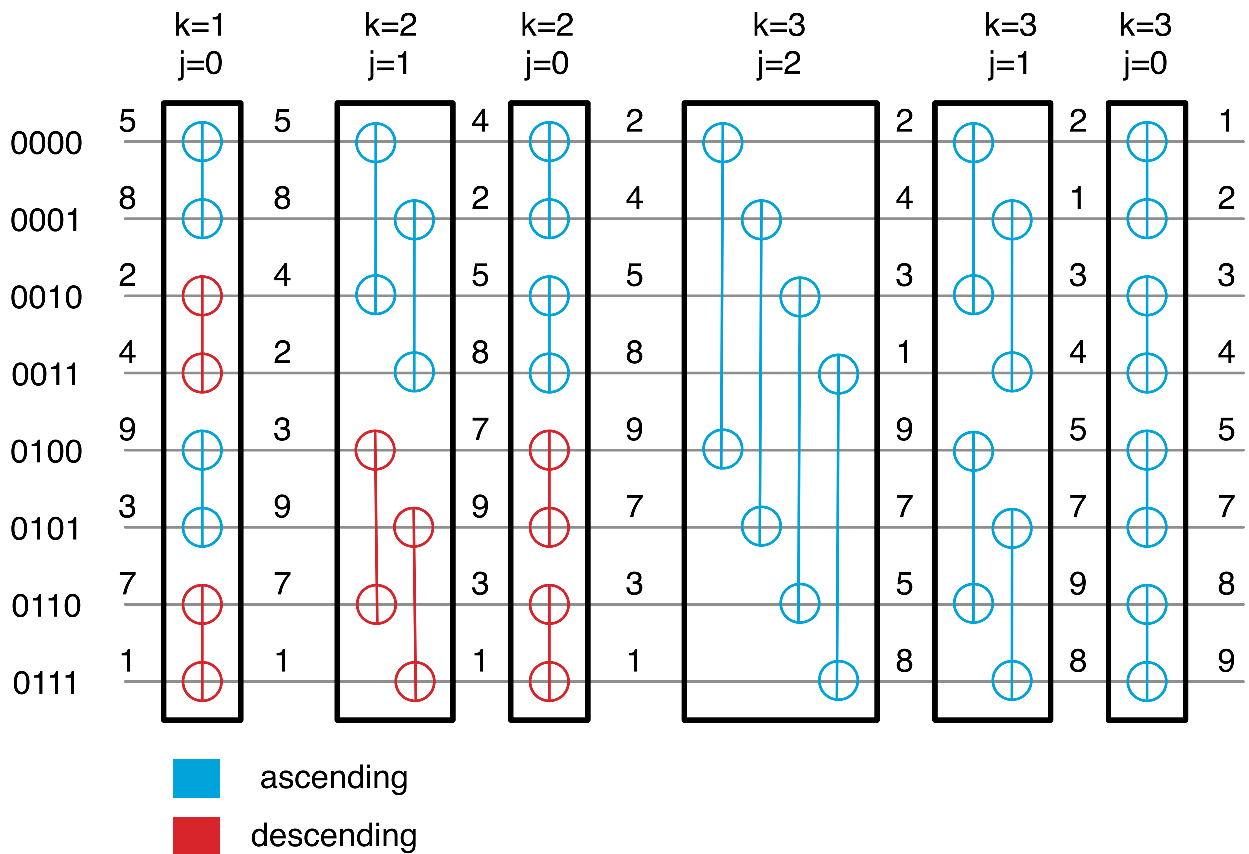


Figure 4.2: A sorting network for eight input values for the parallel version of the bitonic sort. The input values are the same as for the iterative approach mentioned earlier. The black rectangles designate operations that can be done simultaneously. The operations are the same as in Figure 4.1, but no recursive function calls are necessary.

Another important property of this sorting algorithm is that except for the exchange operations, it is independent of the actual values to be sorted, meaning that the number of iterations is known when the number of elements is known. This allows loop unrolling and a well-defined deadline for the whole operation.

4.2 Shader Programming

The classical approach to programming the graphics processors is to adapt the algorithm to be implemented to a graphical representation. An array of values is a texture or a vertex buffer, storing the result of a formula to an entry of an array means drawing the shader's fragment[1] to the texture.

There are three different types of shaders: the fragment shader, the vertex shader and the geometry shader. They all apply to a different stage of the rendering pipeline. For general-purpose programming of the graphics processor (GPGPU), only the fragment shader is relevant. In current graphics card generations, the processors on the graphics card are reassigned dynamically to these shaders, which means that under-utilizing $2/3$ of the programmable parts of the pipeline does not cause any performance degradation.

An important aspect of this approach in OpenGL is the framebuffer objects extension. It allows directly rendering to a texture, without having to go through the screen memory and copying the result to a texture afterwards. It is not part of the official OpenGL specification as of version 2.1, but it is supported by all major graphics card vendors.

In order to draw to the whole texture, a quadrangle has to be generated, triggering the rendering pipeline for every entry in the target framebuffer and running the predefined fragment shader.

Nowadays, shaders use a C-like language. However, this language comes with certain limitations and requires unique performance considerations. For example, there are two ways to apply algorithms: Scatter and Gather.

1. *Scatter* means that for a given input location in the array, the program determines the output location and writes the result there (see Figure 4.3a).
2. *Gather* means that for a given output location in the array, the program collects all information from any source required to determine the value to be written (see Figure 4.3b).

Shaders only support gathering, which means that some algorithms have to be restructured to fit into this paradigm.

Note that these two approaches are directly applicable to the Lattice Boltzmann method to be implemented in this thesis. Equation 3.5 describes a scatter operation. However, it is trivial to restructure the formula to get a gathering operation:

$$f_i(x,t) = f_i^{old}(x - e_i, t - \Delta t) - \frac{\Delta t}{\tau}(f_i^{old}(x - e_i, t - \Delta t) - f_i^{eq}(\rho, u)) \tag{4.3}$$

[1] A "fragment" corresponds to a pixel on a visible framebuffer.

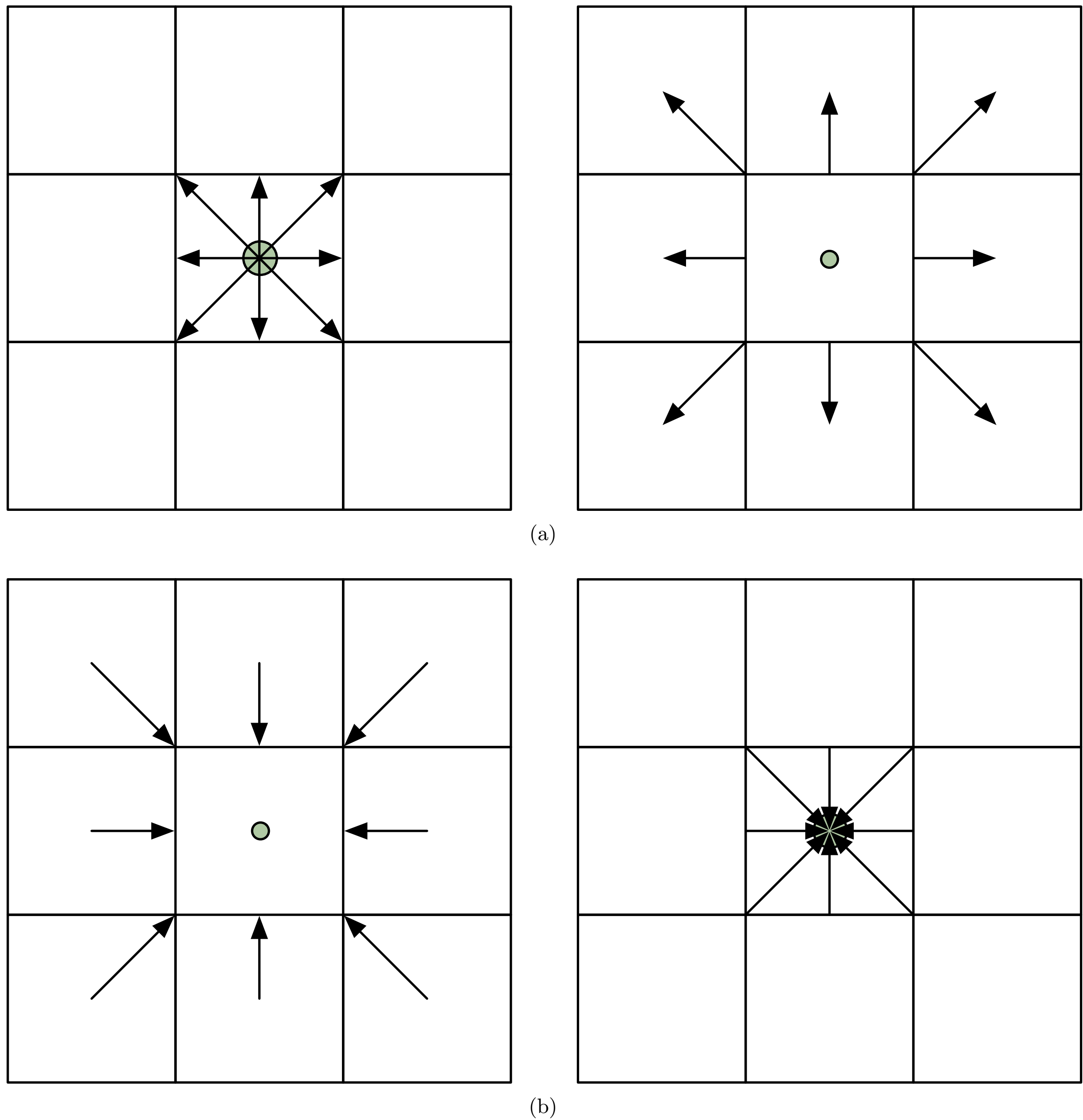


Figure 4.3: The two types of lattice operations applied to LBM D2Q9 (before and after the operation): (a) scatter and (b) gather.

This transformation is applied to D2Q9 in Figure 4.3.

4.3 CUDA

As already mentioned in the introduction, NVidia^{TM} tries to remove the requirement for the knowledge of either OpenGL or Direct3D for the development of GPGPU applications by implementing a new approach, a general purpose C++-derived compiled language, which is run directly on the GPU. This also allows greater control of the processors at the expense of simplicity. It was named "Compute Unified Device Architecture" (CUDA).

Since fluid simulation is a very performance-critical application, the additional speedup that can be provided by CUDA would allow an even more detailed simulation. Additionally, using the simulation mode of the software package allows a direct speed comparison between the GPU-implementation and a CPU-based one using the same codebase.

Further distancing this approach from the graphics sector, NVidia^{TM} also produces graphics cards without the possibility to attach screens under the brand name "Tesla". These are specifically designed for running massively parallel algorithms without having to share the processing power with the display. These cards can be considered as high-performance co-processors. It is conceivable that using these cards would allow implementing a wind tunnel-like offline simulation for aerodynamical tests.

A similar approach was used by IBM^{TM}'s Cell processor. Next to a general-purpose processor core based on the POWER architecture, there are eight "Synergistic Processing Elements" (SPE), which are stream-based single-instruction-multiple-data (SIMD) processors optimized for data processing. All these parts share a single die, which allows a more flexible approach to memory handling than GPU-based solutions (more on this later).

4.3.1 Architecture

Since CUDA approaches the GPGPU topic at a lower level than shading languages, a deeper understanding of the underlying architecture of the NVidia graphics cards is required.

Since the main focus of GPUs is on data processing, its architecture devotes more transistors to arithmetic operations than regular CPUs, sacrificing flow control sophistication (branch prediction for example). Memory latency is hidden by interleaved arithmetical operations instead of data caches. They also employ a data-parallel programming model, meaning that the same operation is applied to multiple input data sets.

The GPU maintains its own memory separate from the host system, but copies between them via a direct memory access controller are possible. Its hardware design is following a layer-approach in all aspects, which are visualized in Figure 4.4 and will now be explained in detail.

- The GPU consists of multiple multiprocessors. Every multiprocessor has a fixed number of SIMD processors, each of which contains a fixed number of registers. The processors are fed by the same

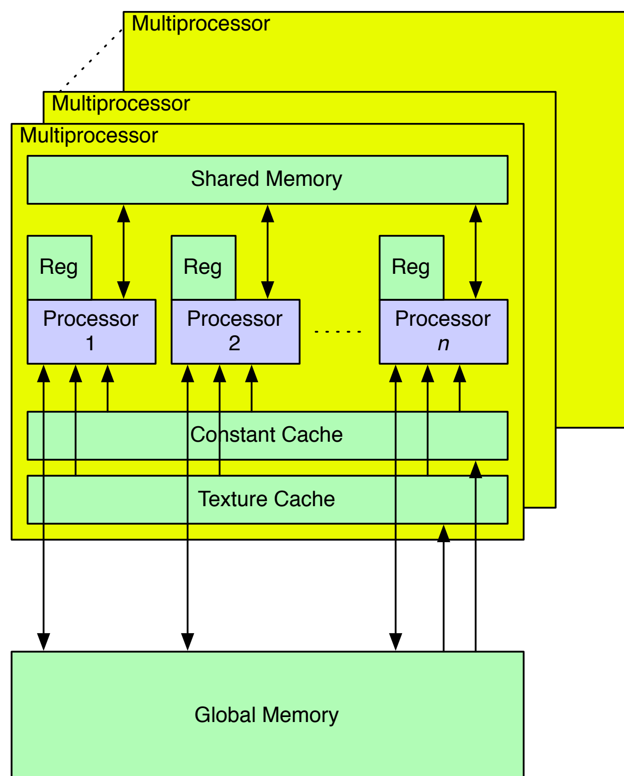


Figure 4.4: The CUDA architecture as a block diagram.

instruction unit. Every multiprocessor also possesses local on-die memory.

- A multiprocessor divides its local memory into three areas:
  1. The shared memory is used for communication between threads in a single block.
  2. The constant cache is a read-only region that has to be initialized by the host before the kernel is executed. Since this cache is located on the multiprocessor die, it can be used by the programmer for optimizing kernels with a high rate of read-only memory accesses.
  3. The texture cache is a read-only region that is similar to the constant cache, but cannot be accessed directly and is optimized to exploiting locality in data access. Its origins lie in the textures used in graphics rendering. Texture memory is accessed through texture units, which have to be set up explicitly.

  In addition to these, the memory on the graphics card can be accessed read/write as global memory. It is larger than any other memory, but is also the slowest, due to the lack of caching. In addition, locking global memory is not possible, which means that multiple kernels writing to the same global memory address results in undefined behavior. Starting with the G92 architecture, atomic integer operations are available that can avert this issue.

- In common data-parallel architectures, including shaders, the SIMD instructions allow using a single operation on an array instead of a single value, e.g. for vector addition in a single instruction.

  However, CUDA uses a different approach: A single processor executes multiple "threads" at the same time, each running the same command, but on different input data. A single batch of these threads running on the same processor is called "warp", while either the first or second half of these is called "half-warp". The collection of warps that have to run on the same multiprocessor is called "thread block". A single thread is executing the "kernel", which is equivalent to a function in C.

  Threads in a single block can synchronize with each other and access the multiprocessor-local memory area mentioned previously. There is a hardware architecture-specific maximum number of threads in a single block, which is specified by the hardware developer. However, CUDA allows scheduling multiple independent blocks running the same kernel, which allows operating on more data sets than the hardware can allow using preemptive multithreading, which is also used in modern operating system kernels for running multiple processes in parallel on a CPU. This technique is encouraged, since during the time a block accesses memory, another block can use the processor for arithmetic operations, thus reaching a higher GPU utilization ratio. The hardware driver is responsible for distributing the blocks to all of the multiprocessors available on the GPU.

  Threads in a block do not share the registers available on a single processor. The compiler determines the amount of registers required for running a specific kernel. This number also limits the number of threads that can run in a single block.

4.3.2 CUDA's Programming Language

Since the CUDA kernels have to be executed on a specialized hardware platform, NVidia^{TM} created a compiler that is adjusted for the architecture's requirements. Since thread blocks have to be managed and uploaded to the GPU by the CPU, a combination of a traditional programming language and API and the GPU-based programming language is required. Unlike in shaders, CUDA allows GPU-based functions to live along with CPU-based ones in the same source file. NVidia^{TM}'s compiler generates a preprocessed file that has to be sent to the regular compiler supplied with the programming environment (this is usually done automatically). The preprocessor replaces the kernel functions with the machine code to be uploaded to the GPU.

For easier integration, CUDA uses a subset of the C++ language with extensions required for a stream-based architecture. It does not allow object-oriented programming, but it's possible to integrate a kernel into an object-oriented program by using functions as middlemen. One important aspect of this is that templates are supported, which allows compile time-optimizations that can enhance performance and cut down on code duplication.

CUDA's headers define some types commonly used in computations, like three and four dimensional vector types (float3, float4), and functions for creating them with the name make_<type>(...). The most important extensions to the C++ programming language itself are:

- Function type qualifiers:
  - __global__ has to be used to declare a function to be a kernel.
  - __device__ is used to declare it to be local to the GPU. It is always implicitly inline.
  - __host__ declares a function to be run on the CPU. This is the default and not required to be specified, unless a function should be compiled for both the CPU and GPU. In this case, both __device__ and __host__ have to be specified.

    One common example for this is the dot operation:

    ```
    __host__ __device__
    float dot(float3 a, float3 b) {
        return a.x*b.x + a.y*b.y + a.z*b.z;
    }
    ```

    This operation can be useful in both host and device functions.

  A useful aspect of CUDA using C++ instead of C is that operator overloading is supported. This means that the new types float3 and float4 can be used in an intuitive way in both host and device functions when defining functions like:

  ```
  __host__ __device__
  float4 operator+(const float4 &a, const float4 &b) {
      return make_float4(a.x+b.x, a.y+b.y, a.z+b.z, a.w+b.w);
  }
  ```

Additionally, templates can be used:

```
template <class T> __host__ __device__
inline void swap(T &a, T &b) {
    T t = a;
    a = b;
    b = t;
}
```

Since all functions are inline in GPU code, implementing these functions allows writing code that is easy to understand without sacrificing runtime performance.

- Variable type qualifiers:
  - __device__ declares a variable to be located on the GPU in global memory. It can only be used from the CPU by using the CUDA API.
  - __constant__ declares a variable to be located in the constant memory space.
  - __shared__ denotes a shared memory block. It is not possible to declare more than one shared memory block in a kernel.
- Invoking a kernel requires more information than supplied with a regular function call, since the API has to know the number of blocks and threads to be started.

  For this, a special syntax was added to the language:

```
Func<<< number of blocks, number of threads per block,
    shared memory size, stream >>>(parameters);
```

For example, a kernel for advecting particles along the fluid velocity field might look like:

```
__global__
void advectParticles_k(particle_t *p, dim3 size,
                       float dt, float count);
```

which would be executed by using

```
advectParticles_k<<<1, particle_count,
  sizeof(particle_t) * particle_count>>>(
                                  particles->array, particles->dim,
                                  dt, particles->count);
```

when shared memory of the size of one particle_t struct per particle is required. Passing 1 as the number of blocks can only be used when the number of particles is a power of two and less or equal to 512 on current GPU generations.

The shared memory size and the stream are optional parameters, both defaulting to 0. The stream parameter allows supplying multiple executions of CUDA API calls like kernel launches and memory copies that have to be run sequentially, but can be run asynchronously from the host application.

The block and thread size can also be two- or three-dimensional vectors, which aids in executing the kernel on data sets of this dimensionality.

The parameters of the kernel are implicitly copied to the GPU before invoking the kernel. They can contain scalars and structs, but not arrays.

- In a kernel, special variables are defined to allow differentiating between all of the kernel invocations running in parallel. Those are:
  - gridDim defines the grid dimensions supplied to the kernel invocation call.
  - blockIdx denotes the index of the current block.
  - blockDim indicates the number of threads in a single block.
  - threadIdx contains the thread index within the block.
- __syncthreads (); can be used in a kernel to make sure that all warps of the current block have reached this point in the kernel. This is required for accessing shared memory in a predetermined fashion.

4.3.3 Optimization Strategies

Optimizing CUDA kernels is a manifold process that requires an understanding of the underlying architecture outlined in the previous section and implementation details not obvious from the CUDA API itself.

The most important optimization technique is determining the balance between block size and the total number of blocks. Since communication between different blocks is not possible, it is also determined by the algorithm implemented in the kernel. The size of the shared memory block limits the number of threads and blocks that can run simultaneously on a single multiprocessor, as does the number of registers required. Additionally, starting only a small number of blocks would prohibit interleaving memory access delays with arithmetic operations by the block scheduler.

The fluid simulation itself requires a comparatively large number of registers, due to the large number of f_i values required for the calculation. This limits the number of threads that can be run in a single block. However, it does not require communication between different threads, and so the block size is not defined by the algorithm itself.

Divergent Threads

Since warps are executed in a SIMD-fashion, the threads in a single warp cannot take different code paths. When a divergent thread is encountered, the execution paths are serialized, which increases the

number of instructions to be executed for this particular warp. The distribution of threads into warps is predefined, and thus can be accounted for when writing the kernel (moving threads that will execute the same code path into the same warp).

This has to be kept in mind when deciding on an algorithm to use for complex obstacles. Since the location of obstacles in the fluid is not known beforehand, having different code paths for solid and fluid cells (or its border) inevitably creates divergent threads.

Memory Access

The GPU supports global memory access for 32-bit, 64-bit and 128-bit. In order to reduce the number of instructions required, as much data as possible should be loaded in a single instruction. For example, the velocity vectors required for fluid simulations require three float values, each 4 bytes. Loading these three values would need three 32-bit load instructions. However, by padding the velocity vector with a single (potentially, but not necessarily unused) float value, a single 128-bit instruction can be used instead.

Additionally, the memory access pattern within a warp should be arranged so that the access can be combined. For this, every thread should access the memory address directly after the one accessed by the previous thread (based on the thread index) by an offset of either 32-bit, 64-bit or 128-bit. The first thread should access an address aligned to multiples of the bit width used for the fetch instruction. If some of the threads do not fetch data from their corresponding memory address, the instruction is nevertheless executed and the result is discarded, avoiding divergent threads.

This is relevant for the D3Q19 geometry, since it requires 19 float values (each 32-bit), that have to be structured in blocks of four (so that the block can be fetched by a single 128-bit fetch instruction), totalling to five blocks, plus four additional floats (i.e. one block) for the macroscopic values.

When implementing the LBM simulation, the efficient memory access pattern can be achieved by collecting values that will be fetched in a single instruction into their own arrays instead of creating large structures that will be fetched in multiple steps. In other words, structuring the memory so that six arrays of 32-bit, 64-bit or 128-bit values are used is more efficient than using a single array containing six 32-bit, 64-bit or 128-bit values.

Since shared memory is on-chip, memory access to it is generally as fast as accessing a register, and a common pattern used in CUDA is to load data from global memory into shared memory, operate on it, and then write the result back to global memory. However, in order to increase bandwidth, this memory region is split into memory banks. When two threads of a halfwarp try to access the same bank, a conflict occurs, which has to be resolved by serializing the memory access.

On the current first-generation CUDA devices, there are 16 memory banks, and a half-warp contains 16 threads. Successive 32-bit words are assigned to successive banks. This is important, since a common access pattern would be that thread n accesses the array at index n. When the size of an array element is so that consecutive elements are on the same bank (for example, 1 byte), a conflict will occur. This can be avoided by letting a thread n of the halfwarp m access the mth element of bank n.

Since the LBM kernel does not require shared memory, this is not relevant for it. However, the fluid/solid-coupling explained later requires calculating a sum using shared memory, which is vulnerable to these kind of access pattern mistakes.

In addition to this, a single 32-bit word per halfwarp-memory access can be broadcast to several threads simultaneously. This avoids a single bank conflict in situations like all threads accessing the same shared memory address. The way in which the broadcast word is selected is driver-dependent and cannot be controlled by the CUDA programmer.

Avoiding Data Copies

Since the bandwidth between the graphics card and the host is constrained, memory copies should be avoided when possible. It is even recommended to run small non-parallelizable computations on the GPU when this avoids copying the data. For computations where the result has to be visualized, like CFD, there is an OpenGL- and Direct3D-interoperability API that allows to use the results from CUDA kernels directly, avoiding the two way-copy.

On the other hand, copying data in GPU-based memory achieves a greater performance and can be used even in performance-critical applications. Due to the limitations of the interoperability API, this is even required for certain operations, like using the result of a CUDA kernel in an OpenGL texture.

Streaming Commands

Since the GPU and CPU run independent of each other, asynchronous programming is required when trying to achieve optimal performance. Stalls occur when one device has to wait for the other to catch up. Since the GPU can not initiate communication with the host system, these stalls can only occur when either the CPU is waiting for the GPU to finish a certain task, or the GPU is waiting for the CPU to get more instructions to execute. (Note that this is also true for shader programming.)

CUDA automatically executes some API commands asynchronously, meaning that they return before the GPU has completed the operation. These are

- Kernel launches,
- Asynchronous memory copies,
- Memory copies from the device to the device and
- memset.

A common pattern for scientific computation is copying the initial data to the graphics card, executing a kernel and copying the result back to the host. However, the kernel is likely to not be completed by the time the final memory copy is executed, resulting in a CPU stall. In order to allow the CPU to run other operations in the meantime, CUDA supplies a stream and events API that allows the programmer to define which operation has to be completed before invoking a specific operation, and to determine

when the commands in a particular stream have completed. This also avoids GPU stalls by letting the host prepare the next batch of commands while the previous batch is still being processed.

4.4 Adapting CFD to the GPU Using CUDA

CFD using the Lattice Boltzmann method is well suited for being adapted to CUDA. Since there are no global operations, every cell can be mapped to exactly one thread, and all threads can be executed independently of each other, increasing the flexibility for the programmer to define the block size.

The input data required for LBM can be stored in arrays residing in global memory. Li *et al.* [Li *et al.*, 2004] propose a certain texture memory layout for shader-based LBM D3Q19 calculations that exploits the possibility to retain data locality even when using bounce back-boundaries. This layout can be used for CUDA for the same reasons and is explained in Table 4.1. This allows fetching all required data by using a single 128-bit fetch instruction.

Array	X	Y	Z	W
u	u_x	u_y	u_z	ρ
f_0	$f(1,0,0)$	$f(-1,\ 0,\ 0)$	$f(0,\ 1,\ 0)$	$f(\ 0,-1,0)$
f_1	$f(1,1,0)$	$f(-1,-1,\ 0)$	$f(1,-1,\ 0)$	$f(-1,\ 1,0)$
f_2	$f(1,0,1)$	$f(-1,\ 0,-1)$	$f(1,\ 0,-1)$	$f(-1,\ 0,1)$
f_3	$f(0,1,1)$	$f(\ 0,-1,-1)$	$f(0,\ 1,-1)$	$f(\ 0,-1,1)$
f_4	$f(0,0,1)$	$f(\ 0,\ 0,-1)$	$f(0,\ 0,\ 0)$	unused

Table 4.1: Distributing the D3Q19 variables in a way to collect values that are required at the same time in the same vector. Based on the work of Li *et al.* [Li *et al.*, 2004].

Since the streaming operation requires reading in adjacent cell values and global synchronization is not possible, the same arrays can not be used for both reading and writing. Thus, the commonly-used flip flop technique is applied, where all arrays are created twice, and on every frame, the input and output arrays are exchanged. This doubles the memory required for the LBM values, but for real-time simulation, the execution speed is a greater limiting factor to the fluid grid size than the memory available on the current GPUs.

More specifically, the memory requirements are as follows: The 24 values listed in Table 4.1 each are of float type, which requires 4 bytes of memory. This means that a single LBM cell requires $24 * 4$, or 96 bytes. A typical simulation with a dimension of $128 \times 128 \times 128$ contains 2097152 cells, which totals to 201326592 bytes. Since all this memory space is required twice for flip flopping, the total amount of memory required for storing the simulation values alone is 402653184 bytes, or 384 megabytes. This is about half the amount of memory available on the graphics cards currently used for GPGPU and is bound to become an even smaller fraction in the near future.

Li *et al.* [Li *et al.*, 2004] use separate shaders for the stream and collide phases. This would be counter-productive for CUDA-based implementations, since the kernel instruction count and the number of read-/write operations are not limited, but they should be avoided when possible.

Both scatter and gather approaches are possible on CUDA and the LBM. A scatter operation was chosen due to a simpler kernel implementation which is also closer to the mathematical representations used in the literature. Since visualization requires the macroscopic fluid information, which is not attainable when scattering without a global lock, this information is calculated in a separate kernel invocation. According to the NVidia^{TM} documentation, this is a cheap operation and should not impose a great burden on the runtime performance.

4.4.1 Visualizing the Flow

Two ways of visualizing the fluid simulation were implemented: Advecting particles and mapping the velocities to a texture, which then is used for drawing an OpenGL quad. Both of these require the OpenGL interoperability feature of the CUDA API, which allows accessing a vertex buffer object or pixel buffer object from a kernel.

For the velocity texture, the resulting pixel buffer object has to be copied to a texture, since accessing an OpenGL texture from CUDA is not possible at this point, and using a buffer object as a texture is not supported either.

For the particles, a vertex buffer object can be used directly as the input for glDrawArrays specifying GL_POINTS as the drawing mode. Since OpenGL allows mapping textures on points, a particle system-based gas visualization can be rendered using the fixed-function pipeline without having to copy the particle positions. The particles' positions were calculated using Euler integration.

Geometry Shaders

Using geometry shaders available on CUDA-enabled graphics cards as an OpenGL extension, arbitrary glyphs (like spheres) can be used for rendering the points generated by the particle visualization, too. Since point particles cannot visualize direction in a still image, and velocity textures can only visualize a single slice of the domain, a particle glyph that can represent direction can be used to visualize the flow velocity in three dimensions. Using a geometry shader, displaying a field with pyramids used as arrows like in Figure 4.5 can be achieved.

A geometry shader uses a primitive as its input (a point in this case) and emits any number of primitives (usually triangles), which are then sent to the rendering pipeline.

Using CUDA, the movement direction of a particle can be stored in the texture coordinates of a point in the vertex buffer object. This information can be used in the geometry shader to rotate the glyph to point to the direction the fluid is moving (note that this code does not include the lighting calculation):

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main(void)
{
```

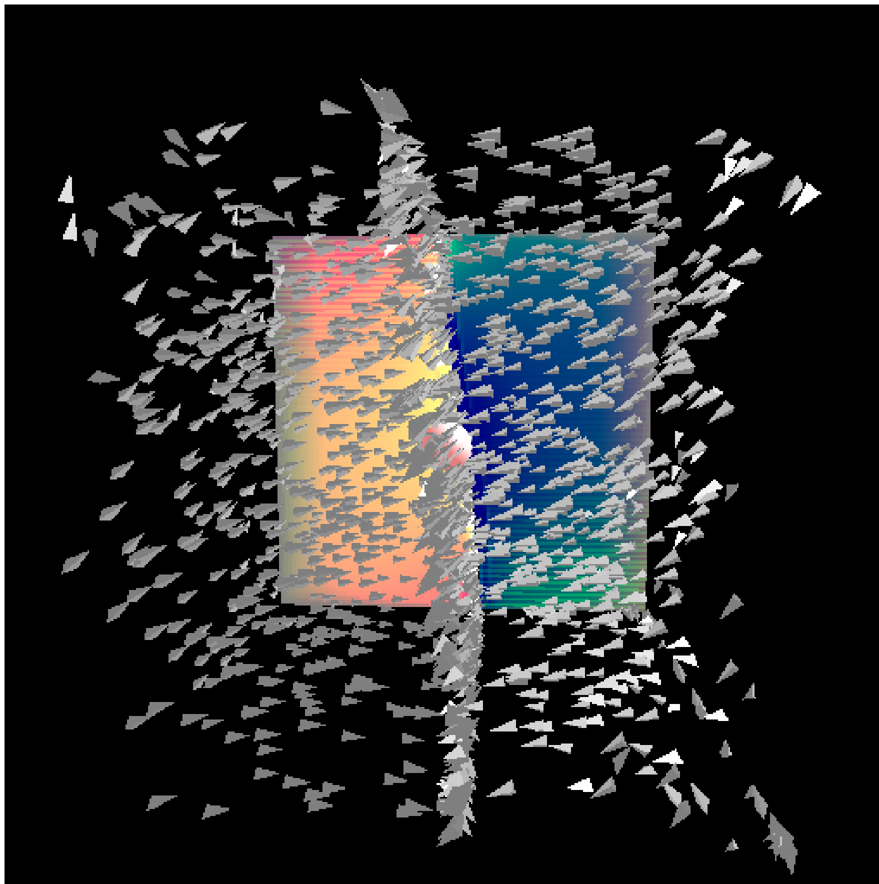


Figure 4.5: Visualizing the fluid domain's velocity using pyramid glyphs representing arrows. Additionally, the velocity texture is visible at the back, showing the center slice's velocity of the fluid domain in red/green/blue color coding for X/Y/Z.

```
const float scale = 0.02;
vec4 pos = gl_PositionIn[0];
vec4 direction = gl_TexCoordIn[0][0];

// normalize
direction.xyz = direction.xyz / length(direction.xyz);

// calculate the rotational matrix required for pointing the glyph
// into the stream direction
mat4 mvp;

vec3 p = vec3(-direction.z,0.0,-direction.x);
float plen2 = dot(p,p);
if(plen2 > 0.00000001) {
    p /= sqrt(plen2); // normalize

    float cosphi = direction.y;
    float oneminuscosphi = 1.0 - cosphi;
    float sinphi = sqrt(1.0-cosphi*cosphi);

    // col row
    rotmat[0][0] = p.x*p.x + p.z*p.z*cosphi;
    rotmat[0][1] = p.z*sinphi;
    rotmat[0][2] = p.x*p.z*oneminuscosphi;
    rotmat[0][3] = 0.0;

    rotmat[1][0] = -p.z*sinphi;
    rotmat[1][1] = (p.x*p.x + p.z*p.z)*cosphi;
    rotmat[1][2] = -p.x*sinphi;
    rotmat[1][3] = 0.0;

    rotmat[2][0] = p.x*p.z*oneminuscosphi;
    rotmat[2][1] = -p.x*sinphi;
    rotmat[2][2] = p.z*p.z + p.x*p.x*cosphi;
    rotmat[2][3] = 0.0;

    rotmat[3][3] = 1.0;
} else
    rotmat = mat4(1.0);

rotmat[3][0] = pos.x;
rotmat[3][1] = pos.y;
rotmat[3][2] = pos.z;
```

```
    mvp = gl_ModelViewProjectionMatrix * rotmat;

    gl_Position = mvp * vec4(-scale,-scale,-scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(scale,-scale,-scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(0.0,scale*3.0,0.0,1.0);
    EmitVertex();

    EndPrimitive();

    gl_Position = mvp * vec4(scale,-scale,-scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(0.0,scale*3.0,0.0,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(0.0,-scale,scale,1.0);
    EmitVertex();

    EndPrimitive();

    gl_Position = mvp * vec4(0.0,scale*3.0,0.0,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(0.0,-scale,scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(-scale,-scale,-scale,1.0);
    EmitVertex();

    EndPrimitive();

    gl_Position = mvp * vec4(0.0,-scale,scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(-scale,-scale,-scale,1.0);
    EmitVertex();
    gl_Position = mvp * vec4(scale,-scale,-scale,1.0);
    EmitVertex();

    EndPrimitive();
}
```

Chapter 5

Complex Obstacles in Fluid Simulations

Since the LBM handles physical interactions at a local level, it can be enhanced to support different fluids (like water and air, or water and oil) and solid interactions with complex boundaries with minimal change. This has been documented by Li; Wei *et al.*; Thürey *et al.* [Li, 2004; Wei *et al.*, 2004; Thürey *et al.*, 2006].

Three different types are possible:

1. Fluid-solid: The fluid affects the solid, but the solids are treated like having no mass and no volume. One application for this are tin cans floating in water.
2. Solid-fluid: The solid affects the fluid. For example, this can be used for simulating water in a non-changing environment. Wei *et al.* [Wei *et al.*, 2004] demonstrate this interaction type in combination with LBM.
3. Two-way interaction: Both are affected by the other. A LBM-based implementation is described in Thürey *et al.* [Thürey *et al.*, 2006]. This is the most reality-like simulation, but can be quite challenging due to the combination of two different kinds of physics (fluid and rigid). GPU-based implementations face an additional challenge here, since the rigid body simulation has also to be implemented on the GPU for optimal performance (NVidia^{TM} has done some research in this area).

Carlson *et al.* [Carlson *et al.*, 2004] propose treating solids like fluids with a high viscosity. However, this approach has certain limitations, namely it doesn't support thin sheets (like paper), self-collisions or rolling animations [Losasso *et al.*, 2006].

5.1 Voxelization on the GPU

In order to integrate a solid object into a fluid grid, the cells that are inside the solid have to be known. When the object is given as volume information, this can be done by resampling to the fluid domain

by applying scale, rotate and translate operations. However, current modelling applications for three dimensional models only generate the surface information due to efficiency reasons.

Since the voxelization result is required to be stored on the GPU, a GPU-based method would be preferred. CPU-based methods like the ones explained by Kaufman and Shimony [Kaufman and Shimony, 1986] do not transform well to GPU-based implementations.

The method currently recommended by NVidia^{TM} is based on the well-known shadow volumes technique. A stencil buffer on the GPU storing signed integer values is initialized to 0, and a orthographic projection with the far plane set to infinity and the near plane set to the slice to be voxelized is set. Then, the whole scene is rendered twice: First, only rendering back-facing trangles. Instead of drawing the triangles, the stencil buffer value is increased by 1 on those pixel locations instead. Then, the front-facing triangles are rendered, decreasing the pixel locations by 1. After these steps, the positive stencil buffer entries denote the location of a voxel in that slice, as demonstrated in Figure 5.1. Then, a rectangle (GL_QUAD) filling the whole viewport has to be drawn with enabled stencil buffer test to get the final voxelization in the color buffer. This operation has to be repeated for every slice, building a three dimensional array of voxels.

Note that using blending operations, a float color buffer can be used instead of the stencil buffer, this removes the need for drawing the rectangle. Color clamping has also to be disabled.

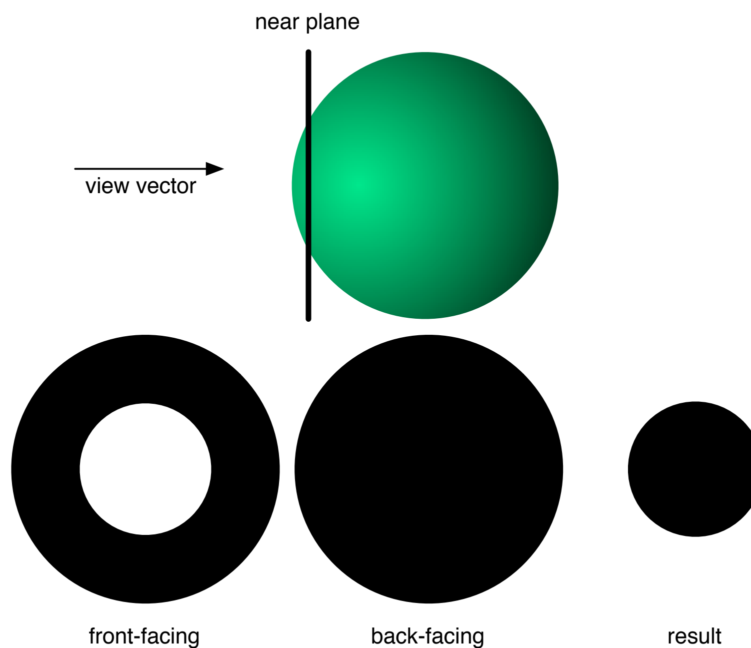


Figure 5.1: A demonstration of the voxelization technique on a sphere.

For easier access, only one texture can be used for the whole voxelization process, where the target viewport is selected using glViewport. The result of applying this technique on a whole scene is shown in Figure 5.2, pseudocode is shown in Listing 5.1. Rendering the whole model multiple times can be

optimized by storing all render operations into an OpenGL display list.

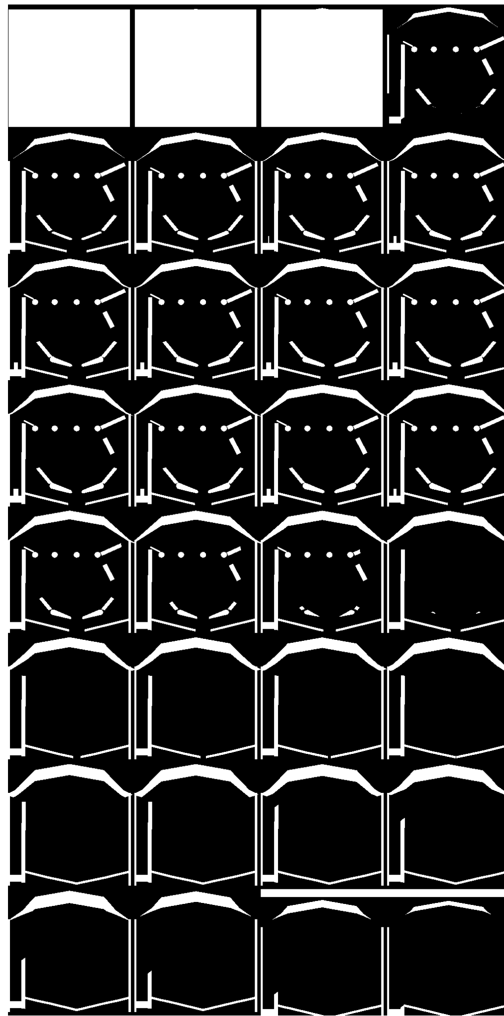


Figure 5.2: The result of applying the voxelization technique for all slices on a pinball geometry.

One major advantage of this technique is that it does not require any high-level knowledge about the geometry to be rendered, since the regular graphics card API calls for rendering any model can be used. This allows using a 3rd party model loader and renderer without modifications. In order to improve the speed, textures and lighting should be disabled for the voxelization process, since they do not have any effect on the end result.

In order to reduce the number of rendering passes required, the well-known technique called "depth-peeling" can be applied, which uses the depth buffer as an additional source of information. When using deforming bodies (such as models using bone-animation), per-vertex velocity has to be determined and rendered (written) to a separate buffer by a geometry and a vertex shader. However, since in this work only rigid bodies are used, this is outside of the scope of this thesis.

Listing 5.1: Voxelization using OpenGL

```
void voxelize() {
    glClampColorARB(GL_CLAMP_FRAGMENT_COLOR_ARB, GL_FALSE);
    clear color buffer;
    load model transformation into OpenGL modelview matrix;
    M ← (fluid domain transformation)^-1;
    glMatrixMode(GL_PROJECTION);
    for(z in every z-slice) {
        glViewport(0, size_y * z, size_x, size_y);
        glLoadIdentity();
        glOrtho(-1.0,1.0,-1.0,1.0, -2.0 * (z - 1.0), 1000.0);
        multiply M into projection matrix;

        glBlendEquation(GL_FUNC_ADD);
        glFrontFace(GL_CW);
        render object;

        glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
        glFrontFace(GL_CCW);
        render object;
    }
    copy framebuffer into vertex buffer object using glReadPixels;
}
```

5.2 Integrating a Physics Engine into a Fluid Simulation

The GPU can be used for rigid body physics calculations. However, further work is required for robust implementations (work on the commercial GPU-based physics engine Havok FX was canceled), and so in the meantime, collaboration between a CPU-based physics engine and a GPU-based fluid engine is desired.

Any physics engine that allows to get the rigid body's current velocity in a point of its surface and then applying force on that point can be used for the integration. However, the Bullet Physics library[1] was chosen, due to it being licensed under the permissive zlib license and having the most active research community.

5.2.1 Rigid Body Simulation using the Bullet Physics Library

Bullet Physics by Coumans [Coumans, 2008] is a commercial-grade open source physics library written in C++ and was used for development of both PC- and console-based games. It is also used for the physics support of the three dimensional modeller Blender for its integrated game engine. Figure 5.3 demonstrates its rigid body capabilities.

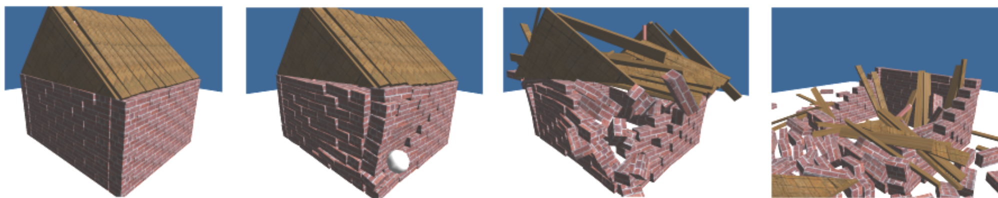


Figure 5.3: Demonstration of the Bullet Physics library's capabilities collapsing a house.

Bullet Physics supports many different collision shapes for rigid bodies (which are represented by the btRigidBody class), like spheres, boxes, cylinders, cones and triangle meshes. It automatically deactivates non-moving bodies to improve processing speed, and also supports vehicle simulations and hinge constraints.

In order to integrate with a fluid simulation, the following three methods of btRigidBody can be used:

```
btVector3 getVelocityInLocalPoint (const btVector3 &rel_pos);
void applyCentralImpulse (const btVector3 &impulse);
void applyTorqueImpulse (const btVector3 &torque);
```

[1] http://bulletphysics.com

5.3 Solid-Fluid Coupling

Of the three basic boundary conditions outlined in Section 2.2.2, only closed boundaries are applicable for objects immersed in the fluid. For this technique, instead of determining boundary cells by checking the cell location, a separate boolean flag has to be stored along with the cell's other information that determines whether any object intersects this cell. When streaming to a cell where this flag is set, the bounceback rule has to be applied instead.

This technique is limited in two ways: First, moving boundaries do not accelerate the fluid, and second, the accuracy is limited to grid spacing. This poses an additional issue, since it creates a “staircase-effect”, which shows non-realistic characteristics (Figure 5.4).

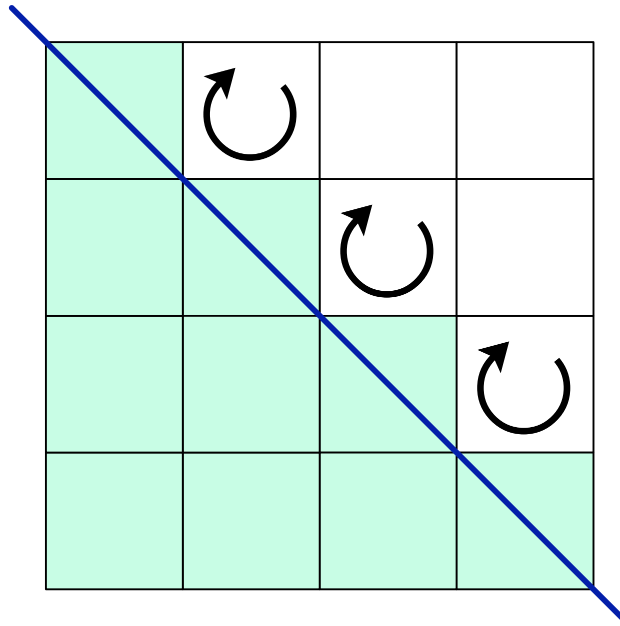


Figure 5.4: When using only the whole-cell boundaries, the border cells show a non-realistic bounce-back behavior due to the “staircase-effect”. The blue line represents the solid boundary.

Thus, Mei *et al.* [Mei *et al.*, 2000] developed a refinement that fixes both problems.

5.3.1 Mei et al.'s Extrapolation Method

A boundary immersed in the fluid is located between two nodes (see Figure 5.5), one of them fluid, the other solid. The location of the boundary has to be determined by

$$\Delta = \frac{|x_f - x_w|}{|x_f - x_b|} \tag{5.1}$$

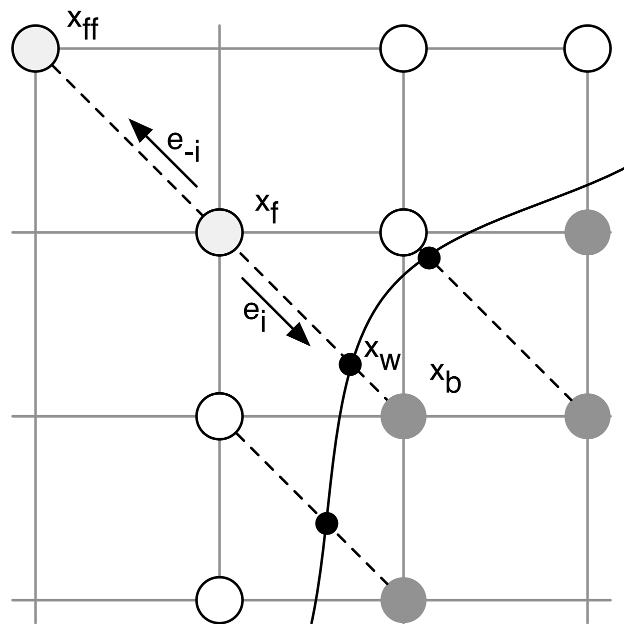


Figure 5.5: Definition of the nomenclature used by Mei *et al.* [Mei *et al.*, 2000]. Grey cells are boundary nodes, white cells are fluid nodes. e_i is the direction vector towards the boundary, x_b is the boundary node, x_w is the virtual wall node, x_f is a fluid node bordering to a boundary node and x_{ff} is the fluid node on the opposite direction of the boundary node, as seen from the bordering fluid node.

In order to apply the LBM to the cell x_f, the fluid particles streamed from x_b have to be determined. Since this cell is solid, the value can be obtained by using

$$f_{-i}(x_b, t) = (1 - \chi) f_i(x_f, t) + \chi f_i^{eq}(x_b, t) + 2\omega_i \rho 3(e_{-i} \cdot u_w) \tag{5.2}$$

with

$$f_i^{eq}(x_b, t) = \omega_i \rho(x_f, t) \left(1 + 3(e_i \cdot u_{bf}) + \frac{9}{2}(e_i \cdot u_f)^2 - \frac{3}{2} u_f^2\right) \tag{5.3}$$

and

$$u_{bf} = \begin{cases} \frac{(\Delta-1)u_f + u_w}{\Delta} \\ u_{ff} = u_f(x_f + e_{-i}, t) \end{cases}, \chi = \begin{cases} \frac{2\Delta-1}{\tau} & \text{for } \Delta \geq 1/2 \\ \frac{2\Delta-1}{\tau-2} & \text{for } \Delta < 1/2 \end{cases} \tag{5.4}$$

The basic idea is to extrapolate virtual fluid particles at x_b based on the fluid surrounding the boundary. The rationale behind the distinction at $\Delta = \frac{1}{2}$ is that when the boundary is too close to the fluid cell, it alone is no longer a good approximation, and a second order interpolation using the next cell in addition is used instead. Note that this method works equally in two dimensions and three dimensions.

While this method is a fast and good approximation, it requires knowledge about the exact location of the solid boundary, which is not possible with the technique shown in Section 5.1 and has to be determined by a technique like ray casting. This works well for primitive objects, but is too slow for real-time use on generic triangular meshes.

An additional downside for implementing this method using CUDA is that boundary nodes would cause a divergent thread, since it requires branching. This would have a noticeable impact on performance.

5.3.2 Noble's Method for Partially Saturated Cells

Noble and Torczynski [Noble and Torczynski, 1998] use a different approach, which is still based on bounce back. Every cell of the fluid lattice contains a certain fraction ε of solid material to fluid material, which is zero for fully fluid and one for fully solid cells, but can be anything inbetween, as demonstrated in Figure 5.6.

A function B is defined as[2]

$$B(\varepsilon, \tau) = \frac{\varepsilon(\tau - \frac{1}{2})}{(1 - \varepsilon) + (\tau - \frac{1}{2})} \tag{5.5}$$

This is a weighting function in the range between 0 and 1, as visualized in Figure 5.7.

[2]Note that the equation in [Noble and Torczynski, 1998] contains a minor error that was later corrected by Strack and Cook [Strack and Cook, 2007].

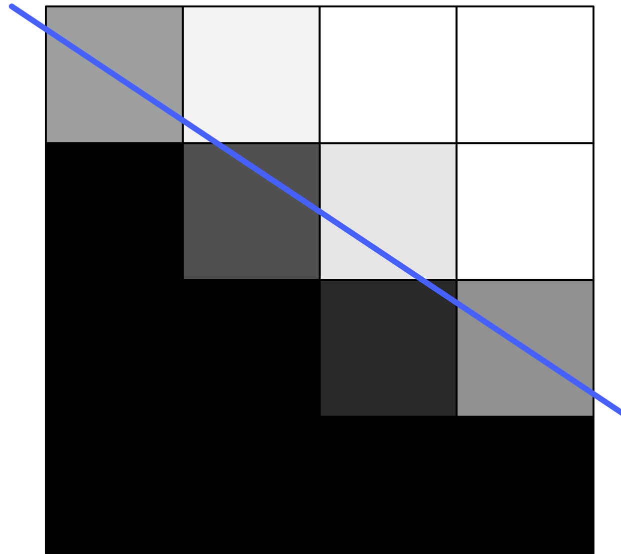


Figure 5.6: Demonstrating the solid fraction ε by using grayscale values. Black means $\varepsilon = 1$ (completely solid), while white means $\varepsilon = 0$ (completely fluid). The blue line represents the solid boundary.

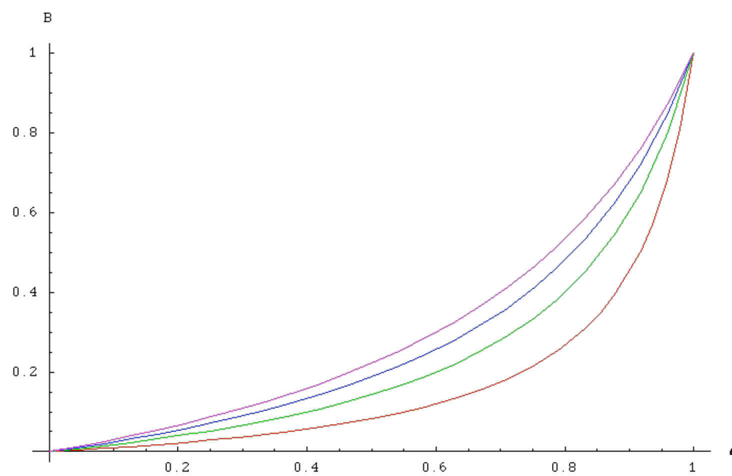


Figure 5.7: The function B with $\tau = 0.6, 0.7, 0.8, 0.9$ in red, green, blue and magenta respectively.

The Equation 3.2 is modified to include a second collision operator:

$$f_i^{new}(x,t) - f_i(x,t) = (1-B)\Omega_i + B\Omega_i^s \tag{5.6}$$

Two different options for Ω_i^s are offered. However, Strack and Cook [Strack and Cook, 2007] demonstrate that Holdych [Holdych, 2003] offers a more stable equation:

$$\Omega_i^s = f_{-i}(x,t) - f_{-i}^{eq}(\rho, u_s) + f_i^{eq}(\rho, u_s) - f_i(x,t) \tag{5.7}$$

In addition to being more stable, since the equilibrium equation contains the same parameters except for the inverted e_i, considering $e_{-i} = -e_i$, this equation can be simplified:

$$\begin{aligned}
\Omega_i^s &= f_{-i}(x,t) - \omega_i\rho\left(1 - \frac{3}{2}u_s^2 + 3(e_{-i}\cdot u_s) + \frac{9}{2}(e_{-i}\cdot u_s)^2\right) + \\
&\quad \omega_i\rho\left(1 - \frac{3}{2}u_s^2 + 3(e_i\cdot u_s) + \frac{9}{2}(e_i\cdot u_s)^2\right) - f_i(x,t) \\
\Omega_i^s &= f_{-i}(x,t) - f_i(x,t) + \\
&\quad \omega_i\rho\left(-1 + \frac{3}{2}u_s^2 - 3(-e_i\cdot u_s) - \frac{9}{2}(-e_i\cdot u_s)^2 + 1 - \frac{3}{2}u_s^2 + 3(e_i\cdot u_s) + \frac{9}{2}(e_i\cdot u_s)^2\right) \\
\Omega_i^s &= f_{-i}(x,t) - f_i(x,t) + 6\omega_i\rho(e_i\cdot u_s)
\end{aligned}$$

This simplification significantly reduces the amount of instructions required. When inserting the collision operators into the equation, the LBM operation now reads:

$$f_i^{new}(x+e_i, t+\Delta t) = f_i(x,t) - (1-B)\frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(\rho,u)) + B(f_{-i}(x,t) - f_i(x,t) + 6\omega_i\rho(e_i\cdot u_s)) \tag{5.8}$$

Note that the method presented by Noble and Torczynski [Noble and Torczynski, 1998] has three significant advantages over the method developed by Mei *et al.* [Mei *et al.*, 2000] for CUDA-based implementations:

1. Since ε is just a multiplication parameter, no branches are required, which avoids divergent threads for any type of obstacle.
2. No ray casting is required for determining the obstacle border. Using the method presented in 5.1, fractional ε can be derived by increasing the voxelization grid's resolution with respect to the fluid grid resolution, and then counting the number of solid voxelization cells in a fluid cell.
3. Due to the limited resolution of the fluid grid, porous media like sand cannot be represented with solid obstacle particles. By multiplying the results of the voxelization process by some factor between 0 and 1, a solid obstacle can be made arbitrarily porous.

A disadvantage of this approach is that solid cells are still handled as fluid cells. This means that the collision and streaming steps are still applied, even when the collision result is multiplied by 0 and the streaming step just bounces the result between two neighboring nodes. Analytically this does not cause problems, but incoming fluid molecules that cannot escape the solid cause the density in border cells to rise. When the float parameter overflows, the GPU treats the value as infinity, which is not rectified by multiplying by 0. Additionally, this can cause problems when a solid moves away from a fluid cell holding high density values. By capping the density value, this can be avoided.

Holdych [Holdych, 2003] proposes a solution to this problem for every cell whose $\varepsilon > 0.95$ that applies the assumption that the cell density difference between two neighboring cells is negligible.

A fluid fraction $\overline{\varepsilon}$ is defined as

$$\overline{\varepsilon} = \sum_i (1 - \varepsilon_i) \tag{5.9}$$

where ε_i is the solid fraction ε of the neighboring cell in direction e_i.

The fluid density for these cells is then calculated depending on $\overline{\varepsilon}$:

$$\rho = \begin{cases} \frac{1}{\overline{\varepsilon}} \sum_i (1 - \varepsilon_i)\rho_i & \text{if } \overline{\varepsilon} > 0.01 \\ 0 & \text{otherwise} \end{cases}$$

where ρ_i is the density value of the cell in direction e_i.

5.4 Fluid-Solid Coupling

In rigid physics simulations, forces applied to rigid objects are stored as only two values with respect to the center of mass: impulse and torque. Since an external physics engine is used, only these values have to be determined by the fluid simulation to be able to let the fluid domain apply any force to the rigid object.

The values calculated by the boundary conditions outlined by Noble and Torczynski [Noble and Torczynski, 1998] can be used to determine these two values, using the equation described by Strack and Cook [Strack and Cook, 2007]:

$$F = \frac{\Delta x \Delta y \Delta z}{\Delta t} \sum_n B_n \sum_i \Omega_i^s e_i \tag{5.10}$$

for the impulse and

$$T = \frac{\Delta x \Delta y \Delta z}{\Delta t} \sum_n (x_n - x_s) \times \left(B_n \sum_i \Omega_i^s e_i \right) \tag{5.11}$$

for the torque, where n is the iterator over the cells[3], x_s is the center of the solid's mass and x_n is the position of the nth cell. Note that $B_n \sum_i \Omega_i^s e_i$ is the same in both equations and has only to be determined once. In addition, Ω_i^s is already required in the collision step for the technique developed by Noble and Torczynski [Noble and Torczynski, 1998] and can be re-used.

Intuitively, $\sum_n B_n \sum_i \Omega_i^s e_i$ describes the number of fluid molecules that bounced off of the solid boundary. Note that Ω^s describes them *after* the bounceback, which means that they point away from the boundary. Thus, in order to determine the force that the molecules transferred to the boundary, the values of F and T have to be inverted.

When implementing this operation in CUDA, the sum has to be implemented as a reduce operation. Since the GPU is a parallel processing unit, summing is non-trivial. However, NVidia^{TM} provides example code called "reduce" to explain on how to implement this without sacrificing the performance advantage of the GPU.

Since the fluid-solid coupling uses the same parameters as the solid-fluid coupling, it should be combined into the same CUDA kernel to provide optimal performance. Pseudocode for this operation is shown in Listing 5.2.

5.5 Two-Way Coupling

In theory, a combination of the techniques presented in the previous two sections would result in full two-way coupling. However, there are certain issues that have to be kept in mind.

Solids moving faster than the speed of sound in the fluid domain would cause a breakdown of the simulation. This can be avoided by limiting the maximum speed of moving boundaries. However, in two-way coupling, this causes the fluid-solid-coupling to generate incorrect results, too.

Since a physics engine and a fluid simulation have to communicate with each other, the physical units have to be kept in sync. For example, a 1m metal sphere with a mass of 1kg might look realistic in a purely rigid simulation, but would generate unexpected behavior when immersed in a fluid. See Section 3.6 for more information on scaling requirements.

Further, when a rigid object moves faster than it would be possible in the given fluid due to aerodynamic resistance, the fluid's recoil into the opposite direction has a greater force than the object's force into its current direction. This is due to skipping the step of applying forces to the fluid and directly using the current speed of the boundary.

[3] Note that this includes all cells in the domain, but the operation can be optimized by not including cells where B is zero, since they do not have any effect on this equation. Since this is not possible in CUDA, another optimization would be to only use the axis-aligned boundary box of the solid.

Listing 5.2: Fluid-solid coupling and solid-fluid coupling in a CUDA kernel.

```
// size is the total number of cells in each lattice dimension
// scale is the number of sub-cell voxelization steps per dimension
// solidicity is a per-object scaling factor to define the object's
//   permeability
// M is the transformation matrix from fluid space to the object's
//   center of mass space
// linearVelocity, angularVelocity are the object's speed as determined by the
//   physics engine
__global__ void collectSolidInformation_k(vector3 size, float scale,
        float solidicity, Mat4x4 M,
        vector3 linearVelocity, vector3 angularVelocity) {
    {x,y,z} ← current position in the lattice;
    const float oneitempart ← 1/(scale^3);
    vector4 u ← (0,0,0,0);
    for(sub_x from 0 to scale) {
        for(sub_y from 0 to scale) {
            for(sub_z from 0 to scale) {
                float voxinfo = voxelization[(scale * x + sub_x)
                  +(scale * size_x * sub_z)
                  +scale * size_x * scale * ((scale * y + sub_y) + scale * size_y * z)];
                u_w ← u_w+ saturate(voxinfo) * oneitempart;
                // saturate() clamps the value to [0,1].
            }
        }
    }
    u_w ← u_w * solidicity;
    vector3 fluidpos ← position of the current cell in the fluid in the
      fluid coordinate system;
    vector3 rel_pos ← M * fluidpos;
    {u_x, u_y, u_z} ← (linearVelocity + angularVelocity × rel_pos) * size;

    float B = u_w * (τ − 0.5)/(1.0 − u_w + τ − 0.5);
    vector3 impulse = −B * u_fluid[current position];
    vector3 torque = rel_pos × impulse;

    reduce impulse and torque;
    write impulse and torque to result array in global memory;
    write u to the solid u array in global memory;
}
```

Chapter 6

Game Design with Fluid Simulations and Their Implementation

When designing a game using a fluid simulation as a game element, special considerations have to be taken into account.

First, nowadays computer hardware is not capable to simulate large fluid domains due to limitations in memory and computational capacity. This means that there are limitations to the extent this element can be applied. For example, whole underwater worlds are better simulated by current approximations using non-interactive forces applied to regions with hit-detection (as soon as an object enters a region like a bounding box, it gets accelerated using a constant force). However, a fluid simulation could be used locally around the player character to give the impression of a fully simulated environment.

Another issue is that currents in air and water are invisible in the real world. On the one hand, striving to make the look as realistic as possible inhibits the feedback required for the user, since touch and temperature sensations are missing on the current output devices. On the other hand, creating a non-realistic visualization (like a velocity field) removes the impression of a fluid, and doesn't let the user apply knowledge about real-world fluid dynamics to the game environment, requiring a steeper learning curve. One possible workaround is to insert small air bubbles in water or leaves in air to keep a fluid impression without altering the game experience. Another visualization approach is used in Plasma Pong shown in Figure 2.3, where the simulation is used for simulating plasma, which is visible to the human eye (e.g. aurora borealis).

Even when a fluid simulation is not part of the game mechanics themselves, it can still contribute to the atmosphere of the game world, especially when used for simulating smoke or fire. Examples of this are shown in Figure 6.1. However, currently these elements have to be used sparingly, because the processing power available is usually already devoted to the game elements.

Another area where fluid simulations can be used in games are water surfaces forming ponds or a sea. For these, a two dimensional simulation usually suffices, greatly enhancing the performance. When localized three dimensional effects (e.g. around a character) are desired, the simulation can be combined with a

(a) Example of the use of smoke as an atmospheric element in the game *Hellgate: London* by *Flagship Studios*.

(b) Example of the use of fire in games using fluid simulation in an NVidia^{TM} example.

Figure 6.1: Examples of uses of fluid simulations for interactive entertainment products.

three dimensional simulation as outlined by Thürey *et al.* [Thürey *et al.*, 2006].

6.1 The Fluid Pinball

The end result of the work described in this thesis is a pinball game with the added element of being underwater, as demonstrated in Figure 6.2. The goal is to demonstrate that it is possible to use CFD for interactive games and to investigate the possible pitfalls for implementations. A pinball was chosen, because it is a well-known game that can benefit from such a change in domain.

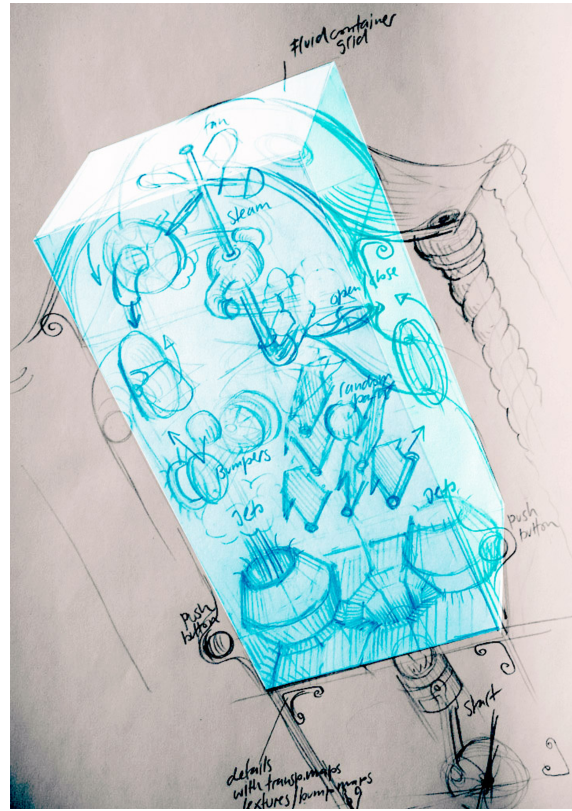
6.1.1 The Pinball Game

Pinball is an arcade game where points have to be scored by manipulating a metal ball on a playfield mounted in a glass-covered box on four feet. This manipulation is done by two or more levers called "flippers". A player can use a defined number of balls in succession, while a central hole is used to remove balls from the game. Nowadays pinball games use electronics to add visual and audible support to the gaming experience.

Scoring is done by using the game elements featured by the pinball machine, which will be outlined in Section 6.1.4.

Starting from the very earliest home computers like the Atari and Apple][, games simulating pinball machines have been a common theme with ever-improving physics. The first pinball simulator was created

(a)



(b)

Figure 6.2: (a) A classical pinball machine called "Theatre of Magic" and (b) a sketch of a possible fluid pinball realization (courtesy of Simon Tschachtli), which can feature many additional elements.

in 1982 and was called "David's Midnight Magic", while nowadays every copy of Microsoft Windows ships with such a game preinstalled under the name "Space Cadet" (licensed from Maxis).

6.1.2 Adapting the LBM Implementation to a Game

Before the game design considerations can be looked at, the technical details for implementing the Lattice Boltzmann method with complex obstacles outside of a testbed have to be examined.

Voxelization can be implemented as outlined in Section 5.1. However, care has to be taken to use only manifold meshes where the object thickness is not smaller than the thickness of a cell. Otherwise, fluid is "leaked", which results in unexpected behavior.

When handling two-way interaction with rigid bodies, a separate voxelization per object is required to differentiate the forces acting on each of them. Since the voxelization process and the interaction between a solid and the fluid is computationally expensive, the amount of objects should be kept to a minium. This can be achieved by treating the whole static geometry of the pinball as a single object. Additionally, since it does not move with respect to the fluid domain, the voxelization has only to be done once.

Another optimization for moving rigid objects is to voxelize them in their bounding box as a preprocessing step, and then resampling the grid at the object's current location and rotation at every frame.This reduces accuracy (depending on the sampling grid used), but avoids re-voxelization on every frame.

The flippers required for interacting with the ball in the original game are replaced by jets accelerating the fluid, which then accelerates the ball instead. This is more efficient, since unlike moving flippers, the jets do not need to be voxelized every frame. Accelerating the fluid is done by setting u_s to the desired velocity instead of using the rigid body's real velocity (which is always 0).

This still leaves multiple voxelized grids that have to be combined into a single grid for the solid collision Ω^s.

For the overall solidicity value ε for a cell, the sum of the voxelizations' ε can be used. This assumes that the sum is not greater than 1, which is analytically true, but might be violated due to rounding errors, and thus the value is clamped to 1.

The solid velocity u_s cannot be determined by linear combination. The ε-weighted average of all voxelizations does not account for the fact that the fluid part of the cell does not affect u_s. So the following equation is used:

$$u_s = \frac{1}{\sum_n \varepsilon_n} \sum_n (\varepsilon_n u_{sn}) \tag{6.1}$$

where ε_n and u_{sn} are ε and u_s respectively of the current cell in the nth voxelization (assuming any order in the set of voxelizations).

Even when the fluid simulation is implemented properly, there is a lot more to making a game based on this concept. For instance, the game engine also requires model loading, a scene graph, audio, a

user interface and scripting. Since this would require a large team of developers, a ready-made rendering engine called Ogre and other libraries integrated into its framework were used.

6.1.3 Ogre

Ogre[1] is a cross-platform open source rendering library for OpenGL and Direct3D. It was described in detail by Junker [Junker, 2006] (also see the web page for more documentation), an example rendering is shown in Figure 6.3.

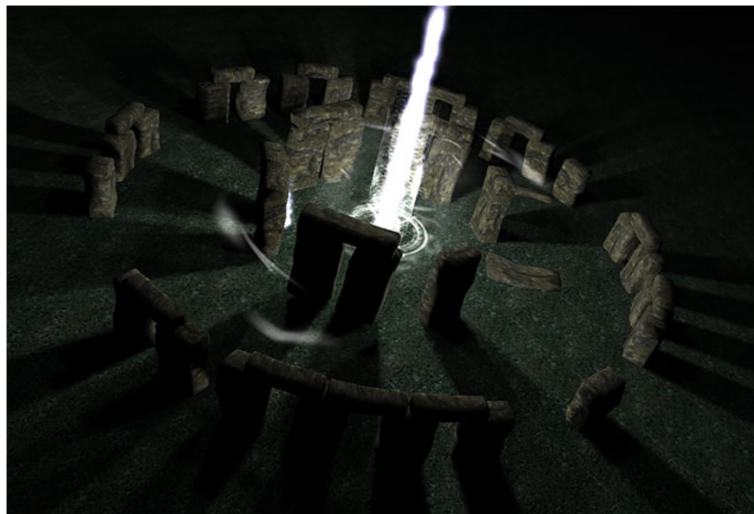


Figure 6.3: A rendering of Stonehenge by Arsen Gnatkivsky called "Magic of Stonehenge" demonstrating Ogre's capabilities.

As a third party addon, integration with Bullet Physics is available, allowing defining physical representations of the models loaded into the application and automatic integration during the simulation process, which means that the physical model and rendering model are always kept in sync.

Rendering the pinball geometry from a model file, adding a user interface and enabling shadows use standard techniques in Ogre. However, a common problem with rendering API-abstracted engines such as this one is that they do not allow low-level immediate rendering commands, since the program only interacts with high-level objects, reducing the whole rendering process to a single method call. This is usually not a problem, because adding rendering features should be done in the engine, where the low-level API is exposed.

However, integrating CUDA into the Ogre engine requires using on-GPU memory as the source for render operations. This is not planned for in the high-level API.

As mentioned in Section 4.4.1, two visualizations were implemented: Particle advection and a velocity texture.

[1] http://www.ogre3d.org/

OpenGL allows rendering a vertex buffer object as GL_POINTS (which can be used as a target memory area in CUDA). Ogre supports particle systems with the Ogre::ParticleSystem class. However, these are expected to be simulated on the CPU, which does not apply here. A class called Ogre::SimpleRenderable is supplied for defining simple geometries in code. This mechanism can be used for rendering points from a vertex buffer object. It contains a virtual method called getRenderOperation(Ogre::RenderOperation& op) which is called whenever the object has to be rendered, which can be overridden in a subclass. The render operation passed to this method is an object where the class is expected to define the way it wants to be rendered. By setting its operation type to Ogre::RenderOperation::OT_POINT_LIST and its vertex data pointer to an instance of the wrapper class for OpenGL vertex buffer objects, wrapping the vertex buffer object supplied by CUDA, Ogre will render it as a point list using OpenGL hardware points (A similar path could be provided for Direct3D-based rendering). The points' properties like size and texture can be set using regular Ogre material definitions, providing a seamless integration. Optionally, vertex and fragment shaders can be used to customize the appearance (geometry shaders are not yet supported by Ogre, but are planned).

Textures are handled as parts of the material definitions in Ogre. This means that in order to add a CUDA-generated texture to the application, a material has to be generated. For this, Ogre's texture manager supplies a createManual() method, where the texture type and dimension can be supplied in code. By downcasting the texture to its Ogre::GLTexture subclass, the texture id used can be determined and used every frame as a target for glTexSubImage2D, which allows copying the data stored in a pixel buffer object by CUDA into its dedicated memory space. Since writing to OpenGL or Direct3D textures is not supported with the current version of CUDA, this copy is unavoidable. However, since this operation is done solely on the GPU, its speed is not limited by the computer's busses.

6.1.4 Pinball Game Elements

As already mentioned, special care has to be taken when integrating a fluid simulation in an already existing game, since it changes the physics, and thus the gaming experience. The most important distinction is that in a regular pinball game, the ball usually only moves along the surface, while in fluids the ball can lift up. This means that the third dimension has to be taken into account. Since a typical pinball machine features many different elements, they have to be examined separately and tested for their suitability for an underwater game. The following list tries to analyze several elements commonly used in pinball games, but is not complete.

- The *ball* is usually 27mm in diameter and made out of steel (Figure 6.4a), although the pinball machine "Twilight Zone" also features a ceramic ball (Figure 6.4b).

  Integrating a ball is essential for preserving the basic concept of pinball, but the correct balance between the fluid's viscosity and the ball's mass has to be determined by game testing. Additionally, since the torque of the ball is more important than in the classic game, it should have a patterned texture that can visualize the movement, as shown in Figure 6.4c.

- The *playfield* (Figure 6.5) is a planar surface usually inclined by 6.5 degrees by convention. The ball is accelerated by gravity while rolling down this surface, so the inclination is essential to determine the game's speed.

For fluid pinballs, the inclination has to be altered to retain the speed, since the fluid reduces the ball's velocity. For example, the underwater pinball game produced by TOMY (Figure 2.1) uses a 90 degree inclination. When simulating the fluid using the Lattice Boltzmann method, care has to be taken that the ball does not gain more than the domain's speed of sound while accelerating due to the simulation of gravity.

- The *plunger* (Figure 6.6) is a spring-loaded rod used to propel the ball over the playing field at the start of the game. It allows determining the force applied to the ball.

  This idea does not translate well to any computer-based pinball game, and can be replaced by a fixed push, or when using fluids, by a fixed stream of fluid from a jet.

- A *flipper* (Figure 6.7) is a lever about 3 to 7 cm in length and is used for directing the ball in the game. Current designs feature at least two of them at the bottom of the playfield. This concept could be used in a fluid pinball, but in order to emphasize the special element of the game, a jet propelling fluid at the push of a button is recommended instead.

- A *bumper* (Figure 6.8) is a round knob in the playfield that pushs the ball away when hit. This element can be used unaltered, but a sphere with the same properties could also be used.

- A *slingshot* (Figure 6.9) is a side of a wall that also pushes the ball away when hit. Current pinball machines have at least one over each of the two bottom flippers. This element can also be used unaltered.

- A *ramp* (Figure 6.10) is an inclined plane used as a skill shot challenge, requiring more force to be passed. Since fluids have a higher viscosity than air, the ball is not necessarily touching either the playfield or the ramp with these game mechanics, making this element impossible to pass. It should be replaced by pipes, either with a full mantle or a mantle with holes when the fluid should be allowed to penetrate from the side.

- A *target* (Figure 6.11) is an element that adds points to the player's score. It can either be stationary or hide when hit. This element is essential to the game and can be used unaltered.

- A *hole* (Figure 6.12) is used for capturing the ball, giving either points or free games. In some pinball games, this hole is connected to a hidden ramp under the playfield. A ball would not move into the hole on its own in a fluid, and so this could be replaced by a vortex.

- A *rotating bumper* (Figure 6.13) is an obstacle that rotates constantly during gameplay, distracting the ball when it is hit. When used for a fluid pinball, this spinning motion would create a vortex, even enhancing the experience.

- A *mini-playfield* (Figure 6.14) is a secondary playfield usually above the main one, sometimes featuring their own flippers. It usually can be reached and left over ramps. This element can be used in fluid games, or even enhanced (e.g. by changing the simulation properties in the mini-playfield, like disabling the fluid simulation in that area or changing the viscosity).

- An *electromagnet* (also Figure 6.14) is used to capture the ball at a certain spot to either make its movement unpredictable, or preserve it for later re-insertion into the game. It can also be under the control of the player, like in the pinball game "Twilight Zone". Although this is easily possible

with a physics simulation, lacking visual hints, it might reduce the player's understanding of the simulation and thus be treated as a programming error rather than a game element.

- A *rollover* (Figure 6.15a) or *rollunder* is a target where score is added when the ball rolls over or under them. This can not be used for fluid pinballs, since the ball doesn't stay on a flat surface.

  A possible replacement is a ring where the ball can pass through, scoring points as it passes, as shown in Figure 6.15b.

- A *spinner rollunder* (Figure 6.16) has the same semantics as a regular rollunder, but uses a small metal plate that can spin around its horizontal axis, usually taking multiple revolutions when hit. This element would cause a distortion of the fluid domain, and thus would enhance the game experience when used. However, a bigger target would have to be used. Also note that it would require a separate rigid object and voxelization and thus is expensive to compute.

6.1.5 Implementation

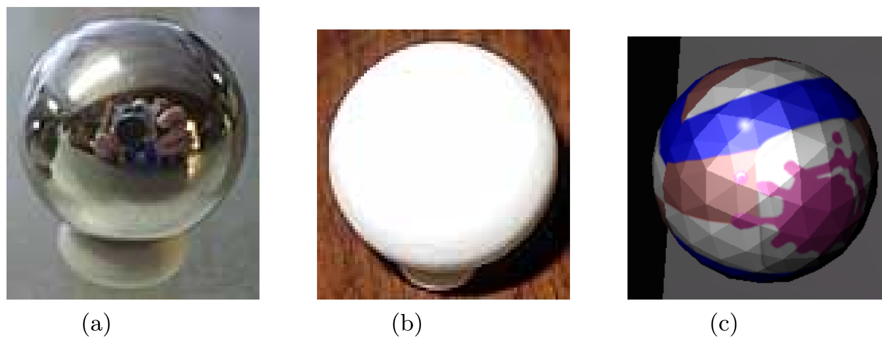
In order to test the balance between ball size and mass, fluid viscosity and the jets' power, a basic fluid interaction testbed is designed and implemented, as shown in Figure 6.17.

Since the particle count is limited by the target framerate, the particles cannot be seeded using an equal distribution. Two seeding areas are defined around the jets using a normal distribution (seeding at the same point would not work, since all particles would be advected in the same way and thus would never separate from each other).

The jets are implemented by using a static geometry, but setting the solid's velocity to a non-zero vector. Another possible implementation is to model a propeller and let it rotate, simulating a physical implementation. However, this would require re-voxelizing the object every frame (unless resampling is used, as described on page 51) and a high resolution of the fluid grid with respect to the propeller's blades, both being a hindrance for real-time simulations.

Note that replacing the flipper of the typical pinball layout with a jet without any other adaptations, as demonstrated in Figure 6.18, has negative effects on the playability. The fluid flow forces the ball to be drawn towards the area between the edge of the playfield and the jet, effectively forcing the ball to be lost. This not only happens when the jet is activated, but also at other times due to the fluid's delay in reacting to these changes on a global scale. This effect can be used intentionally for enhancing the experience level required for mastering the game, but can also limit the playability of it.

A pinball game requires the selection of a specific theme, usually a television series, a movie or an era (for example, the pinball game "Star Trek: The Next Generation" and "Twilight Zone" are both based on the television series with that name). The art designer is then responsible for creating art based on that theme, and the game designer has to map the elements used in the game to the context. For example, in the pinball game "Star Trek: The Next Generation", shooting the warp loop and delta quadrant ramp increases the warp speed [Williams Electronics Games, Inc., 1993]. For fluid pinballs, a water theme or an industrial theme like used in Figure 6.17 (using the fluid for simulating steam) can be used.

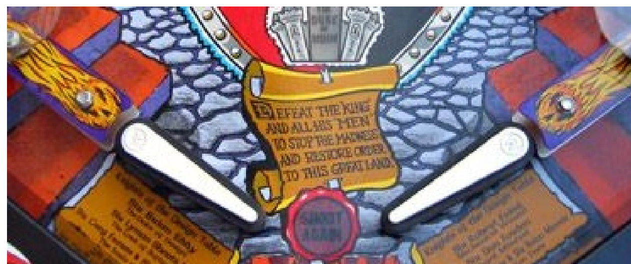


(a) (b) (c)

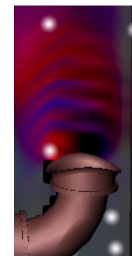
Figure 6.4: (a) A typical chrome ball used in current pinball games, (b) a ceramic ball used in the pinball game “Twilight Zone” and (c) the ball used in the implementation described in this thesis.

Figure 6.5: The playfield of the pinball game “Twilight Zone”.

Figure 6.6: The plunger of the pinball game “Creature from the Black Lagoon”.



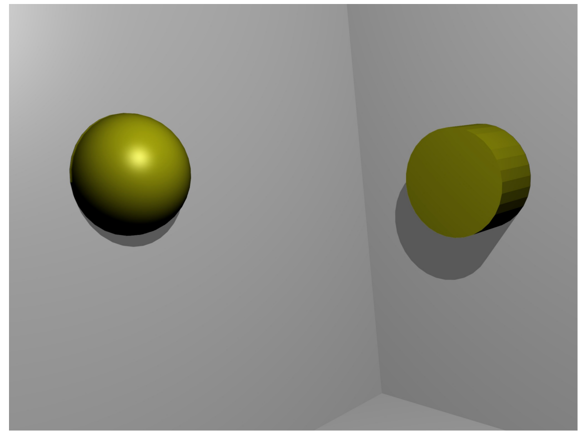
(a)



(b)

Figure 6.7: (a) The two bottom flippers of the pinball game “Medieval Madness” and (b) the replacement jet used in the fluid-based pinball game.

(a)

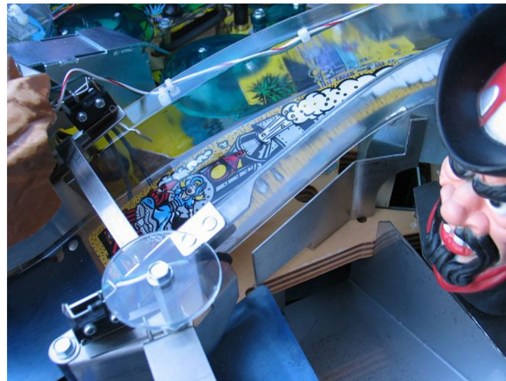


(b)

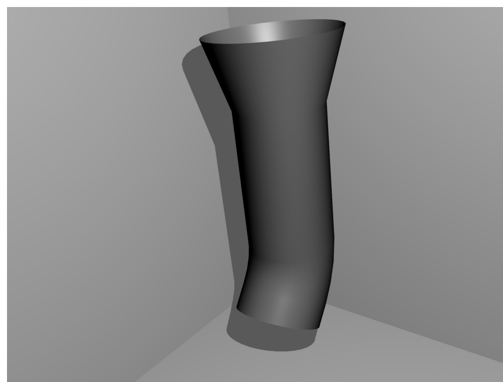
Figure 6.8: (a) Five bumpers of the pinball game “Trade Winds” and (b) two possible adaptions to a three-dimensional pinball game (concept art).

Figure 6.9: A slingshot of the pinball game “Funhouse”.

(a)



(b)

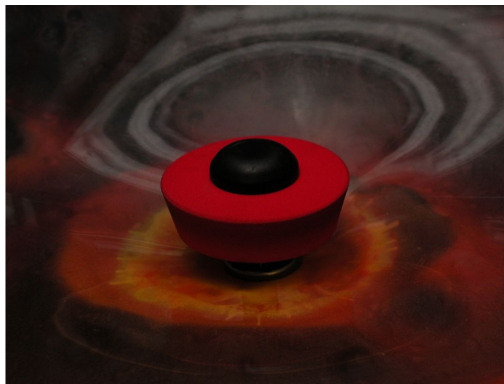


(c)

Figure 6.10: Ramps of the pinball games (a) “Star Trek: The Next Generation” and (b) “Cactus Canyon” and (c) the fluid adaption (concept art).

Figure 6.11: Three targets of the pinball game “Star Trek: The Next Generation”.

Figure 6.12: The hole of the pinball game “Funhouse”.



(a) The rotating bumper from the pinball game "Orbitor 1".

(b) The rotating bumper from the pinball game "Spinball". It does not score any points when hit.

Figure 6.13: Two examples of rotating bumpers.

Figure 6.14: The mini-playfield of the pinball game "Twilight Zone". It uses two electromagnets to simulate the flippers.

(a) (b)

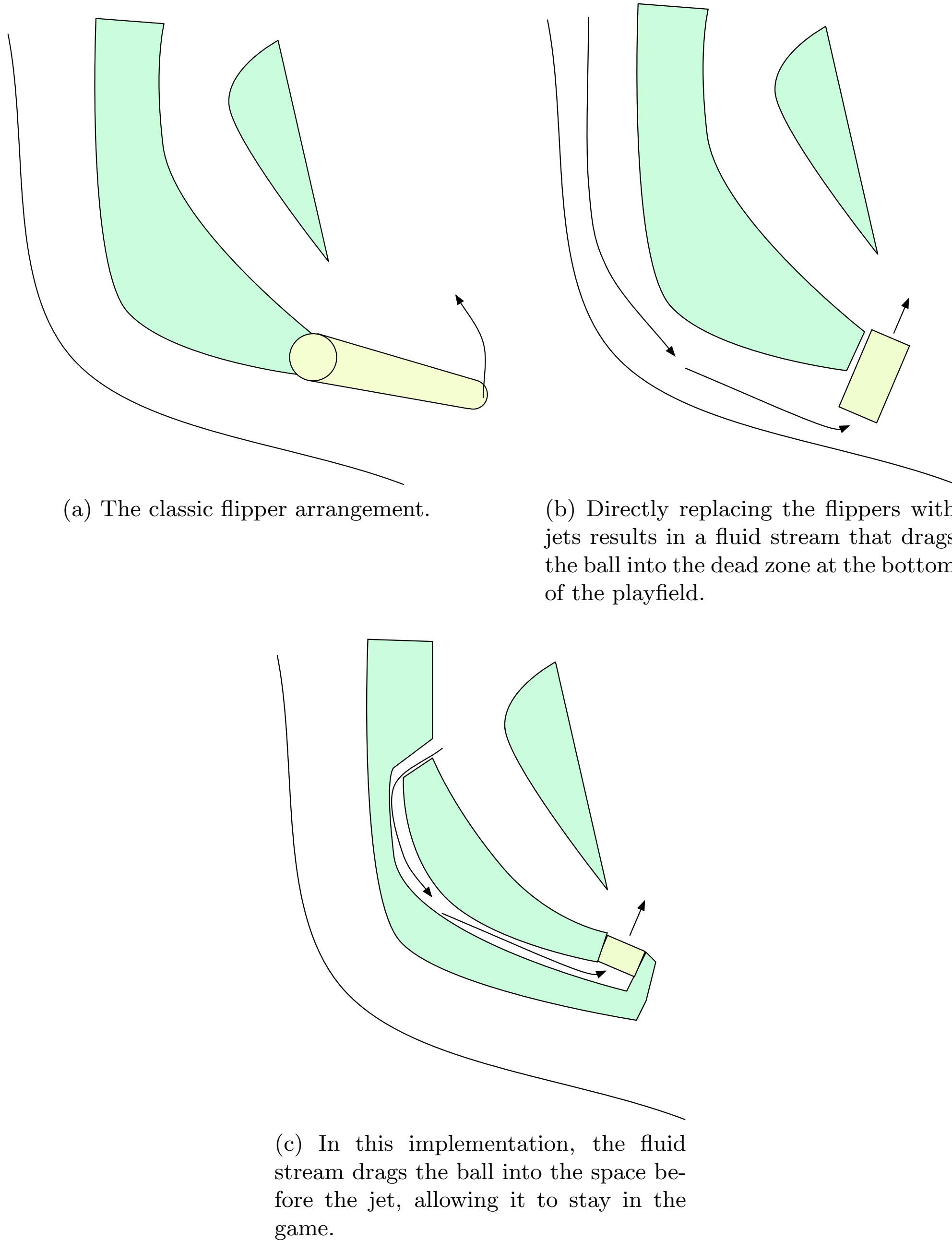
Figure 6.15: (a) A rollover target in the pinball game “Cirqus Voltaire” and (b) the equivalent element in the fluid pinball prototype.

Figure 6.16: A spinner rollunder of the pinball game “Freedom”.

(a) Basic version of the fluid pinball testbed, using large particles, lacking gameplay.

(b) Extended version of the fluid pinball testbed with smaller particles, and a target that enables the player to gain points by passing it.

Figure 6.17: The testbed implementation of the fluid pinball concept.



(a) The classic flipper arrangement.

(b) Directly replacing the flippers with jets results in a fluid stream that drags the ball into the dead zone at the bottom of the playfield.

(c) In this implementation, the fluid stream drags the ball into the space before the jet, allowing it to stay in the game.

Figure 6.18: The layout of (a) a flipper in a typical pinball game, (b) the naïve adaptation to using a fluid jet, showing the fluid streams resulting from the layout and (c) a possible solution.

Chapter 7

Results

The application was tested on an AMD Athlon 64 X2 Dual at 2.21 GHz and 4 GB of RAM running Windows XP Professional x64 SP1. The graphics card is a NVidia^{TM} Geforce 8800 GTX with 768 MB of RAM and 128 stream processors, the theoretical memory bandwidth limit is 86.4 GB/sec. The CUDA version used is 1.1, which does not feature three-dimensional texture access, and thus requires tiling for representing the three-dimensional volume of the fluid domain.

Two versions of the implementation exist: One bare-bone, which contains minimal drawing code, as shown in Figure 7.1a. It contains the particle visualization, with the geometry shader from Section 4.4.1, and the velocity visualization slice outlined in Section 4.4.1. Optionally, a solid sphere can be enabled to measure the performance of the voxalization technique, as shown in Figure 7.1b. The second implementation is a fully-featured application, which includes a menu interface, physics and complex geometry, to simulation a gaming environment.

Note that the fluid simulation itself is independent of any occurrences in the simulation, since the operations themselves don't change. This means that enabling/disabling complex obstacles has only an effect on the speed of the simulation, because the voxelization technique has to be activated/deactivated.

7.1 The Basic Application

In the basic application, the only variable affecting the simulation performance itself is the resolution of the grid used. Thus, a few selected grid sizes were tested to demonstrate a profile of the implementation's performance, the results are shown in Table 7.1. Note that due to the tiling algorithm used for the voxelization technique, the resolution is limited due to the texture size limit of 8192x8192. The algorithm spread the sub-cell z-slice tiles in x-direction and the full-cell z-slice tiles in y-direction. Thus, a $128\times128\times128$ grid with a sub-cell resolution of 4 in all three directions requires $x * sub_x * sub_z = 128 * 4 * 4 = 2048$ pixels in x-direction and $y * z * sub_y = 128 * 128 * 4 = 65536$ pixels in y-direction (where sub_n is the subpixel-resolution in the n-direction). A better algorithm would try to use a square texture, which would

allow larger voxelizations and also provide better performance due to better optimization by the graphics hardware.

$sub_n =$	0	1	2	4
32×32×32	337	232	178	74
64×32×32	256	186	132	45
32×32×64	256	169	114	41
128×32×32	174	130	87	25
32×32×128	174	107	67	-
128×128×16	106	80	51	13
128×128×32	59	45	28	-
128×128×64	32	25	-	-
128×128×128	16	-	-	-

Table 7.1: The performance of the basic application at different cell resolutions, measured in frames per second. sub_n is the sub-cell resolution used for the voxelization technique. $sub_n = 0$ means that the voxelization is deactivated.

The block size for the fluid kernel is set to be equal to the number of cells in the x-direction. The voxelization is done per z-slice, which is the reason why the cell resolution $128 \times 64 \times 64$ provides better performance than $64 \times 64 \times 128$ when it is enabled.

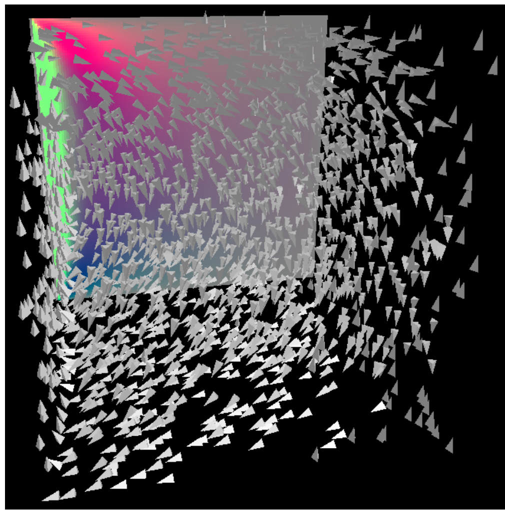
7.2 The Game-like Application

As mentioned in Chapter 6, this application uses the open source tools Ogre 3D and bullet physics to demonstrate how a fluid simulation can be embedded in a game environment. CEGUI is used for implementing the user interface, which allows introspection into the fluid simulation and altering of parameters, but would be used for game mechanics-related operations in a real game.

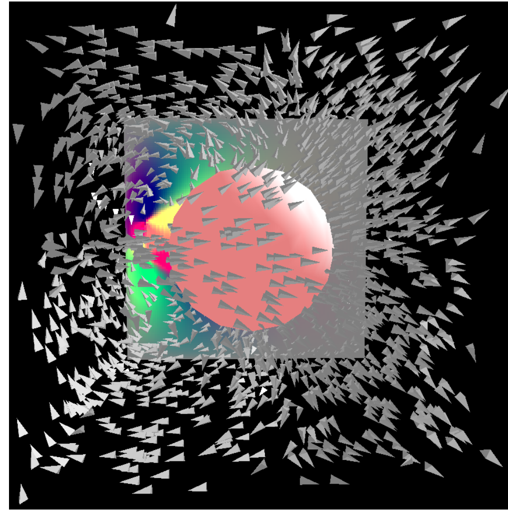
In addition to the features of the basic application, two-way coupling is supported, which degrades performance due to the two-way communication needed between the CPU and the GPU. A screenshot is shown in Figure 6.17b, the performance numbers measured are listed in Table 7.2. In this implementation, the static geometry is only voxelized once at application launch, but the ball is voxelized at every frame.

7.3 Analysis of the Performance Measurement Results

Table 7.1 demonstrates that the main bottleneck in this implementation is not the fluid simulation itself, but the voxelization technique. Thus, optimization should be applied to this area first. For example, the *depth peeling* method might improve performance considerably. Additionally, removing the requirement for a new voxelization from scratch at every frame can provide a frame rate close to the simulation without complex obstacles.



(a) The basic version with particles and the velocity slice in the back.



(b) The basic version plus a sphere for testing complex obstacles immersed in the fluid.

Figure 7.1: The minimal version of the application, which was implemented for measuring the performance of the simulation itself.

$sub_n =$	1	2	4
32×32×32	43	39	26
64×32×32	40	35	19
32×32×64	35	31	18
128×32×32	35	29	-
32×32×128	26	21	-
128×128×16	27	21	-
64×128×32	24	19	-
128×128×32	18	13	-
128×128×64	11	-	-
128×128×128	-	-	-

Table 7.2: The performance of the game-like application at different cell resolutions, measured in frames per second. sub_n is the sub-cell resolution used for the voxelization technique. Since complex obstacles are tightly integrated in this application, they cannot be turned off like in the basic application.

Table 7.2 shows the downside of moving the simulation to the GPU: Since the GPU is the limiting factor, every single additional operation on it has a direct effect on the frame rate. For example, enabling stencil shadows halves the frame rate per light source (not shown in the tables, these were measured with a single light source and shadows turned off). The complexity of the models rendered has a direct impact on the performance, too. Finally, in addition to the potential optimization techniques listed for the basic application, pipelining the GPU commands would allow the CPU-based physics simulation to run in parallel, even when using two-way coupling.

Chapter 8

Conclusion

Due to the advancements in graphics processing hardware, grid-based fluid simulations have moved into the radar of real-time applications.

Graphics processors have become stream-based processing units capable of processing tasks with large amounts of data and a high arithmetic density in a time frame where real-time simulations are possible, even when rendering is also taken into account.

Even though the fluid simulation already executes at interactive rates on graphics cards of the current generation, adding more effects to the game would require faster processors. However, according to NVidia^{TM}, GPU performance currently doubles every 12 months, which will allow to add fluid simulations to current-generation games without affecting the interactivity even in the short term.

The next step to optimize the simulation process would be to avoid the voxelization at every frame. Since only rigid bodies are considered, a single voxelization with the identity transformation could be used, which is then transformed to the current location, rotation and scale of the body for every frame. Another optimization would be to offload the rigid body physics to the GPU, in order to decrease the two-way communication required between the two processing units. A promising approach for achieving this, while still being able to interface with the Lattice Boltzmann method implementation, is described by Harada [Harada, 2007]. Furthermore, a better utilization of the GPU could be achieved by making better use of the memory caches available to the programmer.

It has been demonstrated that the Lattice Boltzmann method is ideally suited for GPU-based implementations due to its simplicity and accuracy. Further, integrating complex objects using two-way coupling is possible with minimal modifications to the original equations and a well-known voxelization technique that allows direct use of the rendering pipeline.

CUDA allows easy integration of CPU and GPU processing with an even greater efficiency than shader-based solutions using a well-know programming language, sacrificing programming architecture simplicity. Even though CUDA-based physics would be preferred, integrating an off-the-shelf CPU-based physics engine is easily possible, but requires awareness of the physical representation of the fluid simulation parameters.

Further, a well-known game can be extended by adding a fluid simulation as a game element, totally changing the user experience, or as a way to enhance the atmosphere, which creates a greater immersion. For pinball games, the simulation enhances the way the game has to be played, because it moves the game to the third dimension. In addition, the objects' movements not only have a short-term effect, but alter the streams in the fluid domain on a global scale, effecting changes on a longer time scale. These applications also allow a layman-centered audience to learn about the behavior of fluids and are an important future opportunity for the edutainment market.

Acknowledgements

I would like to thank Prof. Eduard Gröller and Raphael Fuchs for their help, support and corrections of this thesis. Furthermore, I would like to thank the *VRVis Research Center for Virtual Reality and Visualization* for supplying a workplace and the non-scientific material required for completing this work, and David Holdych for supplying his thesis. Special thanks goes to Simon Tschachtli for providing artistic input for the visual design of the fluid pinball.

List of Figures

2.1 A real-world implementation of the fluid pinball concept from TOMY, 1977. 2
2.2 A screenshot from the game Crysis, demonstrating the wave-based fluid simulation used in most of the current games. 4
2.3 An implementation of the Pong game with the combination of a fluid solver based on the work of Stam [Stam, 1999] called "Plasma Pong" by Steve Taylor. 7
2.4 A breakdown of different fluid computation techniques. 7

3.1 The D2Q9 LBM geometry, including a suggested ordering for the index i. The zero-velocity vector is visualized by a small circle in the center. 9
3.2 The directional vectors e_i in different LBM geometries. 10
3.3 Cells affected by the streaming phase of the center cell. 11
3.4 The basic steps of the LBM operation in a D2Q9 lattice. 12
3.5 The two bounce back methods: Free-slip and no-slip. 14

4.1 The iterative approach to the bitonic sorting algorithm. The arrows on the left indicate the operations the originating line spawns. 18
4.2 A sorting network for eight input values for the parallel version of the bitonic sort. The input values are the same as for the iterative approach mentioned earlier. The black rectangles designate operations that can be done simultaneously. The operations are the same as in Figure 4.1, but no recursive function calls are necessary. 19
4.3 The two types of lattice operations applied to LBM D2Q9 (before and after the operation): (a) scatter and (b) gather. 21
4.4 The CUDA architecture as a block diagram. 23
4.5 Visualizing the fluid domain's velocity using pyramid glyphs representing arrows. Additionally, the velocity texture is visible at the back, showing the center slice's velocity of the fluid domain in red/green/blue color coding for X/Y/Z. 32

5.1 A demonstration of the voxelization technique on a sphere. 36
5.2 The result of applying the voxelization technique for all slices on a pinball geometry. . . 37
5.3 Demonstration of the Bullet Physics library's capabilities collapsing a house. 39
5.4 When using only the whole-cell boundaries, the border cells show a non-realistic bounce-back behavior due to the "staircase-effect". The blue line represents the solid boundary. 40

5.5 Definition of the nomenclature used by Mei *et al.* [Mei *et al.*, 2000]. Grey cells are boundary nodes, white cells are fluid nodes. e_i is the direction vector towards the boundary, x_b is the boundary node, x_w is the virtual wall node, x_f is a fluid node bordering to a boundary node and x_{ff} is the fluid node on the opposite direction of the boundary node, as seen from the bordering fluid node. 41
5.6 Demonstrating the solid fraction ε by using grayscale values. Black means $\varepsilon = 1$ (completely solid), while white means $\varepsilon = 0$ (completely fluid). The blue line represents the solid boundary. 43
5.7 The function B with $\tau = 0.6, 0.7, 0.8, 0.9$ in red, green, blue and magenta respectively. . 43

6.1 Examples of uses of fluid simulations for interactive entertainment products. 49
6.2 A classical pinball machine called “Theatre of Magic” and a sketch of a possible fluid pinball realization (courtesy of Simon Tschachtli). 50
6.3 A rendering of Stonehenge by Arsen Gnatkivsky called “Magic of Stonehenge” demonstrating Ogre’s capabilities. 52
6.4 A typical chrome ball used in current pinball games and a ceramic ball used in the pinball game “Twilight Zone”. 56
6.5 The playfield of the pinball game “Twilight Zone”. 57
6.6 The plunger of the pinball game “Creature from the Black Lagoon”. 58
6.7 (a) The two bottom flippers of the pinball game “Medieval Madness” and (b) the replacement jet used in the fluid-based pinball game. 58
6.8 (a) Five bumpers of the pinball game “Trade Winds” and (b) two possible adaptions to a three-dimensional pinball game (concept art). 59
6.9 A slingshot of the pinball game “Funhouse”. 59
6.10 Ramps of the pinball games “Star Trek: The Next Generation” and “Cactus Canyon” and the fluid adaption. 60
6.11 Three targets of the pinball game “Star Trek: The Next Generation”. 61
6.12 The hole of the pinball game “Funhouse”. 61
6.13 Two examples of rotating bumpers. 62
6.14 The mini-playfield of the pinball game “Twilight Zone”. It uses two electromagnets to simulate the flippers. 63
6.15 (a) A rollover target in the pinball game “Cirqus Voltaire” and (b) the equivalent element in the fluid pinball prototype. 64
6.16 A spinner rollunder of the pinball game “Freedom”. 64
6.17 The testbed implementation of the fluid pinball concept. 65
6.18 The layout of a flipper in a typical pinball game and the naïve adaptation to using a fluid jet, showing the fluid streams resulting from layout. 66

7.1 The minimal version of the application, which was implemented for measuring the performance of the simulation itself. 69

Bibliography

- Batcher, K. E., *Sorting networks and their applications*, in *Spring Joint Computer Conference* (Akron, Ohio, USA) (1968)
- Buick, J. M. and Greated, C. A., *Gravity in a lattice Boltzmann model*, in *Physical Review E*, volume 61(2000)(5): 5307–5320
- Carlson, Mark, Mucha, Peter J. and Turk, Greg, *Rigid fluid: animating the interplay between rigid bodies and fluid*, in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pp. 377–384 (ACM Press, New York, NY, USA) (2004)
- Chen, Jim X. and da Vitoria Lobo, Niels, *Toward interactive-rate simulation of fluids with moving obstacles using Navier-Stokes equations*, in *Graph. Models Image Process.*, volume 57(1995)(2): 107–116
- Coumans, Erwin, *Bullet User Manual* (2008)
  URL `http://bulletphysics.com/ftp/pub/test/physics/Bullet_User_Manual.pdf`
- Fedkiw, Ronald, Stam, Jos and Jensen, Henrik Wann, *Visual simulation of smoke*, in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 15–22 (2001)
- Filippova, Olga and Hänel, Dieter, *Grid refinement for lattice-BGK models*, in *J. Comput. Phys.*, volume 147(1998)(1): 219–228
- Frisch, Uriel, Hasslacher, Brosl and Pomeau, Yves, *Lattice-gas automata for the Navier-Stokes equation*, in *Physical Review Letters*, volume 56(1986)(14): 1505–1508
- Harada, Takahiro, *GPU GEMS 3 Chapter 29, Real-Time Rigid Body Simulation on GPUs* (Addison Wesley) (2007)
- Harris, Mark J., *GPU GEMS Chapter 38, Fast Fluid Dynamics Simulation on the GPU* (Addison Wesley) (2004)
- Holdych, David J., *Lattice Boltzmann methods for diffuse and mobile interfaces*, Ph.D. thesis, University of Illinois at Urbana, Champaign, USA (2003), ph.D. Thesis
- Junker, Gregory, *Pro OGRE 3D Programming* (Apress), 1 edition (2006), ISBN 978-1590597101

- Kaufman, Arie and Shimony, Eyal, *3d scan-conversion algorithms for voxel-based graphics*, in *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pp. 45–75 (Chapel Hill, North Carolina) (1986)
- Krüger, Jens and Westermann, Rüdiger, *GPU simulation and rendering of volumetric effects for computer games and virtual environments*, in *Computer Graphics Forum*, volume 24(2005)(3)
- Li, Wei, *Accelerating Simulation and Visualization on Graphics Hardware*, Ph.D. thesis, Computer Science Department, Stony Brook University (2004)
- Li, Wei, Fan, Zhe, Wei, Xiaoming and Kaufman, Arie, *GPU GEMS 2 Chapter 47, Flow Simulation with Complex Boundaries* (Addison Wesley) (2004)
- Liu, Youquan, Liu, Xuehui and Wu, Enhua, *Real-time 3d fluid simulation on GPU with complex obstacles*, in *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pp. 247–256 (IEEE Computer Society) (2004)
- Losasso, Frank, Irving, Geoffrey and Guendelman, Eran, *Melting and burning solids into liquids and gases*, in *IEEE Transactions on Visualization and Computer Graphics*, volume 12(2006)(3): 343–352, member-Ron Fedkiw
- Mei, Renwei, Shyy, Wei, Yu, Dazhi and Luo, Li-Shi, *Lattice Boltzmann method for 3-d flows with curved boundary*, in *Journal of Computational Physics*, volume 161(2000): 680–699
- Noble, D. R. and Torczynski, J. R., *A lattice-Boltzmann method for partially saturated computational cells*, in *7th Int. Conf. on the Discrete Simulation of Fluids* (1998)
- Peachey, Darwyn R., *Modeling waves and surf*, in *SIGGRAPH Comput. Graph.*, volume 20(1986)(4): 65–74
- Scheidegger, Carlos E., Comba, Joao L. D. and da Cunha, Rudnei D., *Navier-stokes on programmable graphics hardware using smac*, in *Proceedings of XVII SIBGRAPI - II SIACG 2004*, edited by IEEE Press, ISBN 0-7695-2227-0, pp. 300–307 (2004)
- Stam, Jos, *Stable fluids*, in *Siggraph 1999, Computer Graphics Proceedings*, pp. 121–128 (Addison Wesley Longman, Los Angeles) (1999)
- Strack, O. Erik and Cook, Benjamin K., *Three-dimensional immersed boundary conditions for moving solids in the lattice-Boltzmann method*, in *International Journal for Numerical Methods in Fluids*, volume 55(2007): 103–125
- Thürey, Nils, *A single-phase free-surface lattice-Boltzmann method*, Master's thesis, University of Erlangen-Nuremberg (2003)
- Thürey, Nils, Iglberger, Klaus and Rüde, Ulrich, *Free surface flows with moving and deforming objects for LBM*, in *Proceedings of Vision, Modeling and Visualization 2006*, pp. 193–200 (IOS Press) (2006)
- Thürey, Nils, Rüde, Ulrich and Stamminger, Marc, *Animation of open water phenomena with coupled shallow water and free surface simulations*, in *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pp. 157–165 (2006)

- Wei, Xiaoming, Li, Wei, Mueller, Klaus and Kaufman, Arie E., *The lattice-Boltzmann method for simulating gaseous phenomena*, in *IEEE Transactions on Visualization and Computer Graphics*, volume 10(2004)(2): 164–176
- Wejchert, Jakub and Haumann, David, *Animation aerodynamics*, in *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pp. 19–22 (ACM Press, New York, NY, USA) (1991), ISBN 0-89791-436-8
- Williams Electronics Games, Inc., *Star Trek – The Next Generation, Operations Manual* (1993)