Real-time 3D Rendering of Water using CUDA

MSc Thesis

Gonçalo Nuno Paiva Amador

Real-time 3D Rendering of Water using CUDA

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING
(Engenharia Informática)

by

Gonçalo Nuno Paiva Amador
natural of Peniche, Portugal

Computer Graphics and Multimedia Group
Department of Computer Science and Engineering
Univeristy of Beira Interior
Covilhã, Portugal
www.di.ubi.pt

Copyright © 2009 by Gonçalo Nuno Paiva Amador. *All right reserved. No part of this publication can be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the previous written permission of the author.*

Real-time 3D Rendering of Water using CUDA

Author: Gonçalo Nuno Paiva Amador
Student Number: M1420
Email: a14722@ubi.pt

Resumo

Esta tese aborda a simulação de água a 3D em tempo-real, tanto para CPU como para GPU. O método conhecido por stable fluids é extendido para 3D, e implementado tanto para CPU como para GPU. A versão para GPU foi feita usando o NVIDIA Compute Unified Device Architecture API (Application Programming Interface), ou resumidamente CUDA. O método conhecido por stable fluids requer o uso de um método iterativo para resolver sistemas lineares esparsos. Para o efeito, três métodos foram implementados, tanto para a CPU como para a GPU, nomeadamente, os métodos de Jacobi, de Gauss-Seidel, e o Gradiente Conjugado. A renderização da água ou das suas velocidades, dos obstáculos em movimento, dos obstáculos estáticos, e do mundo foram feitas utilizando Vertex Buffer Objects (VBOs). Na versão da CPU foram utilizados OpenGL VBOs padrão, enquanto que na versão da GPU foram utilizados OpenGL-CUDA VBOs e OpenGL VBOs padrão.

Supervisor: Prof. Dr. Abel Gomes, DI, UBI

Real-time 3D Rendering of Water using CUDA

Author: Gonçalo Nuno Paiva Amador
Student Number: M1420
Email: a14722@ubi.pt

Abstract

This thesis addresses the real-time simulation of 3D water, both on the CPU and on the GPU. The stable fluids method is extended to 3D, and implemented both on the CPU and on the GPU. The GPU-based implementation is done using the NVIDIA Compute Unified Device Architecture API (Application Programming Interface), shortly CUDA. The stable fluids method requires the use of an iterative sparse linear system solver. Therefore, three solvers were implemented on both CPU and GPU, namely Jacobi, Gauss-Seidel, and Conjugate Gradient solvers. Rendering of water or its velocities, of the moving obstacles, of the static obstacles, and of the world are done using Vertex Buffer Objects (VBOs). In the CPU-based version standard OpenGL VBOs are used, while on the GPU-based version OpenGL-CUDA interoperability VBOs and standard OpenGL VBOs are used.

Supervisor: Prof. Dr. Abel Gomes, DI, UBI

Preface

I would like to dedicate this thesis to my supervisor, to my parents, to my brother, to my dear sister (not family someone special that I address in that manner), to my friends, and to Rex (my pet iguana).

To my supervisor Abel Gomes for his rigorous (and I really mean rigorous) guidance to every step of my thesis. Also by every help he gave in the several doubts I had in Computer Graphics, CUDA and natural phenomena simulation.

To my parents for making many efforts and sacrifices along the years of my life, that (among many other things) allowed me to conclude my bachelors degree. My brother for being in most aspects the opposite of me, and in doing so it inspires me every day to keep working and fighting for my objectives.

To my friends Sérgio da Piedade for the discussions we had on both our theses, VBOs, and CUDA programming in general, act who gave me many ideas on how not to do things. To my other friends Sara Fernandes, Ana Sofia and Ana Inês Rodrigues for the daily talks about anything other than my thesis, that allowed me tackle problems from different directions. To my friends Alex Cardoso and Ricardo S. Alexandre for annoying me and Sérgio to a point of making us want to work just not to put up with them. And finally but not least to my fried João for the many conversations on life facts and peanuts, and laughs (mostly for out of the context stuff), that were an indirect motivator factor.

To my dear sister for being simply herself.

To my pet iguana for being a green reptile, and for eating all those annoying flies that took my concentration away during the writing of this thesis.

Contents

Preface ix

Contents xi

List of Tables xiii

List of Figures xv

List of Algorithms xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Scheduling the Research Work . . . 2
- 1.4 Contributions . . . 3
- 1.5 Organization of the Thesis . . . 3
- 1.6 Target Audience . . . 4

2 Computational Water Models: The State-of-the-Art 5
- 2.1 Computational Fluid Dynamics and Computer Graphics . . . 5
  - 2.1.1 Off-line and Real-time Water Simulations . . . 6
    - 2.1.1.1 Off-line Simulations . . . 6
    - 2.1.1.2 Real-time Simulations . . . 7
- 2.2 Water Simulation Methods . . . 7
  - 2.2.1 Procedural Method . . . 7
  - 2.2.2 Lagrangian Method . . . 8
  - 2.2.3 Eulerian Method . . . 10
  - 2.2.4 Lattice Boltzmann Method (LBM) . . . 16
- 2.3 Final Remarks . . . 19

3 Water Simulation **21**
3.1 Stable Fluids 21
3.1.1 Add Force (f term in Eq. 3.3 and S term in Eq. 3.4) 22
3.1.2 Advection 22
3.1.3 Diffusion ($v\nabla^2 u$ term in Eq. 3.3 and $k\nabla^2$ term in Eq. 3.4) 23
3.1.3.1 Jacobi and Gauss-Seidel Algorithms 25
3.1.3.2 Conjugate Gradient Algorithm 27
3.1.4 Move ($-(u.\nabla)\,u$ term in Eq. 3.3 and $-(u.\nabla)\,\rho$ term in Eq 3.4) . . 29
3.2 CPU Implementation 30
3.2.1 Extension to 3D 30
3.2.2 Control User Interface 31
3.2.3 Conjugate Gradient Solver 32
3.3 GPU Implementation 33
3.3.1 NVIDIA Compute Unified Device Architecture (CUDA) 35
3.3.2 CPU to GPU version 36
3.3.3 Jacobi and Gauss-Seidel Solvers 38
3.3.4 Conjugate Gradient Solver 39
3.4 Final Remarks 41

4 Water Visualization **45**
4.1 OpenGL VBOs 47
4.2 CUDA-OpenGL Interoperability VBOs 48
4.3 Final Remarks 50

5 Experimental Results **51**
5.1 Computational Performance 51
5.2 Visual Performance 55
5.3 Final Remarks 55

6 Conclusions and Future Work **59**
6.1 Contributions 60
6.2 Future work 60

Bibliography **63**

A Glossary **69**

B CPU-based version source code **73**

C GPU-based version source code **81**

List of Tables

5.1 Frame rate of each solver in both versions, while filling the container. 52
5.2 Frame rate of each solver in both versions, after filling the container. 52
5.3 Time taken by one simulation step for each solver in both versions, while filling the container. 53
5.4 Time taken by one simulation step for each solver in both versions, after filling the container. 53
5.5 Timeouts resulting from device global memory access latency. 54

List of Figures

2.1 A CPU-based SPH fluid simulator running with round 4000 particles (available at `http://www.rchoetzlein.com/art/`). 11
2.2 NVIDIA GPU's version of a SPH fluid simulator with rendering (possibly fast marching cubes, or ray-casting, or ray-marching). 11
2.3 NVIDIA GPU's version of a SPH fluid simulator with no rendering, running with around 61000 particles. 12
2.4 A cell of the 2D MAC-grid (on the left); A cell of the 3D MAC-grid (on the right). 13
2.5 Heightmap representation of a water surface. 13
2.6 Heightfield based simulation. 15
2.7 NVIDIA fluids (GL version). 16
2.8 The most commonly used LBM models, D2F9 (left) and D3F19 (right). 18
2.9 Overview of the stream and collide steps. 18

3.1 Advection computation. 24
3.2 2D grid (left) represented by a 1Darray (right). 25
3.3 Cell interaction with its direct neighbours. 25
3.4 The sparse linear system to solve (for a 2^2 fluid simulation grid). 26
3.5 Gauss-Seidel red black pattern for a 2D grid. 28
3.6 Internal (red dots) and moving boundaries (green dots) representation. 31
3.7 CUDA work flow model. 36
3.8 CUDA memory model. 37

4.1 Screenshot of the GPU-based version, showing only velocity and bounds. . . . 45
4.2 Visible faces of the world. 47

5.1 Frame rate of each solver in both versions, while filling the container. 52
5.2 Frame rate of each solver in both versions, after filling the container. 52
5.3 Time taken by one simulation step for each solver in both versions, while filling the container. 53
5.4 Time taken by one simulation step for each solver in both versions, after filling the container. 53

5.5 Timeouts resulting from device global memory access latency. 55
5.6 CPU-based version using respectively Jacobi, Gauss-Seidel, and Conjugate Gradient solvers. 56
5.7 GPU-based version using respectively Jacobi, Gauss-Seidel, and Conjugate Gradient solvers. 57

List of Algorithms

1 NS fluid simulator. 22
2 Jacobi relaxation. 27
3 Gauss-Seidel relaxation. 27
4 Gauss-Seidel red black. 28
5 Conjugate Gradient method. 29
6 Control User Interface. 32
7 Set the initial values of r and p. 33
8 Set the initial value of ρ (dot product). 33
9 Update q. 34
10 Update x and r. 34
11 Update p. 34
12 Jacobi GPU kernel. 39
13 Gauss-Seidel red black GPU kernel. 40
14 GPU-based set the initial values of r and p (part 1 of _cg GPU kernel). . . . 41
15 GPU-based set the initial values of ρ and ρ_0 (part 2 of _cg GPU kernel). . . 42
16 Update q (part 3 of _cg GPU kernel). 42
17 Update α (part 4 of _cg GPU kernel). 43
18 Update x, r (part 5 of _cg GPU kernel). 43
19 Update β, and p (part 6 of _cg GPU kernel). 44
20 Rendering of the scene. 46
21 OpenGL VBOs methodology. 49
22 CUDA-OpenGL Interoperability VBOs methodology. 49

Chapter 1

Introduction

Natural phenomena simulation is today an area of great interest, mainly for the game and movie industries. Game industry needs visually accurate simulations at interactive rates (around 60 frames per second). Movie industry has no need for interactive rates. However it requires high visually accurate simulations, and the ability to integrate specific effects (i.e. giant wave destroying city, milk being poured into a glass coup, etc) intuitively, required from the movie/animation script.

Specifically, we have four types of methods (i.e. procedural methods, Lagrangian methods, Eulerian methods, and Lattice Boltzmann method) for water simulation (reviewed in the next Chapter). From these four methods, procedural methods are the only ones not physically-based (i.e. it does not use or approximate real physical models). The remaining three methods use (Lagrangian and Eulerian methods) or approximate (Lattice Boltzmann method) real physical models.

Amongst the Eulerian-based methods for full 3D simulations, the stable fluids is the most known because of its unconditional stability (i.e. it can use large times steps). However, its usage in 3D simulations is very limited when running on the CPU (see Chapters 2 to 5 for more details). Stable fluids has been implemented in 2D on GPU using CUDA. However, this implementation does not allow internal or moving boundaries. Also, on the GPU a stable fluids 3D version was implemented, using the Cg language. This last version allows internal and moving boundaries.

1.1 Motivation

Water simulation and rendering is a complex field of study. Since the launch of NVIDIA CUDA in 2007, the topic of graphics cards programming became more active than ever. Other physically-based methods to simulate water (i.e. Lagrangian methods, some Eulerian methods, Lattice Boltzmann method) have been already implemented on GPU using CUDA. However, at our best knowledge, there is no 3D GPU-based implementation of the stable fluids method using CUDA. This was the most significant motivating factor for pursuing the research work behind this thesis. Also to account for my personal interest in computer graphics and parallel computation.

1.2 Problem Statement

The stable fluids method was introduced by Stam in 2D [59]. It is a computationally heavy problem in 3D. Aside from the method computations, one must add the rendering time, also very expensive. The rendering of water is a very delicate subject since we have a deformable mesh varying over time. When simulating water, the only available information are the spatial positions occupied by water. To render water realistically (other quicker alternatives exist but with inferior visual results), a volume reconstruction technique must be used. However, volume reconstruction techniques are expensive (i.e. time consuming). Therefore, speeding up either the physically-based method, or the rendering time, or both is of the utmost importance for real-time simulations. If the physically-based method is quicker, then more time for rendering and more realistic techniques can be used without degradation of overall performance. A quicker physically-based method, in the case of the stable fluids, might also allow the usage of bigger grid sizes. Bigger grid sizes mean better looking water simulations, specifically in the interaction of water with obstacles (i.e. boundaries, moving objects, other liquids, etc). Better volume reconstruction techniques (i.e. quicker but maintaining quality of current techniques) or speeding up the previous techniques is also of great interest for real-time water simulations.

Previous 2D and 3D GPU shader-based versions of stable fluids have already been implemented, but a question arises:

Can we achieve an improved 3D real-time version of stable fluids using CUDA?

Answering this question is the reason for being and the focus of this thesis.

1.3 Scheduling the Research Work

In the research work that has led to this dissertation we have been guided by the following schedule:

- An continuous comprehensive study of all the encountered literature on the subject of fluids (water, fire and smoke) rendering and simulation.
- The implementation of a CPU-based 3D version of the stable fluids method.
- Implementation of several sparse linear systems solvers (i.e. Conjugate Gradient and Jacobi), both on the GPU and CPU-based versions.
- The incorporation of the Jacobi and Conjugate Gradient linear solvers, into the CPU-based 3D version of the stable fluids method.
- Port the CPU-based version to an GPU-based CUDA version.
- The writing of this thesis.

1.4 Contributions

As far as we know, there is no 3D version of stable fluids, that allows internal and moving boundaries, implemented in CUDA. Thus 3D stable fluids is the only method (among the four existing methods) to simulate water that is not implemented in CUDA. As expected, like the 3D GPU shader-based version,this CUDA implementation achieves real-time performance. Therefore, the GPU-based implementation of the 3D stable fluids is the primary contribution of this thesis.

Vertex Buffer Objects (VBOs) are widely used in the computer graphics community. However, there is still some difficulty to find free proper extensive tutorials on using OpenGL VBOs, in spite of being relatively easy to find source code on the subject. OpenGL-CUDA interoperability allows the upload of VBOs data to be done by the graphics card. However, since CUDA is a relatively new architecture, there is not much documentation on using OpenGL-CUDA VBOs. Therefore, another contribution of this thesis is an overview, between the usage of OpenGL-CUDA VBOs (on the GPU) and OpenGL VBOs (on the CPU) in graphics applications.

1.5 Organization of the Thesis

This thesis is organized as follows:

- *Chapter 1* is the current chapter. It introduces the field of real time water rendering on the GPU using CUDA. This chapter lays out the addressed problem, its motivation, and the reasons that justify its importance. The intended target audience is also mentioned in this chapter.

- *Chapter 2* provides an overview of the existing methods to simulate water. An insight of each method is given to the reader, in order to facilitate his/her understanding of the subject. Also, in this chapter, the reader will find information about the related research work found in the literature.

- *Chapter 3* provides a more detailed description of the stable fluids method. This chapter also described the water simulation implementations, carried out in this thesis research work, both CPU and GPU implementations.

- *Chapter 4* describes how the visualization of the water or its velocities, of the moving boundaries, of the internal boundaries, and of the world, was achieved. An explanation on the usage of vertex buffer objects both for the CPU and GPU is given in this chapter.

- *Chapter 5* presents an comparative analysis of performance of both simulations.

- *Chapter 6* concludes the thesis and presents points for further improvement in the future.

1.6 Target Audience

The scope of this dissertation lies in computer graphics. This thesis is also of interest to programmers and researchers in the game industry, who wish or need to implement real-time water in games and virtual worlds. This thesis is particularly relevant to researchers who study natural phenomena simulation, namely water simulation.

Chapter 2

Computational Water Models: The State-of-the-Art

The first mathematical models that describe the movement or flow of fluids (i.e. liquids and gases movement) go back to 1700's, and are due to Sir Isaac Newton. Among the earliest models for incompressible/compressible flows we find the 1730's famous Bernoulli equation for blood flows. In the early 1800's G.G. Stokes (in England) and M. Navier (in France) independently derived the equations known as Navier-Stokes (NS) equations, which describe the movement of viscous fluids. These equations describe the relation between pressure, temperature, and density of a moving fluid (i.e. water, fire, smoke, etc). With the advent of the digital computers in 1940's, the doors for the possibility of computer fluids simulation were open. Since then fluid models have been extensively studied in the fields of Computational Fluid Dynamics and Computer Graphics.

2.1 Computational Fluid Dynamics and Computer Graphics

There are mainly two fields with interest in water simulation: Computational Fluid Dynamics (CFD) and Computer Graphics (CG). CFD is a branch of fluid mechanics that, by the usage of numerical methods and algorithms, tries to solve and analyse problems involving fluid flows as necessary in engineering analysis (i.e. fluid problems in mechanical engineering, civil engineering, etc). CFD aims, by the usage of simulation, to solve or as best as possible give an approximation to the solution, of real-life fluid problems (Component Modelling, System Analysis, Design Optimization, Design Verification, etc) formulated in terms of systems of equations. The more interested reader in the field of CFD is referred to the CFD terms dictionary [58] and the 1st edition of "Fundamentals of Computational Fluid Dynamics" [37].

On the other hand, the CG purpose is to generate appealing, convincing, plausible simulations of fluid effects, mostly for the game and movie industries. Therefore, CG water models are simplifications (i.e. simplified NS equations) or approximations (i.e. Lattice Boltzmann method) of real models, or non-physically-based approaches (i.e. procedural water). Physically-based water models used in CG are simplified versions of CFD water

models. The simplifications intend to make the methods faster, but maintaining their visual quality (i.e. the fluid-like behaviour).

2.1.1 Off-line and Real-time Water Simulations

In computer graphics, there are two categories of water simulation:

- *Off-line*: used in movies/animations as special effects.
- *Real-time*: used in games and virtual reality.

Models used in real-time can be always used in off-line, but the opposite is not true. Off-line simulations have less time restrictions than real-time simulations, but require higher visual quality. Therefore, the distinction between off-line and real-time water simulation models lies in the parameters of the simulation (i.e. smaller simulation areas for real-time methods, less number of particles in particle-based methods, etc), and the rendering techniques used (i.e. off-line simulations can resort to computational intensive volume reconstruction techniques). Off-line simulations can use efficient volume reconstruction techniques to achieve great visual accuracy (i.e. marching cubes [38], ray-tracing [22], ray-casting [12] [35], etc). However, real-time simulations need faster techniques to visualize water at the cost of less visual accuracy if necessary.

A good reference to understand the core of fluid simulation, either for real-time or off-line can be found in Bridson et al. work [7, 8]. In this work the reader might find many important concepts, considerations and explanation of the cores of many of the fluid simulation issues and solving methodologies. The work of Foster and Metaxas [18] presents a rapid method for computing the motion and appearance of fluids, allowing users model and interact in real time with virtual water.

2.1.1.1 Off-line Simulations

Off-line simulations aim at achieving quality. They are not under the strong time restrictions of real-time simulation. The simulation should be done in acceptable time (i.e. less than the average life longevity of an human being), and must allow an intuitive and quick generation of user intended localized specific water effects (i.e. waves at a given point in space, splash with foam in a non-boundary region, etc).

Off-line simulations control have been addressed in several papers. Foster and Fedkiw introduced an efficient pressure iteration technique to the previous Navier-Stokes-based methods [16]. Basically, the proposed a modified semi-Lagrangian method and a new approach to calculate fluid flow around objects. Also, they introduced significant contributions to the simulation and control of three dimensional fluid simulations by using a hybrid liquid volume model that combines implicit surfaces and massless marker particles. Later, Enright et al. [13] improved the visually accuracy of Foster's and Fedkiw's work, but at the cost of more time. Enright et al. also fixed the volume loss problem in the simulation of very complex scenes using the particle level-set method. McNamara et al. introduced a novel method for controlling physics-based off-line fluid simulations [40]. This method uses the adjoint method technique. It was the first method to achieve the full control of free-surface

liquids. McNamara et al. also describe a new set of control parameters for liquids. For off-line simulation, and to allow more realistic water effects (e.g. splash with foam), Génevaux et al. [21] proposed an interface between Eulerian NS fluid simulator and a Lagrangian spring-mass solids simulator.

2.1.1.2 Real-time Simulations

Real-time simulations have more restrictive requirements, than off-line simulations, namely:

- *Real-time computaion*. They have to be done as quickly as possible within 15-30 milliseconds.
- *Low memory consumption*. Memory has to be shared by several tasks running on a single computer.
- *Stability*. Games run at fixed frame rates, so it is imperative that the algorithm remains stable unconditionally for a given time step (i.e. unlike off-line simulations, no adaptive time-stepping can be used in real-time simulations, because stability problems may arise).
- *Plausibility*. The obtained visual result must look and behave as in the real world, i.e. convince the observer.

In the literature, we find some techniques and algorithms to simulate water in real-time. Belyaev gave an explanatory evaluation of previous algorithms for real-time simulation of water surface with waves [5]. Also a good overview of the current algorithms for real-time physics simulation, not only water simulation, can be found in Müller et al. work [47].

2.2 Water Simulation Methods

There are mainly four methods to simulate water: procedural method, Lattice Boltzmann method, Lagrangian methods, and Eulerian methods. Procedural water simulates the physical effect, not its causes. Lagrangian methods (i.e. particle systems) are ideal for simulating splash, foam, jets, etc. for small amounts of water. Eulerian methods can be either 2.5D (i.e. heightfields and shallow waters) or 3D (full 3D NS). Eulerian 2.5D methods are used for modelling surfaces of large water bodies (i.e. lakes, sea, etc). Lattice Boltzmann method can be used in 2D or 3D simulations.

2.2.1 Procedural Method

Among the water simulation methods, procedural water is the only one that does not make usage of a physical approach. Because no relation between pressure, viscosity, forces or velocities is considered in this method (not a physical approach), internal boundaries and objects (either in the surface or inside the fluid) have to be treated locally. This method focus on the desired effect, not on its causes. This method was one of the first methods to

simulate water in CG. Also, it is the only method used exclusively in CG, since it does not follow any physics laws. Consequently, it has no practical use in CFD.

Ocean water surface (near the shore or not) is described by a set of waves. Waves can be divided into classes: tides, seismic waves, internal waves and wind waves (surface gravity waves and surface tension or capillary waves). See Peachey's work for further details [50]. Wave shape can be described by a function or set of functions. The simplest surface gravity wave, for example, is described by the following parametric equation:

$$f(x,t) = A\cos\left(\frac{2\pi(x - Ct)}{L}\right) \tag{2.1}$$

were x is the distance from an origin point, C the wave propagation speed, A the amplitude of the wave, t is the time instant and L is the wavelength. Eq. 2.1 can be used to determine where each water element is at a given time, after departing from an user given starting position. Since the starting point can be variable, for a given set of functions, we can make various waves that go in different directions, empowering this method with an extremely high controllability. We can use specific approximated functions to draw different wave shapes (described by Fournier and Reeves [19]). Effects such as foam and breaking waves are not supported in this method. These effects cannot be represented from continuous functions, but they can be simulated as explained Jeschke et al. work [28].

Procedural methods are based on a known set of mathematical functions that change over time, as those described by Eq. 2.1. These functions are used to perform a generalized parametric water surface representation. An update of the water surface consists in the generation of the new values for the parameters. These new values result from variations in the values of the parametric function parameters (i.e. time, amplitude of the wave, propagation speed, distance from the origin point). It is clear that rendering this kind of water surface requires its triangulation. Other water simulation methods have to update the physical model values and then reconstruct the geometry to be rendered. Therefore, water simulation procedural methods are computationally less expensive methods than any other physically-based water simulation methods used in real-time and off-line simulations.

Parametric representation of water surfaces (procedural approach) for off-line water simulations was addressed by Peachey [50]. This is an introductory paper to the subject of ocean waves modelling, being this work exclusively for off-line purposes. Peachey attempts to model realistic wave behaviour, including the change in wave behaviour as a function of the depth of the water. Ts'o and Barsky [70] extended Peachey's work, with a more detailed explanation of the previous method. A good introduction to procedural water was presented by Jeschke et al. [28]. More recent work in ocean surface simulation has been addressed by Tessendorf's [65] and Fréchot's [20].

2.2.2 Lagrangian Method

In nature, all matter is made of small elements (atoms), and these elements are made of smaller elements (electrons, neutrons, protons), which are in turn made of even smaller elements (an so on). A portion of water may posses billions of smaller water elements. To

simulate water in real-time using such number of elements on current computers is practically impossible.

Therefore, while keeping the same logic (i.e. water is made of smaller elements designated particles) the number of used particles is extremely reduced when compared with real world water. That is, each particle represents a water element in a macroscopic way. The Lagrangian approach consists in tracking the movement of a set of particles (i.e. particle systems). From an Lagrangian viewpoint, each particle occupies a spatial position and has an velocity associated to it. In some cases, particles might also have a time to live, mass, phase, and information about external forces that might influence their behaviour. The phase property is very important in liquid-liquid interaction, provided that it ensures that materials (i.e. liquids) with different densities do not mix.

Particle systems were introduced to computer graphics by Reeves [52], but the first particle system for fluid simulation was presented by Miller and Pearce [42]. Müller et al. [44] introduced the first particle system to simulate fluids with free surfaces using SPH. Particle systems allow, specifically for water simulations, several effects that, in most cases, are not possible using grid approaches. For example, splashing and pooling water effects were made possible in particle systems with Stein and Max work [63]; Takahashi et al. work [64] showed how to use particle systems to simulate splashes and foam; Greenwood and House [24] contributed with the addition of bubbles to off-line fluid simulations; and Müller et al. [46] added boiling water, trapped air and the dynamics of a lava lamp to the set of special effects enabled by particle systems.

Particle systems also are adequate to simulate solid-liquid and liquid-liquid interactions. For example, Müller et al. [45] studied the interaction of fluids with deformable solids in real-time using a particle-system. Later, Müller et al. [46] presented a new method to model fluid-fluid interaction based on the SPH method. Schläfli work [53] addressed solid-liquid interaction using SPH. However, Schläfli proposed a different approach, for the representation of solid objects. Schläfli considered solid objects and fluids as particles of different kinds, within the same particle system.

Traditional particle systems consider interactions of each particle with all existing particles and objects/obstacles/boundaries in the system. If such approach is taken, specifically for real-time water simulations, the processing time is unacceptable. To significantly reduce the processing time (in an attempt to get real-time performance) and to model water's shape, two models were proposed: the Smoothed Particle Hydrodynamics (SPH) method and the Moving Particle Semi-implicit method (MPS). The interested reader is referred to Li and Liu work [36] for more details on mesh free and particle methods and its applications (in fields other than computer graphics).

MPS and SPH methods approximate Partial Differential Equations (PDEs), as those used to solve in the NS equations. Both methods differ in the way the solution to the referred PDEs is obtained. MPS uses a semi-implicit prediction-correction process, while SPH uses a fully explicit prediction-correction process. Also, MPS method does not consider the gradient of the kernel function. Instead, MPS method uses a simplified differential operator, which is solely based on a local weighted averaging process. MPS method was addressed in computer graphics by Premoze et al. work [51].

SPH originally appeared in CFD as a method for fluid flow simulation in several re-

search fields such as, for example, astrophysics, ballistics, vulcanology and oceanology. It is a mesh-free Lagrangian method. It was originally designed to simulate the dynamics of stars, but it is also quite useful in some fluid simulations (i.e. water, fire, explosions, etc). As with other particle system approaches, SPH uses the NS equations to describe the behaviour of individual elements of water (i.e. particles). That is, SPH is a physically-based system, so realistic off-line and real-time water simulations can be, in principle, achieved. In fact, SPH allows for remarkable results in real-time, even if particle-particle interactions are not considered.

Both MPS and SPH methods assign a smoothing length to each particle, allowing the simulation to automatically adapt itself to local conditions. This smoothing length is associated with a smooth kernel. This length is the area of interaction of the particle with its neighbours.

SPH and MPS methods have several advantages over grid methods, namely:

- Mass conservation guarantees, with no extra computation requirements.
- Pressure is computed from weighted contributions of neighbour particles making it faster than methods that solve linear systems of equations.
- Both make possible to simulate effects (out of reach of purely grid-based methods) such as drops of water when water splashes, and small details in fluid simulations (i.e. bubbles, foam, etc).

Nevertheless, SPH and MPS methods also have drawbacks when compared to grid methods, namely:

- Expensive rendering, since both methods use polygonization techniques such as metaballs and marching cubes.
- Both methods require an high number of particles to produce simulations of equivalent resolution to grid-based methods.

To better understand the mathematical/physical fundamentals of the SPH method the reader is referred to Schlatter work [54]. The source code for a SPH CPU-based version (see Fig. 2.1) is available online at `http://www.rchoetzlein.com/art/`. There is also a port of SPH for the GPU (PhysX fluids demo from NVIDIA), available at `http://www.nvidia.com`, however with no access to the source code (see Fig. 2.2 and 2.3). The GPU version clearly gains in performance over the CPU version (around 4000 particles to 61000 particles each time step). Besides, also the GPU version includes various types of rendering (such is not the case of the CPU-based version).

2.2.3 Eulerian Method

Similar to the Lagrangian method, the Eulerian approach tracks variations of the momentum (i.e. movement of the fluid). But, unlike Lagrangian methods, the Eulerian method is a mesh-based method (or grid-based method) for fluids. Thus, instead of following each individual particle variation, it discretizes space into a limited grid of cells and analyses

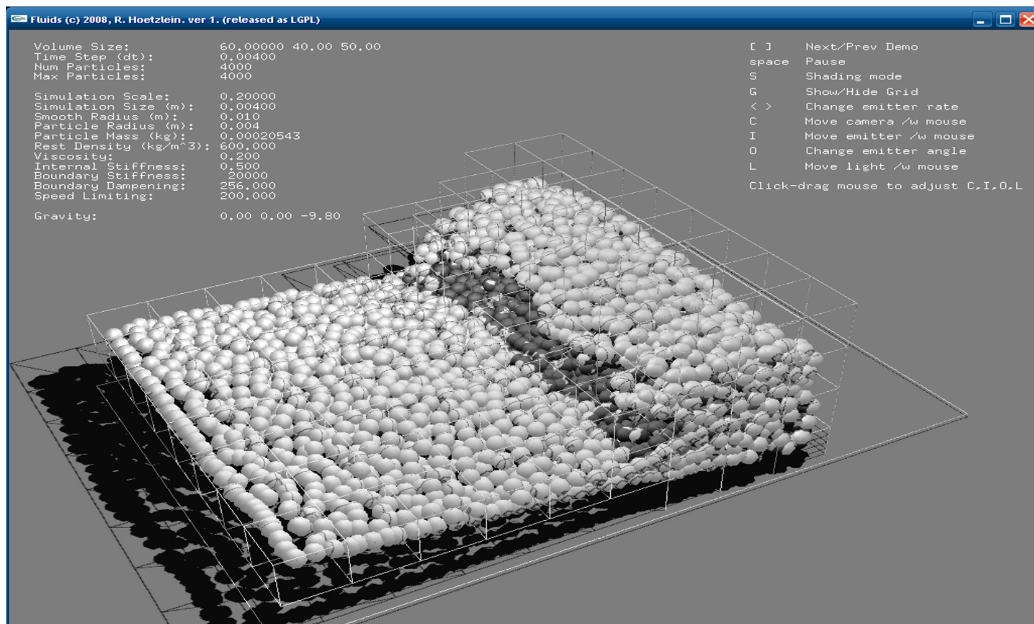

Figure 2.1: A CPU-based SPH fluid simulator running with round 4000 particles (available at `http://www.rchoetzlein.com/art/`).

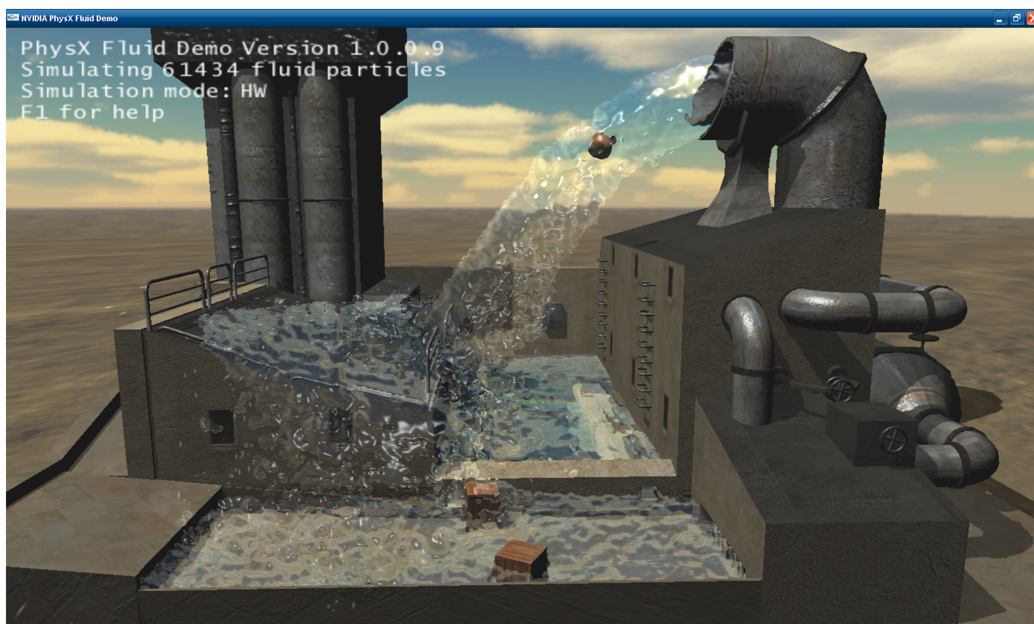

Figure 2.2: NVIDIA GPU's version of a SPH fluid simulator with rendering (possibly fast marching cubes, or ray-casting, or ray-marching).

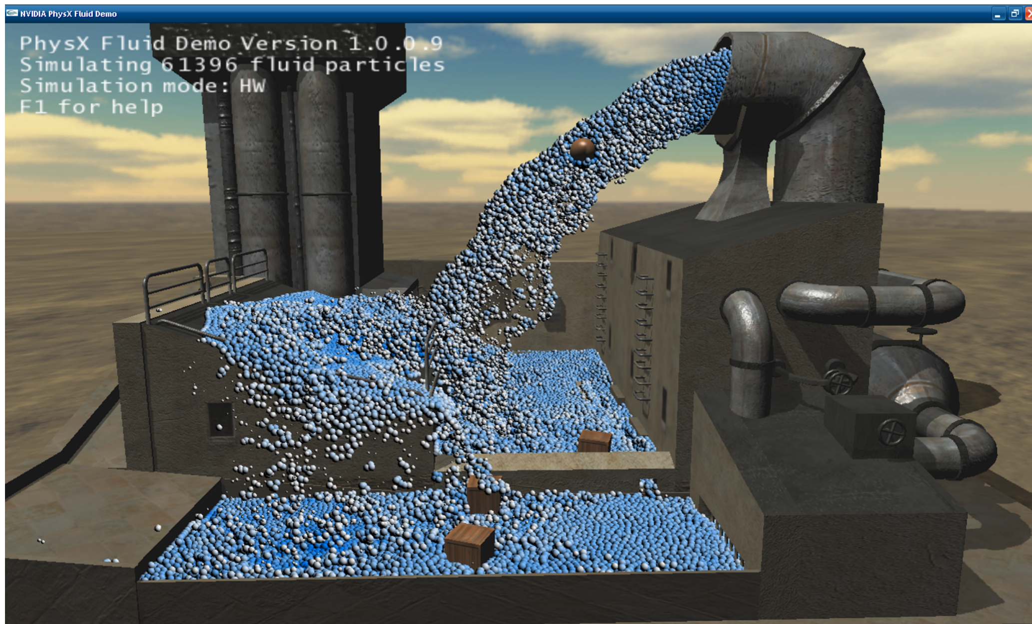

Figure 2.3: NVIDIA GPU's version of a SPH fluid simulator with no rendering, running with around 61000 particles.

the variations of pressure, velocities, density, mass, etc. within each grid cell. The major disadvantage of the Eulerian approach is its memory space requirements because many of the grid cells are unnecessary most of the time. To alleviate this problem octree structures can be used, as in Losasso et al. [39]. However, using this kind of data structure is not so intuitive as regular grid approaches. Besides, it increases the memory access time to retrieve octree cell data.

There are mainly two kinds of grids: the coarse grid and the Marker-and-Cell (MAC) grid. In a coarse grid, the values of pressure, velocities, density, mass, etc. of each cell are stored at the cell center. In a MAC grid, for each grid cell, the pressure is sampled at the cell center, while velocities are split into their Cartesian components, and sampled at the center of the cell faces that are perpendicular to their Cartesian axis (see Fig. 2.4).

The MAC grid method was introduced by Harlow and Welch [25] as a way of solving incompressible flows problems (other uses are not recommended). The method introduces the so-called "marker particles" (hence the designation of the method) that are used to identify which grid cells contain liquid or not.

In the literature, we find two types of Eulerian methods:

- *Hybrid method of heightfields and Shallow Water Equations (SWE)*. This method is used to simulate a large water surface such as the ocean.

- *3D Navier-Stokes equations (NS)*. This method is used when heightfields and SWE cannot be used (only for effects within the possibilities of grid-based methods).

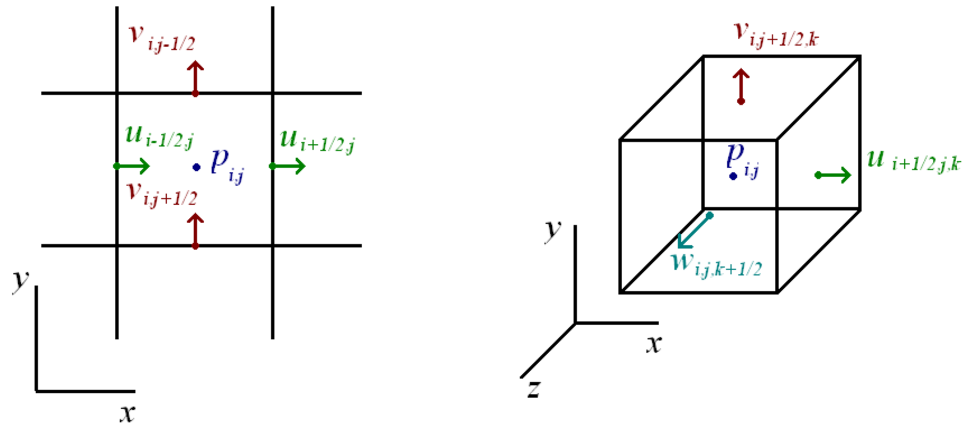

Figure 2.4: A cell of the 2D MAC-grid (on the left); A cell of the 3D MAC-grid (on the right).

Whenever possible, heightfields combined with the SWE are used, instead of 3D NS, for the simple reason that they are less computationally demanding. In fact, solving SWE reduces the problem of 3D water simulation on a grid to 2D water simulation on a surface mesh. Miklós [41] also show how to use heightfields to simulate pouring water.

In the heightfields approach, the fluid surface is represented as a 2D function. The height values of the water surface are stored in a 2D matrix, usually known as heightmap (see Fig. 2.5).

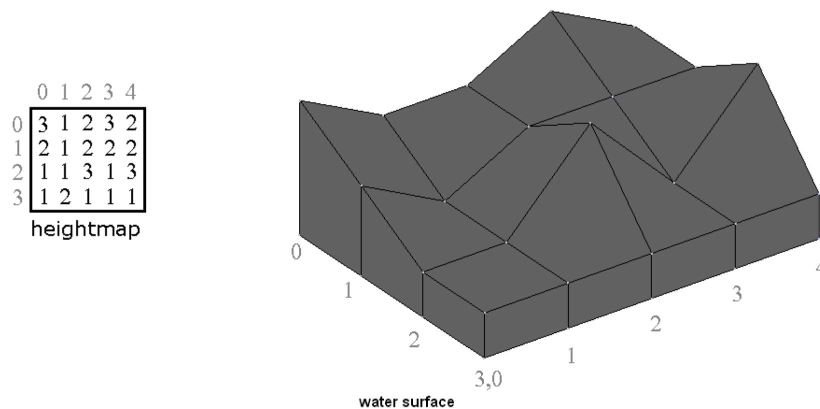

Figure 2.5: Heightmap representation of a water surface.

The SWE are a simplified version of the NS equations, and are adequate for the case of small depth water (i.e. ocean near shore, rivers, etc). SWE describe the movement of fluids by a set of 2D equations (Eqs. 2.2 and 2.3), and resort to heightfields to represent the water surface.

$$\frac{\partial \eta}{\partial t} + (\nabla \eta)\, v = -\eta \nabla \cdot v \tag{2.2}$$

$$\frac{\partial v}{\partial t} + (\nabla v)\, v = a_n \nabla h \tag{2.3}$$

where a_n denotes a vertical acceleration of the fluid, η denotes the height of the fluid above ground, t is time, v is the velocity of the fluid in the horizontal plane, and h denotes the height of the fluid above zero-level. Then, the SWE are used to recalculate the fluid evolution, the heightmap is updated, and the water surface is rendered.

SWE can either use an explicit or implicit (the differences and motivations for each are addressed further ahead in this chapter) time integration scheme. However, the explicit approach makes the solution of the SWE considerably more simple. Layton and Panne [34] presented a semi-Lagrangian advection with an implicit time integration scheme.

Using heightfields combined with the SWE has two major drawbacks. First, it is not possible to simulate breaking waves. Second, when using explicit integration these methods are only conditionally stable (i.e. they need a specific time interval depending on the intended simulation). The first drawback however can somewhat be reduced by interfacing this method with a particle system, as it was proposed by Holmberg and Wünsche [27] to model turbulent water. Kass and Miller were probably the first to approximate the full 3D Navier-Stokes equations by using heightfields combined with SWE [30]. O'Brien and Hodgins [49] extended Kass and Miller work by adding off-line splashing fluids. For liquids animation, Foster and Metaxas [17] combined the MAC method, described by Harlow and Welch, with heightfields and SWE. Johanson [29] analysed real-time water rendering using Microsoft Direct3D shading language HLSL (i.e. High Level Shading Language), for heightfield-based simulations. There is a GPU open-source CUDA-based version of the heightfields combined with SWE (see Fig. 2.6), which is part of CUDA SDK examples available at `http://www.nvidia.com/object/cuda_get.html`.

Often NS based-simulations use the Euler equations instead. Euler equations are the particular case of NS for inviscid liquids (i.e. without the pressure term). However, fluid simulations that use Euler equations still seem to posses viscosity. This fake viscosity is the result of the numerical error present in the simulator steps. Almgren et al. [3] simulated a fluid by calculating time-dependent incompressible Euler equations for inviscid flow. There are two kinds of solvers that use the Eulerian approach. One uses explicit integration, while the other uses implicit integration instead (the details of each implementation will be addressed in a further chapter). Most explicit integration works are based on Kass and Miller [30]. Explicit methods suffer from a limitation that forces the usage of very small time steps in order to avoid blowing up the simulation. Implicit integration was introduced by Jos Stam [59] method, also known as *stable fluids* method.

Since the appearance of the stable fluids method, much research work has been done with this method. To reduce fluid dissipation (i.e. loss of fluid outer-limits mass) caused by

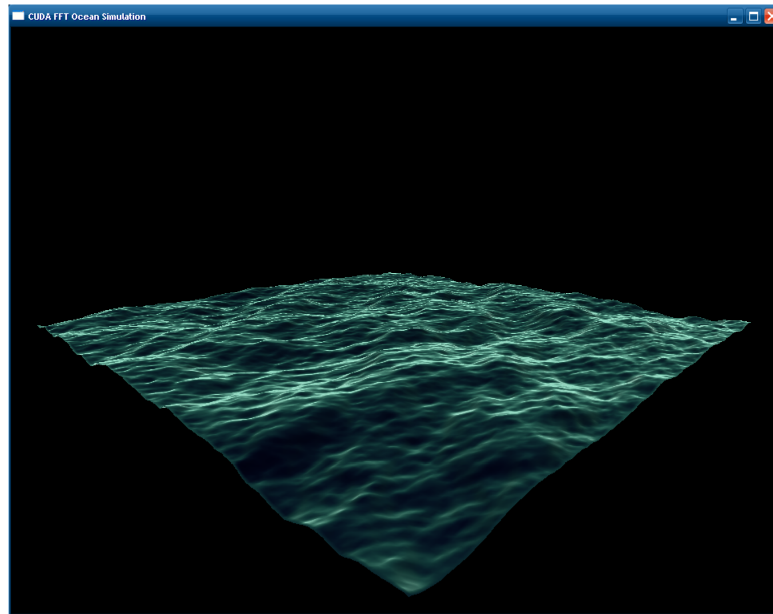

Figure 2.6: Heightfield based simulation.

numerical error in the semi-Lagrangian advection, several solutions have been presented, namely vorticity confinement [14], vortex particles [56], and MacCormack advection [55]. Stable fluids require the resolution of two sparse linear systems (addressed in the following chapter). To solve the sparse linear systems for an 2D simulation, both the Fast Fourier Transform (FFT) in Stam's [60] and the Gauss-Seidel relaxation in Stam's [62] were used (both implementations run on the CPU). The Gauss-Seidel version source code is available online at `http://www.dgp.toronto.edu/~stam/reality/Research/pub.html`. The Gauss-Seidel version was the only one that could support internal and moving boundaries (tips in how to implement internal and moving boundaries are given in Stam's [62]). Later on, in 2005, Stam's Gauss-Seidel version was extended to 3D, also for the CPU, by Ash [4]. In 2007, Ash's 3D version of stable fluids, was implemented for C'Nedra (open source virtual reality framework) by Bongart [6]. This version also runs on the CPU. Recently, in 2008, Kim presented in [32] a full complexity and bandwidth analysis of Stam's stable fluids version [62]. Stam also addressed the simulation of fluid flows on smooth surfaces of arbitrary topology (a flow running on the surface of a geometry) using again the stable fluids method [61]. Carlson [9] addressed the problem of applying the Conjugated Gradient method to fluids simulations, and to liquid-solid interaction in grid-based approaches.

In 2005, Stam's stable fluids version was implemented on the GPU by Harris [26] using the Cg language. This version supported internal and moving boundaries, but used Jacobi relaxation instead of the Gauss-Seidel relaxation. In 2007, an extension to 3D of Harris'work was carried out by Keenan et al [31]. Still in 2007, when CUDA was released, Goodnight's OpenGL-CUFFT version of Stam's [60] became available [23] (see Fig. 2.7) .

The code of this implementation is still part of the CUDA SDK examples, and is available for download at `http://www.nvidia.com/object/cuda_get.html`.

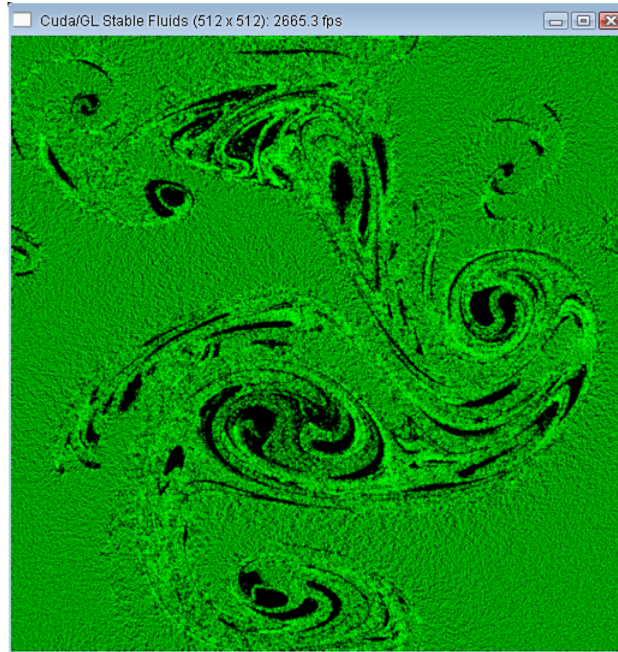

Figure 2.7: NVIDIA fluids (GL version).

The stable fluids method addressed the stability problem of previous solvers, namely Kass and Miller method in [30]. The Kass and Miller method could not be used for large time steps. Consequently, it was not possible to use it in real-time computer graphics applications. The reason behind this was related to the usage of explicit integration, done with the Jacobi solver, instead of stable fluids implicit integration. In spite of the limitations of Kass and Miller method, in 2004, there was a GPU Cg-based version of this method implemented by Noe [48]. This GPU version is used in real-time, thanks to the gains obtained from using the GPU.

2.2.4 Lattice Boltzmann Method (LBM)

LBM is the most recent water simulation method. Nevertheless, it has attracted interest from researchers in computational physics and in computer graphics. LBM has, due to its nature, several advantages over other conventional CFD methods, namely:

- It allows the incorporation of microscopic interactions when dealing with complex boundaries.
- It is easily parallelizable.

Other CFD methods numerically solve the conservation equations of macroscopic properties (i.e. mass, momentum, and energy). LBM models the fluid as a set of fictitious particles. These particles consecutively propagate and collide in a discrete lattice mesh. LBM is not a Lagrangian method, since it tests for particles interactions at each node of a grid. Each grid node consists in one particle, and only interactions with the particle's direct neighbours (i.e. direct neighbour grid cells) are considered.

LBM originated from the lattice gas automata (LGA) method studied in the field of statistical physics. LGA method is a simplified fictitious molecular dynamics model where all parameters are discrete (i.e. space, time, and particle velocities). LGA method is a model that describes gases in space, while LBM is a model that describes fluids such as water. LGA suffers from many drawbacks, namely (among others):

- Lack of Galilean invariance.
- Statistical noise.
- Exponential complexity for three-dimensional lattices.

LBM does not suffer from the same statistical noise as the LGA method. LBM might also be interpreted as a discrete-velocity Boltzmann equation. When applying the numerical methods to solve the system of partial differential equations a discrete map rises. This discrete mapping consists in performing consecutive propagations and collisions between fictitious particles (i.e. stream and collide steps). LBM approximates the NS equations with good accuracy an only requires a single pass over the computational grid per time step, which makes it adequate to parallelize in architectures such as programmable, multi-threaded GPU's. LBM is more memory-consuming that NS methods, but this issue can be overcome using grid compression.

In LBM, each lattice node is connected to its neighbours by a number of lattice velocities, 9 in 2D and 19 in 3D (commonly referred as D2Q9 and D3Q19 models), as shown in Fig. 2.8. Other configurations are possible, but these are the most common in the literature. The lattice velocity vectors take the following values $e_1 = (0,0,0)^T$, $e_{2,3} = (\pm1,0,0)^T$, $e_{4,5} = (0,\pm1,0)^T$, $e_{6,7} = (0,0,\pm1)^T$, $e_{8\ldots11} = (\pm1,\pm1,0)^T$, $e_{12\ldots15} = (0,\pm1,\pm1)^T$ and $e_{16\ldots19} = (\pm1,0,\pm1)^T$. The values of e_i, with $i = 1,\cdots,9$, are identical in the 2D and 3D models. In the LBM, all formulas are only dependent of particle distribution functions (DFs). A DF is a function with seven variables ($f(x,y,z,t,vx,vy,vz)$), which gives the number of particles that travel at an approximate velocity (vx,vy,vz) near a spatial position (x,y,z) at a given time (t).

At each node, there is either 0 or 1 particles moving in a particular lattice direction. At each time step, each particle will move to one of its neighbour nodes (i.e. propagation or streaming step). This movement or propagation of each particle results from advecting all DFs with their respective velocities. If more than one particle moves to the same node a collision occurs, which is then treated with a set of collision rules (i.e. all but one particle are moved to other neighbour nodes until a collision does not occur). This is the collision step. The LBM algorithm performs the stream and collide steps for each particle in each grid cell, as shown in Fig. 2.9.

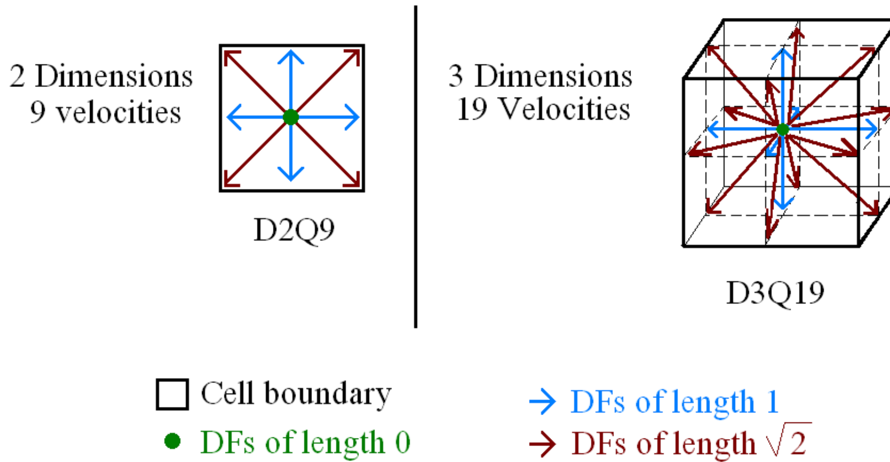

Figure 2.8: The most commonly used LBM models, D2F9 (left) and D3F19 (right).

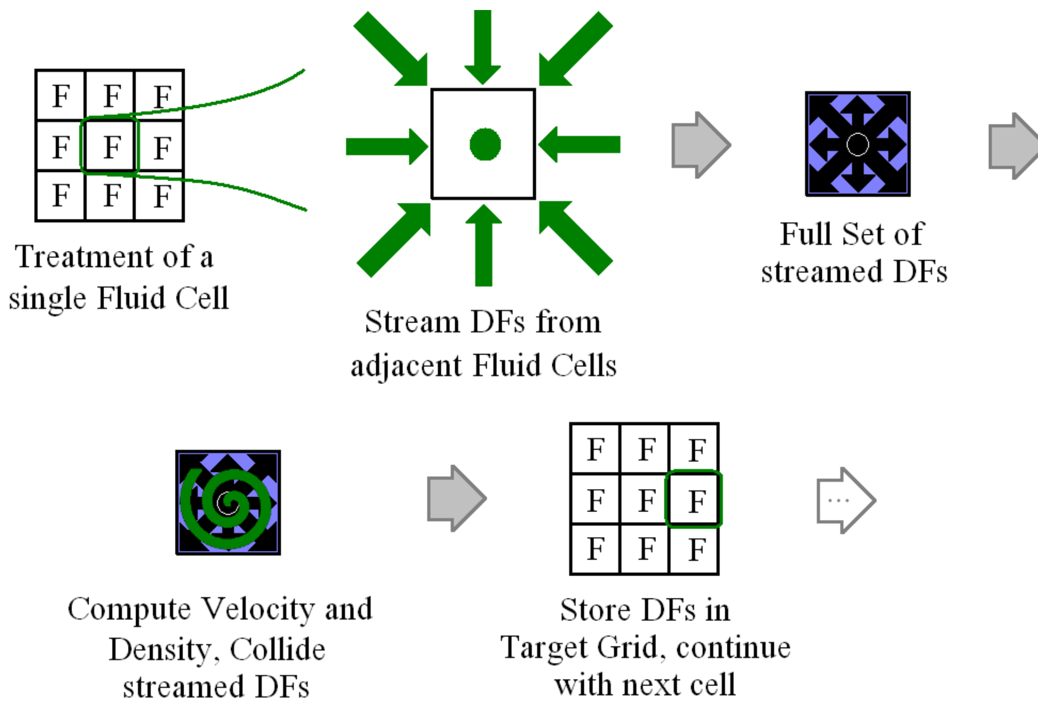

Figure 2.9: Overview of the stream and collide steps.

LBM has been addressed in many publications. A comprehensive study of the physics and mathematics of the LBM was given by Chen and Doolen [10]. Thürey and Rüde [67] presented an algorithm to perform free surface flow simulations using LBM on adaptive

grids in order to achieve better performance. Later Chentanez et al. [11] introduced a tetrahedral dynamic mesh LBM fluid simulation as an alternative to the classical a grid based methodology. Körner et al. [33] presented a 2D and 3D LBM for the treatment of free surface flows including gas diffusion, and a bubbles and foam algorithm. Zhao et al. [71] also addressed 3D fluid flow simulation using a locally-refined LBM. Thürey and Rüde [68] extended the possibilities of the LBM for fluid simulations, with an implementation of free surface Lattice-Boltzmann fluid simulations with and without level sets. LBM has also been implemented in several parallel architectures, namely in OpenMP and MPI (Thürey et al. [66]), and CUDA (Tölke [69] and Monitzer [43]). SIGGRAPH 2008 "Real-time physics" course documentation on LBM [47], and source code for an 2D CPU-based version of an LBM simulation is available online at `http://www.matthiasmueller.info/realtimephysics/`.

2.3 Final Remarks

There are mainly two fields with interest in water simulation: Computational Fluid Dynamics (CFD) and Computer Graphics (CG). CFD is a branch of fluid mechanics that tries to solve and analyse problems involving fluid flows as necessary in engineering analysis. CG purpose is to generate appealing, convincing, plausible simulations of fluid effects, mostly for the game and movie industries.

CG water models are simplifications or approximations of real models, or non-physically-based approaches. Physically-based water models used in CG are simplified versions, to make the methods faster, of CFD water models. Procedural water is a non-physically-based method. Lagrangian methods allow water effects such as splash, foam, jets, etc. for small amounts of water. Eulerian methods and LBM are grid-based methods.

Individually, none of the CG four water models is complete (i.e. none allows to simulate all water effects for large masses of water). However, hybrid models that interface the Eulerian and the Lagrangian methods are the closest to a complete model.

Chapter 3

Water Simulation

This chapter describes the physically-based model, working behind the water simulation, presented in this thesis. First, we explain the stable fluids method, which is used to solve the Navier-Stokes equations. In the stable fluids method explanation, we also justify the need of an implicit sparse linear system solver (e.g. Jacobi, Gauss-Seidel, Conjugate Gradient, etc.). The three solvers implemented on both CPU and GPU-based version, are explained also. After explaining the stable fluids method, we explain its implementation details both for the CPU and for the GPU-based versions.

3.1 Stable Fluids

The motion of a viscous fluid can be described by the Navier-Stokes (NS) equations. They are a set of partial differential equations (PDEs) that establish a relation between pressure, velocity and forces during a given time step. Thus, all fluid properties are exclusively described in function of its density and velocity. The NS equations derive from Newton's second law of motion, which state that:

$$f = m \cdot a \tag{3.1}$$

where m is the object mass, a is the object acceleration, and f is the force applied to the object. NS equations describe the fluid acceleration as the sum of all forces acting on the fluid (including forces introduced by the fluids own movement).

Most physically based fluid simulations use NS equations, or simplified versions of them (i.e. SWE or Euler equations), to describe the motion of fluids. These simulations are based on three NS equations. One equation just ensures mass conservation, and states that variation of the velocity field equals zero (Eq. 3.2). The other two equations describe the evolution of velocity (Eq. 3.3) and density (Eq. 3.4) over time as follows:

$$\nabla v = 0 \tag{3.2}$$

$$\frac{\partial u}{\partial t} = -\left(u \cdot \nabla\right) u + v \nabla^2 u + f \tag{3.3}$$

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla)\rho + k\nabla^2\rho + S \tag{3.4}$$

where u represents the velocity field, v is a scalar describing the viscosity of the fluid, f is the external force added to the velocity field, ρ is the density of the field, k is a scalar that describes the rate at which density diffuses, S is the external source added to the density field, and $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ is the gradient.

Stable fluids is a computer graphics method to simulate fluids. It uses implicit integration to solve the NS equations, in order to ensure unconditional stability. This means that, unlike explicit integration methods, the simulation does not blow up for large time steps. Stable fluids-based fluid simulators usually come with some sort of control user interface (CUI) to allow the user to interact with the simulation (see steps 2 and 3 of Algorithm 1). In order to solve Eqs. 3.3 and 3.4, the stable fluids-based fluid simulator performs a number of steps:

Algorithm 1 NS fluid simulator.

Output: Updated fluid at each time-step

1: **while** simulating **do**
2: Get forces from UI
3: Get density source from UI
4: Update velocity (Add force, Diffuse, Move)
5: Update density (Add force, Advect, Diffuse)
6: Display density
7: **end while**

Eqs. 3.3 and 3.4 are solved in steps 4 and 5 of Algorithm 1. Also, during step 4, mass conservation (i.e. Eq. 3.2) is ensured. To better understand velocity and density updates, let us detail steps 4 and 5 of Algorithm 1.

3.1.1 Add Force (f term in Eq. 3.3 and S term in Eq. 3.4)

This step considers the influence of external forces to the field. It consists in adding a force f to the velocity field or a source S to the density field. For each grid cell, the new velocity u is equal to its previous value u_0 plus the result of multiplying the simulation time step Δt by the force f to add, i.e. $u = u_0 + \Delta t \times f$. The same applies to the density, i.e. $\rho = \rho_0 + \Delta t \times S$, where ρ_0 is the density previous value, ρ is the new density value, Δt is the simulation time step, and S is the source to add to the density.

3.1.2 Advection

Advection is the term used to describe the transport of objects, densities, the fluid itself (self-advection) and other quantities. This transport is the result of the fluid's velocity when moving. Fedkiw's work [15] addressed on the connection between the Lagrangian and Eulerian methods in the advection of the velocities step.

To best understand advection let us consider that each grid cell represents a fluid particle. Taking into account that we are interested in calculating the variation of position of each particle along the velocity field, it becomes intuitive that advection of a quantity must be performed for each grid cell. To compute the result of advection, we can use either an explicit method (Euler method, or other more accurate methods such as the midpoint method and the Runge-Kutta methods), or an implicit method (Stam's 1999).

When using an explicit method (i.e. Euler's method), updates to the grid are performed as in a particle system. Therefore, the new particle position $r(t + \delta t)$ equals its old value $r(t)$ plus the distance covered in time δt, while travelling along the velocity field u (Eq. 3.5).

$$r(t + \delta t) = r(t) + u(t)\delta t \tag{3.5}$$

In spite of being an intuitive approach, advection with an explicit method is unstable for large time steps. When the magnitude of $u(t)\delta t$ is greater than the size of a single grid cell the simulation may blow up.

Stam's stable fluids approach overcomes this problem of explicit methods. Its solution consists in rewriting Eq. 3.5 in an implicit form. With the stable fluids method, the trajectory of the particle from each grid cell is traced back in time, to its former position. The particle value at its former position is then copied to the starting grid cell. This approach is also referred as semi-Lagrangian advection. In order to update any quantity q carried by the fluid (i.e. velocity, temperature, density, etc), one uses Eq. 3.6.

$$q(x, t + \delta t) = q(x - u(x,t)\delta t, t) \tag{3.6}$$

Stam's method is not only unconditional stable for arbitrary time steps and velocities, but also implementable on the GPU.

In Fig. 3.1 illustrates Stam's method semi-Lagrangian advection. Each grid cell has a velocity direction, which is part of an uniform velocity field (blue arrows). First, we track the particle back in time (green arrow) to its previous position. Then, we swap the values (i.e. velocities and density) of its former and current position. Thus, we need to store each grid cell previous and current values.

3.1.3 Diffusion ($v\nabla^2 u$ term in Eq. 3.3 and $k\nabla^2$ term in Eq. 3.4)

Viscosity describes the fluid's internal resistance to flow. This resistance results in diffusion of the momentum (i.e. dissipation of velocity). Viscous diffusion partial differential equations (Eqs. 3.7 and 3.8) can be extracted from Eqs. 3.3 and 3.4:

$$\frac{\partial u}{\partial t} = v\nabla^2 u \tag{3.7}$$

$$\frac{\partial \rho}{\partial t} = k\nabla^2 \rho \tag{3.8}$$

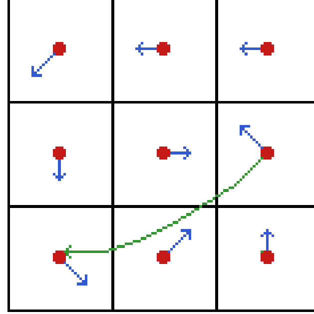

Figure 3.1: Advection computation.

Similar to advection computation, diffusion can be done using either an explicit (Eq. 3.9), discrete form, or an implicit form (Eq. 3.10). In the explicit, discrete form we must solve the following equation:

$$u(x, t + \delta t) = u(x, t) + v\delta t \nabla^2 u(x, t) \tag{3.9}$$

where ∇^2 is the discrete form of the Laplacian operator. In a similar way to the Euler method in explicit advection, the previous formulation is unstable for large time steps δt and v (i.e. it may blow up).

In Stam's explicit formulation, we must solve the following equation:

$$\left(I - v\delta t \nabla^2\right) u(x, t + \delta t) = u(x, t) \tag{3.10}$$

where I is the identity matrix. Like for advection, this formulation is stable for arbitrary time steps and viscosities. This equation is a (somewhat disguised) Poisson equation (see Eq. 3.11) for velocity, which can be solved by an iterative solver (i.e. Jacobi relaxation, Gauss-Seidel relaxation, Conjugate Gradient [57]).

When solving the full 3D diffusion problem, the simulation grid is represented as an 1D array for memory efficient storage and access purposes. Fig. 3.2 illustrates the mapping from an 2D grid to an 1D array.

During diffusion, each grid cell interacts with its direct neighbours, as shown in Fig. 3.3.

If we discretize Eq. 3.10, we obtain the following Poisson equation for each grid cell:

$$\begin{aligned} D^{n}_{i,j,k} = D^{n+1}_{i,j,k} - \frac{kdt}{h^3} \Big(& D^{n+1}_{i-1,j,k} + D^{n+1}_{i,j-1,k} + D^{n+1}_{i,j,k-1} + \\ & + D^{n+1}_{i+1,j,k} + D^{n+1}_{i,j+1,k} + D^{n+1}_{i,j,k+1} - 6D^{n+1}_{i,j,k} \Big) \end{aligned} \tag{3.11}$$

where D^n and D^{n+1} are a grid cell velocity or density values, respectively, before and after diffusion, i, j, k are the cell coordinates (i.e. the cell spatial position in the grid), dt is the simulation time step value, h^3 is the volume of a cell, and k is the diffusion rate.

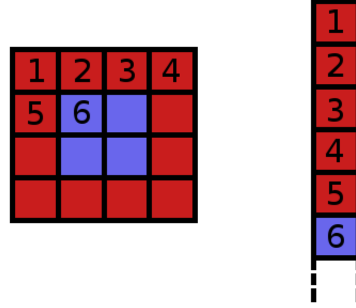

Figure 3.2: 2D grid (left) represented by a 1Darray (right).

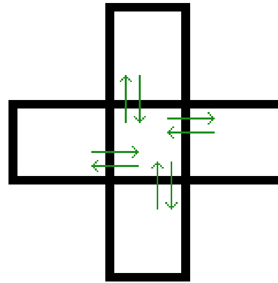

Figure 3.3: Cell interaction with its direct neighbours.

Eq. 3.11 must be rewritten, for us to know the value of $D^{n+1}_{i,j,k}$ (Eq. 3.12). The obtained equation follows:

$$D^{n+1}_{i,j,k} = \frac{D^{n}_{i,j,k} + \frac{kdt}{h^3}\left(D^{n+1}_{i-1,j,k} + D^{n+1}_{i,j-1,k} + D^{n+1}_{i,j,k-1} + D^{n+1}_{i+1,j,k} + D^{n+1}_{i,j+1,k} + D^{n+1}_{i,j,k+1}\right)}{1 + \frac{kdt}{h^3}} \quad (3.12)$$

If one writes down all the equations for a 4×4 2D fluid simulation, the resulting system is the following Fig. 3.4:

This is the sparse linear system in the form $Ax = b$ to be solve, using an iterative method. Among the possible algorithms to solve this sparse linear system, we selected three of them: namely Jacobi relaxation, Gauss-Seidel relaxation, and Conjugate Gradient. These three solvers are described in sequel.

3.1.3.1 Jacobi and Gauss-Seidel Algorithms

The Jacobi and Gauss-Seidel solvers, iterate a given number times (line 1 of Algorithms 2 and 3) for each grid cell (line 2 of Algorithms 2 and 3) to calculate the grid cell new value (line 3 of Algorithms 2 and 3). What distinguishes both solvers is that Gauss-Seidel

-2	1			1													x[0,0]		b[0,0]
1	-3	1			1												x[1,0]		b[1,0]
	1	-3	1			1											x[2,0]		b[2,0]
		1	-2				1										x[3,0]		b[3,0]
1				-3	1			1									x[0,1]		b[0,1]
	1			1	-4	1			1								x[1,1]		b[1,1]
		1			1	-4	1			1							x[2,1]		b[2,1]
			1			1	-3				1						x[3,1]		b[3,1]
				1				-3	1			1				×	x[0,2]	=	b[0,2]
					1			1	-4	1			1				x[1,2]		b[1,2]
						1			1	-4	1			1			x[2,2]		b[2,2]
							1			1	-3				1		x[3,2]		b[3,2]
								1				-2	1				x[0,3]		b[0,3]
									1			1	-3	1			x[1,3]		b[1,3]
										1			1	-3	1		x[2,3]		b[2,3]
											1			1	-2		x[3,3]		b[3,3]

Figure 3.4: The sparse linear system to solve (for a 2^2 fluid simulation grid).

uses the already calculated values of a grid cell direct neighbours (i.e. density or velocities values of each grid cell), but Jacobi does not. Therefore, Jacobi convergence rate will be slower, when compared to the Gauss-Seidel solver. Since the Jacobi solver does not use the already updated grid cell direct neighbours values, it requires the storage of the new values, in a temporary auxiliary 1D array (aux). When all new values of the grid cells have been determined (i.e. density or velocities values of each grid cell), the old values of the grid cells (i.e. density or velocities values of each grid cell) will be replaced with new the values stored in the auxiliary 1D array (lines 5 to 7 of Algorithm 2). Stam's 2003 version of stable fluids [62] solves diffusion and part of the move step (addressed further ahead), using the same sparse system linear solver. When using Jacobi or Gauss-Seidel, the solver becomes generic after some maths, where only $iter$ and a will differ (line 3 of Algorithms 2 and 3). The arrays x and $x0$ are the grid current and previous values (i.e. velocities and density of each grid cell). After each iteration we must set the correct values in cells that are either internal, or moving, or external boundaries (lines 8 and 5 of Algorithms 2 and 3).

The Gauss-Seidel solver is a sequential algorithm since it requires the already calculated values of a grid cell direct neighbours. The Gauss-Seidel implementation for a parallel architecture, such as the GPU, has two interleaved passes. First it updates the red cells (line 3 of Algorithm 4) and then the black cells (line 8 of Algorithm 4), according to the pattern shown in Fig. 3.5.

Thus, the already calculated values of a grid cell direct neighbours are used, as in the sequential algorithm. Nevertheless, Gauss-Seidel red black takes more iterations to converge than the sequential implementation.

Algorithm 2 Jacobi relaxation.

Input:

x: 1D array with the grid current values.

$x0$: 1D array with the grid previous values.

aux: auxiliary 1D array temporarily store the updated grid cell values.

a: $\frac{kdt}{h^3}$ (see Eq. 3.12).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12).

Output:

x: 1D array with the grid new interpolated values.

1: **for** $iteration = 0$ to max_iter **do**
2: **for all** grid cells **do**
3: $aux(i, j, k) = (x0(i, j, k) + a \times (x(i-1, j, k) + x(i, j-1, k) + x(i, j, k-1) + x(i+1, j, k) + x(i, j+1, k) + x(i, j, k+1)))/iter$
4: **end for**
5: **for all** grid cells **do**
6: $x(i, j, k) = aux(i, j, k)$
7: **end for**
8: Enforce Boundary Conditions
9: **end for**

Algorithm 3 Gauss-Seidel relaxation.

Input:

x: 1D array with the grid current values.

$x0$: 1D array with the grid previous values.

a: $\frac{kdt}{h^3}$ (see Eq. 3.12).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12).

Output:

x: 1D array with the grid new interpolated values.

1: **for** $iteration = 0$ to max_iter **do**
2: **for all** grid cells **do**
3: $x(i, j, k) = (x0(i, j, k) + a \times (x(i-1, j, k) + x(i, j-1, k) + x(i, j, k-1) + x(i+1, j, k) + x(i, j+1, k) + x(i, j, k+1)))/iter$;
4: **end for**
5: Enforce Boundary Conditions
6: **end for**

3.1.3.2 Conjugate Gradient Algorithm

The Conjugate Gradient method (Algorithm 5) takes a different approach than the previous relaxation techniques. Instead of successive relaxations that eventually will converge, Conjugate Gradient directly searches for the iterated values. However, despite achieving

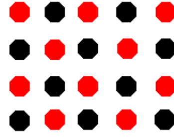

Figure 3.5: Gauss-Seidel red black pattern for a 2D grid.

Algorithm 4 Gauss-Seidel red black.

Input:

x: 1D array with the grid current values.

$x0$: 1D array with the grid previous values.

a: $\frac{kdt}{h^3}$ (see Eq. 3.12).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12).

Output:

x: 1D array with the grid new interpolated values.

1: **for** $iteration = 0$ to max_iter **do**
2: **for all** grid cells **do**
3: **if** $i + j$ is pair **then**
4: $x(i,j,k) = (x0(i,j,k) + a \times (x(i-1,j,k) + x(i,j-1,k) + x(i,j,k-1) + x(i+1,j,k) + x(i,j+1,k) + x(i,j,k+1)))/iter$;
5: **end if**
6: **end for**
7: **for all** grid cells **do**
8: **if** $i + j$ is even **then**
9: $x(i,j,k) = (x0(i,j,k) + a \times (x(i-1,j,k) + x(i,j-1,k) + x(i,j,k-1) + x(i+1,j,k) + x(i,j+1,k) + x(i,j,k+1)))/iter$;
10: **end if**
11: **end for**
12: Enforce Boundary Conditions
13: **end for**

far better convergence rates than relaxation techniques, its sequential implementation has an heavy computational cost. The Conjugate Gradient will for a given number of iterations (line 5 of Algorithm 5) choose a new iterated values for each grid cell, and store them in the 1D array x. To choose these new values, at each iteration, for each of the grid cells, an optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors, is chosen (line 14 of Algorithm 5). Moving along p by a distance of α will get us closer to the new iterated value of each grid cell. The optimal search vectors for each grid cell are stored in the 1D array p. As long as A is symmetric positive definite, Conjugate Gradient will converge eventually, which is always true for the sparse linear system to solve (Fig. 3.4).

Algorithm 5 Conjugate Gradient method.

Input:
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
a: $\frac{kdt}{h^3}$ (see Eq. 3.12).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12).
max_iter: number of iterations to perform tol: tolerance after which is safe to state that the values of x converged optimally **Output:**
x 1D array with the grid new interpolated values.

1: $r = b - Ax$
2: $p = r$
3: $\rho = r^T \cdot r$
4: $\rho_0 = \rho$
5: **for** $iteration = 0$ to max_iter **do**
6:   **if** $(\rho! = 0)$ and $\rho > tol^2 \times \rho_0$ **then**
7:     $q = Ap$
8:     $\alpha = \rho/(p^T \cdot q)$
9:     $x+ = \alpha \times p$
10:     $r- = \alpha \times q$
11:     $\rho_old = \rho$
12:     $\rho = r^T \cdot r$
13:     $\beta = \rho/\rho_old$
14:     $p = r + \beta \times p$
15:     Enforce Boundary Conditions
16:   **end if**
17: **end for**

The CPU and GPU-based implementations of each of the Conjugate Gradient method steps is addressed and explained further ahead in this chapter.

3.1.4 Move ($-(u.\nabla)\,u$ term in Eq. 3.3 and $-(u.\nabla)\,\rho$ term in Eq 3.4)

Real fluids, like water, change their volume (if not, you would not be able to hear underwater), but that does not occur often. In computer graphics, the water volume changes are discarded, since ensuring compressibility is difficult and computationally expensive.

Move is the step where mass conservation (Eq. 3.2), and fluid incompressibility are ensured. Move is actually five steps: an projection of velocity, one advection for each of the three velocity components (i.e. x, y, and z velocities), and another projection of velocity. When the fluid moves, mass conservation must be ensured. This means that the flow leaving each cell (of the grid where the fluid is being simulated) must equal the flow coming in. But the previous steps (Add force and diffuse for velocity) violate the principle of mass conservation. Stam uses a Hodge decomposition of a vector field (the velocity vector field

specifically) to address this issue. Hodge decomposition states that every vector field is the sum of a mass conserving field and a gradient field. To ensure mass conservation we then simply subtract the gradient field from the vector field. In order to do this we must find the scalar function that defines the gradient field. Computing the gradient field is therefore a matter of solving the following Poisson equation for each grid cell.

$$P_{i-1,j,k} + P_{i,j-1,k} + P_{i,j,k-1} + P_{i+1,j,k} + P_{i,j+1,k} + P_{i,j,k+1} - 6P_{i,j,k} = \\ = (U_{i+1,j,k} - U_{i-1,j,k} + V_{i,j+1,k} - V_{i,j-1,k} + W_{i,j,k+1} - W_{i,j,k-1})h \tag{3.13}$$

where P is a grid cell velocity value after projection, i, j, k are the cell coordinates (i.e. the cell spatial position in the grid), U, V, W denote the velocity components (i.e. x, y and z), and h is the size of a cell side.

Solving this Poisson equation for each grid cell is the same as solving a sparse symmetrical linear system, similar to the one in the diffusion step. Therefore, this system can be solved with the solver used in the diffuse step, considering the value of $iter = 6$ and $a = 1$ in Algorithms 2, 3, 4, and 5.

3.2 CPU Implementation

Stam's 2003 version of stable fluids was implemented for 2D, without considering internal or moving boundaries [62]. On the contrary, the experimental scenario mounted for this thesis work takes into consideration internal and moving boundaries. More specifically, the scenario simulates water being poured into a tank, with the possibility of adding a falling sphere (i.e. moving obstacle). Thus, our work extends stable fluids no only to 3D, but also to scenarios with internal and moving boundaries. Aside from the extension to 3D, in order to achieve the intended scenario a control user interface was also required. Finally, as previously mentioned, a more detailed explanation of each of the Conjugate Gradient solver steps is given, both in the CPU and GPU versions.

3.2.1 Extension to 3D

The first change made to Stam's version was the creation of a data structure to store the grid related information. In this data structure named $_GRID$ the values of the previous ($d0$, $vx0$, $vy0$, and $vz0$) and current (d, vx, vy, and vz) of density and velocity x, y, and z components are taken in consideration. There is also an 1D array (bd) for tracking the internal and moving boundaries occupied position. This data structure also possesses the total number o grid cell ($_GRID_cells$), the number of grid cells per axis (NX, NY, and NZ), and the width ($sizeZ$), height ($sizeY$), and length ($sizeX$) of each grid cell.

```
typedef struct
{
    int _GRID_cells; // total number of grid cells
    int NX, NY, NZ; // fluid grid number of partitions per axis
    float sizeX, sizeY, sizeZ; // fluid cell size per axis
```

```
    float *vx, *vy, *vz, *vx0, *vy0, *vz0; // velocities and previous velocities
    float *d, *d0; // density values
    char *bd; // fluid internal and external boundaries
} _GRID;
```

Internal boundaries are read in from a file. This file contains a description of the grid in slices. Each slice contains information about which grid cells in that slice are internal boundaries (i.e. those having the value B), and which are not (i.e. those having the value _). In respect to moving boundaries (i.e. those having the value M), a moving object position is tracked and the values which are not boundaries are temporally considered as moving boundaries. To better visually distinguish internal and moving boundaries, a different color is used to distinguish between them green for moving boundaries and red for internal boundaries (see Fig. 3.6).

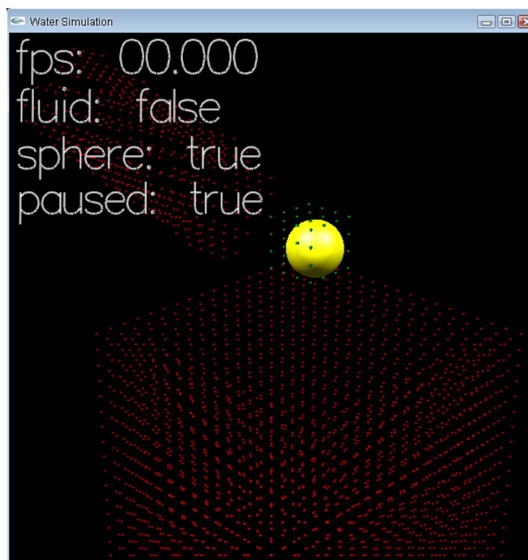

Figure 3.6: Internal (red dots) and moving boundaries (green dots) representation.

Any grid cell values (i.e. velocity and density values) in a boundary position (internal or moving boundary) equal zero. In spite of achieving better visual results, this is not physically accurate for moving boundaries. For moving boundaries, the correct physical interpretation would be to consider the velocity in the moving boundary cell as having the inverse value of the velocity field in the same cell. The external boundaries conditions remain the same; they only need to be adapted from a 2D to 3D grid.

Apart the internal and moving boundaries, the remaining adaptations to extend stable fluids to 3D CPU-based versions were already addressed in previous works, [4, 6]. For more details on this subject, the reader is referred to these citations, well as to Appendix B.

3.2.2 Control User Interface

The control user interface (Algorithm 6) consists in a series of steps (i.e. add density, add forces, increase water level, etc), to achieve the desired simulation (i.e. water is poured

from a pipe to a water container until it is full, and any given time a falling sphere may be added to the simulation).

Algorithm 6 Control User Interface.

1: clear $vx0$, $vy0$, $vz0$ and $d0$ values.
2: Introduce water until container reaches a certain level of water.
3: **for all** grid cells **do**
4: Add horizontal velocity inside the pipe to simulate a water pump
5: Ensure gravities force is considered only where fluid lies.
6: Calculate the position of the sphere (i.e. moving boundary) in grid coordinates.
7: Add horizontal velocity that opposes gravity attraction, to make a splash when the sphere hits the surface.
8: Ensure that the water is only add until a certain water level.
9: **end for**
10: Increase water level until tank is full.

Line 7 of Algorithm 6 might be interpreted as a contradiction, to the previous sentence that moving boundaries equal zero. However, it is not true as what happens is the introduction of a local velocity at an instant of the simulation. To consider the moving boundaries velocity value equal to the inverse of the velocity field in the same cell would result in a velocity tunnel. This tunnel generates a visual unwanted swirling water effect (i.e. a localized small scale tornado). Therefore the splash was interpreted as the result of the energy released (in the form of velocity) when two masses (i.e. the sphere and the water surface) collide.

Line 8 of Algorithm 6 is where the already filled positions inside the water container must be progressively refilled (i.e. marked as filled) in order to avoid density dissipation over time. However, doing so eliminates the waving effect on the water surface when the sphere hits it.

3.2.3 Conjugate Gradient Solver

The CPU-based Conjugate Gradient algorithm (Algorithm 5) consists in a series of calls to functions, each one of which performs each solver step. The variables arrays x and $x0$ stand for the current and previous values 1D data arrays, that store the values of either density or velocity components. Before iterating, it is first required (lines 1 to 4 of Algorithm 5) to set the initial values of r and p (Algorithm 7), and of ρ_0 and ρ (Algorithm 8):

Now we are ready to iterate until all iterations are done or the stop criterion is achieved (lines 5 and 6 of Algorithm 5). For each iteration, the first step (line 7 of Algorithm 5) is to update q (Algorithm 9). After updating q, the next step (lines 8 to 12 of Algorithm 5) is to determine the new distance to travel along p, α. During the update of α, the dot product of p by q must be determined. Dot product is a generic function, so we just need to change the function arguments (Algorithm 8). After updating α, we need to determine the iterated values of x, and the new r residues (lines 9 and 10 of Algorithm 5 as detailed of Algorithm 10). Before updating each grid cell previous optimal search vector (gradient), that is or-

Algorithm 7 Set the initial values of r and p.

Input:
r: residue for each grid cell.
p: 1D array with each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors.
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).
Output:
r: updated r.
p: updated p.

1: **for all** grid cells **do**
2: $r[i,j,k) = x0(i,j,k) - ((x(i,j,k) \times iter) - a \times (x(i-1,j,k) + x(i,j-1,k) + x(i,j,k-1) + x(i+1,j,k) + x(i,j+1,k) + x(i,j,k+1)))$
3: $p(i,j,k) = r(i,j,k)$
4: **end for**

Algorithm 8 Set the initial value of ρ (dot product).

Input:
r: residue for each grid cell.
ρ: $r^T \cdot r$.
a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).
Output:
ρ: updated ρ.

1: $result = 0$
2: **for all** grid cells **do**
3: $result += r(i,j,k) \times r(i,j,k)$
4: **end for**

thogonal (conjugate) to all the previous search vectors, p (line 13 of Algorithm 5), $\rho_o ld$, ρ and β must be updated (lines 11 to 13 of Algorithm 5). This is no more than performing a call to the generic dot product function with different arguments. After updating β the new search directions (p values) must be set (Algorithm 11).

3.3 GPU Implementation

Now we address to the GPU-based version of stable fluids. More specifically, we describe how the CPU-based version was modified to achieve the corresponding CUDA version.

Algorithm 9 Update q.

Input:

q: 1D auxiliary array.

p: 1D array with each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors.

x: 1D array with the grid current values.

a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).

Output:

q: updated q.

1: **for all** grid cells **do**
2: $q(i,j,k) = (p(i,j,k) \times iter) - a \times (p(i-1,j,k) + p(i+1,j,k) + p(i,j-1,k) + p(i,j+1,k) + p(i,j,k-1) + p(i,j,k+1))$
3: **end for**

Algorithm 10 Update x and r.

r: residue for each grid cell.

p: 1D array with each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors.

q: 1D auxiliary array.

x: 1D array with the grid current values.

α: distance to travel along p for each grid cell.

Output:

x: new interpolated values of x.

r: updated r.

1: **for all** grid cells **do**
2: $x(i,j,k) += \alpha \times p(i,j,k)$
3: $r(i,j,k) -= \alpha \times q(i,j,k)$
4: **end for**

Algorithm 11 Update p.

r: residue for each grid cell.

p: 1D array with each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors.

β: auxiliary scalar.

Output:

p: updated p.

1: **for all** grid cells **do**
2: $p(i,j,k) = r(i,j,k) + \beta \times p(i,j,k)$
3: **end for**

Also, we will explain in more detail how the Conjugate Gradient, Jacobi, and Gauss-Seidel algorithms were modified to run using CUDA. Before proceeding, a small explanation on the CUDA programming model is given.

3.3.1 NVIDIA Compute Unified Device Architecture (CUDA)

With the advent of the first graphics systems it became clear that the parallel work required by graphics computations should be delegated to another component other than the CPU. Thus, the first graphics cards arrived to alleviate graphics processing load of the CPU. However, graphics programming was basically done using a kind of assembly language. With the appearance of the graphics programming APIs (such as OpenGL or DirectX), and later the high-level shading languages (such as Cg or GLSL), programming graphics became easier.

When, in 2007, NVIDIA CUDA (Programming Guide [1] and Reference Manual [2]) was released, it was made possible to specify how and what work should be done in the NVIDIA graphics cards. It became possible to program directly the GPU, using the C\C++ programming language. CUDA has two programming modes, namely the driver API and the runtime API. The runtime programming mode is a high-level API built on top of the low-level driver API. The programming mode used in this thesis was the runtime API. CUDA is available for Linux and Windows operating systems, and includes the BLAS and the FFT libraries.

The CUDA programming logic separates the work (see Fig. 3.7) that should be performed by the CPU (host), from the work that should be performed by the GPU (device). The host runs its own code and launches kernels, to be executed by the GPU. After a kernel call, the control returns immediately to the host, unless a lock is activated with `cudaThreadSynchronize`. The GPU and the CPU work simultaneously after a kernel call, each running its own code (unless a lock is activated in the host). The GPU runs the kernels by order of arrival ($kernel$ 1 then $kernel$ 2 as shown in Fig. 2), unless they are running on different streams, i.e. kernel execution is asynchronous. If a lock is activated the next kernel will be only called when the previously invoked kernels finish their jobs.

A kernel has a set of parameters, aside from the pointers to device memory variables or copies of CPU data. The parameters of a kernel specify the number of blocks in a grid (in 2D only), the number of threads (in x, y, z directions) of each grid block, the size of shared memory per block (0 by default) and the stream to use (0 by default). The maximum number of threads allowed by block is 512 ($x \times y \times z$ threads per block). All blocks of the same grid have the same number of threads. Blocks work in parallel either asynchronously or synchronously. With this information the kernel specifies how the work will be divided over a grid.

When talking about CUDA, four kinds of memory are considered (see Fig. 3.8). Host memory refers to the CPU-associated RAM, and can only be accessed by the host. The device has three kinds of memory: constant memory, global memory and shared memory. Constant and global memory are accessible by all threads in a grid. Global memory has read/write permissions from each thread of a grid. Constant memory only allows read permission from each thread of a grid. The host may transfer data from RAM to the device

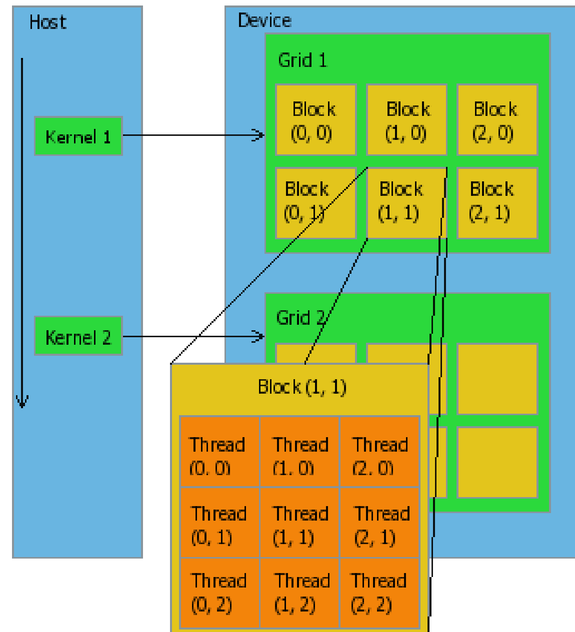

Figure 3.7: CUDA work flow model.

global or constant memory or vice-versa. Shared memory is the memory shared by all threads of a block. All threads within the same block have read/write permissions to use the block shared memory.

3.3.2 CPU to GPU version

Several modifications were made to allow transform the CPU-based into a GPU-based version. Velocity components (i.e. velocity in the x, y, and z directions), and density are stored in the GPU global memory. Therefore, the structure used to store the grid relevant data from the CPU-based version was downsized in the GPU-based version, as follows:

```
typedef struct
{
    int _GRID_cells; // total number of grid cells
    int NX, NY, NZ; // fluid grid number of partitions per axis
    float sizeX, sizeY, sizeZ; // fluid grid size per axis
    char *bd; // fluid internal and external boundaries
} _GRID;
```

CUDA is launched after initializing the grid. This is done only once in the application. The same applies to shutting down CUDA, which is done only once, before closing the application. The simulation internal and external boundaries are also read in from an file, like in the CPU-based version. Unfortunately, so far it is not possible to read data from a

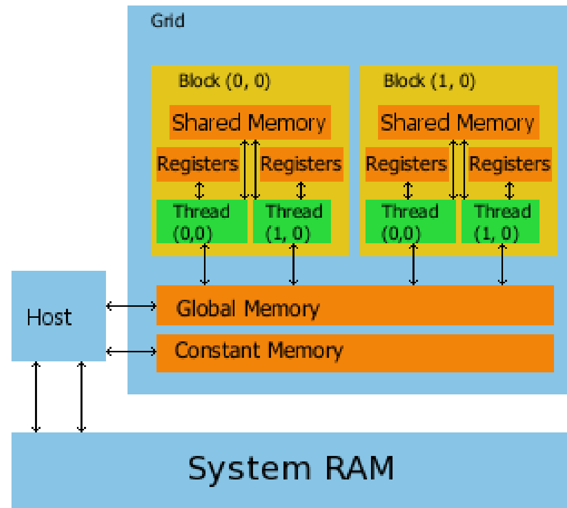

Figure 3.8: CUDA memory model.

file in a kernel. Therefore, the file is read to an 1D array (`bd`) on the host side, like in the CPU-based version, being the boundaries data array then copied to an 1D array in device global memory.

In the first GPU implementation, the solver was the only work done by the GPU. However, the continuous host-to-device and device-to-host memory transfers (as expected) degraded simulation performance to a level below the CPU-based version. Unsurprisingly, the final version now runs entirely on the GPU, and all simulation data (i.e. velocities, densities, control variables, VBOs, etc) resides in the graphics global memory. The device global memory is reserved in the beginning of the application, and free when the application shuts down. There are two pointers to the CUDA-OpenGL interoperability VBOs (i.e. color and vertex data). These two pointers are generic. Therefore, boundaries, velocities or density share the same pointers. These generic pointers must be resized depending on the drawing primitive required (i.e. velocities use GL_LINE, while boundaries and density use GL_POINTS).

In CUDA, several concerns arise when transforming the CPU-based functions into kernels. First, for better performance, the number of kernels must be minimized to a minimum, because many calls to the same kernel will result in noticeable timeouts (i.e. the simulation temporarily freezes). Second, the size of the blocks affects the performance of the simulation (i.e. block sizes that do not maximize GPU usage degrade the overall simulation performance). Therefore, the right balance between block size and the grid dimensions is of major concern. Finally, the most critical issue is the device global memory required. Stable fluids consumes lots of memory space. Summing up the memory required by the VBOs, to the device global memory required by the Jacobi or Conjugate Gradient solvers, we came to the conclusion that the maximum grid size allowed is 128^3. In order to minimize the device global memory requirements, Jacobi or Conjugate Gradient required extra memory is only

allocated when they are used.

To migrate the functions in the CPU-based version to the GPU-version, we have to identify which work can be done in parallel. The majority of these functions performed operations over the grid of cells (in CUDA a grid of threads). To access the grid data in the CPU-based version, we sequentially pass by all grid positions, respectively in the z, y and x directions. In the GPU-based version, this access is performed by a single thread, assigned to a specific grid cell. There are two ways of accessing to the z coordinate in the GPU-based version. First, one may use the thread block dimension in z. Secondly, each block treats all grid slices in z direction for the threads in x and y. After some experiments we concluded that the second approach works better than the first. Our block size is 16×16. Each block treats all the slices in z, for the same x and y elements of the grid. When calling a kernel, we have to specify the block size (i.e. `threads`) and the number of blocks in the grid of threads (i.e. `grid`), as follows:

```
dim3 threads(BLOCK_DIM_X,BLOCK_DIM_Y,BLOCK_DIM_Z);
dim3 grid(NX/BLOCK_DIM_X,NY/BLOCK_DIM_Y);
```

where NX and NY are the dimensions of the grid in x and y directions. Each block has $BLOCK_DIM_X$ threads in x, $BLOCK_DIM_Y$ threads in y, and $BLOCK_DIM_Z$ threads in z directions (i.e. 16 threads in x, 16 threads in y, and 1 thread in z). The code of the kernels is available at Appendix C.

3.3.3 Jacobi and Gauss-Seidel Solvers

On the GPU-based version, of the Jacobi and the Gauss-Seidel solvers, we have a call to a kernel (Algorithms 12 and 13) with two parameters: `grid` stands for the number of blocks in `X` and `Y` axes, and `threads` denotes the number of threads per block, as follows:

```
// Jacobi kernel call
_jcb<<<grid,threads>>>(x,x0,aux,a,iter,max\_iter);
CUT_CHECK_ERROR("Kernel execution failed");

// or

// Gauss-Seidel red black kernel call
_gs_rb<<<grid,threads >>>(x,x0,a,iter,max\_iter);
CUT_CHECK_ERROR("Kernel execution failed");
```

In the GPU-based Jacobi and Gauss-Seidel red black algorithms, the values of i, j (cell coordinates) are obtained from the blocks, threads, and grid information $dim3$ type variables (lines 1 amd 2 of Algorithms 12 and 13). The Jacobi solver presents two drawbacks when compared to Gauss-Seidel solver. First, the Jacobi solver converges slower. Secondly, it requires an auxiliary 1D array (aux) stored in device global memory, to temporarily store the already updated grid cell direct neighbours values. The reader with experience in programming with CUDA might question why not use shared memory instead. The reason is that the the data to be stored in the 1D array is for all individual slices in z. Therefore, it surpasses the maximum allowed size for shared memory per block threads. The 1D arrays x and $x0$ are for storing the current and previous values of either velocities or densities. The simulation grid as NX columns, NY lines, and NZ slices.

Algorithm 12 Jacobi GPU kernel.

Input:

x: 1D device global memory array with the grid current values.

$x0$: 1D device global memory array with the grid previous values.

aux: auxiliary 1D device global memory array temporarily store the updated grid cell values.

a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).

Output:

x: new interpolated values of x.

1: $i = threadIdx.x + blockIdx.x \times blockDim.x$
2: $j = threadIdx.y + blockIdx.y \times blockDim.y$
3: **for** $iteration = 0$ to $maxiter$ **do**
4:   **for** $k = 0$ to NZ **do**
5:     **if** $(i! = 0)\,\&\&\,(i! = NX - 1)\,\&\&\,(j! = 0)\,\&\&\,(j! = NY - 1)\,\&\&\,(k! = 0)\,\&\&\,(k! = NZ - 1)$ **then**
6:       $aux(i,j,k) = (x0(i,j,k) + a \times (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)))/iter$
7:       $__syncthreads$
8:       $x(i,j,k) = aux(i,j,k)$
9:     **end if**
10:     Enforce Boundary Conditions
11:   **end for**
12: **end for**

3.3.4 Conjugate Gradient Solver

The GPU-based Conjugate Gradient algorithm (Algorithms **??** to **??**) was also implemented as a kernel, called `_cg`, which is invoked as follows:

```
_cg<<<1,NX>>>(r,p,q,x,b,alpha,beta,rho,rho0,rho_old,a,iter,max_iter);
CUT_CHECK_ERROR("Kernel execution failed");
```

The most intuitive way to migrate the Conjugate Gradient from a sequential to a parallel algorithm, is to perform its steps (i.e. dot products, update of grid positions, etc) by kernels or using the CUDA BLAS library kernels. However, most of these kernels must be called for a certain number of iterations. Therefore, the successive invocation of kernels will result in timeouts in the simulation. The best solution found was to build up a massive kernel. However, this results in losing much of the CUDA performance gains. The reason is related with the parallel version of dot product, which forces the use of an one block, with NX threads in x. Much of the steps of the Conjugate Gradient performance degrades with this restriction. Even worse this version has worst performance than the CPU-based version.

The GPU-based Conjugate Gradient kernel first requires (lines 1 to 4 of Algorithm 5) to set the initial values of ρ_0, ρ (Algorithms 14 and 15):

Algorithm 13 Gauss-Seidel red black GPU kernel.

Input:

x: 1D device global memory array with the grid current values.

$x0$: 1D device global memory array with the grid previous values.

a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).

$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).

Output:

x: new interpolated values of x.

1: $i = threadIdx.x + blockIdx.x \times blockDim.x$
2: $j = threadIdx.y + blockIdx.y \times blockDim.y$
3: **for** $iteration = 0$ to $maxiter$ **do**
4: **for** $k = 0$ to NZ **do**
5: **if** $(i! = 0)\,\&\&\,(i! = NX - 1)\,\&\&\,(j! = 0)\,\&\&\,(j! = NY - 1)\,\&\&\,(k! = 0)\,\&\&\,(k! = NZ - 1)$ **then**
6: **if** $(i + j)\%2 == 0$ **then**
7: $x(i, j, k) = (x0(i, j, k) + a \times (x(i-1, j, k) + x(i+1, j, k) + x(i, j-1, k) + x(i, j+1, k) + x(i, j, k-1) + x(i, j, k+1)))/iter$
8: **end if**
9: $__syncthreads$
10: **if** $(i + j)\%2! = 0$ **then**
11: $x(i, j, k) = (x0(i, j, k) + a \times (x(i-1, j, k) + x(i+1, j, k) + x(i, j-1, k) + x(i, j+1, k) + x(i, j, k-1) + x(i, j, k+1)))/iter$
12: **end if**
13: **end if**
14: Enforce Boundary Conditions
15: **end for**
16: **end for**

Now we are ready to iterate (line 37 of Algorithm 16), until all iterations are done or the stop criterion is achieved (line 38 of Algorithm 16). For each iteration, the first step (line 7 of Algorithm 5) is to update the values of the auxiliary 1D array q (lines 39 to 46 of Algorithm 16).

After updating the values of the auxiliary 1D array q, the next step (lines 8 to 12 of Algorithm 5) is to determine the new distance to travel along p, α (Algorithm 17). During the update of α, the dot product of p by q must be determined (lines 47 to 49 of Algorithm 17). The dot product was already performed when ρ was initialized (lines 11 to 31 of Algorithm 15). Therefore, is just a matter of performing small changes. However, most of the source code is identical.

After updating α, we need to determine the iterated values of x, and the new r residues (lines 9 and 10 of Algorithm 5 as detailed of Algorithm 18).

Before updating each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous search vectors, p (line 13 of Algorithm 5 as detailed in

Algorithm 14 GPU-based set the initial values of r and p (part 1 of $_cg$ GPU kernel).

Input:
r: residue for each grid cell.
p: 1D array with each grid cell previous optimal search vector (gradient), that is orthogonal (conjugate) to all the previous earch vectors.
q: 1D auxiliary array.
ρ: $r^T \cdot r$.
x: 1D array with the grid current values.
$x0$: 1D array with the grid previous values.
α: distance to travel along p for each grid cell.
β: auxiliary scalar.
a: $\frac{kdt}{h^3}$ for diffusion (see Eq. 3.12), 1 for move (see Eq. 3.13).
$iter$: $1 + \frac{kdt}{h^3}$ (see Eq. 3.12), 6 for Move (see Eq. 3.13).
max_iter: number of iterations to perform.
tol: tolerance after which is safe to state that the values of x converged optimally. **Output:**
x: new interpolated values of x.

1: $i = threadIdx.x$
2: **for** $k = 0$ to NZ **do**
3: **for** $j = 0$ to NY **do**
4: **if** $(i! = 0) \&\& (i! = NX - 1) \&\& (j! = 0) \&\& (j! = NY - 1) \&\& (k! = 0) \&\& (k! = NZ - 1)$ **then**
5: $r[i, j, k) = x0(i, j, k) - ((x(i, j, k) \times iter) - a \times (x(i - 1, j, k) + x(i, j - 1, k) + x(i, j, k - 1) + x(i + 1, j, k) + x(i, j + 1, k) + x(i, j, k + 1)))$
6: $p(i, j, k) = r(i, j, k)$
7: **end if**
8: **end for**
9: **end for**
10: $__syncthreads$

lines 92 to 99 of Algorithm 19), $\rho_o ld$, ρ and β must be updated (lines 11 to 13 of Algorithm 5). The updated value of ρ_old equals the current value of ρ (lines 74 to 77 of algorithm 19). To update ρ (lines 78 to 83 of Algorithm 19) is an already familiar process (Algorithm 15 lines 11 to 35). After the new value of ρ is determined β can be updated (lines 84 to 91 of Algorithm 19). With the updated value of β the new search directions (p values) must be set (lines 92 to 99 of Algorithm 19).

3.4 Final Remarks

Stable fluids is an grid-based method. The main purpose of the stable fluids approach is to withdraw the time restrictions of explicit approaches (i.e. explicit methods blow up for big time steps). Stable fluids method consumes a lot of memory in the 3D version, as a

Algorithm 15 GPU-based set the initial values of ρ and ρ_0 (part 2 of $_cg$ GPU kernel).

11: $sum = 0$
12: **for** $j = 0$ to NY **do**
13: **if** $(i! = 0)\&\&(i! = NX - 1)\&\&(j! = 0)\&\&(j! = NY - 1)$ **then**
14: **for** $k = 0$ to NZ **do**
15: **if** $(k! = 0)\&\&(k! = NZ - 1)$ **then**
16: **if** $k == 1$ **then**
17: $t[i - 1] = 0$
18: **end if**
19: $t[i - 1]+ = r(i, j, k)x \times r(i, j, k)$
20: **end if**
21: **end for**
22: $__syncthreads$
23: **for** $stride = (NX - 2)/2$ to $stride > 0$ with $stride/ = 2$ at each step **do**
24: **if** $(i - 1) < stride$ **then**
25: $t[i - 1]+ = t[stride + i - 1]$
26: **end if**
27: $__syncthreads$
28: **end for**
29: $sum+ = t[0]$
30: **end if**
31: **end for**
32: **if** $i == 1$ **then**
33: $\rho = sum$
34: $\rho_0 = sum$
35: **end if**
36: $__syncthreads$

Algorithm 16 Update q (part 3 of $_cg$ GPU kernel).

37: **for** $iteration = 0$ to max_iter **do**
38: **if** $(\rho! = 0)\&\&(\rho > tol^2 \times \rho 0)$ **then**
39: **for** $k = 0$ to NZ **do**
40: **for** $j = 0$ to NY **do**
41: **if** $(i! = 0)\&\&(i! = NX - 1)\&\&(j! = 0)\&\&(j! = NY - 1)\&\&(k! = 0)\&\&(k! = NZ - 1)$ **then**
42: $q(i, j, k) = (p(i, j, k) \times iter) - a \times (p(i-1, j, k) + p(i+1, j, k) + p(i, j-1, k) + p(i, j + 1, k) + p(i, j, k - 1) + p(i, j, k + 1))$
43: **end if**
44: **end for**
45: **end for**
46: $__syncthreads$

Algorithm 17 Update α (part 4 of $_cg$ GPU kernel).

```
47:     Algorithm: 15 lines 11 to 18
48:     t[i − 1]+ = p(i, j, k)x × q(i, j, k)
49:     Algorithm: 15 lines 20 to 31
50:     if i == 1 then
51:         α = sum
52:     end if
53:     __syncthreads
54:     if i == 1 then
55:         if α == 0 then
56:             α = 0
57:         else
58:             α = ρ/α
59:         end if
60:     end if
61:     __syncthreads
```

Algorithm 18 Update x, r (part 5 of $_cg$ GPU kernel).

```
62:     Algorithm: 15 lines 11 to 18
63:     t[i − 1]+ = p(i, j, k)x × q(i, j, k)
64:     Algorithm: 15 lines 20 to 31
65:     for k = 1 to NZ do
66:         for j = 1 to NY do
67:             if (i! = 0)&&(i! = NX − 1)&&(j! = 0)&&(j! = NY − 1)&&(k! = 0)&
    &(k! = NZ − 1) then
68:                 x(i, j, k)+ = α × p(i, j, k)
69:                 r(i, j, k)− = α × q(i, j, k)
70:             end if
71:         end for
72:     end for
73:     __syncthreads
```

result of its implicit formulation. The implemented sparse linear system iterative solvers give good results in 2D. In 3D the small number of iterations performed, to keep real-time performance, affect the visual quality. This results from the numerical error in the advection step. This can be minimized by using other techniques such as MacCormack advection.

Specific water simulation scenarios are difficult to achieve. Therefore, one of the most difficult parts in fluid simulations (as previously mentioned) is its controllability. Thus, one of the most excruciating parts of this thesis was the implementation of the control user interface.

CUDA is not complicated to learn. However, when using CUDA many implementation details must be taken in consideration (e.g. size of the blocks, grid distribution, etc.). Therefore, when migrating the CPU-based version of 3D stable fluids to CUDA, the biggest

Algorithm 19 Update β, and p (part 6 of $_cg$ GPU kernel).

```
74:     if i == 1 then
75:       ρ_old = ρ
76:     end if
77:     __syncthreads
78:     Algorithm: 14 lines 11 to 35
79:     __syncthreads
80:     if i == 1 then
81:       ρ = sum
82:     end if
83:     __syncthreads
84:     if i == 1 then
85:       if ρ_old == 0 then
86:         β = 0
87:       else
88:         β = ρ/ρ_old
89:       end if
90:     end if
91:     __syncthreads
92:     for k = 1 to NZ do
93:       for j = 1 to NY do
94:         if (i! = 0)&&(i! = NX − 1)&&(j! = 0)&&(j! = NY − 1)&&(k! = 0)&
      &(k! = NZ − 1) then
95:           p(i, j, k) = r(i, j, k) + β × p(i, j, k)
96:         end if
97:       end for
98:     end for
99:     __syncthreads
100:    Enforce Boundary Conditions
101:   end if
102: end for
```

obstacle encountered was in the optimization of the CUDA code. The Jacobi and Conjugate Gradient versions under CUDA require more device global memory consumption. The CUDA implementation of stable fluids consumes a lot of device global memory, which limits maximum grid size allowed.

Chapter 4

Water Visualization

The simulation scenario consists in water, coming from a pipe with velocity given by a water pump, that pours to a container. At any time, a sphere may be dropped inside the container (see Fig. 4.1), resulting in a splash when the sphere hits the water surface.

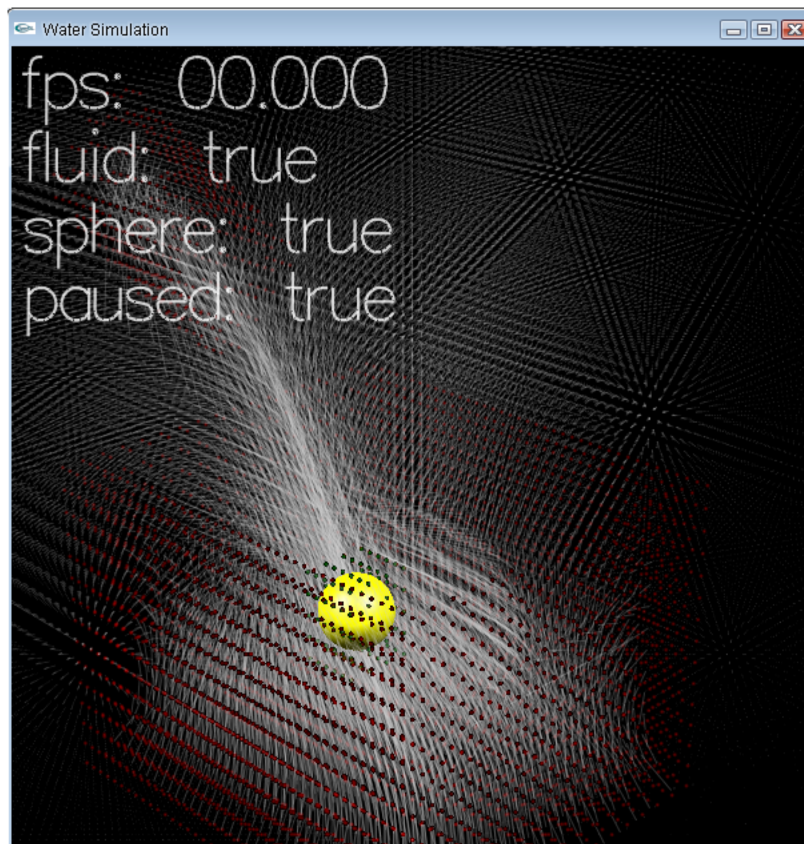

Figure 4.1: Screenshot of the GPU-based version, showing only velocity and bounds.

The scene is repeatedly rendered whenever the physically-based solver (i.e. stable fluids

method) updates the values of velocities and density. The scene rendering procedure is described in Algorithm 20.

Algorithm 20 Rendering of the scene.

```
 1: Render world
 2: if Render bounds is active then
 3:     Render bounds
 4: end if
 5: Render falling sphere
 6: Deactivate depth mask
 7: if Render fluid is active then
 8:     Render fluid density
 9: else
10:     Render fluid velocity
11: end if
12: if Render bounds is not active then
13:     Render pipe
14:     Render water container
15: end if
16: Render information
```

As observed in Algorithm 20, rendering the scene means rendering the world, the falling sphere, the fluid density or velocity, the pipe, and the water container. The user may choose between rendering the world, the pipe, and the water container, or the simulation internal and moving bounds.

The pipe is drawn using the `gluCylinder` subroutine (from the GLU library). The pipe rendering is equal in both CPU and GPU-based versions. After drawing the pipe, with a certain size, we must place it at the correct position, in the scene, applying rotations with `glRotatef` and translations with `glTranslatef`.

With the exception of the pipe, everything in the scene is rendered using vertex buffer objects (VBOs). In this thesis two types of VBOs were used: standard OpenGL VBOs and CUDA-OpenGL VBOs. The container, the falling sphere, and the world, in the CPU and GPU-based versions, were all rendered using standard OpenGL VBOs. The world and container VBOs data (i.e. number of points and normals) was static and of small size, while the falling sphere VBOs data was for a small number of points. Therefore, either for the container, the falling sphere, or the world rendering no significant gains would be obtained by the use of CUDA-OpenGL interoperability VBOs. Also, doing so avoided unnecessary kernel calls. The simulations boundaries (i.e. internal and moving), and the fluids velocity or density were the only renderings done using CUDA-OpenGL interoperability VBOs, in the GPU-based version.

The world rendering requires the loading of three textures. The scenario is rendered using a texture cube map (Fig. 4.2). Only the visible sides of the scenario are rendered. The camera is placed at a position such the user only sees the back, left and floor sides of the world. In Fig. 4.2 the camera was moved backward to better illustrate the world rendered

faces.

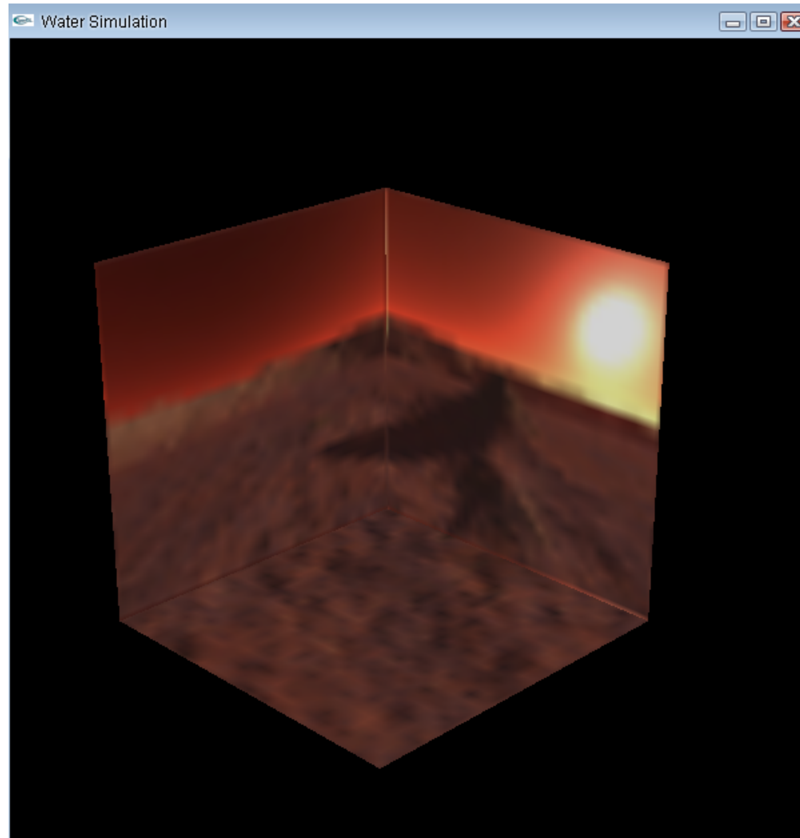

Figure 4.2: Visible faces of the world.

No volume reconstruction technique was used to render water; we used splatting instead. Splatting the volume means to render it with a set of disks using the back-to-front order. Each disk is a grid cell that contains water (i.e. its density is superior to 0). Since the goals of this thesis were oriented to use CUDA and OpenGL, this chapter is dedicated to explain in which way CUDA and OpenGL can be used to render water. To understand the mechanics of CUDA-OpenGL VBOs, we used a try-and-error approach;hence the importance of this chapter.

4.1 OpenGL VBOs

Beginners learning the first steps to OpenGL are introduced to its immediate mode. This mode consists in providing the data (i.e. coordinates, color) to render each of the vertexes, including normals for illumination purposes. However, immediate mode is ineffective in rendering amounts of data. In this case, vertex arrays or display lists are better alternatives. VBOs are an improvement over vertex arrays and display lists to get an even better OpenGL rendering performance.

The first to take in consideration in using VBOs is that rendering in done by the graphics card, also referred as server. A process running on the CPU side, also referred as client, is the entity that instructs the server to render when necessary. In conventional approaches (i.e. immediate mode or displays lists or vertex arrays) the data to required render is stored exclusively either in the client or in the server. If the data is stored in the client, the server needs to receive such data before rendering it. This communication is expensive and has impact on overall performance.

Vertex array approach stores data in the client side, and the data is sent to the server all in once when required. The vertex array approach significantly reduces the number of functions calls, when compared to OpenGL immediate mode. However, using vertex arrays implies that data must be resent to the server, each time rendering is performed.

Display lists approach stores data directly in the server. However, this data is static, which means it cannot be modified. A great advantage of both display lists and textures is that their data can be accessed by several clients simultaneously.

VBOs approach involves the creation of a transfer buffer; hence the term vertex buffer objects (VBOs). This buffer is in the server side. The data on the buffer can be mapped to the client memory space, to be altered, and then uploaded back to the buffer. This approach promotes efficient data transfer, while keeping the positive aspects of both display lists and vertex arrays, without their respective drawbacks. VBOs approach allows for sharing the buffer objects among several clients.

The general procedure to use VBOs is described in Algorithm 21. The data uploaded to the buffer (line 3 of Algorithm 21), comes from clients memory. Drawing the data in the buffer (line 4 of Algorithm 21), is similar to draw vertex arrays. First, the client state is enabled for the data contained in the VBOs (i.e. vertex, normals, colours, texels). This is done calling `glEnableClientState`. This means you may have a generic VBO with all kinds of permitted data (i.e. vertex, normals, colours, texels). Besides, we may select only part of data (e.g. only vertex and color data) in the VBO for rendering. This is done by enabling the respective client states. After enabling the target client states, we are ready for drawing. In this thesis only `glDrawArrays` was used. However, there are other alternatives. Finally, the clients state are deactivated with `glDisableClientState`. Also, it is important to point out that this is the simplest approach to use VBOs. Other alternatives exist in the literature, but they are not described here since they have not been used in this work.

4.2 CUDA-OpenGL Interoperability VBOs

When trying to learn how to use VBOs, one may encounter several VBOs examples for standard OpenGL VBOs (abundantly available on the internet). However, the same can not be said about finding free proper extensive tutorials. It becomes even a worse scenario when trying to learn how to use CUDA-OpenGL VBOs. Aside from sparse examples from CUDA SDK and one or two papers, not much information can be found when trying to learn about this subject. This section is intended to alleviate that lack of information.

CUDA-OpenGL interoperability allows the VBOs data to be uploaded in the server side.

Algorithm 21 OpenGL VBOs methodology.

1: Create buffer object (`glGenBuffers`).
2: **while** graphics application requires rendering **do**
3: Bind the buffer object (`glBindBuffer`).
4: Upload data from the clients to the buffer (`glBufferData`).
5: Point to the buffer data to draw (`glNormalPointer` or `glVertexPointer` or `glColorPointer` or `glTexCoordPointer`).
6: Draw data in the buffer (`glEnableClientState`, `glDrawArrays`, and `glDisableClientState`).
7: **end while**
8: Delete buffer object (`glDeleteBuffers()`).

The procedure to use CUDA-OpenGL interoperability VBOs is slightly different from the standard OpenGL VBOs (see Algorithm 22).

Algorithm 22 CUDA-OpenGL Interoperability VBOs methodology.

1: Create buffer object (`glGenBuffers`).
2: **while** graphics application requires rendering **do**
3: Register the buffer data that is going to be uploaded using CUDA (`glBindBuffer`, `glBufferData`, `glNormalPointer` or `glVertexPointer` or `glColorPointer` or `glTexCoordPointer`, `cudaGLRegisterBufferObject`).
4: Map buffer object for writing from CUDA (`cudaGLMapBufferObject`).
5: Fill VBOs data from the server side (using a CUDA kernel).
6: Unmap buffer object for writing from CUDA (`cudaGLUnmapBufferObject`).
7: Bind the buffer object (`glBindBuffer`).
8: Point to the buffer data to draw (`glNormalPointer` or `glVertexPointer` or `glColorPointer` or `glTexCoordPointer`).
9: Draw data in the buffer (`glEnableClientState`, `glDrawArrays`, and `glDisableClientState`).
10: Unregister each of the buffer object data fields (i.e. vertex, color, etc) in CUDA (`cudaGLUnregisterBufferObject`).
11: **end while**
12: Delete buffer object (`glDeleteBuffers()`).

The main difference from the standard OpenGL VBOs is that CUDA needs to know that the buffer exists, before mapping the data to it (lines 2 to 5 of Algorithm 22). This is achieved by temporarily register the buffer, to perform data upload using CUDA (line 2 of Algorithm 22). During registration it is also required to state the size of the buffer with `glBufferData`. Than it remains to map the data to the buffer using CUDA (lines 3 to 5 of Algorithm 22). With the data in the buffer, we have only to indicate which data is going to be drawn, and to draw it (lines 6 to 8 of Algorithm 22). After drawing the data, the buffer must be unregistered. The main gain in performance, when using CUDA-OpenGL

interoperability VBOs, comes from the reduction in the VBOs data filling time.

4.3 Final Remarks

For learning purposes, OpenGL immediate mode is clearly the most intuitive choice. However, for full graphics applications, specifically for real-time purposes rendering of massive data, they are inadequate. VBOs are the most complete alternative approach, replacing both display lists and vertex arrays alternatives. VBOs possess better performance than display lists and vertex arrays, while maintaining most of their positive aspects. The main drawback of VBOs is the update of the data arrays, such as it was the case when using vertex arrays. However, using OpenGL-CUDA interoperability alleviates considerably this problem. For small data amounts, VBOs performance differs almost nothing, regardless of whether using either OpenGL VBOs or CUDA-OpenGL interoperability VBOs. However, the gains are significant when we use CUDA-OpenGL interoperability VBOs for large amounts of data.

Chapter 5

Experimental Results

This chapter is dedicated to a comparative analysis between the GPU and CPU-based versions. These implementations were tested on a Intel(R) Core(TM)2 Quad CPU Q6600 at 2.40GHz with 4096MBytes of DDR2 RAM, and an NVIDIA GeForce 8800 GT graphics card. First, we will provide a comparative analysis of the computational cost of both implementations. Second, we will compare the simulation visual output between both versions. Finally, we draw some conclusions from the results.

5.1 Computational Performance

Taking into account both CPU-based and GPU-based implementations, the following four tables and respective charts describe the performance of each solver: Jacobi, Gauss-Seidel, and Conjugate Gradient. The first two tables (Tables 5.1 and 5.2), and respective charts (Figs. 5.1 and 5.2), describe the frame rate (`Frames Per Second`) of the simulations for each solver, for a given number of iterations (`#Iterations`). The last two tables (Tables 5.3 and 5.4), and respective charts (Figs. 5.3 and 5.4), account the time took to perform one simulation time step (`Time (ms)`), also for a given number of iterations (`#Iterations`), without considering rendering of the simulation. In the first and second two tables, the performance results were extracted in two stages of the simulation. First, while water was being poured from the pipe to the container. Second, after the container has been filled with water, and no more water came from the pipe. We only study here maximum grid sizes (`Grid Size`) that ensure real-time performance, i.e. $64 \times 32 \times 32$ in the GPU-based version, and $32 \times 32 \times 32$ in the CPU-based version.

The first conclusion to draw from these tables (Tables 5.1 to 5.4) an respective charts (Figs. 5.1 to 5.4), is that the simulation time and, consequently, its frame rate vary, from the first (i.e. while filling the container) to the second stage (i.e. after filling the container) of the simulation, either in the CPU-based or in the GPU-based versions. This variation results from the simulation time, which is consumed by the control user interface. Depending on the stage of the simulation, the user control interface will take more time while filling the container or less time after filling the container. These variations are only noticeable, in the GPU-based version frame rate results (`Frames Per Second`), described in Tables 5.1

	Grid Size	Frames Per Second (fps)	#Iterations
Jacobi CPU		20	6
Gauss-Seidel CPU	$32 \times 32 \times 32$	22	4
Conjugate Gradient CPU		24	4
Jacobi GPU		58	6
Gauss-Seidel GPU	$64 \times 32 \times 32$	48	8
Conjugate Gradient GPU		8	4

Table 5.1: Frame rate of each solver in both versions, while filling the container.

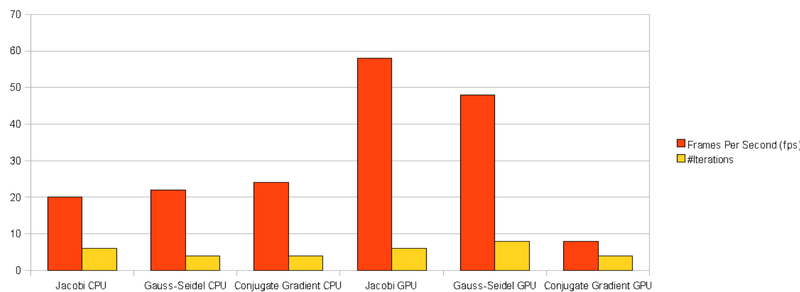

Figure 5.1: Frame rate of each solver in both versions, while filling the container.

	Grid Size	Frames Per Second (fps)	#Iterations
Jacobi CPU		24	6
Gauss-Seidel CPU	$32 \times 32 \times 32$	26	4
Conjugate Gradient CPU		25	4
Jacobi GPU		55	6
Gauss-Seidel GPU	$64 \times 32 \times 32$	45	8
Conjugate Gradient GPU		8	4

Table 5.2: Frame rate of each solver in both versions, after filling the container.

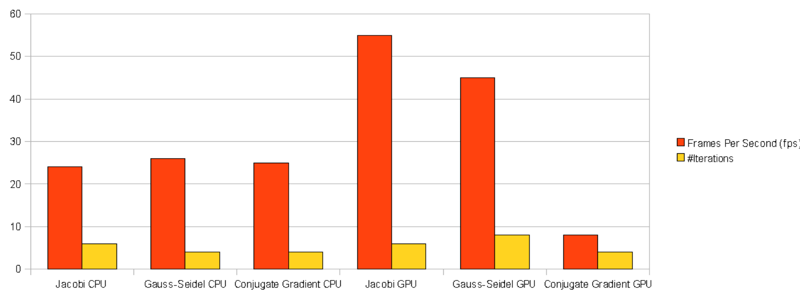

Figure 5.2: Frame rate of each solver in both versions, after filling the container.

	Grid Size	Time (ms)	#Iterations
Jacobi CPU		$5,01$	6
Gauss-Seidel CPU	$32 \times 32 \times 32$	$4,17$	4
Conjugate Gradient CPU		$4,05$	4
Jacobi GPU		$1,39$	6
Gauss-Seidel GPU	$64 \times 32 \times 32$	$1,34$	8
Conjugate Gradient GPU		$1,48$	4

Table 5.3: Time taken by one simulation step for each solver in both versions, while filling the container.

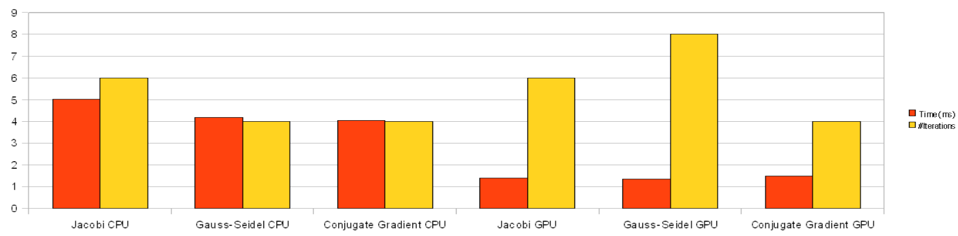

Figure 5.3: Time taken by one simulation step for each solver in both versions, while filling the container.

	Grid Size	Time (ms)	#Iterations
Jacobi CPU		$4,01$	6
Gauss-Seidel CPU	$32 \times 32 \times 32$	$3,57$	4
Conjugate Gradient CPU		$3,85$	4
Jacobi GPU		$1,39$	6
Gauss-Seidel GPU	$64 \times 32 \times 32$	$1,34$	8
Conjugate Gradient GPU		$1,48$	4

Table 5.4: Time taken by one simulation step for each solver in both versions, after filling the container.

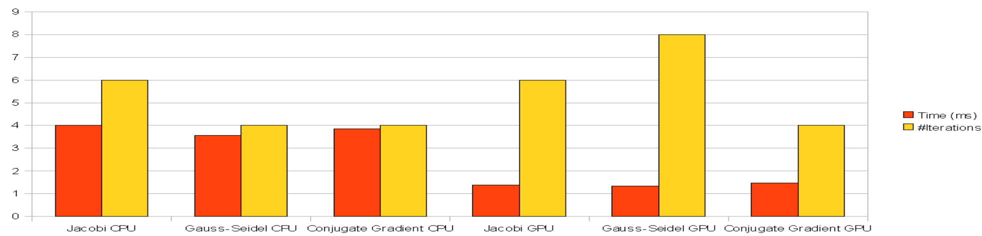

Figure 5.4: Time taken by one simulation step for each solver in both versions, after filling the container.

and 5.2. However, if this behaviour results from the control user interface, this might seem a contradiction. The reason why it is not a contradiction can be explained by the fact that the time values (`Time (ms)`) in Tables 5.3 and 5.4 were rounded to two digits after the floating point. The only exception that does not present these variations is the GPU-based conjugate gradient.

It is clear that the gains obtained by the GPU-based version, both in rendering and simulation time, justify the usage of CUDA and CUDA-OpenGL interoperability VBOs. As expected, the GPU-based implementation is faster than the CPU-based one. However, for grid sizes bigger than $64 \times 32 \times 32$, the GPU takes to much time in rendering. Therefore, to expand the GPU-based simulation to allow bigger grid sizes, a different rendering technique (i.e. just render the water surface elements, or use a real-time parallel volume reconstruction technique) should be used.

The best CPU-based solver is the conjugate gradient. Conjugate gradient is not always the fastest solver. However, it does not suffer from variations in the frame rate, and the difference to Gauss-Seidel solver is barely zero. On the other hand, the Jacobi solver is the best GPU-based implementation. Also, as expected, it clearly surpasses the time performance results (`Time (ms)`) of the best CPU-based solver (Tables 5.3 and 5.4, and respective charts in Figs. 5.3 and 5.4).

The reader may notice that according to Tables 5.3 and 5.4, and respective charts in Figs. 5.3 and 5.4, the conjugate gradient is quicker in the GPU-based version, but its frame rate (`Frames Per Second (fps)`) is worse than the one of the CPU-based version. The reason is that, all the GPU-based solvers suffer from timeouts. These timeouts result from device global memory access latency. These timeouts appear in average once at every ten simulation steps. The Jacobi and Gauss-Seidel GPU-based solvers, however, hide them quit well. The same could not be ensured in the implementation of the GPU-based conjugate gradient solver. Since each solver must be done for a certain number of iterations to avoid a kernel invocation bottleneck the conjugate gradient has been converted to one massive kernel. This kernel does not allow the use of an optimized block size, because the conjugate gradient solver requires dot product operations. Therefore, in this kernel the global memory access latency cannot be hidden, which will result in visible timeouts. These timeouts are so big, that the gain in the other average nine steps does not pay off.

Table 5.5, and the respective chart in Fig. 5.5, show the maximum timeouts (`Time Out (ms)`) observed for each solver, given a number of iterations ($\#Iterations$). If we reduced the number of iterations to one, the conjugate gradient kernel might be redesigned to alleviate these timeouts.

	Time Out (ms)	#Iterations
Jacobi	15.35	6
Gauss-Seidel	$18,87$	8
Conjugate Gradient	$124,77$	4

Table 5.5: Timeouts resulting from device global memory access latency.

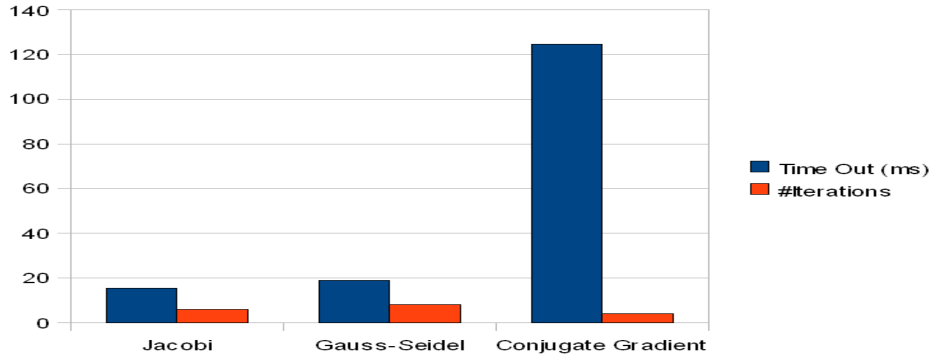

Figure 5.5: Timeouts resulting from device global memory access latency.

5.2 Visual Performance

Looking at Figs. 5.6 and 5.7, the reader may observe several details. The visual quality of both CPU and GPU-based Gauss-Seidel and Jacobi solvers is very similar. The Conjugate Gradient method, in both versions, gives slightly better water shaped effects than in the other two methods. However, the GPU-based version of Conjugate Gradient is only used on the diffusion of velocities and densities. Possibly due to floating point errors, the GPU-version does not give the correct results in the projection step. Therefore, instead of using the Conjugate Gradient, we use the Gauss-Seidel or Jacobi solvers in the projection step. The visual aspect of both the GPU and CPU-based versions differ mostly from the usage of different grid sizes. The rendering technique used for grid sizes smaller then $128x64x64$ does not produce an uniform volume; consequently, we cannot distinguish the individual elements (i.e. we see the dots individually and not a continuous water volume). For a grid size of $128x64x64$, the rendering technique resulted in a uniform water volume, where the individual water elements could not be distinguished from each other. However, none of the versions allowed such grid size because all water elements, and not only the water surface elements, were rendered. The reader will notice also that the textures of the GPU-based version look better. This results from the fact that the texture dimensions used are equivalent to the biggest grid dimension allowed. Therefore, the CPU-based version uses 32×32 textures, while the GPU-based version uses 64×64 textures. Another

5.3 Final Remarks

The GPU-based version with a grid size of $64 \times 32 \times 32$ clearly surpassed the results of the CPU-based version with a grid size of $32 \times 32 \times 32$. The GPU-based version has better results both in rendering speed (i.e. the simulation frame rate is higher than the one of the CPU-based version) and simulation time, with the exception of the conjugate gradient solver, where the kernel device global memory latency timeouts are to big too compensate its gains. When compared to the CPU based version, the GPU-based version

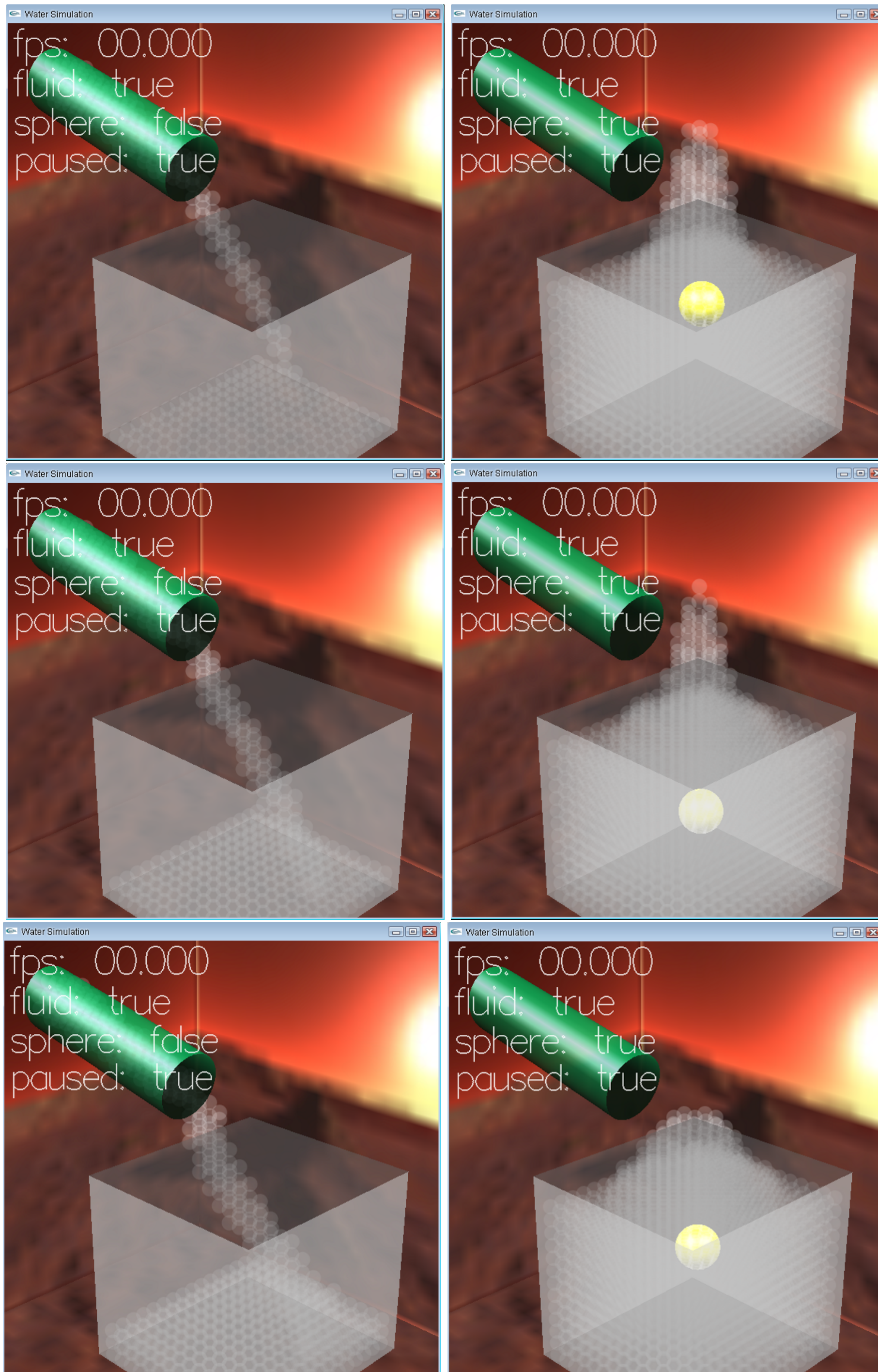

Figure 5.6: CPU-based version using respectively Jacobi, Gauss-Seidel, and Conjugate Gradient solvers.

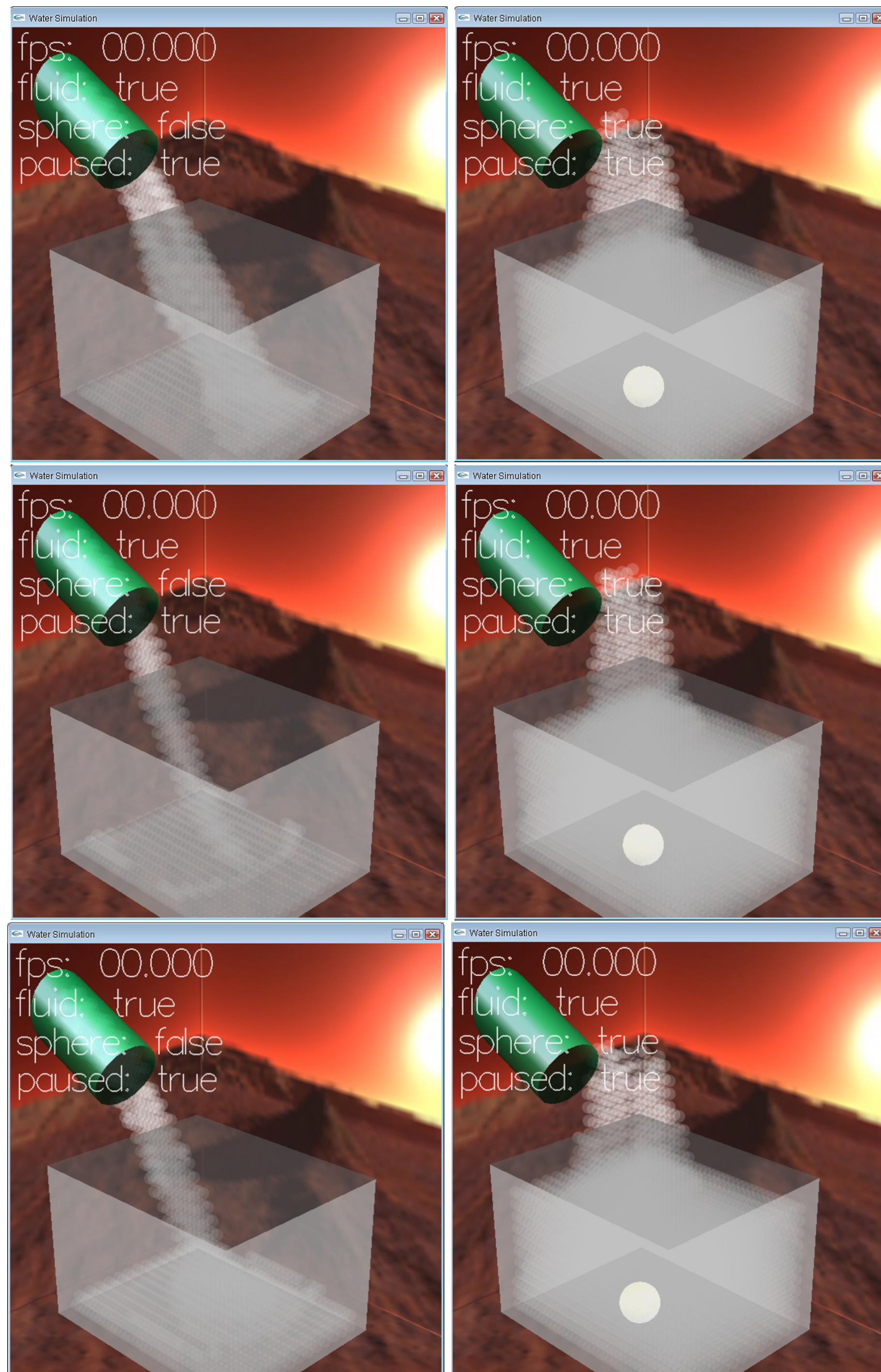

Figure 5.7: GPU-based version using respectively Jacobi, Gauss-Seidel, and Conjugate Gradient solvers.

is more scalable, because, with a better (i.e. faster) rendering technique, it allows real-time simulations with grid sizes up to $128 \times 128 \times 128$. This grid size limitation exists as a result of the global memory required by the current GPU-based version.

The control user interface can affect the overall performance. Therefore, a particular care must be taken while designing/programming it.

The best solver in the CPU-based version is the conjugate gradient, for a grid size of $32 \times 32 \times 32$. Despite not being always the fastest, its frame rate is the one that varies less, and its the solver where better looking water shapes are achieved. In the GPU-based version Jacobi is the best solver, because it is the fastest. The current GPU-based implementation of conjugate gradient is useless for real-time simulations. However, for one iteration its kernel may be redesigned, to allow real-time performance.

The last important consideration is that realistic-looking water is not obtainable with the current rendering technique. However, when tested for a grid size of $128 \times 64 \times 64$ the fluid looks realistic (i.e. we add an uniform water mass, where we could not distinguish the individual water elements).

Chapter 6

Conclusions and Future Work

This dissertation describes the implementation of a CUDA 3D stable fluids version that supports internal and moving boundaries. This CUDA implementation allows real-time performance, for bigger grid sizes than the CPU-based version. The GPU shader-based version of 3D stable fluids, implemented with the Cg language, from Keenan et al [31], supports a grid size of $128 \times 64 \times 64$. The CUDA implementation, described in this dissertation, does not achieve real time performance, for a grid size of $128 \times 64 \times 64$, with the rendering technique used . Therefore, the GPU CUDA-based implementation made in this dissertation is better than the CPU-based version, but inferior than the GPU Cg language shader-based version. Thus, the GPU Cg language shader-based version of 3D stable fluids is the best implementation of 3D stable fluids for the GPU.

CUDA is not complicated to learn. However, when using CUDA many implementation details must be taken in consideration (e.g. size of the blocks, grid distribution, etc.). Therefore, when migrating the CPU-based version of 3D stable fluids to CUDA, the biggest obstacle encountered was in the optimization of the CUDA code. There are several concerns and limitations that CUDA imposes which difficult its usage, namely, limited data structures usage, recursivity can not be used, small amount available global memory when compared with the systems RAM, global memory is not cached, etc.. In spite of these difficulties, the migration of stable fluids to CUDA was achieved.

The CUDA-based version has space for improvement, namely, with the use of a different rendering technique. The Jacobi and conjugate gradient versions using CUDA require additional device global memory consumption. The CUDA implementation of stable fluids consumes a lot of device global memory, which limits the maximum allowed grid size, depending on the graphics card global memory (i.e. $128 \times 128 \times 128$ for the NVIDIA GeForce 8800 GT graphics card). The GPU-based implementation of conjugate gradient is useless for real time applications. However, as already referred, the solver's kernel can be improved to achieve real-time performance (i.e. hiding its timeouts to allow real-time performance).

There has clearly been some effort in running physically-based water simulations on the GPU over the past half decade. However, aside from the recent LBM, 3D grid-based methods do not seem appropriate for large grid size real-time applications. In terms of visual quality, off-line computer graphics fluid simulations are far more advanced than real-time fluid simulations. However, this is not surprising because off-line techniques allow the

usage of high quality volume reconstruction techniques.

Currently, most research on water simulation focus on the improvement of the existing methods (i.e. procedural water, LBM, Lagrangian methods, and Eulerian methods) to consume less resources or to run faster (i.e. faster in rendering and in the physical model). Nevertheless, new methods can appear, as it was the case with the recent LBM. Taking in consideration the recent interest in parallel computing, such new methods will preferably run on parallel architectures, in particular GPU. Also, there is not a fully complete model for real-time that allows large masses of water (e.g. ocean near shore), with all possible water effects (i.e. splashes, bubbles, foam, breaking waves, etc). Particle systems allow most, not to say all, water effects. However, particle approach-based simulations require huge amounts of particles for large water masses. Grid-based methods (i.e. LBM or Eulerian methods) are able to simulate large water masses (e.g. ocean surface). However, they are not capable of simulation effects such as, for example, splashes, bubbles, foam, breaking waves. Hybrid methods that interface the Lagrangian and Eulerian methods can become the most complete (i.e. that allow water simulations with all water desired effects) water simulation methods.

The principal disadvantages of grid-based techniques are their memory requirements, and the difficulty in rendering an 3D water volume. Therefore, we need better data structures to storage the grid information that do not degrade performance of parallel implementations of fluid simulations. Also, rendering is one of the major concerns for the current state-of-the-art of water simulation models. Alleviating the rendering load of volume reconstruction techniques or searching new rendering techniques is of the utmost importance for physically-based water simulations. Despite the considerable visual quality of off-line simulations, their acceleration needs further investigations in the future, namely, for the movie industry.

The full 3D stable fluids still hold some possibilities of imposing in real-time simulations, as a replacement to procedural water or heightfields. However, this is mainly dependent on reducing 3D stable fluids memory requirements, and an the development of efficient real-time volume reconstruction technique.

6.1 Contributions

A CUDA-based 3D version of stable fluids, that allows internal and moving boundaries, is the main contribution of this thesis. Even so, many improvements can be made to this implementation, as noted in the next section.

An explanation on how to use CUDA-OpenGL interoperability VBOs, is the second but far less valuable, contribution of this thesis.

6.2 Future work

Further work will focus on extending the water simulation, and on improving the performance of the current GPU-based version (if possible). Among other improvements, I intend to:

- Add refraction, reflection, and underwater caustics to the simulation, to improve its visual realism.
- Reduce the dissipation due to numerical errors in the Semi-Lagrange advection step. This can be achieved by using the MacCormack advection, instead of the Semi-lagrange advection.
- Implement other, less time consuming, rendering techniques or a volume reconstruction technique, such as ray-tracing, or marching cubes (MCs), etc. to improve the water visual quality.
- Allow the loading of scenario models. The internal boundaries data would be extracted from these scenarios data (instead of the data files as they are now), using MCs or other volume reconstruction technique.
- Transform the simulation into an hybrid simulation, by incorporating SPH along aside the stable fluids method.
- Add an graphical interface to the simulation, that would allow the generation of different simulations. This interface would allow to change the parameters of the simulation, to add user positioned moving boundaries, to load different kinds of terrain modules, etc.

Bibliography

[1] NVIDIA CUDA programming guide 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, Jul 2008.

[2] NVIDIA CUDA reference manual 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf, Jun 2008.

[3] Ann S. Almgren, John B. Bell, Phillip Colella, and Tyler Marthaler. A cartesian grid projection method for the incompressible euler equations in complex geometries. *SIAM J. Sci. Comput.*, 18(5):1289–1309, Sep 1997.

[4] Michael Ash. Simulation and visualization of a 3d fluid. Master's thesis, Université d'Orléans, France, Sep 2005.

[5] Vladimir Belyaev. Real-time simulation of water surface. In *GraphiCon-2003*, pages 131–138. OOO "MAX. Press", 2003.

[6] Robert Bongart. Efficient simulation of fluid dynamics in a 3d game engine. Master's thesis, KTH Computer Science and Communication, Stockholm Sweden, 2007.

[7] Robert Bridson, Ronald Fedkiw, and Matthias Müller-Fischer. Fluid simulation: SIGGRAPH 2006 course notes. In *ACM SIGGRAPH 2006 Course Notes (SIGGRAPH '06)*, pages 1–87, New York, NY, USA, 2006. ACM Press.

[8] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: SIGGRAPH 2007 course notes. In *ACM SIGGRAPH 2007 Course Notes (SIGGRAPH '07)*, pages 1–81, New York, NY, USA, 2007. ACM Press.

[9] Mark Thomas Carlson. *Rigid, melting, and flowing fluid*. PhD thesis, Atlanta, GA, USA, Jul 2004.

[10] Shiyi Chen and Gary D. Doolen. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30:329–364, 1998.

[11] Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. Liquid simulation on lattice-based tetrahedral meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '07)*, pages 219–228, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association Press.

[12] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH '88)*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[13] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21(3):736–744, 2002.

[14] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*, 15–22, New York, NY, USA, 2001. ACM Press.

[15] Ronald P. Fedkiw. Coupling an eulerian fluid calculation to a lagrangian solid calculation with the ghost fluid method. *J. Comput. Phys.*, 175(1):200–224, 2002.

[16] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*, pages 23–30, New York, NY, US, 2001. ACM Press.

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

[18] Nick Foster and Dimitris Metaxas. Modeling water for computer animation. *Commun. ACM*, 43(7):60–67, Jul 2000.

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

[20] Fréchot-Jocelyn. Realistic simulation of ocean surface using wave spectra. In *Proceedings of the First International Conference on Computer Graphics Theory and Applications (GRAPP 2006)*, pages 76–83, 2006.

[21] Génevaux-Olivier, Arash Habibi, and Jean-Michel Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pages 31–38, Halifax, Nova Scotia, Jun 2003. CIPS, Canadian Human-Computer Commnication Society, A K Peters Press. ISBN 1-56881-207-8, ISSN 0713-5424.

[22] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, 1989.

[23] Nolan Goodnight. CUDA/Opengl fluid simulation. `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#postProcessGL`, 2007.

[24] S. T. Greenwood and D. H. House. Better with bubbles: enhancing the visual realism of simulated fluid. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '04)*, pages 287–296, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association Press.

[25] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, Dec 1965.

[26] Mark J. Harris. Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Course Notes (SIGGRAPH '05)*, number 220, New York, NY, USA, 2005. ACM Press.

[27] Nathan Holmberg and Burkhard Wünsche-C. Efficient modeling and rendering of turbulent water over natural terrain. In *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '04)*, pages 15–22, New York, NY, USA, 2004. ACM Press.

[28] Stefan Jeschke, Hermann Birkholz, and Heidrun Schumann. A procedural model for interactive animation of breaking ocean waves. In *The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003 (WSCG '2003)*, 2003.

[29] Claes Johanson. Real-time water rendering - introducing the projected grid concept. Master's thesis, Lund University, Mar 2004.

[30] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'90)*, pages 49–57. ACM Press, 1990.

[31] Crane Keenan, Llamas Ignacio, and Tariq Sarah. Real-time simulation and rendering of 3d fluids. In Nguyen Hubert, editor, *GPU Gems 3*, chapter 30, pages 633–675. Addison Wesley Professional, Aug 2007.

[32] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games (SI3D '08)*, pages 99–106, 2008.

[33] Körner-C., M. Thies, T. Hofmann, Thürey-Nils, and Rüde-Ulrich. Lattice Boltzmann model for free surface flow for modeling foaming. *Journal of Statistical Physics*, 121(1–2):179–196, Oct 2005.

[34] Anita T. Layton and Michiel van de Panne. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*, 18(1):41–53, Feb 2002.

[35] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988.

[36] Shaofan Li and Wing Kam Liu. Meshfree and particle methods and their applications. *Applied Mechanics Reviews*, 55(1):1–34, 2002.

[37] Harvard Lomax, Thomas H. Pulliam, and David W. Zingg. Fundamentals of computational fluid dynamics. http://maji.utsi.edu/courses/07_681_advanced_viscous_flow/ref_af6_Fundamentals_of_CFD.pdf, Aug 1999.

[38] William E Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[39] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 (SIGGRAPH '04)*, pages 457–462, New York, NY, USA, 2004. ACM Press.

[40] Antoine McNamara, Adrien Treuille, Popović-Zoran, and Jos Stam. Fluid control using the adjoint method. *ACM Trans. Graph.*, 23(3):449–456, 2004.

[41] Bálint Miklós. Real time fluid simulation using height fields semester thesis. http:\www.balintmiklos.com/layered_water.pdf, 2009.

[42] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, 1989.

[43] Andreas Monitzer. Fluid simulation on the GPU with complex obstacles using the Lattice Boltzmann method. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria, Jul 2008.

[44] Müller-Matthias, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '03)*, Aire-la-Ville, Switzerland, 2003. Eurographics Association Press.

[45] Müller-Matthias, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus Gross. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds (CAVW)*, 15(3–4):159–171, Jul 2004.

[46] Müller-Matthias, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '05)*, pages 237–244, New York, NY, USA, 2005. ACM Press.

[47] Müller-Matthias, Jos Stam, Doug James, and Thürey-Nils. Real time physics: SIGGRAPH 2008 class notes. In *ACM SIGGRAPH 2008 Class Notes (SIGGRAPH '08)*, pages 1–90, New York, NY, USA, 2008. ACM Press.

[48] Karsten Noe. Implementing rapid, stable fluid dynamics on the GPU. http://projects.n-o-e.dk/?page=show&name=GPU%20water%20simulation, 2004.

[49] James F. O'Brien and Jessica K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation (CA '95)*, pages 198–205, Washington, DC, USA, Apr 1995. IEEE Computer Society Press.

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

[51] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron E. Lefohn, and Ross T. Whitaker. Particle-based simulation of fluids. *Comput. Graph. Forum*, 22(3):401–410, 2003.

[52] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, Apr 1983.

[53] Schläfli-Jür. Simulation of fluid-solid interaction. Diploma thesis. `http://www.ifi.uzh.ch/archive/mastertheses/DA_Arbeiten_2005/Schlaefli_Juerg.pdf`, Dec 2005.

[54] Brian Schlatter. A pedagogical tool using smoothed particle hydrodynamics to model fluid flow past a system of cylinders. Technical report, Oregon State University, 1999.

[55] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35(2–3):350–371, Jun 2008.

[56] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. In *ACM SIGGRAPH 2005 (SIGGRAPH '05)*, pages 910–914, New York, NY, USA, 2005. ACM Press.

[57] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, Aug 1994.

[58] Althea De Souza. How to understand computational fluid dynamics jargon. `http://www.nafems.org/downloads/edocs/how_to_understand_cfd_jargon-nafems.pdf`, 2005.

[59] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, pages 121–128, New York, NY, USA, Aug 1999. ACM Press.

[60] Jos Stam. A simple fluid solver based on the FFT. *J. Graph. Tools*, 6(2):43–52, 2001.

[61] Jos Stam. Flows on surfaces of arbitrary topology. In *ACM SIGGRAPH 2003 (SIGGRAPH '03)*, volume 22, pages 724–731, Jul 2003.

[62] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, Mar 2003.

[63] Clifford M. Stein and Nelson L. Max. A particle-based model for water simulation. Technical Report UCRL-JC-129378, Lawrence Livermore National Laboratory, Jan 1998.

[64] Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki. Realistic animation of fluid with splash and foam. *Comput. Graph. Forum*, 22(3):391–400, Sep 2003.

[65] Jerry Tessendorf. Simulating ocean water. In *ACM SIGGRAPH 2004 Course Notes (SIGGRAPH '04)*, pages 13–38, New York, NY, USA, 2004. ACM Press.

[66] Thürey-Nils, Thomas Pohl, and Rüde-Ulrich. Hybrid parallelization techniques for Lattice Boltzmann free surface flows. In *Proceedings of Parallel Computational Fluid Dynamics 2007*, pages 1–8, 2007.

[67] Thürey-Nils and Rüde-Ulrich. Free surface Lattice-Boltzmann fluid simulations with and without level sets. In *Workshop on Vision, Modelling, and Visualization VMV*, volume 2, pages 199–207, 2004.

[68] Thürey-Nils and Rüde-Ulrich. Stable free surface flows with the Lattice Boltzmann method on adaptively coarsened grids. *Computing and Visualization in Science*, 12(5):247–263, Mar 2008.

[69] Tölke-Jonas. Implementation of a Lattice-Boltzmann kernel using the compute unified device architecture developed by NVIDIA. *Computing and Visualization in Science*, Feb 2008.

[70] Pauline Y. Ts'o and Brian A. Barsky. Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Trans. Graph.*, 6(3):191–214, Jul 1987.

[71] Ye Zhao, Feng Qiu, Zhe Fan, and Arie Kaufman. Flow simulation with locally-refined lbm. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*, pages 181–188, New York, NY, USA, 2007. ACM Press.

Appendix A

Glossary

CPU: Central Processing Unit the key component of a computer system, which contains the circuitry necessary to interpret and execute program instructions.

GPU: Graphics Processing Unit specialized parallel processor. Intended to offload from the CPU, the 3D graphics rendering.

CUDA: Compute Unified Device Architecture NVIDIA parallel computing architecture, that extends the C language to allow to program to NVIDIA GPUs.

CFD: Computational Fluid Dynamics branch of fluid mechanics, dedicated to the study of fluid flows.

CG: Computer Graphics a branch of computer science dedicated to the study of methods for digitally synthesizing and manipulating visual content.

Real-time sub field of computer graphics focused on producing and studying visual appealing /convincing effects at interactive frame rates.

Off-line sub field of computer graphics focused on producing and studying high quality visual appealing /convincing effect, with no major time constrains.

Differential equations a mathematical equation that establishes a relation between an unknown function or functions, with one or more variables, and its derivatives of various orders.

PDEs: partial differential equations type of differential equation.

NS: Navier-Stokes set of partial differential equations that describe the movement of viscous fluids.

Euler equations set of partial differential equations that describe the movement of inviscid fluids (i.e. NS equations without the pressure term).

Parametric functions a curve defined by a set of parameters.

Parametric surface a Eucledian space surface defined by an parametric equation with two parameters.

Procedural methods a non physically-based computer graphics approach, that by the usage of parametric surface representations models water surfaces. One of the first computer graphics methods to simulate water.

Lagrangian viewpoint a way of describing the movement of a fluid, that consists in tracking the movement of the fluids individual elements (i.e. particles).

Eulerian viewpoint a way of describing the movement of a fluid, that consists in tracking the fluids variations over a non-moving mesh or grid.

MPS: Moving Particle Semi-implicit method CFD Lagrangian computational method used for the simulation of incompressible free surface flows.

SPH: Smoothed Particle Hydrodynamics CFD Lagrangian computational method used for the simulation of fluid flows.

SWE: Shallow Water Equations a set of hyperbolic partial differential equations that describe fluids movement bellow a pressure surface.

Heightfields a visual representation of a 2D function that for each two-dimensional point given returns a scalar value ("height") as output.

LGA: Lattice Gas Automata a simplified fictitious molecular dynamics model where all parameters are discrete, from the field of statistical physics.

LBM: Lattice Boltzmann Method a CFD model originated from the LGA model. LBM models the fluid as a set of fictitious particles, that consecutively propagate and collide in a discrete lattice mesh.

DFs: Particle Distribution Functions functions with seven variables ($f(x, y, z, t, vx, vy, vz)$), which gives the number of particles that travel at an approximate velocity (vx, vy, vz) near an spatial position (x, y, z) at a given time (t).

Stable fluids a computer graphics method to simulate fluids. It uses implicit integration to solve the NS equations, in order to ensure unconditional stability (i.e. for large time steps the simulation does not blow up, as is the case when using explicit integration methods).

Advection the transport of objects, densities, the fluid itself (self-advection) and other quantities, as the result of the fluid's velocity when moving.

Diffusion dissipation of velocity as a result of a fluid's internal resistance to flow (i.e. viscosity).

Move fluid simulation step where mass conservation, and fluid incompressibility are ensured.

Euler method a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value.

Poisson's equation a partial differential equation useful for the of electrostatics, mechanical engineering an theoretical physics.

Sparse linear system a linear system of equations in the form $Ax = b$, where most elements of the matrix A are 0.

Jacobi method an stationary iterative method, used to solve linear systems of equations.

Gauss-Seidel method an stationary iterative method, used to solve linear systems of equations.

CG method:Conjugate Gradient Method an Krylov subspace iterative method, used to solve linear systems of equations.

VBOs: Vertex Buffer Objects an OpenGL extension that allow the upload of data (i.e. vertex, normal vector, color, etc) directly to the graphics card for non-immediate-mode rendering. an OpenGL extension that allow the upload of data (i.e. vertex, normal vector, color, etc) directly to the graphics card for non-immediate-mode rendering.

HLSL: High Level Shader Language or High Level Shading Language a proprietary shading language developed for Microsoft Direct3D API.

Appendix B

CPU-based version source code

```
/*
Grid structure and used Macros
*/
typedef struct {
    int _GRID_cells; // total number of grid cells
    int NX, NY, NZ; // fluid grid number of partitions per axis
    float sizeX, sizeY, sizeZ; // fluid cell size per axis
    float *vx, *vy, *vz, *vx0, *vy0, *vz0; // velocities and previous velocities
    float *d, *d0; // density values
    char *bd; // fluid internal and external boundaries
} _GRID;

#define _NX 32 // number of fluid cells in X
#define _NY 32 // number of fluid cells in Y
#define _NZ 32 // number of fluid cells in Y
#define _IX(x, y, z, NX, NY) ((x)+(y)*(NX)+(z)*(NX)*(NY)) // index to access the ↩
    corresponding position of grid

#define FOR_EACH_CELL1\
    for(k=((g->NZ)+1);k>=0;k--) {\
    for(j=((g->NY)+1);j>=0;j--) {\
    for(i=0;i<=((g->NX)+1);i++) {
#define FOR_EACH_CELL2\
    for(k=1;k<=(g->NZ);k++) {\
    for(j=1;j<=(g->NY);j++) {\
    for(i=1;i<=(g->NX);i++) {
#define FOR_EACH_CELL3\
    for(k=1;k<=(g.NZ);k++) {\
    for(j=1;j<=(g.NY);j++) {\
    for(i=1;i<=(g.NX);i++) {
#define FOR_EACH_CELL4\
    for(k=1;k<=(g->NZ);k++) { z=((k-0.5f)*g->sizeZ)/(g->NZ);\
    for(j=1;j<=(g->NY);j++) { y=((j-0.5f)*g->sizeY)/(g->NY);\
    for(i=1;i<=(g->NX);i++) { x=((i-0.5f)*g->sizeX)/(g->NX);
#define FOR_ALL_CELLS\
    for(i=0;i<(g->_GRID_cells);i++) {
#define END_FOR_ALL }
#define END_FOR_EACH_CELL }}}

//#define _JACOBI
//#define _GAUSS_SEIDEL
#define _CG
```

```
/*
Global variables
*/
float dt; // simulation time step
float diff; // fluid diffusion
float visc; // fluid viscosity;
float force;
float source;
float posYS; // position of the fallig sphere
float water_lvl; // max water level allowed
int more_source; // toggle on/off  pouring more water into the tank
float vec_scale; // scaling of hedgehogs
int fluid; // draw the fluid or is velocities
int paused; // toggle on/off the animation
int falling_sphere; // toggle on/off drawing falling sphere
int bounds; // toggle on/off drawing internal and moving bounds

/*
Control User Interface function
*/
void _clear__GRID_data0(_GRID *g) {
    int i;

    FOR_ALL_CELLS
    g->vx0[i]=g->vy0[i]=g->vz0[i]=g->d0[i]=0.0f;
    END_FOR_ALL
}

void control_user_interface() {
    int i, j, k;
    float pos;

    _clear__GRID_data0(&g);

    // introduce water until tank reaches a certain level of water
    if(more_source)
        for(k=15;k<19;k++)
            for(j=25;j<29;j++) {
                // stop introducing liquid at a certain water level
                if(water_lvl<16)
                    for(i=1;i<5;i++)
                        g.d0[_IX(i,j,k,g.NX+2,g.NY+2)]=source;
                else {
                    // ensure no more water is introduced
                    for(i=1;i<5;i++)
                        g.d[_IX(i,j,k,g.NX+2,g.NY+2)]=0;
                    more_source=0;
                }
            }

    FOR_EACH_CELL3
    // add horizontal velocity inside the pipe to simulate a water pump
    if((j>(water_lvl+1))&&(water_lvl<16)&&(g.d[_IX(i,j,k,g.NX+2,g.NY+2)]>0.1))
        g.vx0[_IX(i,j,k,g.NX+2,g.NY+2)]=(float)(rand()%(int)force)*0.22;

    // ensure gravities force is considered only where fluid lies
    if(g.d[_IX(i,j,k,g.NX+2,g.NY+2)]>0.1) {
        if(water_lvl<16)
            g.vy0[_IX(i,j,k,g.NX+2,g.NY+2)]=-0.88;
        else
            g.vy0[_IX(i,j,k,g.NX+2,g.NY+2)]=-0.44;
    }
```

```
    if(j<(water_lvl+1))
        g.vy[_IX(i,j,k,g.NX+2,g.NY+2)]=0.0;

    // calculate the position of the sphere in grid coordinates
    pos=posYS*(g.NY+1);

    // add horizontal velocity that opposes gravity attraction, to make a splash ↩
        when the sphere hits the surface
    if((pos>(water_lvl-1.0))&&(pos<(water_lvl+1.0))&&(i>20)&&(i<25)&&(k>14)&&(k↩
        <19))
        g.vy0[_IX(i,j,k,g.NX+2,g.NY+2)]=9.0;

    // ensure that the water is only add until a certain water level
    if((j<water_lvl)&&(j>3)&&(13<i)&&(i<32)&&(7<k)&&(k<26))
        g.d0[_IX(i,j,k,g.NX+2,g.NY+2)]=source;
    END_FOR_EACH_CELL

    // increase water level until container is full
    if((water_lvl<=16)&&(g.d[_IX(28,11,16,g.NX+2,g.NY+2)]>0.1))
        water_lvl+=0.10;

    return;
}



/*
Solver functions
*/
void add_source(_GRID *g, float *x, float *s, float dt) {
    int i, size=(g->_GRID_cells);
    FOR_ALL_CELLS
    x[i]+=dt*s[i];
    END_FOR_ALL
}

void add_vel_source(_GRID *g, float *vx, float *vy, float *vz, float *vx0, float ↩
    *vy0, float *vz0, float dt) {
    int i, size=(g->_GRID_cells);
    FOR_ALL_CELLS
    vx[i]+=dt*vx0[i];
    vy[i]+=dt*vy0[i];
    vz[i]+=dt*vz0[i];
    END_FOR_ALL
}

void diffuse(_GRID *g, int b, float *x, float *x0, float diff, float dt) {
    float a=dt*diff;

    switch (b)
    {
        case 1: a *= (g->NX) * (g->NX); // Velocity x
            break;
        case 2: a *= (g->NY) * (g->NY); // Velocity y
            break;
        case 3: a *= (g->NZ) * (g->NZ); // Velocity z;
            break;
        case 0: a *= (g->NX) * (g->NY) * (g->NZ); // Density
    }

    lin_solve(g, b, x, x0, a, 1+6*a);
```

```
}

void advect(_GRID *g, float *d, float *d0, float *vx, float *vy, float *vz, float↩
     dt) {
    int i, j, k, i0, j0, k0, i1, j1, k1;
    float xf, yf, zf, s0, t0, u0, s1, t1, u1, dtx, dty, dtz;

    dtx=dt*(g->NX);
    dty=dt*(g->NY);
    dtz=dt*(g->NZ);

    FOR_EACH_CELL2
    xf=(float) i-(dtx*vx[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);
    yf=(float) j-(dty*vy[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);
    zf=(float) k-(dtz*vz[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);

    if(xf<0.5f) xf=0.5f;
    if(xf>((g->NX)+0.5f)) xf=(g->NX)+0.5f;
    i0=(int) xf; i1=i0+1;

    if(yf<0.5f) yf=0.5f;
    if(yf>((g->NY)+0.5f)) yf=(g->NY)+0.5f;
    j0=(int) yf; j1=j0+1;

    if(zf<0.5f) zf=0.5f;
    if(zf>((g->NZ)+0.5f)) zf=(g->NZ)+0.5f;
    k0=(int) zf; k1=k0+1;

    s1=xf-i0; s0=1-s1;
    t1=yf-j0; t0=1-t1;
    u1=zf-k0; u0=1-u1;

    d[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=s0*(t0*(u0*d0[_IX(i0, j0, k0, (g->NX)↩
        +2, (g->NY)+2)]+
        u1*d0[_IX(i0, j0, k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*d0[_IX(i0, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*d0[_IX(i0, j1, k1, (↩
            g->NX)+2, (g->NY)+2)])))
        +s1*(t0*(u0*d0[_IX(i1, j0, k0, (g->NX)+2, (g->NY)+2)]+u1*d0[_IX(i1, j0, ↩
            k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*d0[_IX(i1, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*d0[_IX(i1, j1, k1, (↩
            g->NX)+2, (g->NY)+2)])));
    END_FOR_EACH_CELL

    _set_bnd(g, 0, d);
}

void advect_vel(_GRID *g, float *vx, float *vy, float *vz, float *vx0, float *vy0↩
    , float *vz0, float dt) {
    int i, j, k, i0, j0, k0, i1, j1, k1;
    float xf, yf, zf, s0, t0, u0, s1, t1, u1, dtx, dty, dtz;

    dtx=dt*(g->NX);
    dty=dt*(g->NY);
    dtz=dt*(g->NZ);

    FOR_EACH_CELL2
    xf=(float) i-(dtx*vx0[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);
    yf=(float) j-(dty*vy0[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);
    zf=(float) k-(dtz*vz0[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]);

    if(xf<0.5f) xf=0.5f;
    if(xf>((g->NX)+0.5f)) xf=(g->NX)+0.5f;
```

```
    i0=(int) xf; i1=i0+1;

    if(yf<0.5f) yf=0.5f;
    if(yf>((g->NY)+0.5f)) yf=(g->NY)+0.5f;
    j0=(int) yf; j1=j0+1;

    if(zf<0.5f) zf=0.5f;
    if(zf>((g->NZ)+0.5f)) zf=(g->NZ)+0.5f;
    k0=(int) zf; k1=k0+1;

    s1=xf-i0; s0=1-s1;
    t1=yf-j0; t0=1-t1;
    u1=zf-k0; u0=1-u1;

    vx[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=s0*(t0*(u0*vx0[_IX(i0, j0, k0, (g->NX)↩
        +2, (g->NY)+2)]+
        u1*vx0[_IX(i0, j0, k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vx0[_IX(i0, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vx0[_IX(i0, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])))
        +s1*(t0*(u0*vx0[_IX(i1, j0, k0, (g->NX)+2, (g->NY)+2)]+u1*vx0[_IX(i1, j0,↩
            k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vx0[_IX(i1, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vx0[_IX(i1, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])));
    vy[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=s0*(t0*(u0*vy0[_IX(i0, j0, k0, (g->NX)↩
        +2, (g->NY)+2)]+
        u1*vy0[_IX(i0, j0, k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vy0[_IX(i0, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vy0[_IX(i0, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])))
        +s1*(t0*(u0*vy0[_IX(i1, j0, k0, (g->NX)+2, (g->NY)+2)]+u1*vy0[_IX(i1, j0,↩
            k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vy0[_IX(i1, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vy0[_IX(i1, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])));
    vz[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=s0*(t0*(u0*vz0[_IX(i0, j0, k0, (g->NX)↩
        +2, (g->NY)+2)]+
        u1*vz0[_IX(i0, j0, k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vz0[_IX(i0, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vz0[_IX(i0, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])))
        +s1*(t0*(u0*vz0[_IX(i1, j0, k0, (g->NX)+2, (g->NY)+2)]+u1*vz0[_IX(i1, j0,↩
            k1, (g->NX)+2, (g->NY)+2)])+
        (t1*(u0*vz0[_IX(i1, j1, k0, (g->NX)+2, (g->NY)+2)]+u1*vz0[_IX(i1, j1, k1,↩
            (g->NX)+2, (g->NY)+2)])));
    END_FOR_EACH_CELL

    _set_bnd(g, 1, vx);
    _set_bnd(g, 2, vy);
    _set_bnd(g, 3, vz);
}

void project(_GRID *g, float *vx, float *vy, float *vz, float *p, float *div) {
    int i, j, k;

    FOR_EACH_CELL2
    div[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=-0.5f*(
        (vx[_IX(i+1, j, k, (g->NX)+2, (g->NY)+2)]-vx[_IX(i-1, j, k, (g->NX)+2, (g↩
            ->NY)+2)])/(g->NX)+
        (vy[_IX(i, j+1, k, (g->NX)+2, (g->NY)+2)]-vy[_IX(i, j-1, k, (g->NX)+2, (g↩
            ->NY)+2)])/(g->NY)+
        (vz[_IX(i, j, k+1, (g->NX)+2, (g->NY)+2)]-vz[_IX(i, j, k-1, (g->NX)+2, (g↩
            ->NY)+2)])/(g->NZ));
    p[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]=0;
    END_FOR_EACH_CELL
```

```
    _set_bnd(g, 0, div);
    _set_bnd(g, 0, p);

    lin_solve(g, 0, p, div, 1, 6);

    FOR_EACH_CELL2
    vx[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]-=0.5f*(g->NX)*
        (p[_IX(i+1, j, k, (g->NX)+2, (g->NY)+2)]-p[_IX(i-1, j, k, (g->NX)+2, (g->↩
            NY)+2)]);
    vy[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]-=0.5f*(g->NY)*
        (p[_IX(i, j+1, k, (g->NX)+2, (g->NY)+2)]-p[_IX(i, j-1, k, (g->NX)+2, (g->↩
            NY)+2)]);
    vz[_IX(i, j, k, (g->NX)+2, (g->NY)+2)]-=0.5f*(g->NZ)*
        (p[_IX(i, j, k+1, (g->NX)+2, (g->NY)+2)]-p[_IX(i, j, k-1, (g->NX)+2, (g->↩
            NY)+2)]);
    END_FOR_EACH_CELL

    _set_bnd(g, 1, vx);
    _set_bnd(g, 2, vy);
    _set_bnd(g, 3, vy);
}

void lin_solve(_GRID *g, int b, float *x, float *x0, float a, float iter) {
#ifdef _JACOBI
    _jacobi(g, x, x0, a, iter, b, 6);
#endif

#ifdef _GAUSS_SEIDEL
    _gauss_seidel_relax(g, x, x0, a, iter, b, 4);
#endif

#ifdef _CG
    _cg(g, x, x0, a, iter, 1e-5, b, 4);
#endif
}

void dens_step(_GRID *g, float diff, float dt) {
    add_source(g, (g->d), (g->d0), dt);
    diffuse(g, 0, (g->d0), (g->d), diff, dt);
    advect(g, (g->d), (g->d0), (g->vx), (g->vy), (g->vz), dt);
}

void vel_step(_GRID *g, float visc, float dt) {
    add_vel_source(g, (g->vx), (g->vy), (g->vz), (g->vx0), (g->vy0), (g->vz0), dt↩
        );
    diffuse(g, 1, (g->vx0), (g->vx), visc, dt);
    diffuse(g, 2, (g->vy0), (g->vy), visc, dt);
    diffuse(g, 3, (g->vz0), (g->vz), visc, dt);
    project(g, (g->vx0), (g->vy0), (g->vz0), (g->vx), (g->vy));
    advect_vel(g, (g->vx), (g->vy), (g->vz), (g->vx0), (g->vy0), (g->vz0), dt);
    project(g, (g->vx), (g->vy), (g->vz), (g->vx0), (g->vy0));
}

void _vel_dens_step(_GRID *g, float visc, float diff, float dt) {
    vel_step(g, visc, dt);
    dens_step(g, diff, dt);
}

void _set_bnd(_GRID *g, int b, float *x) {
    int i, j, k;

    // simulation external boudaries
```

```
for(j=1;j<=(g->NY);j++)
    for(i=1;i<=(g->NX);i++) {
        x[_IX(i,j,0,(g->NX)+2,(g->NY)+2)]=b==3?-x[_IX(i,j,1,(g->NX)+2,(g->NY)↩
            +2)]:x[_IX(i,j,1,(g->NX)+2,(g->NY)+2)];
        x[_IX(i,j,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]=b==3?-x[_IX(i,j,(g->NZ),(g↩
            ->NX)+2,(g->NY)+2)]:
            x[_IX(i,j,(g->NZ),(g->NX)+2,(g->NY)+2)];
    }
for(k=1;k<=(g->NZ);k++)
    for(i=1;i<=(g->NX);i++) {
        x[_IX(i,0,k,(g->NX)+2,(g->NY)+2)]=b==2?-x[_IX(i,1,k,(g->NX)+2,(g->NY)↩
            +2)]:x[_IX(i,1,k,(g->NX)+2,(g->NY)+2)];
        x[_IX(i,(g->NY+1),k,(g->NX)+2,(g->NY)+2)]=b==2?-x[_IX(i,(g->NY),k,(g↩
            ->NX)+2,(g->NY)+2)]:
            x[_IX(i,(g->NY),k,(g->NX)+2,(g->NY)+2)];
    }
for(k=1;k<=(g->NZ);k++)
    for(j=1;j<=(g->NY);j++) {
        x[_IX(0,j,k,(g->NX)+2,(g->NY)+2)]=b==1?-x[_IX(1,j,k,(g->NX)+2,(g->NY)↩
            +2)]:x[_IX(1,j,k,(g->NX)+2,(g->NY)+2)];
        x[_IX((g->NX)+1,j,k,(g->NX)+2,(g->NY)+2)]=b==1?-x[_IX((g->NX),j,k,(g↩
            ->NX)+2,(g->NY)+2)]:
            x[_IX((g->NX),j,k,(g->NX)+2,(g->NY)+2)];
    }

// simulation corner boundaries
x[_IX(0,0,0,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX(1,0,0,(g->NX)+2,(g->NY)+2)]+
    x[_IX(0,1,0,(g->NX)+2,(g->NY)+2)]+x[_IX(0,0,1,(g->NX)+2,(g->NY)+2)]);
x[_IX(0,(g->NY)+1,0,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX(1,(g->NY)+1,0,(g->NX)↩
    +2,(g->NY)+2)]+
    x[_IX(0,(g->NY),0,(g->NX)+2,(g->NY)+2)]+x[_IX(0,(g->NY)+1,1,(g->NX)+2,(g↩
        ->NY)+2)]);
x[_IX(0,0,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX(1,0,(g->NZ)+1,(g->NX)↩
    +2,(g->NY)+2)]+
    x[_IX(0,1,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]+x[_IX(0,0,(g->NZ),(g->NX)+2,(g↩
        ->NY)+2)]);
x[_IX((g->NX)+1,0,0,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX((g->NX),0,0,(g->NX)↩
    +2,(g->NY)+2)]+
    x[_IX((g->NX)+1,1,0,(g->NX)+2,(g->NY)+2)]+x[_IX((g->NX)+1,0,1,(g->NX)+2,(↩
        g->NY)+2)]);
x[_IX(0,(g->NY)+1,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX(1,(g->NY)+1,(g↩
    ->NZ)+1,(g->NX)+2,(g->NY)+2)]+
    x[_IX(0,(g->NY),(g->NZ)+1,(g->NX)+2,(g->NY)+2)]+x[_IX(0,(g->NY)+1,(g->NZ)↩
        ,(g->NX)+2,(g->NY)+2)]);
x[_IX((g->NX)+1,(g->NY)+1,0,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX((g->NX), (g->↩
    NY)+1, 0,(g->NX)+2,(g->NY)+2)]+
    x[_IX((g->NX)+1, (g->NY), 0,(g->NX)+2,(g->NY)+2)] + x[_IX((g->NX)+1, (g->↩
        NY)+1, 1,(g->NX)+2,(g->NY)+2)]);
x[_IX((g->NX)+1,0,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]=0.33f*(x[_IX((g->NX), 0, (g↩
    ->NZ)+1,(g->NX)+2,(g->NY)+2)]+
    x[_IX((g->NX)+1, 1, (g->NZ)+1,(g->NX)+2,(g->NY)+2)] + x[_IX((g->NX)+1, 0,↩
         (g->NZ),(g->NX)+2,(g->NY)+2)]);
x[_IX((g->NX)+1,(g->NY)+1,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]=0.33f*(
    x[_IX((g->NX),(g->NY)+1,(g->NZ)+1,(g->NX)+2,(g->NY)+2)]+x[_IX((g->NX)+1,(↩
        g->NY),(g->NZ)+1,(g->NX)+2,(g->NY)+2)]+
    x[_IX((g->NX)+1,(g->NY)+1,(g->NZ),(g->NX)+2,(g->NY)+2)]);

// simulation internal and moving boundaries
FOR_EACH_CELL2
if(((g->bd[_IX(i,j,k,(g->NX)+2,(g->NY)+2)])=='B')||((g->bd[_IX(i,j,k,(g->NX)↩
    +2,(g->NY)+2)])=='M'))
    x[_IX(i,j,k,(g->NX)+2,(g->NY)+2)]=0;
```

```
    END_FOR_EACH_CELL
}
```

Appendix C

GPU-based version source code

```
/*
Grid structure and used Macros
*/
typedef struct {
    int _GRID_cells; // total number of grid cells
    int NX, NY, NZ; // fluid grid number of partitions per axis
    float sizeX, sizeY, sizeZ; // fluid cell size per axis
    char *bd; // fluid internal and external boundaries
} _GRID;

#define _JACOBI
//#define _GAUSS_SEIDEL
//#define _CG

#define _ITERATIONS_J 1
#define _ITERATIONS_GS 1
#define _ITERATIONS_CG 1
#define _TOL 1e-5

#define _NX 64 // number of fluid cells in X
#define _NY 32 // number of fluid cells in Y
#define _NZ 32 // number of fluid cells in Z
#define _sizeX 0.5 // size of fluid cell in X
#define _sizeY 0.5 // size of fluid cells in Y
#define _sizeZ 0.5 // size of fluid cells in Z
#define _BLOCK_DIM_X 1
#define _BLOCK_DIM_Y 1
#define _BLOCK_DIM_Z 1
#define _IX(x,y,z,NX,NY) ((x)+(y)*(NX)+(z)*(NX)*(NY)) // index to access the ↩
    corresponding position of grid

/*
Global variables
*/
float dt; //simulation time step
float diff; //fluid diffusion
float visc; //fluid viscosity;
float force;
float source;
float vec_scale; //scaling of hedgehogs
float pos; // position of the fallig sphere center of mass
int fluid; //draw the fluid or is velocities
int paused; //toggle on/off the animation
```

```
int bounds; // toggle on/off drawing internal and moving bounds
int falling_sphere; //toggle on/off drawing falling sphere

/*
Global pointes for the device memory
 */
float *vx, *vy, *vz, *d, *vx0, *vy0, *vz0, *d0; // Velocity and density pointers
char *bd; // internal bounds data pointer
int *more_source; // toggle on/off  pouring more water into the tank
float *water_lvl; // max water level allowed in the tank
float *aux; // Jacobi auxiliar array
float *rho, *rho0, *rho_old, *alpha, *beta; // CG variable pointers
float3 *r_p_q; // CG vector pointers

/*
Control User Interface kernels
*/
__global__ void _clear_moving_bounds_data(char *bd) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    for(int k=0;k<_NZ;k++)
        if(bd[_IX(i,j,k,_NX,_NY)]=='M')
            bd[_IX(i,j,k,_NX,_NY)]='_';
}

__global__ void _add_pipe_source(float *d, float *d0, float *water_lvl, int *↩
    more_source, float source) {
    int i=threadIdx.x+1;
    int j=blockIdx.x+24;

    if(*more_source)
        for(int k=14;k<18;k++) {
            // stop introducing liquid at a certain water level
            if(*water_lvl<11) {
                d0[_IX(i,j,k,_NX,_NY)]=source;
                __syncthreads();
            }
            else {
                // ensure that no more density exists in the introduction area
                d[_IX(i,j,k,_NX,_NY)]=0;
                __syncthreads();

                *more_source=0;
            }
        }
}

__global__ void _add_force_source(float *d, float *d0, float *vy, float *vx0, ↩
    float *vy0, float *water_lvl, float scm, float source, float force, float rn↩
    ) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    float pos;

    for(int k=0;k<_NZ;k++) {
        // horizontal velocity is only introduced inside the pipe to simulate a ↩
            water pump
        if((j>(*water_lvl+1))&&(*water_lvl<14)&&(d[_IX(i,j,k,_NX,_NY)]>0.1))
            vx0[_IX(i,j,k,_NX,_NY)]=(rn-(force*floorf(rn/force)))*0.11;
```

```
        // ensure gravities force is considered only where there is fluid
        if(d[_IX(i,j,k,_NX,_NY)]>0.1) {
            if(*water_lvl<14)
                vy0[_IX(i,j,k,_NX,_NY)]=-0.62;
            else
                vy0[_IX(i,j,k,_NX,_NY)]=-0.88;
        }

        if(j<(*water_lvl+1))
            vy[_IX(i,j,k,_NX,_NY)]=0.0;

        // calculate the position of the sphere in grid coordinates
        pos=scm*(_NY-1);

        // add horizontal velocity that opposes gravity attraction, to make a ↩
             splash when the sphere hits the surface
        if((pos>(*water_lvl-1.0))&&(pos<(*water_lvl))&&(*water_lvl>6)&&(i>26)&&(i↩
             <38)&&(k>12)&&(k<22))
            vy0[_IX(i,j,k,_NX,_NY)]=8.0;

        // ensure that the water is only add until a certain water level
        if((j<*water_lvl)&&(j>3)&&(16<i)&&(i<54)&&(7<k)&&(k<25))
            d0[_IX(i,j,k,_NX,_NY)]=source;
    }
    __syncthreads();

    // increase water level until tank is full.
    if((*water_lvl<=14)&&(d[_IX(40,5,16,_NX,_NY)]>0.1))
        *water_lvl+=0.02;
}

/*
Solver kernels and device functions
*/
__device__ void set_bnd(float *x, int i, int j, int k, int b, char *bd) {
    // simulation external boundaries
    if(i==0)
    x[_IX(i,j,k,_NX,_NY)]=b==1?-x[_IX(i+1,j,k,_NX,_NY)]:x[_IX(i+1,j,k,_NX,_NY)];
    if(i==(_NX-1))
    x[_IX(i,j,k,_NX,_NY)]=b==1?-x[_IX(i-1,j,k,_NX,_NY)]:x[_IX(i-1,j,k,_NX,_NY)];
    if(j==0)
    x[_IX(i,j,k,_NX,_NY)]=b==2?-x[_IX(i,j+1,k,_NX,_NY)]:x[_IX(i,j+1,k,_NX,_NY)];
    if(j==(_NY-1))
    x[_IX(i,j,k,_NX,_NY)]=b==2?-x[_IX(i,j-1,k,_NX,_NY)]:x[_IX(i,j-1,k,_NX,_NY)];
    if(k==0)
    x[_IX(i,j,k,_NX,_NY)]=b==2?-x[_IX(i,j,k+1,_NX,_NY)]:x[_IX(i,j,k+1,_NX,_NY)];
    if(k==(_NZ-1))
    x[_IX(i,j,k,_NX,_NY)]=b==2?-x[_IX(i,j,k-1,_NX,_NY)]:x[_IX(i,j,k-1,_NX,_NY)];

    // simulation corner boundaries
    if((i==0)&&(j==0)&&(k==0))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i+1,j,k,_NX,_NY)]+x[_IX(i,j+1,k,_NX,↩
             _NY)]+x[_IX(i,j,k+1,_NX,_NY)]);
    if((i==0)&&(j==(_NY-1))&&(k==0))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i+1,j,k,_NX,_NY)]+x[_IX(i,j-1,k,_NX,↩
             _NY)]+x[_IX(i,j,k+1,_NX,_NY)]);
    if((i==0)&&(j==0)&&(k==(_NZ-1)))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i+1,j,k,_NX,_NY)]+x[_IX(i,j+1,k,_NX,↩
             _NY)]+x[_IX(i,j,k-1,_NX,_NY)]);
    if((i==0)&&(j==(_NY-1))&&(k==(_NZ-1)))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i+1,j,k,_NX,_NY)]+x[_IX(i,j-1,k,_NX,↩
             _NY)]+x[_IX(i,j,k-1,_NX,_NY)]);
```

```
    if((i==(_NX-1))&&(j==0)&&(k==0))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i-1,j,k,_NX,_NY)]+x[_IX(i,j+1,k,_NX,↩
            _NY)]+x[_IX(i,j,k+1,_NX,_NY)]);
    if((i==(_NX-1))&&(j==(_NY-1))&&(k==0))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i-1,j,k,_NX,_NY)]+x[_IX(i,j-1,k,_NX,↩
            _NY)]+x[_IX(i,j,k+1,_NX,_NY)]);
    if((i==(_NX-1))&&(j==0)&&(k==(_NZ-1)))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i-1,j,k,_NX,_NY)]+x[_IX(i,j+1,k,_NX,↩
            _NY)]+x[_IX(i,j,k-1,_NX,_NY)]);

    if((i==(_NX-1))&&(j==(_NY-1))&&(k==(_NZ-1)))
        x[_IX(i,j,k,_NX,_NY)]=0.33f*(x[_IX(i-1,j,k,_NX,_NY)]+x[_IX(i,j-1,k,_NX,↩
            _NY)]+x[_IX(i,j,k-1,_NX,_NY)]);

    // simulation internal and moving boundaries
    if(((bd[_IX(i,j,k,_NX,_NY)])=='B')||((bd[_IX(i,j,k,_NX,_NY)])=='M'))
        x[_IX(i,j,k,_NX,_NY)]=0;
}

__global__ void _add_source(float *x, float *s, float dt) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    for(int k=0;k<_NZ;k++) x[_IX(i,j,k,_NX,_NY)]+=dt*s[_IX(i,j,k,_NX,_NY)];

}

__global__ void _add_vel_source(float *vx, float *vy, float *vz, float *vx0, ↩
    float *vy0, float *vz0, float dt) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    for(int k=0;k<_NZ;k++) {
        vx[_IX(i,j,k,_NX,_NY)]+=dt*vx0[_IX(i,j,k,_NX,_NY)];
        vy[_IX(i,j,k,_NX,_NY)]+=dt*vy0[_IX(i,j,k,_NX,_NY)];
        vz[_IX(i,j,k,_NX,_NY)]+=dt*vz0[_IX(i,j,k,_NX,_NY)];
    }
}

__global__ void _project1(float *vx, float *vy, float *vz, float *p, float *div, ↩
    char *bd) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    for(int k=0;k<_NZ;k++) {
        if((i!=0)&&(j!=0)&&(k!=0)&&(i!=(_NX-1))&&(j!=(_NY-1))&&(k!=(_NZ-1))) {
            div[_IX(i,j,k,_NX,_NY)]=-0.5f*((vx[_IX(i+1,j,k,_NX,_NY)]-vx[_IX(i-1,j↩
                ,k,_NX,_NY)])/_NX+
                (vy[_IX(i,j+1,k,_NX,_NY)]-vy[_IX(i,j-1,k,_NX,_NY)])/_NY+
                (vz[_IX(i,j,k+1,_NX,_NY)]-vz[_IX(i,j,k-1,_NX,_NY)])/_NZ);
            p[_IX(i,j,k,_NX,_NY)]=0;
        }
        __syncthreads();

        set_bnd(div, i, j, k, 0, bd);
        set_bnd(p, i, j, k, 0, bd);
    }
}

__global__ void _project2(float *vx, float *vy, float *vz, float *p, char *bd) {
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
```

```
    for(int k=0;k<_NZ;k++) {
        if((i!=0)&&(j!=0)&&(k!=0)&&(i!=(_NX-1))&&(j!=(_NY-1))&&(k!=(_NZ-1))) {
            vx[_IX(i,j,k,_NX,_NY)]-=0.5f*_NX*(p[_IX(i+1,j,k,_NX,_NY)]-p[_IX(i-1,j↩
                ,k,_NX,_NY)]);
            vy[_IX(i,j,k,_NX,_NY)]-=0.5f*_NY*(p[_IX(i,j+1,k,_NX,_NY)]-p[_IX(i,j↩
                -1,k,_NX,_NY)]);
            vz[_IX(i,j,k,_NX,_NY)]-=0.5f*_NZ*(p[_IX(i,j,k+1,_NX,_NY)]-p[_IX(i,j,k↩
                -1,_NX,_NY)]);
        }
        __syncthreads();

        set_bnd(vx, i, j, k, 1, bd);
        set_bnd(vy, i, j, k, 2, bd);
        set_bnd(vz, i, j, k, 3, bd);
    }
}

__global__ void _advect(float *d, float *d0, float *vx, float *vy, float *vz, ↩
    float dt, char *bd) {
    int i0, j0, k0, i1, j1, k1;
    float xf, yf, zf, s0, t0, u0, s1, t1, u1, dtx, dty, dtz;

    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    dtx=dt*(_NX-2);
    dty=dt*(_NY-2);
    dtz=dt*(_NZ-2);

    for(int k=0;k<_NZ;k++) {
        if((i!=0)&&(j!=0)&&(k!=0)&&(i!=(_NX-1))&&(j!=(_NY-1))&&(k!=(_NZ-1))) {
            xf=(float) i-(dtx*vx[_IX(i,j,k,_NX,_NY)]);
            yf=(float) j-(dty*vy[_IX(i,j,k,_NX,_NY)]);
            zf=(float) k-(dtz*vz[_IX(i,j,k,_NX,_NY)]);

            if(xf<0.5f) xf=0.5f;
            if(xf>(_NX+0.5f)) xf=_NX+0.5f;
            i0=(int) xf;i1=i0+1;

            if(yf<0.5f) yf=0.5f;
            if(yf>(_NY+0.5f)) yf=_NY+0.5f;
            j0=(int) yf;j1=j0+1;

            if(zf<0.5f) zf=0.5f;
            if(zf>(_NZ+0.5f)) zf=_NZ+0.5f;
            k0=(int) zf;k1=k0+1;

            s1=xf-i0;s0=1-s1;
            t1=yf-j0;t0=1-t1;
            u1=zf-k0;u0=1-u1;

            d[_IX(i,j,k,_NX,_NY)]=s0*(t0*(u0*d0[_IX(i0,j0,k0,_NX,_NY)]+u1*d0[_IX(↩
                i0,j0,k1,_NX,_NY)])+
                (t1*(u0*d0[_IX(i0,j1,k0,_NX,_NY)]+u1*d0[_IX(i0,j1,k1,_NX,_NY)])))↩
                    +
                s1*(t0*(u0*d0[_IX(i1,j0,k0,_NX,_NY)]+u1*d0[_IX(i1,j0,k1,_NX,_NY)↩
                    ])+
                (t1*(u0*d0[_IX(i1,j1,k0,_NX,_NY)]+u1*d0[_IX(i1,j1,k1,_NX,_NY)])))↩
                    ;
        }
        __syncthreads();
```

```
        set_bnd(d, i, j, k, 0, bd);
    }
}

__global__ void _advect_vel(float *vx, float *vy, float *vz, float *vx0, float *↩
    vy0, float *vz0, float dt, char *bd) {
    int i0, j0, k0, i1, j1, k1;
    float xf, yf, zf, s0, t0, u0, s1, t1, u1, dtx, dty, dtz;

    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;

    dtx=dt*(_NX-2);
    dty=dt*(_NY-2);
    dtz=dt*(_NZ-2);

    for(int k=0;k<_NZ;k++) {
        if((i!=0)&&(j!=0)&&(k!=0)&&(i!=(_NX-1))&&(j!=(_NY-1))&&(k!=(_NZ-1))) {
            xf=(float) i-(dtx*vx0[_IX(i,j,k,_NX,_NY)]);
            yf=(float) j-(dty*vy0[_IX(i,j,k,_NX,_NY)]);
            zf=(float) k-(dtz*vz0[_IX(i,j,k,_NX,_NY)]);

            if(xf<0.5f) xf=0.5f;
            if(xf>(_NX+0.5f)) xf=_NX+0.5f;
            i0=(int) xf;i1=i0+1;

            if(yf<0.5f) yf=0.5f;
            if(yf>(_NY+0.5f)) yf=_NY+0.5f;
            j0=(int) yf;j1=j0+1;

            if(zf<0.5f) zf=0.5f;
            if(zf>(_NZ+0.5f)) zf=_NZ+0.5f;
            k0=(int) zf;k1=k0+1;

            s1=xf-i0;s0=1-s1;
            t1=yf-j0;t0=1-t1;
            u1=zf-k0;u0=1-u1;

            vx[_IX(i,j,k,_NX,_NY)]=s0*(t0*(u0*vx0[_IX(i0,j0,k0,_NX,_NY)]+u1*vx0[↩
                _IX(i0,j0,k1,_NX,_NY)])+
                (t1*(u0*vx0[_IX(i0,j1,k0,_NX,_NY)]+u1*vx0[_IX(i0,j1,k1,_NX,_NY)])↩
                    ))+
                s1*(t0*(u0*vx0[_IX(i1,j0,k0,_NX,_NY)]+u1*vx0[_IX(i1,j0,k1,_NX,_NY↩
                    )])+
                (t1*(u0*vx0[_IX(i1,j1,k0,_NX,_NY)]+u1*vx0[_IX(i1,j1,k1,_NX,_NY)])↩
                    ));
            vy[_IX(i,j,k,_NX,_NY)]=s0*(t0*(u0*vy0[_IX(i0,j0,k0,_NX,_NY)]+u1*vy0[↩
                _IX(i0,j0,k1,_NX,_NY)])+
                (t1*(u0*vy0[_IX(i0,j1,k0,_NX,_NY)]+u1*vy0[_IX(i0,j1,k1,_NX,_NY)])↩
                    ))+
                s1*(t0*(u0*vy0[_IX(i1,j0,k0,_NX,_NY)]+u1*vy0[_IX(i1,j0,k1,_NX,_NY↩
                    )])+
                (t1*(u0*vy0[_IX(i1,j1,k0,_NX,_NY)]+u1*vy0[_IX(i1,j1,k1,_NX,_NY)])↩
                    ));
            vz[_IX(i,j,k,_NX,_NY)]=s0*(t0*(u0*vz0[_IX(i0,j0,k0,_NX,_NY)]+u1*vz0[↩
                _IX(i0,j0,k1,_NX,_NY)])+
                (t1*(u0*vz0[_IX(i0,j1,k0,_NX,_NY)]+u1*vz0[_IX(i0,j1,k1,_NX,_NY)])↩
                    ))+
                s1*(t0*(u0*vz0[_IX(i1,j0,k0,_NX,_NY)]+u1*vz0[_IX(i1,j0,k1,_NX,_NY↩
                    )])+
```

```
                (t1*(u0*vz0[_IX(i1,j1,k0,_NX,_NY)]+u1*vz0[_IX(i1,j1,k1,_NX,_NY)])↩
                    ));
        }
        __syncthreads();

        set_bnd(vx, i, j, k, 1, bd);
        set_bnd(vy, i, j, k, 2, bd);
        set_bnd(vz, i, j, k, 3, bd);
    }
}
```