Programa de Doctorat en Computació
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Efficient algorithms for the realistic simulation of fluids

A Doctoral Thesis
by

Jesús Ojeda Contreras

Tesi presentada per obtenir el títol de Doctor per la
Universitat Politècnica de Catalunya

Advisor: Antonio Susín Sánchez

Barcelona, March 2013

Abstract

Nowadays there is great demand for realistic simulations in the computer graphics field. Physically-based animations are commonly used, and one of the more complex problems in this field is fluid simulation, more so if real-time applications are the goal. Videogames, in particular, resort to different techniques that, in order to represent fluids, just simulate the consequence and not the cause, using procedural or parametric methods and often discriminating the physical solution.

This need motivates the present thesis, the interactive simulation of free-surface flows, usually liquids, which are the feature of interest in most common applications. Due to the complexity of fluid simulation, in order to achieve real-time framerates, we have resorted to use the high parallelism provided by actual consumer-level GPUs. The simulation algorithm, the Lattice Boltzmann Method, has been chosen accordingly due to its efficiency and the direct mapping to the hardware architecture because of its local operations.

We have created two free-surface simulations in the GPU: one fully in 3D and another restricted only to the upper surface of a big bulk of fluid, limiting the simulation domain to 2D. We have extended the latter to track dry regions and is also coupled with obstacles in a geometry-independent fashion. As it is restricted to 2D, the simulation loses some features due to the impossibility of simulating vertical separation of the fluid. To account for this we have coupled the surface simulation to a generic particle system with breaking wave conditions; the simulations are totally independent and only the coupling binds the LBM with the chosen particle system.

Furthermore, the visualization of both systems is also done in a realistic way within the interactive framerates; raycasting techniques are used to provide the expected light-related effects as refractions, reflections and caustics. Other techniques that improve the overall detail are also applied as low-level detail ripples and surface foam.

Resum

Avui dia hi ha una gran demanda per les simulacions realistes en el camp de la computació gràfica. Animacions amb base física s'utilitzen comunament, i un dels problemes més complexos en aquest camp és la simulació de fluids, tant més si aplicacions en temps real són l'objectiu. Els videojocs, en particular, recórren a diferents tècniques que, amb la finalitat de representar els fluids, simplement simulen la conseqüència i no la causa, fent servir mètodes procedurals o paramètrics i, sovint, discriminen la solució física.

Aquesta necessitat és la que motiva aquesta tesi, la simulació interactiva dels fluids amb superfície lliure, en general líquids, que són la característica d'interès en la majoria de les aplicacions comunes. A causa de la complexitat de la simulació de fluids, amb la finalitat d'aconseguir una taxa de fotogrames en temps real, s'ha recorregut a utilitzar l'alt paral·lelisme proporcionat per les GPUs de consum actuals. L'algorisme de simulació, el Mètode Lattice Boltzmann (LBM), ha estat triat, per tant, per la seva eficiència i l'assimilació gairebé directa a l'arquitectura de maquinari gràcies a la natura local de les seves operacions.

Hem creat dos simulacions de superfície lliure a la GPU: una completament en 3D i una altra restringida només a la superfície superior d'un volum gran de fluid, la qual cosa limita el domini de simulació a 2D. Hem estès aquest últim model per tal de poder rastrejar regions seques, on el fluid no ha arribat, i també l'hem acoblat amb els objectes dinàmics d'una manera independent de la geometria. Com està restringit a 2D, la simulació perd algunes característiques a causa de la impossibilitat de simular la separació vertical del fluid. Per solventar aquesta limitació, s'ha acoblat la simulació de superfície amb un sistema de partícules genèric amb condicions de detecció del trencament de les onades, així les simulacions són totalment independents i només l'acoblament uneix el LBM amb el sistema de partícules escollit.

A més, la visualització d'ambdós sistemes també es fa d'una manera realista dins de les taxes de refresc interactives; s'empren tècniques de raycasting per tal de proporcionar els esperats efectes relacionats amb la llum com refraccions, reflexions i càustiques. També s'han aplicat altres tècniques que milloren el detall general com ones de detall de baix nivell i l'escuma de la superfície.

Acknowledgements

First, I would like to thank my advisor, Toni Susín, for introducing me into the fluid simulation world, allowing me to play and experiment with the methods and for the helpful discussions in critical moments.

I want to thank all the members of the Moving Group, present and past, for their comments and the shared seminar sessions, but specially Pere Brunet, who agreed to supervise my master thesis, introducing me to Computer Graphics.

My lab mates have also been very helpful, specially in making me procrastinate, sometimes much needed, and amuse me in our lunch times. A salute to you, kind ladies and gentlemen.

I would like to specially thank M.Àngels, for inspiring me into become a fake doctor, for her valuable support, for listening to my random ramblings and trying to help whenever I was stuck with some problem. Now it is your turn.

Finally, I want to thank my family for their support throughout the time this thesis has lasted. They had to endure the roller coaster of achievements and disappointments as the fluid simulations evolved.

This work has been funded by a FI grant from the Generalitat de Catalunya.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

God help us; we're in the hands of engineers.

Dr. Ian Malcolm
Jurassic Park

As visualization of computer generated scenes is every day nearer to full photo-realism, physically-based effects and animation related to natural phenomena still remain a complex problem, more so in the real-time applications. One particular hard field within this physics-related category of problems is the simulation of fluids.

Fluids are a very common feature of nature that takes part in every day life and it is quite desirable to include them in our virtual worlds and applications, being them off-line or real-time. As such, numerical simulation of fluids has become an important tool, from industrial processes to entertainment-oriented products as animation movies or videogames. While applications with engineering purposes focus on the accuracy of the physical simulations, a more realistic and appealing appearance is desired for computer generated animations.

In any case, we can distinguish between the simulation of single-phase gases or liquids; the former expand to the full size of their container, thus using the full simulation domain in a computational solution, the latter, however, may just exist in a limited volume inside the simulation domain, so it is required that the surface of the liquid is somehow tracked in order to conserve mass. Of course, there are situations in which multiple different fluids take part and interact between each other, that is known as multiphase simulations.

In the real-time domain, i.e., videogames basically, simplifications or even tricks are used for the simulation of liquids; they can deceive the user into believing that complex fluid systems are used. These fluid representations aren't usually based in physically-correct simulations due to the complexity involved and the small timeframe physic computations are allowed to take from the total time in a game loop.

We can find some examples of these kind of simplifications in quite recent games like *Left 4 Dead 2*(2009) and *Portal 2*(2011), where the fluid is simplified to a mesh and the textures associated (debris, normal map, etc.) are advected using the technique from [MB96] from a Flow map, a vector field computed off-line (see Figure 1.1) [Vla10]. *The Elder Scrolls V: Skyrim*(2011) does a similar thing and adds some sprites to simulate mist generated from waterfalls like in Figure 1.2. The *Uncharted*(2007-2011) series for the *Playstation 3* system (depicted in Figure 1.3) use a handful of techniques; from advecting normal and displacement maps like in the previous cases, to a variation of the simulation from [YHK07] and even using pre-baked fluid simulation that animates a skeletal mesh with one joint per vertex and additional particle systems for further details [GOH12, Coo12].

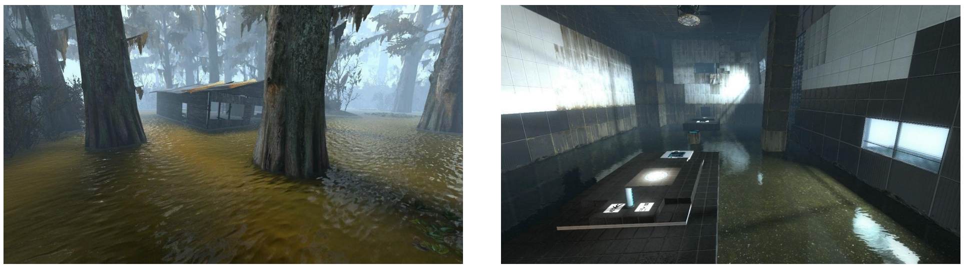


Figure 1.1: *Left 4 Dead 2* (left) and *Portal 2* (right), developed and published by Valve Corporation.

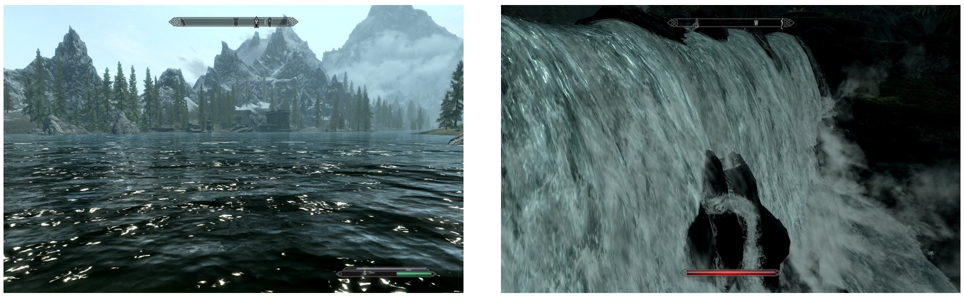


Figure 1.2: *The Elder Scrolls V: Skyrim*, developed by Bethesda Game Studios and published by Bethesda Softworks.

On the other hand, the most advanced games can even use full physical simulations like the case of *Borderlands 2*(2012) that can make use of the Nvidia PhysX library in the PC version, if the hardware supports it. In this case, however, only small fractions of fluids can be simulated due to the requirements the particle system used to accomplish the effect has (see Figure 1.4).

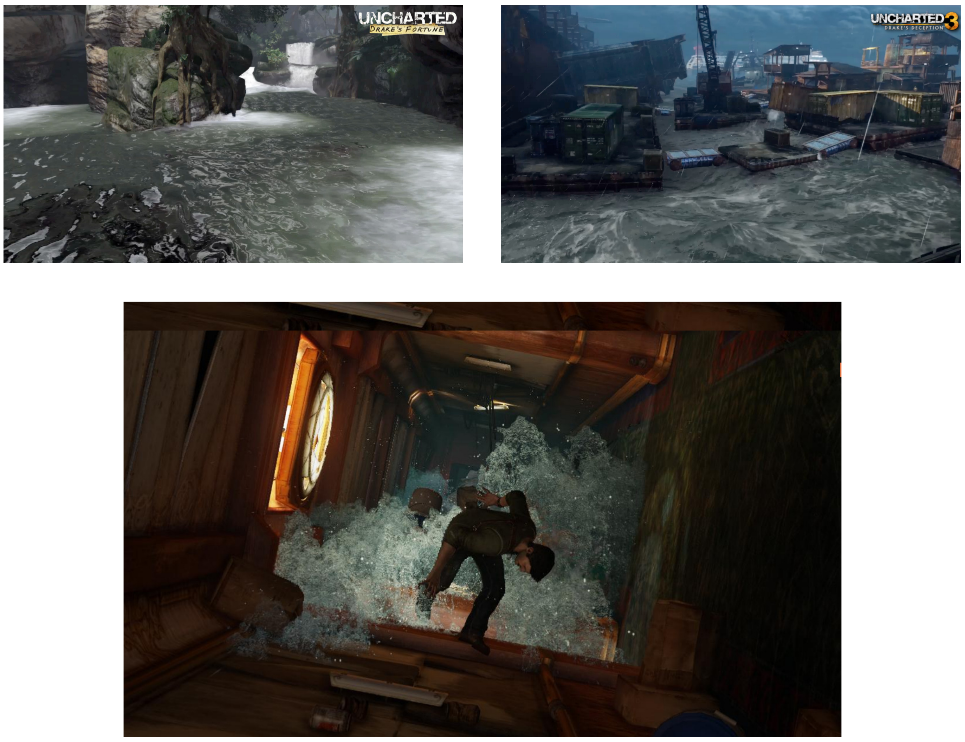


Figure 1.3: The *Uncharted* series, developed by Naughty Dog and published by Sony Computer Entertainment.

The previous argument, the need for fluid simulations, liquids in their majority, from a physically correct standpoint in real-time environments, motivates the present work; this thesis focuses in the simulation of two-phase problems, in which one phase is a liquid and the other is a gas, generally handled with simplified treatments or just dismissed completely.

Generally, the simulation of fluids is based on the Navier-Stokes equations, a set of partial differential equations, which are difficult to solve due to their non-linearity. This makes them hard to solve in real-time environments, where the simulation has to be stepped as well as visualized in just milliseconds. For this reason, we decided early on to use the raw power and heavy parallelization GPUs offer nowadays to make fluid simulations viable on interactive applications executed in regular PCs, which restrict even more the final performance.

Additionally, we have used an alternative formulation that approximates the Navier-Stokes equations, the Lattice Boltzmann Method (LBM), which is based on inter-molecular interactions. This approach is relatively novel and is becoming increasingly popular due to its simple and efficient algorithm.

Figure 1.4: *Borderlands 2*, developed by Gearbox Software and published by 2K Games.

1.1 Objectives and Goals

The final goal of this thesis is to achieve real-time fluid simulations and realistic visualization on commodity hardware, more specifically from free-surface flows. In order to leverage the maximum performance from GPUs, the best suited algorithm for this goal is the LBM, the local operations it is based on make it easily parallelizable and map perfectly with the GPU architecture.

The ideal situation is to achieve full 3D free-surface fluid simulation, although it still imposes some constraints in interactive applications depending on the size of the domain simulated, and bearing in mind that no obstacles are introduced.

As the previous solution is quite heavy and not as efficient as we would like, a different approach can be applied. Instead of simulate the whole bulk of fluid, we can restrict ourselves to a surface simulation, represented as a heightfield. This limits the spectrum of effects the fluid can provide, e.g., waves can't break. The solution to this is achieved by coupling the surface simulation with another, different system; in this case a particle system. Breaking wave conditions are used to detect when and where to generate particles that evolve independently and are finally reintegrated once they fall to the surface fluid representation again. As the fluid is only represented by a heightfield and a particle system, we have also to deal with dry regions, zones inside the domain without fluid, akin to the free-surface of the 3D fluid simulation. The surface algorithm is consequently modified to account for this possibility. Additionally, two-way coupling of rigid bodies is introduced. Objects can now react to the fluid, they drift, but can also interact with the fluid as they fall into or move around like a boat may do.

In every case, correspondent techniques for realistic visualization are developed. A raycasting algorithm is used for the 3D free-surface simulation, allowing reflections and refractions. In the hybrid particle-surface simulation, not only reflections and refractions are there, but also caustics as well as further methods to improve the final look and feel of the fluid.

Contributions

To fulfill the prime goal of real-time fluid simulation and visualization, the main contributions of this thesis can be summarized as:

1. A CUDA implementation of the LBM in 3D with the coupling of a free-surface algorithm from [Thü07] with deterministic results, explained in Sections 3.6 and 4.2.1.

2. The implementation of the Shallow waters LBM in CUDA, shown in Sections 3.7 and 4.2.2.

3. A new method to track dry regions in the shallow waters simulation, introduced in Section 3.7.3.

4. A geometry independent coupling of external rigid bodies to the shallow waters algorithm, exposed in Section 3.7.4.

5. Fluid feature-enhanced by coupling a generic particle system to the shallow waters implementation. This method allows the use of any particle system as long as it is also simulated in the GPU and enables additional effects like the breaking of waves. This will be sorted out in Sections 3.8 and 4.3.

6. An extension to screen-space for the caustics mapping technique from [SKP07], shown in Section 5.2.2.

7. A screen-space technique to simulate refractions and reflections, based in raycasting through depth-maps, presented in Section 5.2.3.

8. Texture-based techniques that improve the fluid surface detail. FFT Ocean simulation, a modified version of Perlin noise and a foam advection scheme are used here. These techniques are explained in Sections 5.2.1 and 5.2.5.

Almost all these contributions have been published or are nearly ready for submission. A prior result which lead to the first contribution was [OS09]. Contributions 2 to 5 are published in [OS13b, OS13a] and contributions 6 to 8 are included in [OS13c].

1.2 Thesis overview

From this point, we will firstly introduce the state of the art on fluid simulation in Chapter 2.

Then, Chapter 3 will go deeper in the details of the LBM algorithm in the various versions used throughout this work: from 3D free-surface fluid simulation to an hybrid surface simulation coupled with obstacles and particle systems.

The implementation of these algorithms in GPU will be discussed in Chapter 4, making extensive use of CUDA.

As we want to simulate as well as visualize realistically the fluids, we used a variety of algorithms to account for the different effects that can arise from fluids; they are described in Chapter 5.

Finally, we will conclude this document with some results obtained, some discussion about them as well as some ideas that could improve the present techniques in the future in Chapter 6.

Chapter 2

Simulation of Fluids

Curioser and curioser.

Alice in Wonderland
Lewis Carroll

In this chapter we will give a brief overview of related work on fluid simulations for computer animations; further references are provided in the corresponding sections of each chapter. Here, we will review the major alternatives for physically-based fluid simulation, from eulerian methods that simulate the fluid in fixed points in space, i.e., inside a grid, like direct discretizations of the Navier-Stokes equations, to lagrangian methods that evaluate the fluid properties at points in space that are advected with the own fluid, like particle systems, finally introducing and comparing the chosen algorithm, the Lattice Boltzmann Method.

2.1 Navier-Stokes equations

The common ground on computational fluid dynamics are the Navier-Stokes equations (NS), a reformulation of Newton's Second Law that describe the motion of fluid substances.

For incompressible and Newtonian fluids, those which have a constant density ρ and dynamic viscosity μ, the NS equations are

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

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

where the unknowns $\mathbf{u}$ and p are velocity and the pressure, respectively, ν is the kinematic viscosity ($\nu = \mu/\rho$) and $\mathbf{g}$ represents external body forces as gravity.

Equation 2.1 is known as the *momentum equation* and comprises the terms for advection, pressure, diffusion and external forces, respectively.

Equation 2.2 receives the name of *continuity equation* and ensures the conservation of mass, that is, the incompressibility of the fluid.

With appropriate initial and boundary conditions, these equations can be discretized, using finite differences and some grid representation like the staggered grid depicted in Figure 2.1, and solved using numerical methods such as Gauss-Seidel. An implementation of this kind of solution can be found in, e.g., [GDN98, Suá06].

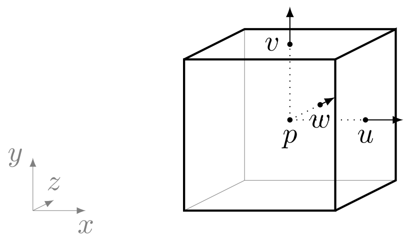


Figure 2.1: Staggered grid discretization. Pressure p is evaluated at the center of the cells, velocity $\mathbf{u} = (u, v, w)$ at the center of the faces.

From the visual standpoint, the most interesting feature of a fluid simulation is its surface and among the first to perform physically-based animations of free surfaces we find Foster and Metaxas [FM96, FM97a], which based their implementation on the algorithm from [HW65]. They also simulated gases with a similar discretization of the NS equations [FM97b]. However, at the time, the simulations were quite limited in scale because of the algorithms, as well as the computational resources, not being fully adequate. A breakthrough came when Stam [Sta99] introduced a semi-Lagrangian method; it achieved significant performance improvements, ensuring the solver was unconditionally stable at larger time steps but with a higher dissipation. This method has also been adapted to GPUs in, e.g., [Har04].

Some years later, the topic of free surface animations was brought back to life in [FF01]. They used Stam's technique and introduced the level-sets from [OS88], already used in [FAMO99]. This work, in addition to Stam's, represent the base of the principal type of level-set based free surface simulations. Several works have derived from them as, e.g., enhancing the surface tracking of the level-set with additional particles [EFFM02], improve the simulations with octrees for small scale details [LGF04] (see Figure 2.2) or using tall cells under the surface of the fluid in order to reduce the number of cells simulated in 3D [IGLF06, CM11].

Other methods to track the free surface have also been used. For example, Sussman [Sus03] used a Volume of Fluid method [HN81] combined with the level-set to achieve the smooth representation of the latter, but with improved mass conservation provided by the former one. Zhu and Bridson [ZB05], on the other hand, animated sand as a fluid, using the Fluid-Implicit-Particle method (FLIP) [BR86], which advects the fluid quantities

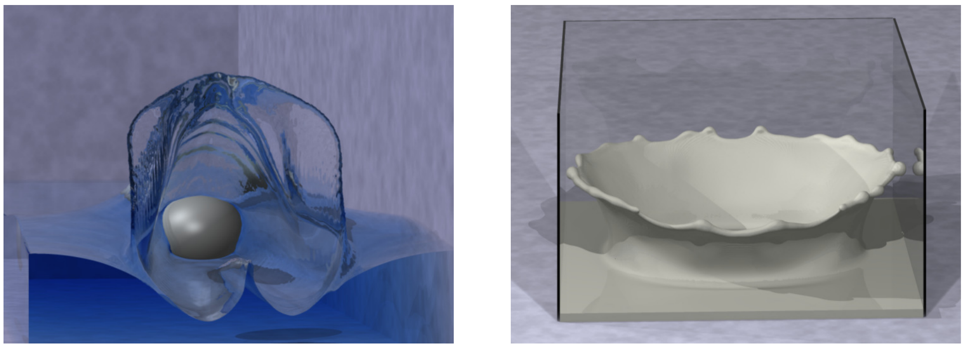


Figure 2.2: Octree-based level-set fluid simulations from [LGF04].

on particles, in contrast to do it on a grid as the other methods, although an auxiliary grid is still needed for the velocity field.

Alternatively, the Navier-Stokes equations have also been simplified to simulate fluids as heightfields, i.e., [KM90] considered the fluid volume divided in columns and assumed the fluid varied slowly. Later, [OH95] improved Kass et al. work, modelling the flow between fluid columns with virtual pipes, coupling objects as they impacted the surface of the fluid and adding splashes. In [CLHM97], the authors used a finite difference approach to solve the NS equations in 2D, with penalization instead of the continuity equation.

The Shallow Waters framework (SW), a simplification of the NS equations assuming vertical pressure gradients are nearly hydrostatic, introduced in [LvdP02] to computer graphics has seen several applied and derived works. As an example, [TMFSG07] simulated breaking waves and [ATBG08] created fluid characters, with inspiration from [Sta03] where flow was simulated on surfaces of arbitrary topology. Quite related to our work, [CM10], using an improved advection model from [SFK$^+$08] on GPUs, couples the SW with a particle system to cope with the limitations of a heightfield-based simulation, e.g., waves that should break or discontinuities on the fluid level, like in a waterfall.

2.2 Smoothed Particle Hydrodynamics

In contrast to the previous methods, where a grid is needed to evaluate the fluid, we find the particle-based systems that can simulate fluids; the fluid properties are interpolated among particles and mass is inherently conserved.

The more popular approach in this kind of methods, known as Smoothed Particle Hydrodynamics (SPH), arose from the astrophysics field [GM77, Luc77]. Its use has become quite common in the field of graphics as the particle representation renders it as a high-detail method, although a great count of particles is needed for nice results and the time step needed has to remain low enough to ensure stability.

As the fluid is not continuous, to calculate the motion of each particle it is used an interpolation technique based on kernel functions which are radius limited. At location $\mathbf{r}$, the scalar quantity A (ρ, for example) is interpolated by a weighted sum of contributions from all particles:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, j), \tag{2.3}$$

$$\nabla A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, j), \tag{2.4}$$

$$\nabla^2 A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, j), \tag{2.5}$$

where j iterates over all particles, m_j is the mass of particle j, $\mathbf{r}_j$ its position, ρ_j the density and A_j the field quantity at $\mathbf{r}_j$. W is the smoothing kernel with radius h and is the only one affected by the derivatives as shown in Equations 2.4 and 2.5. The kernels describe the behaviour of the associated fluid characteristics to the particle at hand in its neighbourhood (see Figure 2.3) and are defined so they solve the Navier-Stokes equations[1].

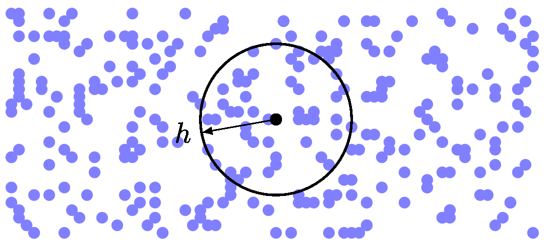


Figure 2.3: Only particles inside the kernel radius h will affect the computations for the central black particle. The kernels are designed to fall off with regards to the distance.

One problem, however, is that the SPH method simulates compressible flows, although the more common implementation uses an Equation of State to compute the pressure values in order to achieve near-incompressibility as introduced in [DG96]. More efforts have been developed to achieve full incompressibility as in, e.g., [SL03, HA07, SP09]; some results from the latter are shown in Figure 2.4.

It was introduced to the graphics community with fire and gas simulations in [SF95] and Müller et al. showed that interactive simulations were also possible [MCG03], although the particle number influenced heavily on the visual result.

[1]The smoothing kernels can be freely defined, independently for each scalar quantity. As long as the kernel is even ($W(\mathbf{r}, h) = W(-\mathbf{r}, h)$) and normalized ($\int W(\mathbf{r}d\mathbf{r} = 1$), the interpolation is of second order accuracy.

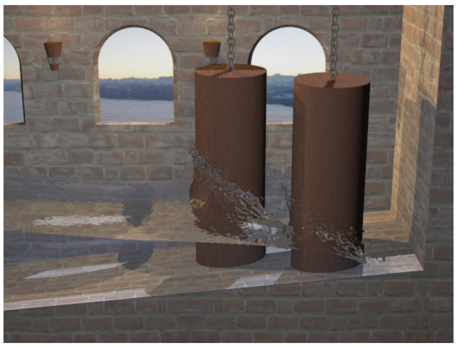


Figure 2.4: Predictive-corrective SPH simulations from [SP09].

The higher level of detail for smaller fluid features as splashes has also made this method the preferred choice in hybrid algorithms, where the bulk of a fluid is simulated with other, maybe grid-based, methods and the finer detail is the responsibility of the SPH particles; for example, [LTKF08] coupled the SPH with the Particle level-set.

Over time, as the GPUs have been improving and now provide general computing with higher parallelism, the SPH method has also received GPU implementations like those in [HKK07, GSSP10].

Additionally, similar to the NS case, the SPH has also been implemented in order to solve the Shallow Waters equations, being [LH10, SBC+11] a couple of examples of this subject.

An alternative method quite similar to SPH but fully incompressible is the Moving-Particle Semi-Implicit [KO96, PTB+03]. Here, the particles interact with each other as in the SPH but the pressure is computed differently; a Poisson equation is solved from the particle density values in order to find the pressure and correct, therefore, the particle velocities.

2.3 Lattice Boltzmann Method

The last contestant in this comparative has reclaimed some interest in previous years, the Lattice Boltzmann Method (LBM), which is now used for a variety of applications. An example of this could be the PowerFLOW commercial package [Cor13], used in the automobile or aircraft industries.

This method is based in statistical physics and can solve the original Navier-Stokes equations. It uses a grid for the simulation of the fluid and discretizes the directions the fluid molecules can travel along. The LBM can be summarized in one equation:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) - f_i(\mathbf{x}, t) = \Omega_i(f(\mathbf{x}, t)), \tag{2.6}$$

where i is the direction of the molecular travel, f_i is the statistical distribution function for the i direction and Ω is the collision operator, which defines how the molecules interact with each other for each lattice point. We will expand on the mathematical details and general background in Chapter 3.

From Equation 2.6, it is easy to see that the computations needed are totally local and, depending on the complexity of the chosen collision operator, they could potentially remain as simple arithmetic operations. These arguments make the LBM an ideal method for parallel implementations, being the GPU ones that have arisen as, e.g., [LWK03, Tö10], more interesting for our goals.

The LBM has also been coupled with multiple free surface algorithms, being [GRZZ91, GS03, Thü07] a little example, shown in Figure 2.5 a sample of the latter, and even has been adapted to simulate Shallow Waters problems like in [Sal99, Zho02].

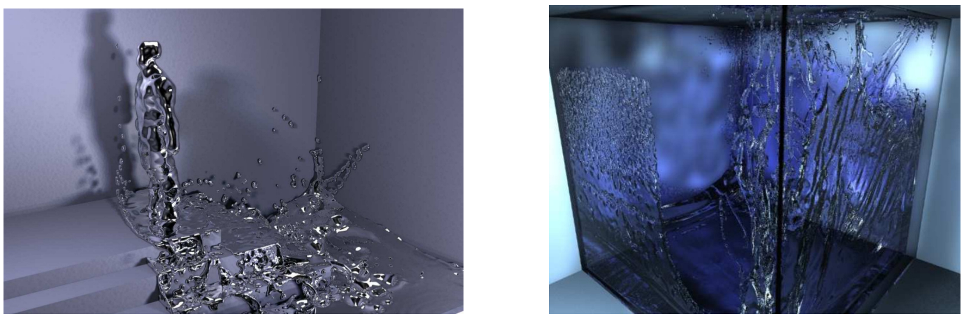


Figure 2.5: Fluid control and adaptive grids for LBM free-surface simulations from [Thü07].

In contrast to the other fluid simulation methodologies, the LBM is quite efficient for its simple operations and that makes it ideal for parallelization, the free surface algorithm which we have applied ensures mass conservation, although there are restrictions for the time step and, overall, the LBM is a quite consuming memory algorithm.

2.4 Present fluid simulation limits

As we have seen, fluid simulation is one of the classic problems, greatly studied in the computational fluid dynamics field. For engineering purposes the algorithm to use depends on the physical flow properties that have to be computed. Furthermore, to acquire accurate results, it is prioritized a correct physical simulation before the time needed for those computations. Traditionally, the most used methods are the Navier-Stokes equations and SPH model for aerodynamic applications.

In contrast, in the computer graphics field, as there is no actual flow property to compute accurately, the priority is rooted on a consistent and completely realistic visual result.

In this field, full 3D fluid simulation were first seen in [FM96] using the Navier-Stokes discretization which provided a rigid simulation inside a grid of fixed domain size. In this direction, nowadays, a state of the art simulation using a NS solver can be integrated with arbitrary sized time steps. Additionally, if the commonly used level set approach is applied for tracking the free surface, we can obtain a smooth surface representation, although by itself is not mass conserving and other additional techniques, as auxiliary particles, are needed to guarantee it. Unfortunately, the same algorithm that allows arbitrary time steps [Sta99] also causes additional dissipation which has to be dealt with if liquids are to be simulated. Other inconvenience for the NS solver include the need of a global pressure correction step to achieve a divergence free velocity field.

In contrast, the SPH model introduced by [SF95, DG96] allows more flexibility due to the nature of the particles, they conserve mass naturally and allow for a simple tracking of the free surface of the fluid. Furthermore, the use of particles as the conveyors of the fluid provides higher low-scale details as splashes and spray, which has provided SPH great publicity in film production using packages like RealFlow [Tec13]. On the bad side, SPH doesn't enforce incompressibility by itself, so additional efforts have to be made at this respect. Additionally, greater simulation domains require a very large number of particles which, in turn, make the method more computationally expensive.

The LBM as a third counterpart for the simulation of fluids, has an efficient algorithm which uses only arithmetic operations and is totally local, providing it a head-start for a parallel implementation. Moreover, it can also conserve mass when tracking free surfaces if an algorithm like the Volume of Fluid implemented in this thesis is applied. As noted before, however, it has an increased memory requirement and forces the use of small time steps to ensure stability. It is worth mentioning that, for the same time step, a single LBM step is usually significantly faster than the update step of a NS solver.

Another thing that has an important role in the fluid simulations is the coupling with external dynamic objects. In the case of SPH, the particles can interact directly with other objects without additional restrictions, but in the case of NS and LBM simulations, they both need ad hoc processes that couple external simulations of dynamic objects to the fluid, which in fact poses an additional drawback for their implementation.

The previous discussion is summarized in Figure 2.6.

Although all three techniques have received interactive implementations like, e.g., [Har04, CM10] with an implementation of Stam's work and a shallow water adaptation with particles similar to the present work, [MCG03, HKK07] with an interactive implementation of SPH and a GPU one and [Thü07] that provides interactive framerates with its basic free surface LBM, none of them have been used in videogames, except maybe SPH (as seen

NS Level set	SPH	LBM VOF
✓ Large time steps	✓ No grid required	✓ Simple algorithm
✓ Smooth surface	✓ Mass conserving	✓ Mass conserving
✓ Largely known in CG	✓ Dynamic objects interaction	✓ Totally local
✗ Auxiliary Particles required	✗ Compressibility	✗ Time step restriction
✗ Pressure Correction Step	✗ High number of particles	✗ High memory usage
✗ Ad hoc dynamic objects	✗ Smoothing of kernels	✗ Ad hoc dynamic objects

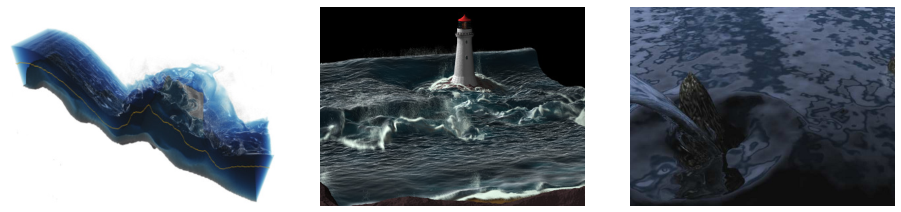


Figure 2.6: Overview of the main advantages and drawbacks of the three main fluid methods: NS, SPH and LBM. Images from [CM11, LTKF08, Thü07], respectively.

in Figure 1.4) for very small domains and only using the latest hardware available if only a single GPU is used[2].

For this reason, the parallel nature of the LBM attracted us at the beginning of this thesis. We implemented a full 3D simulation with this method on a mid-level GPU at the time (see Sections 3.6.2 and 4.2.1) but, although it was feasible with an appropriate visualization for small domains, it didn't meet our requirements for it to be fully applicable to real-time software with commodity hardware. This setback made us realize that full simulations were still unmanageable at the scale we wanted, so we opted for a 2.5D solution: a shallow waters approach to the problem, where fluid is represented by a heightfield; as the method was faster, it allowed to include dynamic objects and particles to supply some of its deficiencies (see Sections 3.7, 3.8 and 4.3).

The previous discussion also showed us a different aspect from these techniques: it is usually not discussed how the fluids are visualized because, as it may be assumed from their computational requirements that they are directed toward off-line rendering, then external visualization packages as, e.g., PRMan[Pix13] or Povray[oVRPL13], are used. These packages are intended for photorealistic results and may require from seconds to hours to render just a frame, depending on the internal lighting algorithms used.

[2]The SPH is executed with the PhysX library only implemented in GPU for Nvidia cards. If the user does not use one of this brand, the simulation runs in CPU with the correspondent performance hit. Alternatively, a multi-GPU configuration may be used, where a less powerful Nvidia GPU card would go to execute the physics simulation alone.

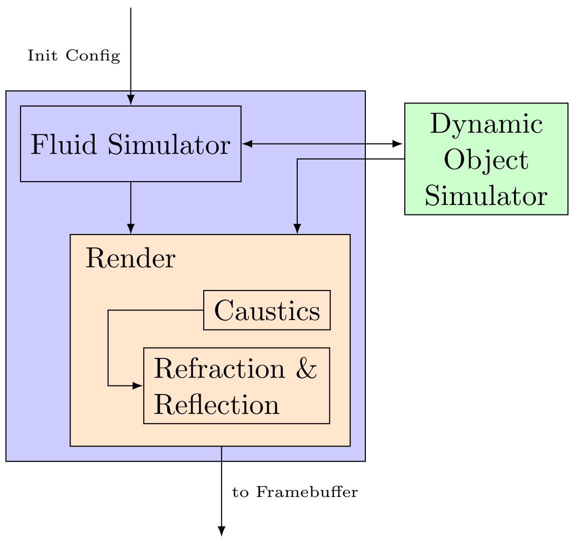


Figure 2.7: The final pipeline applicable to a receiving software with both simulation and visualization of fluids in real-time. From an initial domain configuration and a group of objects initialized in a dynamic object simulator, the fluid simulation can start providing the adequate data to the visualization module, who also needs some data from the dynamic objects (for correctly proceed with caustics, refractions and reflections), and the result is finally provided to the framebuffer.

From this scenario, it is clear that, for real-time applications, an appropriate visualization must be created that runs as fast as can be but with results as physically faithful to photorealism as possible. Then, for a final software, it is desirable to view simulation and visualization as a whole, like the pipeline shown in Figure 2.7. This lead us to use raymarching techniques (see Sections 5.2.2 and 5.2.3) through the scene to reproduce the light effects as caustics, reflections and refractions, but restricting the amount of information available to screen-space, as global illumination techniques are prohibitive, given that everything, simulation and visualization, are executed at the same time and must do so in just a fraction of a second for each frame.

In general, although different implementations can make each method to vary significantly, we can conclude that there is no silver bullet in the computational fluid dynamics field and the best solution lies in the combination of multiple simulation methods to obtain the best of them. In any case, the Lattice Boltzmann Method is up to the challenge of simulating fluids with realistic results and it is quickly making its entrance in the graphics field.

Chapter 3

Lattice Boltzmann Method

Back off, man. I'm a scientist.

Peter Venkman
The Ghostbusters

This chapter goes in depth with our chosen fluid model, the Lattice Boltzmann Method (LBM). Firstly, we give a brief revision of its history and explain its basic form. Then, a mathematical derivation from the Boltzmann equation to the discretized Lattice Boltzmann equation is given. Next, the needed parametrization and an improved stabilization through a turbulence model are presented. We proceed to explain how a free-surface model can be adapted to the LBM. From a complete 3D simulation, we step back to a surface only representation, using an adapted LBM to solve the Shallow Water equations (LBMSW) and expand on how to include dry sections and dynamic rigid bodies. Finally, we present an hybrid model using this surface simulation coupled with a particle system which allows to achieve results that the LBMSW by itself can not reproduce.

3.1 Evolution: how it became to existence

Devised by Ludwig Boltzmann in 1872, the Boltzmann equation is part of the classical statistical physics and describes the behaviour of a gas in a microscopic scale through the interaction of particles. Although mainly developed for ideal gases, in the limit of small mean free path between molecular collisions, a gas may be considered as a continuum fluid.

The LBM evolved, however, from methods that simulated gases as the interaction from particles, solely using boolean or integer operations inside a lattice. These methods are closely related to the concept of Cellular Automata [Wol86]; great complex structures are created from a simple set of local rules. [HdPP76] were the first to attempt to perform fluid simulations with this approach. It was not until [FHP86, FDH$^+$87] with their Lattice

Gas Automata (LGA), though, that it was discovered that the isotropy of the lattice vectors is crucial to perform a correct approximation of the NS equations.

Replacing the operations of the LGA with the calculation of the time evolution of a probability density distribution of particles led to the new LBM [MZ88, HJ89]. A great advance was the inclusion of a simplified collision operator by the name BGK for their creators [BGK54], which was derived independently in [CCM92, QDL92].

Further improvements as a different collision operator of the form of a multi-relaxation scheme [LL00, DGK$^+$02], better boundary conditions [Zie93, Sko93, NCGB95, IYO95, ZH97, MLS99, LCM$^+$08] or local mesh refinement and non-uniform grids [NS92, HLD96] have increased the popularity and use of the LBM in the computational fluid dynamics area.

Although a general comparison between LB solvers and NS ones is difficult, [GKT$^+$06] compared solvers of both types. Their conclusion is that the computational efficiency of the LBM solver is quite competitive with regard to a discretization of the corresponding problem with NS. For special problems, each solver has its advantages and disadvantages; for example, the LBM can handle problems with a wide range of Knudsen numbers, whereas a solution with NS can not be applied.

Overall, LBM performs very well for complex geometries and, if the simplified BGK operator is used, the computations needed are simple arithmetic operations. This reasons motivated us to use LBM as the fluid model to implement in the GPU to reach a high performance and achieve real-time simulations.

3.2 Algorithm skeleton

The most basic incarnation of LBM is based on two simple steps executed for all the cells in the lattice: stream and collision. The simulated microscopic particles can only move in a restricted number of directions, which is determined by the model used. As LBM is a statistical method, there are no particles per se, but particle distribution functions (dfs) which account for that microscopic movement. Figure 3.1 shows the most common distribution models used throughout the literature.

As the most part of the formulas for the LBM only depend on the dfs, we will proceed with the general algorithm explanation without further detail on the distribution models; they will be discussed in Sections 3.6.1 and 3.7.1 for the 3D and 2D case, respectively. We will, however, use images from the D2Q9 model for clarity.

For each of the restricted directions or velocity vectors $\mathbf{e}_{0..n}$, a floating point number $f_{0..n}$ is stored, representing the fraction of particles moving along that direction. These are the already introduced distribution func-

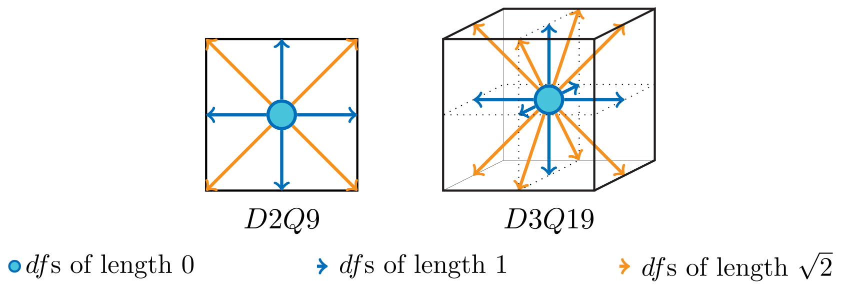


Figure 3.1: The most popular LBM models used.

tions.

During the first part of the algorithm, the stream step, the dfs move along their velocity vector to the next cell, they are advected; as shown in Figure 3.2. In terms of the dfs, it can be formulated as

$$f_i'(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t), \tag{3.1}$$

where f_i' is the value of f_i after streaming. Δx is the lattice size and Δt is the time step, but if they are normalized ($\Delta t/\Delta x = 1$) the stream step can be implemented as a simple copy operation, avoiding float point operations.

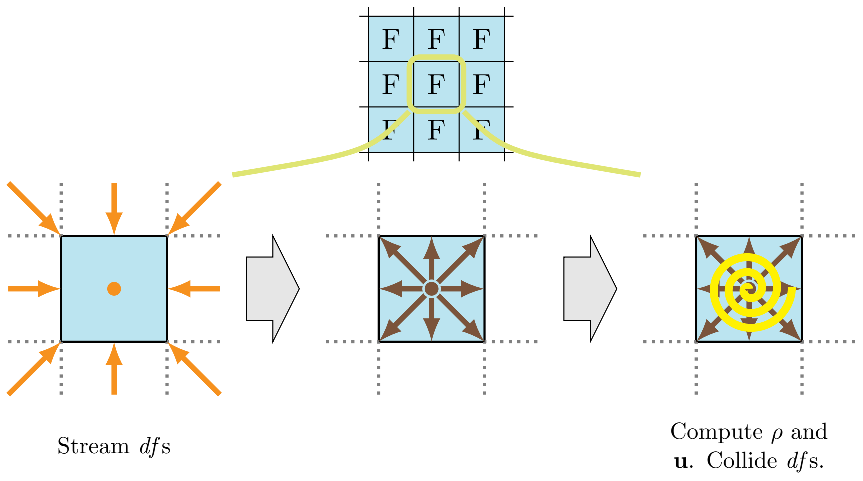


Figure 3.2: Stream and Collision steps for a fluid cell.

The second part of the algorithm is the responsible of the behaviour of the fluid. This part is the collision step, which is expressed as

$$f_i(\mathbf{x}, t) = f_i'(\mathbf{x}, t) + \Omega_i(f(\mathbf{x}, t)), \tag{3.2}$$

where Ω_i is the collision operator that represents the rate of change of f_i resulting from collision.

Using the BGK collision operator, the incoming *df*s are weighted with the so called equilibrium distribution functions, denoted here as f_i^{eq}. Equation 3.2 can be rewritten as

$$f_i(\mathbf{x}, t) = (1 - \omega) f_i'(\mathbf{x}, t) + \omega f_i^{eq}, \tag{3.3}$$

where ω is the relaxation parameter and is related to the kinematic viscosity of the fluid. It can also be found in the literature as $\tau = 1/\omega$. How ω is defined, as well as other proper details on the parametrization, will be given in Section 3.4.

These equilibrium *df*s are obtained from a Taylor expansion of the Maxwell-Boltzmann equilibrium distribution function and, in the limit of low Mach numbers, take the form

$$f_i^{eq} = \rho w_i \left(1 + 3\mathbf{e}_i \cdot \mathbf{u} + \frac{9}{2}(\mathbf{e}_i \cdot \mathbf{u})^2 - \frac{3}{2}\mathbf{u}^2\right), \tag{3.4}$$

where w_i depends on the distribution model and ρ and $\mathbf{u}$ are the local macroscopic density and velocity of the fluid. These macroscopic values are computed by summation of all *df*s for each cell as

$$\rho = \sum f_i, \tag{3.5}$$

$$\rho\mathbf{u} = \sum \mathbf{e}_i f_i. \tag{3.6}$$

The values obtained from Equation 3.3 are finally stored as the *df*s for time $t + \Delta t$.

3.2.1 Boundary conditions

The fluid simulation can not be complete without proper boundary conditions. Although there are various alternatives, [Zie93, IYO95, ZH97] among others, there is a very commonly used one: the no-slip boundary condition which provides normal and tangential velocities equal to 0. It is also known as the bounce-back rule.

To ensure that no particle of fluid leaves the domain and no unknown values are introduced, when a fluid cell should receive a *df* from a boundary cell, that *df* is set as the inverted one from the own set of the fluid cell as seen in Figure 3.3:

$$f_i'(\mathbf{x}, t + \Delta t) = f_{\tilde{i}}(\mathbf{x}, t), \tag{3.7}$$

where $\tilde{i}$ denotes the inverse direction of i, $\mathbf{e}_{\tilde{i}} = -\mathbf{e}_i$. The collision step remains the same as for normal fluid cells.

The counterpart to the no-slip boundary condition is the free-slip boundary, also in Figure 3.3. This one results in a normal velocity equal to 0, but maintains the tangential velocity unchanged. In contrast to Equation 3.7

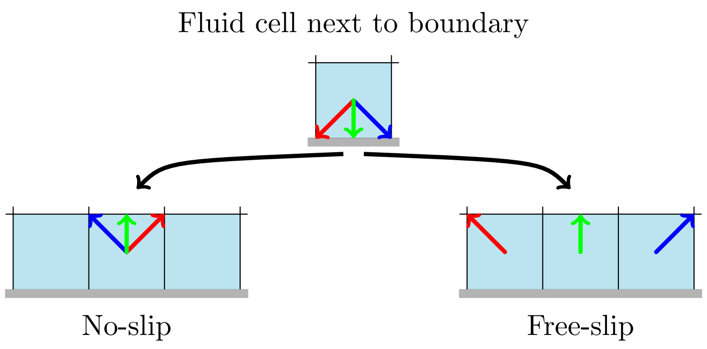


Figure 3.3: Simple boundaries for LBM simulations: no-slip and free-slip.

where the dfs are reflected along normal and tangential direction, the free-slip scheme only requires the reflection along the normal of the surface. This, however, comes at a price: the computations are not local as the neighbour cells will be required. Additionally, if the neighbouring cell is not a fluid one, the free-slip handling will behave as in the no-slip case.

3.3 Mathematical Derivation

This section will give an overview of the Boltzmann equation. The derivation of the Lattice Boltzmann equation will be also described later on.

3.3.1 Boltzmann Equation

Although LBM historically evolved from LGA, it was soon realized that it was a special discretization of the Boltzmann equation

$$\frac{\partial f}{\partial t} + \boldsymbol{\xi} \cdot \nabla f = \Omega(f), \tag{3.8}$$

where $f \equiv f(\mathbf{x}, \boldsymbol{\xi}, t)$ gives the amount of particles moving with a given microscopic velocity $\boldsymbol{\xi}$ through space and time. Ω is the collision operator and models the interaction of these particles.

A simplified approximation that preserves the collision invariants and tends towards a Maxwellian distribution is the BGK model [BGK54]:

$$\Omega_{BGK} = -\omega(f^{eq} - f). \tag{3.9}$$

Here, f^{eq} is the Maxwell-Boltzmann equilibrium distribution function

$$f^{MB} \equiv \frac{\rho}{(2\pi RT)^{D/2}} \exp\left(-\frac{(\boldsymbol{\xi} - \mathbf{u})^2}{2RT}\right), \tag{3.10}$$

where R is the ideal gas constant or Boltzmann constant, D is the dimension of the space, and ρ, $\mathbf{u}$ and T are the macroscopic density, velocity and temperature, respectively. These macroscopic variables are computed from the moments of the distribution function f:

$$\rho = \int f d\boldsymbol{\xi}, \tag{3.11}$$

$$\rho\mathbf{u} = \int \boldsymbol{\xi} f d\boldsymbol{\xi}, \tag{3.12}$$

$$\rho\varepsilon = \frac{1}{2}\int (\boldsymbol{\xi} - \mathbf{u})^2 f d\boldsymbol{\xi}, \tag{3.13}$$

where $\varepsilon = TD/2$ and it is the internal energy.

It is possible to derive the Navier-Stokes equations from the Boltzmann equation, through a multi-scale analysis called Chapman-Enskog. It splits the Boltzmann equation according to a hierarchy of different scales for space and time, for which the expansion[1] parameter is the Knudsen number $K_n = \lambda/L_C$. This value is the ratio between the mean free path length λ and the characteristic shortest scale of the macroscopic system that needs to be considered, L_C. The Knudsen number has to be much smaller than one, in order to be able to treat the fluid as a continuous system. A full derivation of the equations can be found in, e.g., [CD98, WG00].

3.3.2 Lattice Boltzmann Equation

We will present here the derivation of the lattice Boltzmann equation from the continuous Boltzmann equation, which is based on [HL97b]. We will follow their same nomenclature.

Starting with the BGK Boltzmann equation

$$\frac{\partial f(t)}{\partial t} + \boldsymbol{\xi} \cdot \nabla f(t) = -\frac{1}{\lambda}(f(t) - g(t)), \tag{3.14}$$

where $f(t) \equiv f(\mathbf{x}, \boldsymbol{\xi}, t)$ is the single particle distribution function for position $\mathbf{x}$, time t and microscopic velocity $\boldsymbol{\xi}$. λ is the relaxation time due to collision and g is the Maxwell-Boltzmann distribution f^{MB} from Equation 3.10. For simplicity reasons, it will be also used $f(t + \delta_t) \equiv f(\mathbf{x} + \boldsymbol{\xi}\delta_t, \boldsymbol{\xi}, t + \delta_t)$ as an abbreviation in the following equations.

Discretization of time

Equation 3.14 can be rewritten as an ordinary differential equation:

$$\frac{Df}{Dt} + \frac{1}{\lambda} f = \frac{1}{\lambda} g, \quad \text{where } \frac{D}{Dt} = \frac{\partial}{\partial t} + \boldsymbol{\xi} \cdot \nabla \tag{3.15}$$

is the time derivative along the characteristic velocity $\boldsymbol{\xi}$. Equation 3.15 is further integrated and simplified over a small time step δ_t and assuming g

[1]The expansion is usually truncated after terms of second order.

is smooth enough locally as

$$f(t+\delta_t) - f(t) = -\frac{\delta_t}{\lambda}(f(t) - g(t)), \tag{3.16}$$

where δ_t/λ is the dimensionless relaxation time (in the units of δ_t) and is better known as ω or $1/\tau$.

Although g is written as dependent on t, it only relies on the hydrodynamic variables ρ, $\boldsymbol{\xi}$ and T. However, a momentum space discretization is needed in order to allow them to be integrated from Equations 3.11, 3.12 and 3.13. We will introduce this additional discretization in the next sections.

Equilibrium distribution approximation

In the lattice Boltzmann equation, the Maxwell-Boltzmann equilibrium distribution from Equation 3.10 is used. It is Taylor expanded up to second order achieving a valid approximation for low Mach numbers, resulting in

$$f^{(eq)} = \frac{\rho}{(2\pi RT)^{D/2}} \exp\left(\frac{-\boldsymbol{\xi}^2}{2RT}\right)\left(1 + \frac{\boldsymbol{\xi}\cdot\mathbf{u}}{RT} + \frac{(\boldsymbol{\xi}\cdot\mathbf{u})^2}{2(RT)^2} - \frac{\mathbf{u}^2}{2RT}\right). \tag{3.17}$$

What remains to be done is the discretization of momentum space or the microscopic velocities, which we will explain below.

Phase space discretization

From the Boltzmann Equation (Equation 3.8), the moving microscopic particles can do so in an infinite number of directions or more precisely, with an infinite number of microscopic velocities $\boldsymbol{\xi}$. This is called the phase space or momentum space. In order to be able to implement the Boltzmann Equation the phase space should be discretized, restricting the directions the particles can travel along. For simplicity and conciseness, we will only present the D2Q9 model derivation from [HL97b].

There are two considerations to be made in the needed discretization. On one hand, the discretization of momentum space is coupled to the configuration space; a lattice structure is obtained. On the other hand, isotropy has to be retained. This is the most important of the symmetries of Navier-Stokes equations. The lattice should, then, be invariant to rotations of the problem.

For the Lattice Boltzmann derivation, the moments are used directly as constraints for the numerical integration method. For isothermal models, as the one used here, only the first moment, the velocity is required. Calculating these moments for Equation 3.17 is equivalent to evaluate the following integral:

$$I = \int \psi(\boldsymbol{\xi}) f^{(eq)} d\boldsymbol{\xi}, \quad \text{where } \psi(\boldsymbol{\xi}) = \xi_x^m \xi_y^n. \tag{3.18}$$

Here, ψ is a polynomial of $\boldsymbol{\xi}$: ξ_x and ξ_y are the x and y components of $\boldsymbol{\xi}$. This can be calculated numerically with a Gaussian-Hermite quadrature of third-order

$$I^m = \sum_{j=1}^{3} \mathsf{w}_j (\zeta_j)^m. \tag{3.19}$$

The three abscissas of the quadrature are

$$\zeta_1 = -\sqrt{3/2}, \quad \zeta_2 = 0, \quad \zeta_3 = \sqrt{3/2}, \tag{3.20}$$

and the corresponding weights

$$\mathsf{w}_1 = \sqrt{\pi}/6, \quad \mathsf{w}_2 = 2\sqrt{\pi}/3, \quad \mathsf{w}_3 = \sqrt{\pi}/6. \tag{3.21}$$

Then, the moment function of Equation 3.18 can be defined as

$$I = \frac{\rho}{\pi} \sum_{i=1}^{3} \sum_{j=1}^{3} \mathsf{w}_i \mathsf{w}_j \psi(\boldsymbol{\zeta}_{i,j}) \left(1 + \frac{\boldsymbol{\zeta}_{i,j} \cdot \mathbf{u}}{RT} + \frac{(\boldsymbol{\zeta}_{i,j} \cdot \mathbf{u})^2}{2(RT)^2} - \frac{\mathbf{u}^2}{2RT} \right), \tag{3.22}$$

where $\boldsymbol{\zeta}_{i,j} = \sqrt{2RT}(\zeta_i, \zeta_j)^T$. Here, we can already identify the equilibrium distribution function

$$f_{i,j}^{(eq)} = \frac{\mathsf{w}_i \mathsf{w}_j}{\pi} \rho \left(1 + \frac{\boldsymbol{\zeta}_{i,j} \cdot \mathbf{u}}{RT} + \frac{(\boldsymbol{\zeta}_{i,j} \cdot \mathbf{u})^2}{2(RT)^2} - \frac{\mathbf{u}^2}{2RT} \right). \tag{3.23}$$

We can get the final weights w from

$$w_\alpha = \frac{\mathsf{w}_i \mathsf{w}_j}{\pi} = \begin{cases} 4/9, & i = j = 2 & \alpha = 0 \\ 1/9, & i = 2 \oplus j = 2 & \alpha = 1..4 \\ 1/36, & i \neq j \neq 2 & \alpha = 5..8 \end{cases} \tag{3.24}$$

Similarly, and with the substitution of $\sqrt{2RT}\sqrt{3/2} = \sqrt{3RT} = c$, we can find the final discretized velocity vectors:

$$\begin{aligned} \mathbf{e}_0 &= \boldsymbol{\zeta}_{2,2} = (0,0)^T c \\ \mathbf{e}_{1..4} &= \boldsymbol{\zeta}_{1,2}, \boldsymbol{\zeta}_{2,1}, \boldsymbol{\zeta}_{2,3}, \boldsymbol{\zeta}_{3,2} = (\pm 1, 0)^T c, (0, \pm 1)^T c \\ \mathbf{e}_{5..8} &= \boldsymbol{\zeta}_{1,1}, \boldsymbol{\zeta}_{1,3}, \boldsymbol{\zeta}_{3,1}, \boldsymbol{\zeta}_{3,3} = (\pm 1, \pm 1)^T c. \end{aligned} \tag{3.25}$$

Equation 3.23 can be rewritten to its final form as the equilibrium distribution function for each of the 9 velocity vectors:

$$f_\alpha^{eq} = w_\alpha \rho \left(1 + \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3\mathbf{u}^2}{2c^2} \right). \tag{3.26}$$

Accordingly, the hydrodynamic moments from Equations 3.11, 3.12 and 3.13, on which depends the equilibrium distribution function[2], can now be evaluated as

$$\rho = \sum_\alpha f_\alpha, \tag{3.27}$$

$$\rho\mathbf{u} = \sum_\alpha \mathbf{e}_\alpha f_\alpha, \tag{3.28}$$

$$\rho\varepsilon = \frac{1}{2}\sum_\alpha (\mathbf{e}_\alpha - \mathbf{u})^2 f_\alpha. \tag{3.29}$$

It is worth noting that the discretized velocity vectors were given by the chosen quadrature, as was the configuration of the lattice from these vectors. Other models like the D3Q27 can be derived similarly although the same is not possible for other more irregular models, as the common D3Q19. For these models, the *ansatz method* must be used [WG00].

3.4 Parametrization

For the LBM to behave properly, we need to assign some of its values from real-world units. The parameters needed are Δx, Δt and ω.

As stated earlier, we would like a normalization between Δx and Δt to minimize floating point operations in the implementation, like the simplification of the direct copy of *df*s for the stream step. To achieve that, we will use adimensional quantities.

Let ν^* $[m^2/s]$, D^* $[m]$, $\mathbf{g}^*$ $[m/s^2]$ and r be the real kinematic viscosity, domain size[3], gravitational force and desired resolution of the lattice[4], respectively. The cell size used by the LBM can be computed as $\Delta x^* = D^*/r$.

From the gravitational force we can now compute the dimensional time step Δt^*. We can, additionally, limit the compressibility due to that force as in [Thü07],

$$\Delta t^* = \sqrt{\frac{g_c \Delta x^*}{|\mathbf{g}^*|}}, \tag{3.30}$$

where g_c is the maximum factor of compression caused by gravitational acceleration. We have used a value of g_c in the range of $[0.001, 0.01]$ in our tests.

With Δx^* and Δt^* we can now compute the adimensional gravitational

[2]As we are working with an isothermal model, the energy density $\rho\varepsilon$ will not be needed.

[3]A characteristic length; the width of the simulation, for example.

[4]It should be the matching resolution to the length from the domain size variable; the width of the lattice, for the same example.

acceleration and kinematic viscosity as

$$\mathbf{g} = \mathbf{g}^* \frac{\Delta t^{*2}}{\Delta x^*}, \tag{3.31}$$

$$\nu = \nu^* \frac{\Delta t^*}{\Delta x^{*2}}. \tag{3.32}$$

Finally, the relaxation parameter ω is computed from the adimensional viscosity as

$$\omega = \frac{2}{6\nu + 1}. \tag{3.33}$$

The relaxation parameter will remain in the range $(0, 2]$, where values near 0 will result in very viscous fluids and, conversely, values near 2 will result in more turbulent flows. For values close to 2, however, the method can become unstable, although the Smagorinsky stabilization in Section 3.5 improves the situation.

The Δx^* and Δt^* values will be further needed in other sections like the coupling with rigid body simulations (Section 3.7.4) and particle simulations (Section 3.8), as well as for the render of the fluid (Section 5.2) as scaling factors.

3.5 Large Eddy Simulation & Improved Stability

Turbulent flows, represented by high Reynolds numbers and caused by low viscosities, introduce instabilities in the LBM as the ω relaxation parameter approximates 2. In order for the LBM to be able to simulate these kind of flows, the Smagorinsky sub-grid model [Sma63] was introduced and adapted in [HSCD96]. Here, the Smagorinsky model will be used as in [LWK03, Thü07, Zho04] for its stabilization properties.

This turbulence model adds the computation of the local stress tensor, which is used to modify the relaxation parameter of each cell. This is possible as each cell contains information about the derivatives of the hydrodynamic variables in its *df*s. For the calculation of the modified relaxation parameter, the Smagorinsky constant $\mathcal{C}$ is used, which should be computed dynamically per problem depending on the smallest resolved scales that contribute to the Reynolds stresses, as shown in [GPMC91]. It is more usual, however, to choose a fixed value for the Smagorinsky constant; we have used the value 0.04 for our testing.

This additional computation is easily integrated in the basic algorithm: the modified relaxation parameter ω' is calculated between the stream and collision steps as follows:

1. Using Einstein summation convention, the non-equilibrium stress tensor $\Pi_{\alpha,\beta}$ is computed for each cell:

$$\Pi_{\alpha,\beta} = \sum_i e_{i_\alpha} e_{i_\beta} (f_i - f_i^{eq}). \tag{3.34}$$

2. Calculate the local stress tensor $\mathcal{S}$ as
$$\mathcal{S} = \frac{\sqrt{\nu^2 + 18\mathcal{C}^2\sqrt{\Pi_{\alpha,\beta}\Pi_{\alpha,\beta}}} - \nu}{6\mathcal{C}^2}. \tag{3.35}$$

3. Finally, the modified relaxation parameter that will be used in the collision step is given by
$$\omega' = \frac{2}{6(\nu + \mathcal{C}^2\mathcal{S}) + 1}. \tag{3.36}$$

This modification to the basic LBM algorithm ensures a local increment to the viscosity depending on the size of the stress tensor calculated from the non-equilibrium parts of the *df*s of each cell. Doing so, instabilities due to close values of ω to 2 are eliminated, as shown in Figure 3.4.

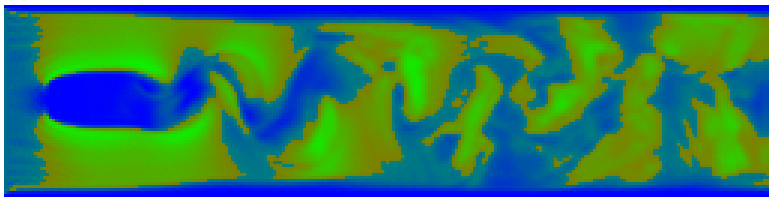


Figure 3.4: Kármán vortex street. 2D slice from the speeds of the velocity field of a D3Q19 LBM simulation in a lattice of 256x64x64 with $\omega = 2$. The speeds are color-graded; blue is for the lower ones, going through green as the red ones are the highest.

It is worth noting that for engineering purposes it is important to evaluate the deviations on accuracy caused by this turbulence model. For graphical uses like those intended in this thesis, however, they can be ignored, as the primary focus is the stability of the simulation.

Another method that leads to the LBM stabilization is the multi-relaxation time (MRT) [LL00, DGK$^+$02]. It uses a different collision operator for Equation 3.2 that relaxes the different hydrodynamic moments separately and provides better accuracy than the BGK model. Furthermore, MRT can be also combined with the Smagorinsky turbulence model as in, e.g., [KTL03]. However, as we want to maximize performance for our simulations, we have chosen the simpler BGK collision operator that, combined with the Smagorinsky turbulence model, is stable and accurate enough for our simulations.

3.6 LBM 3D with a Free-surface

Free surface fluid simulations are still a great challenge to solve with good performance. The added complexity is due to the incompressibility restriction and the surface tracking required, as in a simulation of this type two fluids are used (liquid and gas), although usually only one is computed: the liquid. In contrast to liquid simulations where the fluid occupies a finite portion of the simulation domain, gas simulations don't require this surface tracking as they expand the whole domain of the simulation; to be able to visualize them a tint is used, usually as the form of marker particles advected with the fluid. Here, however, we are interested in the simulation of liquids, thus, the boundary of the fluid phase has to be determined and updated in correspondence to the simulation.

We limit the discussion of free surfaces to Eulerian simulations (e.g. LBM), as Lagrangian ones (e.g. SPH) represent the fluid with primitives as particles; the mass is conserved without further calculations and the free surface is just determined by the external particles from the bulk of the fluid.

There are three major algorithms for tracking the free surface of a fluid:

- **Marker and Cell**(MAC). Introduced by [HW65], it was brought to light again by [FM96, FM97a] with a complete 3D formulation. It is based on the introduction of massless particles which are advected with the fluid's velocity field. Then, the cells of the grid used for the simulation can be categorized in three types:

  - Empty: Cells that do not contain any particle at all.
  - Interface: Cells containing at least one particle that are adjacent to Empty cells.
  - Fluid: Cells containing at least one particle and are not Interface cells.

  Although the particles do not represent any mass, extra care has to be taken when an Empty cell changes to Interface, or vice versa, in order for the incompressibility to be preserved. We suggest [GDN98, Suá06] to the reader for detailed instructions on a complete NS solver using this method.

- **Level-set**(LS). Although developed by [OS88], it was [FF01] who introduced it as a method to track the surface of fluids. Level-sets are functions one dimension higher than the one used in the problem. They define closed curves or curved surfaces that are the interface between spaces.

  To update the level-set, denoted here by ϕ, the following equation has to be solved:

  $$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0, \tag{3.37}$$

  where $\mathbf{u}$ is the velocity of the fluid. ϕ is implemented as the distance to the surface of the fluid, thus, points inside the fluid will have a positive value and points outside will have a negative value.

The Level-set method can suffer from mass loss and extra efforts have to be made to maintain incompressibility. [FF01] already used an hybrid method with advected particles; the particles helped to refine the surface when the curvature of the level-set was above a certain threshold. Likely, [EFFM02] presented a different hybrid solution where marker particles were introduced in the two sides of the level-set and were advected seamlessly; if a particle was detected to have changed sides, the level-set was reinitialized using the particle information.

- **Volume of Fluid**(VOF). Presented in [HN81], the VOF method solves the same equation as the LS method. In this case, the property advected is not the surface itself, but some variable φ that each cell stores. This variable is the quantity of fluid that cell has at time t, in the range $[0, 1]$. Cells with $\varphi = 0$ are Empty cells, cells with $\varphi = 1$ are Fluid cells, and cells with $0 < \varphi < 1$ are Interface cells. This makes it differ from the LS method, as VOF is able to conserve the mass and allows for easily traced changes on topology of the fluid, e.g., interfaces can join or break apart without ambiguities.

  This method has been used to achieve results of high quality; for example, [Sus03] coupled it with the LS method and the result was later used in [MMS04] to control and animate breaking waves. For more details, [SZ99] gives a review of the VOF method.

We discarded the MAC method early on, as the marker particles could potentially cluster or segregate, losing sampling of the fluid. To solve this, the fluid is resampled each time step, creating or removing particles where needed. This dynamic control on the number of particles involves an additional problem in a GPU implementation for real-time simulations.

Although there are some more alternatives for the free surface LBM models, like [GRZZ91, GS03], [Thü07] presented a more simpler loosely VOF-based model, in which the mass fluxes are easily computed from the streaming *df*s. [TR04] already compared this method with a LBM level-set implementation; no clear winner arose, the level-set gave a smoother surface but it suffered from mass loss, in contrast to the VOF-based algorithm.

As it wasn't done before, we, therefore, chose to implement in CUDA this VOF-based algorithm for its ease of integration with the LBM. It will be explained here for completeness, but the implementation details will be given in Section 4.2.1.

3.6.1 D3Q19 model and modified equilibrium distribution function

For a full 3D simulation we have chosen to use the D3Q19 model (previously shown in Figure 3.1). In contrast, the D3Q15 has a decreased stability due to the reduced discretization of the phase space, and the D3Q27, although more complete, has no apparent advantages over the D3Q19 model but needs more memory requirements.

The discretized velocities for this model are defined as:

$$\begin{aligned}
\mathbf{e}_0 &= (0,0,0)^T, \\
\mathbf{e}_{1,2} &= (\pm 1,0,0)^T, \\
\mathbf{e}_{3,4} &= (0,\pm 1,0)^T, \\
\mathbf{e}_{5,6} &= (0,0,\pm 1)^T, \\
\mathbf{e}_{7..10} &= (\pm 1,\pm 1,0)^T, \\
\mathbf{e}_{11..14} &= (0,\pm 1,\pm 1)^T, \text{ and} \\
\mathbf{e}_{15..18} &= (\pm 1,0,\pm 1)^T.
\end{aligned} \tag{3.38}$$

Furthermore, we use the incompressible model from [HL97a], who introduce a modified equilibrium distribution with regard to Equation 3.4

$$f_i^{eq} = w_i\left(\rho + 3(\mathbf{e}_i\cdot\mathbf{u}) - \frac{3}{2}\mathbf{u}^2 + \frac{9}{2}(\mathbf{e}_i\cdot\mathbf{u})^2\right), \text{ where } w_i = \begin{cases} 1/3 & i = 0, \\ 1/18 & i = 1..6, \\ 1/26 & i = 7..18. \end{cases} \tag{3.39}$$

The velocity computation is changed, besides, in contrast to Equation 3.6, the normalization with the density is not needed,

$$\mathbf{u} = \sum_i \mathbf{e}_i f_i. \tag{3.40}$$

Aside from these two changes, however, the stream and collision steps for the LBM are the same as those described in Section 3.2. Next, we will introduce the additions to the algorithm that allow for a free surface simulation based on [Thü07].

3.6.2 LBM Volume of Fluid

In the simulation of free surfaces, we can distinguish two main regions: empty space and fluid (or more precisely, liquid). This requires us to classify the LBM cells depending what region they belong to at a given time. Therefore, we can classify the cells in three types: Empty, Fluid and Interface. Empty cells will be those that do not contain fluid, Fluid cells will be those completely filled and Interface cells, which are not completely filled, will be the separation layer between Fluid and Empty cells. For Empty cells, as for boundary ones, there will not be any computation. Fluid cells only need the usual steps of the LBM algorithm: stream and collision. Interface cells, however, form a closed layer and are the ones carrying the responsibility to track the surface of the fluid, as shown in Figure 3.5.

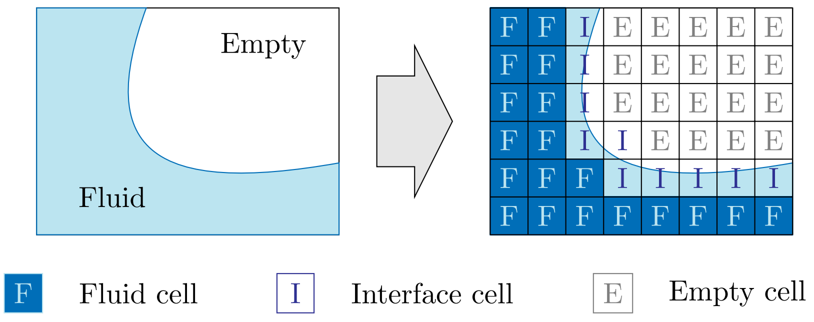


Figure 3.5: Classification of cells depending on the region they are on.

The movement of the fluid surface is tracked by the computation of the mass each cell contains. Two new properties are introduced for each cell: the mass m of the cell and its fluid fraction φ, which has to be in the range $[0, 1]$ and is just the ratio between the mass m and the density ρ of the cell (from Equation 3.5) as

$$\varphi = \frac{m}{\rho}. \tag{3.41}$$

For each iteration, the additional operations needed for interface cells, illustrated in Figure 3.6, proceed as follows:

1. Calculate the mass exchange. This is computed directly from the values streamed between two adjacent cells for each direction of the model, averaging with the fluid fraction of the cells:

   $$\begin{aligned} \Delta m_i(\mathbf{x}, t + \Delta t) &= m_{c_i} \frac{\varphi(\mathbf{x} + \mathbf{e}_i \Delta t, t) + \varphi(\mathbf{x}, t)}{2}, \\ \text{where } m_{c_i} &= (f_{\bar{i}}(\mathbf{x} + \mathbf{e}_i \Delta t, t) - f_i(\mathbf{x}, t)) \end{aligned} \tag{3.42}$$

   This equation is symmetric; the amount of fluid leaving one cell enters the other one, and vice versa.

   Now, the mass value is updated for the next time step:

   $$m(\mathbf{x}, t + \Delta t) = m(\mathbf{x}, t) + \sum_i \Delta m_i(\mathbf{x}, t + \Delta t). \tag{3.43}$$

2. Stream step. Only *df*s coming from Fluid and Interface cells are accepted, as the ones from Empty cells are invalid.

3. Reconstruct incoming *df*s from Empty cells. Here a constant value of atmospheric pressure, and therefore, a constant density $\rho_A = 1$ is used, as it is also the reference value for the density of the fluid. It is assumed that the viscosity of the fluid is lower than the one of the gas[5], while having a higher density. This implies that the gas follows

[5]Not simulated but represented as the Empty cells.

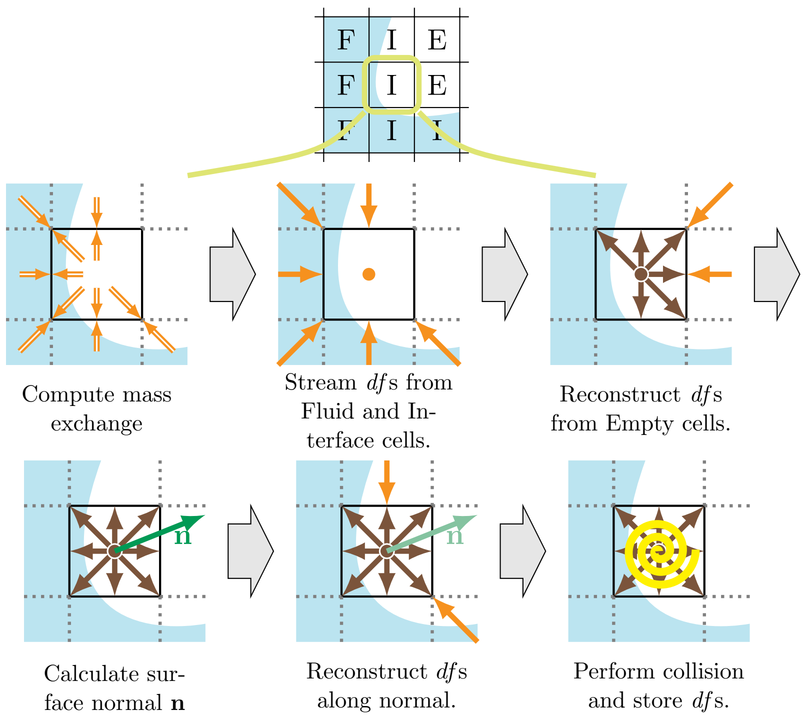

F
I
E
F
I
E
F
I
I
Compute mass exchange
Stream dfs from Fluid and Interface cells.
Reconstruct dfs from Empty cells.
n
Calculate surface normal $\mathbf{n}$
n
Reconstruct dfs along normal.
Perform collision and store dfs.

Figure 3.6: Steps done for an interface cell.

the motion of the fluid at the interface. The required dfs are finally computed as

$$f'_i(\mathbf{x}, t + \Delta t) = f_i^{eq}(\rho_A, \mathbf{u}) + f_{\tilde{i}}^{eq}(\rho_A, \mathbf{u}) - f_{\tilde{i}}(\mathbf{x}, t), \tag{3.44}$$

where $\mathbf{u}$ is the velocity of the cell at position $\mathbf{x}$ and time t.

4. Compute the surface normal. For the next step the surface normal at this Interface cell will be needed; it is computed using finite differences from the fluid fraction:

$$\mathbf{n} = \frac{1}{2}\begin{pmatrix} \varphi(\mathbf{x}_{i-1,j,k}) - \varphi(\mathbf{x}_{i+1,j,k}) \\ \varphi(\mathbf{x}_{i,j-1,k}) - \varphi(\mathbf{x}_{i,j+1,k}) \\ \varphi(\mathbf{x}_{i,j,k-1}) - \varphi(\mathbf{x}_{i,j,k+1}) \end{pmatrix} \tag{3.45}$$

5. Reconstruct dfs along normal. To balance the forces on each side of the fluid interface, the dfs coming from the direction of the surface normal are also reconstructed. Equation 3.44 will be used for all the dfs that hold

$$\mathbf{u} \cdot \mathbf{e}_{\tilde{i}} > 0. \tag{3.46}$$

6. Collision step. Finally, with the full set of dfs, the collision step can be executed. The fluid fraction φ is also updated according to Equation 3.41.

Additionally, after the iteration is over and all cells have been updated, Interface cells that have filled or emptied during this time step have to be reflagged, that is, changed their type, accordingly. The density calculated during collision to check if the cells filled or emptied:

$$\begin{aligned} m(\mathbf{x}, t+\Delta t) &> (1+\epsilon)\rho(\mathbf{x}, t+\Delta t) \rightarrow \quad \text{cell has filled,} \\ m(\mathbf{x}, t+\Delta t) &< (0-\epsilon)\rho(\mathbf{x}, t+\Delta t) \rightarrow \quad \text{cell has emptied.} \end{aligned} \tag{3.47}$$

An additional offset ϵ is used to prevent change of cells in subsequent time steps.

These cells have to be converted. For filled cells, their neighbourhood of Empty cells are converted to Interface. These neighbour cells are initialized with the equilibrium *df*s using the average ρ^{avg} and $\mathbf{u}^{avg}$ from the surround non-Empty cells. Any Interface cell around these new filled cells that is marked for emptying has to be maintained for the boundary to remain valid as a closed layer. The excess of mass $m^{ex} = m - \rho$ of these filled cells has to be distributed among the surrounding Interface cells, as it means that the boundary moved beyond the actual cell. Finally, the cell is changed to Fluid. The exceeding mass, however, is not distributed evenly; it is weighted depending on the direction of the surface normal $\mathbf{n}$:

$$m(\mathbf{x} + \mathbf{e}_i \Delta t) = m(\mathbf{x} + \mathbf{e}_i \Delta t) + (\eta_i / \eta_{total}) \cdot m^{ex}, \tag{3.48}$$

where η_{total} is the sum of all weights η_i, which are computed as

$$\begin{aligned} \eta_i &= \begin{cases} \mathbf{n} \cdot \mathbf{e}_i & \text{if } \mathbf{n} \cdot \mathbf{e}_i > 0 \\ 0 & otherwise \end{cases} \quad \text{for filled cells, and} \\ \eta_i &= \begin{cases} -\mathbf{n} \cdot \mathbf{e}_i & \text{if } \mathbf{n} \cdot \mathbf{e}_i < 0 \\ 0 & otherwise \end{cases} \quad \text{for emptied cells.} \end{aligned} \tag{3.49}$$

Likewise, the fluid neighbours of emptied cells have to be changed to Interface. These emptied cells could have negative mass values, it should be distributed as for the exceeding mass of filled cells. At last, the cell is also changed to Empty.

As the mass of the adjacent cells change, their fluid fraction should be updated accordingly. It should be noted that these computations must provide the same results independently of the order in which filled and emptied cells (as well as their neighbourhood) are converted.

Figure 3.7 shows some screenshots from our CUDA implementation for the Drop and Breaking Dam examples, with a low value for ω, thus, using higher viscosity. Other examples using the turbulence model from Section 3.5, with a ω value of 2 and the OpenGL raycasting visualization from Section 5.1 can be seen in Figures 5.2 and 6.4. The timings for these examples will be provided in Section 6.2.

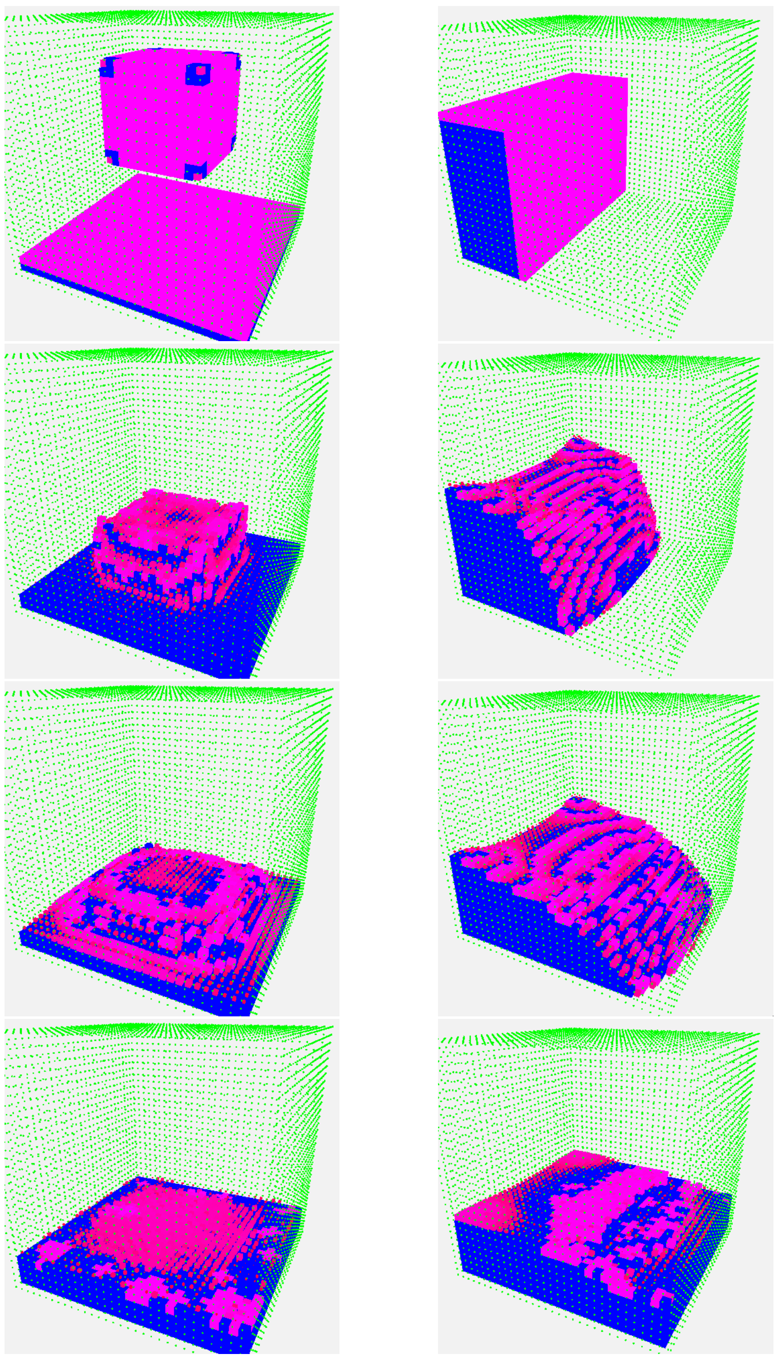


Figure 3.7: Image stills from the drop example (left column) and the breaking dam example (right column). A 32^3 grid is used in both examples. Blue cells are full fluid ones, pink cells are Interface cells with a size proportional to their fluid fraction and green points indicate boundary cells.

3.7 Shallow Waters Simulation

The full simulation of a 3D fluid flow, although feasible as shown in the previous scenes, is still a very heavy problem for real-time simulations. To reduce complexity and assuming that the water depth is much smaller than the horizontal scale, we can simplify the problem to the simulation of the surface of the fluid by using the Shallow Waters Equations, as [KM90] pioneered in the computer graphics field. Here, the surface is tracked as a height field. This imposes some restrictions; the fluid can not break apart vertically, e.g., we can't simulate breaking waves or the pour of wine in a glass. We will present, however, an hybrid approach to solve these types of limitations in Section 3.8.

For now, we will introduce the simpler surface-only simulation based on the Shallow Waters Equations, also known as St. Venant Equations. These equations are derived from depth-integration of the original Navier-Stokes Equations (2.1 and 2.2) and are usually used to simulate waves whose wavelength is similar to the overall fluid depth with a wave propagation speed constant for all amplitudes.

They can be written as in [Zho04]:

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial (h u_j)}{\partial x_j} &= 0, & (3.50)\\ \frac{\partial h u_i}{\partial t} + \frac{\partial (h u_i u_j)}{\partial x_j} + \frac{g}{2}\frac{\partial h^2}{\partial x_i} &= F_i, & (3.51) \end{aligned}$$

where i and j are Cartesian indices and Einstein summation convention is used, h is the water depth, u_i is the depth-averaged velocity component in the i direction, t is the time and g is the vertical gravity. F_i are the external forces in the i direction and, as pictured in Figure 3.8, can be defined as

$$\begin{aligned} F_i &= -gh\frac{\partial z_b}{\partial x_i} - \tau_{bi} + \tau_{wi}, \text{ with} & (3.52)\\ \tau_{bi} &= C_b u_i \sqrt{u_j u_j}, & (3.53)\\ \tau_{wi} &= C_w u_{wi} \sqrt{u_{wj} u_{wj}}, & (3.54) \end{aligned}$$

where z_b is the bed elevation, τ_{wi} is the wind shear stress and τ_{bi} is the bed shear stress. C_b and C_w are the bed friction coefficient and the wind resistance coefficient, respectively. u_{wi} is the component of the wind velocity in the i direction. We also define the value $\eta = h + z_b$ for further use.

Getting back to the LBM, we can apply it to the Shallow Waters (SW) like in, e.g., [TSB07, Thü07, Tub10]. Derivations for the full LBMSW can be found in, e.g., [Sal99, Del02, Zho02].

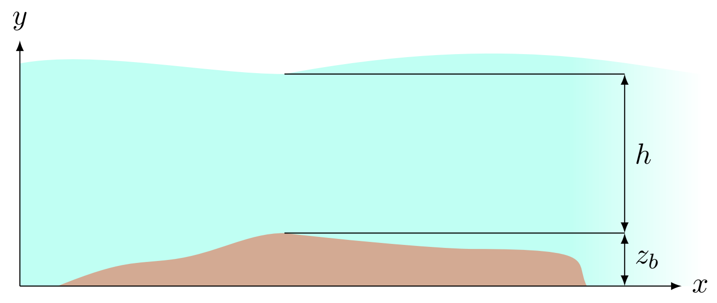


Figure 3.8: Sketch for the Shallow Water Simulation with arbitrary underlying terrain.

3.7.1 D2Q9 model and equilibrium distribution function

As the fluid simulation is only two-dimensional, the common D2Q9 model from Figure 3.1 is used. Its discretized velocities are:

$$\begin{aligned}
\mathbf{e}_0 &= (0,0)^T, \\
\mathbf{e}_{1,2} &= (\pm 1,0)^T, \\
\mathbf{e}_{3,4} &= (0,\pm 1)^T, \\
\mathbf{e}_{5..8} &= (\pm 1,\pm 1)^T. \qquad (3.55)
\end{aligned}$$

In the LBMSW, instead of computing fluid pressure (or density), a height value is considered. Equations 3.5 and 3.6 should be rewritten as

$$h = \sum f_i, \qquad (3.56)$$

$$\mathbf{u} = \frac{1}{h}\sum \mathbf{e}_i f_i. \qquad (3.57)$$

Likewise, the equilibrium distribution function is different:

$$f_i^{eq} = \begin{cases} h - w_i h\left(\frac{15}{6}gh - \frac{3}{2}\mathbf{u}^2\right) & i = 0, \\ w_i h\left(\frac{3}{2}gh + 3(\mathbf{e}_i\cdot\mathbf{u}) + \frac{9}{2}(\mathbf{e}_i\cdot\mathbf{u})^2 - \frac{3}{2}\mathbf{u}^2\right) & i = 1..8, \end{cases}$$

$$\text{where } w_i = \begin{cases} 4/9 & i = 0, \\ 1/9 & i = 1..4, \\ 1/36 & i = 5..8. \end{cases} \qquad (3.58)$$

The rest of the LBM algorithm remains exactly the same, stream and collide; the same boundary conditions can be applied as well as the turbulence model from Section 3.5.

3.7.2 Underlying terrain and other forces

Forces from Equation 3.52 are also introduced into the LBMSW adding a couple of terms to the LBM collision from Equation 3.3 as in [Zho11]

$$f_i(\mathbf{x},t) = f_i'(\mathbf{x},t) - \omega(f_i'(\mathbf{x},t) - f_i^{eq}) - \mathcal{X}_i + \mathcal{Z}_i, \tag{3.59}$$

in which, using $\bar{h} = [h(\mathbf{x} + \mathbf{e}_i\Delta t, t) + h(\mathbf{x}, t)]/2$,

$$\mathcal{X}_i = \begin{cases} \frac{g\bar{h}}{2}[z_b(\mathbf{x} + \mathbf{e}_i\Delta t) - z_b(\mathbf{x})], & i = 1..4, \\ 0, & \text{otherwise,} \end{cases} \tag{3.60}$$

and

$$\mathcal{Z}_i = \begin{cases} 0 & i = 0, \\ \frac{F_\alpha}{6e_{i_\alpha}} & \text{otherwise,} \end{cases} \tag{3.61}$$

where F_α is the component of the external forces in the α direction from Equations 3.53 and 3.54, as $\mathcal{X}_i$ already takes into account the bed elevation. Other forces like those to take into account the Coriolis effect can be also added, but we have dismissed them.

This approach, in contrast to the forcing terms used in, e.g., [TSB07, GRGT10], allows us to preserve the simple arithmetic computations of the LBM, avoiding first order derivatives for the underlying terrain but achieving the same accuracy results.

3.7.3 Dry sections

The LBMSW described so far interacts with an arbitrary bed surface but is limited to subcritical flows (slow moving deep water) as it has to fulfill the following condition [Zho04]:

$$\frac{\mathbf{u} \cdot \mathbf{u}}{gh} < 1 \tag{3.62}$$

In order to allow the dynamic drying and wetting of the bed topography, we have to extend the LBMSW algorithm to account for these new dry sections and how they behave with regard to the rest of the fluid. We do this imposing a minimal height limit for the fluid cells, as a small threshold parameter. Cells with a computed height below this threshold will be converted to dry cells and be discarded in further computations. When the fluid comes in again, due to the propagation step of the LBM, the cell will come wet and usable in following steps.

To ensure that no wet cells are left behind, e.g., in a case of fluid descending a slope, we enforce that this kind of cells, which already have a low height, propagate their remainder fluid in the opposite direction of the gradient of the underlying terrain as follows:

$$f_i(\mathbf{x} + \mathbf{e}_i\Delta t) = f_i(\mathbf{x} + \mathbf{e}_i\Delta t) + h(\mathbf{x}) \cdot (\eta_i/\eta_{total}), \tag{3.63}$$

where η_{total} is the sum of all weights η_i, which are computed as

$$\eta_i = \begin{cases} -(\nabla z_b \cdot \mathbf{e}_i) & \text{if } -(\nabla z_b \cdot \mathbf{e}_i) > 0 \text{ and cell at} \\ & (\mathbf{x} + \mathbf{e}_i) \text{ is a Fluid one,} \\ 0 & \text{otherwise,} \end{cases} \tag{3.64}$$

However, for very small heights, this method can lead to numerical instabilities when computing the macroscopic velocity in Equation 3.57. To avoid such a situation, the correct solution to keep stability and physical validity would be the reparametrization of the entire simulation as in [Thü07], reducing the value of the time step of an iteration, thus increasing the number of iterations needed.

In contrast to [GRGT10], who solved this problem using a modified minmod flux limiter, we have resorted to a simpler approach. Based on the restriction of Equation 3.62 we introduce an upper limit for that ratio as a parameter $\lambda \in [0, 1)$. For cells that have a ratio above λ, we rescale the velocity so the subcritical flow condition is met and compute new *df*s from the equilibrium distribution function. We have used a $\lambda = 0.95$ for our tests.

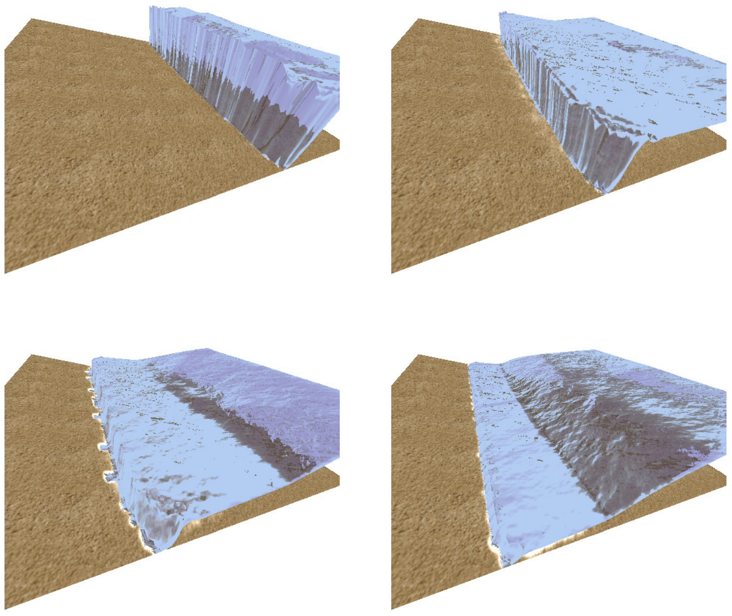


Figure 3.9: Image stills from the shore breaking dam example.

Although not physically correct, this method ensures stability in a similar fashion to the Smagorinsky's change of local viscosity from Section 3.5, dampening effectively local high velocities; as can be seen in Figures 3.9 and 3.10. Additionally, as the subcritical flow condition has to be guaranteed

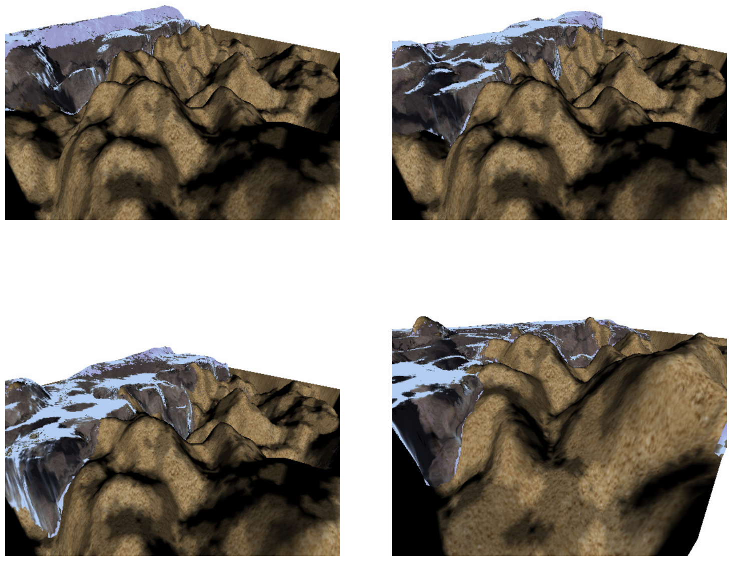


Figure 3.10: Image stills from the breaking dam over noisy ground.

for all the fluid, we can use this method to dampen high velocities in the fluid caused by external forces, whatever the height of the cell.

3.7.4 Dynamic Obstacles

In this section we will describe how dynamic obstacles, more specifically rigid bodies, may be introduced to LBMSW simulation. These solid objects react to the fluid movement but also influence its behaviour. This means that a two-way coupling between obstacles and fluid must be done.

When a rigid body is introduced in the fluid simulation, a compromise between the complexity of the model and time needed for interation must be taken. We decided to define a proxy object that simplifies geometry and keeps body weight in order to preserve the dynamics. This proxy object will be responsible for the interaction with the fluid and will provide the final geometric transformation for the real model to be visualized.

To get a simple physical model, the proxy object we have defined is a set of spheres with some associated properties. These properties are the radius r, the position $\mathbf{p} = (p_x, p_y, p_z)^T$ in object space and the normal $\mathbf{n} = (n_x, n_y, n_z)^T$. During the simulation, the spheres will also carry a velocity $\mathbf{v} = (v_x, v_y, v_z)^T$.

The model discretization into a set of spheres and their properties has to be provided externally; we have created a basic application to allow the

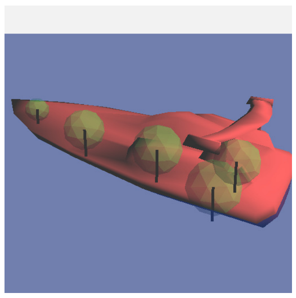


Figure 3.11: Sphere discretization example for a boat model done with our basic editor application. The spheres are positioned and sized within the model, their normal vectors represented by the black short lines.

positioning of an arbitrary number of spheres and their parametrization inside an object. A basic example of this type of discretization is shown in Figure 3.11.

It should be noted that, the more spheres and accurately positioned and sized inside the model, the more precise and physically realistic will be its behaviour. When a model is uniformly discretized with spheres of a radius r lower than the Δx from the fluid simulation, our results are comparable to [CM10], where they use a tessellated mesh from the real model mesh, whose triangles have sides with lengths lower than Δx.

Fluid to Obstacle coupling

There are three major forces that define the movement of an obstacle due to the action of the fluid, as seen in Figure 3.12: buoyancy, drag and lift. The buoyancy force points upward and is proportional to the weight of the displaced fluid. Drag force is a resistive force and is dependent on the actual velocity of the obstacle with regard to the fluid. Finally, lift is a force perpendicular to the oncoming flow direction, it contrasts with the drag force as that one is parallel to the flow direction.

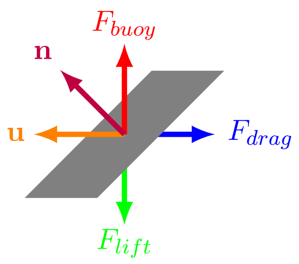


Figure 3.12: Buoyancy, Drag and Lift forces of an object with velocity $\mathbf{u}$ relative to the fluid.

We follow the same strategy as [YHK07, CM10] for the computation of these forces at the position of each sphere of the proxy object. The values for the fluid's level and velocity at different positions are bilinearly interpolated. In the following, we assume that the simulation plane is xz.

The buoyancy force can be defined as $-g\rho V$, where V is the displaced volume of fluid. For sphere i is defined by

$$\mathbf{f}_i^{buoy} = \begin{cases} 0 & \text{if } S_i^p - S_i^r > \eta_p, \\ g\rho V_{sub}\hat{y} & \text{otherwise,} \end{cases} \tag{3.65}$$

where η_p is the water level at the sphere position, S_i^r is the sphere radius, S_i^p is the y coordinate of the location of the sphere and V_{sub} is the volume of the submerged part of the sphere calculated as $V_{sub} = \int_{-S_i^r}^{top} \pi({S_i^r}^2 - x^2)dx$, with $top = (\eta_p - (S_i^p - S_i^r))$. Finally, $\hat{y} = (0, 1, 0)^T$.

The drag and lift forces are computed, as well, as

$$\mathbf{f}_i^{drag} = -\frac{1}{2} C_D A_{2D} \|\mathbf{u}_{rel}\| \mathbf{u}_{rel}, \tag{3.66}$$

$$\mathbf{f}_i^{lift} = -\frac{1}{2} C_L A_{2D} \|\mathbf{u}_{rel}\| \left(\mathbf{u}_{rel} \times \frac{S_i^n \times \mathbf{u}_{rel}}{\|S_i^n \times \mathbf{u}_{rel}\|} \right), \tag{3.67}$$

where C_D and C_L are the drag and lift coefficients, $\mathbf{u}_{rel}$ is the relative velocity of the sphere with respect to the fluid, S_i^n is the normal defined for the sphere and A_{2D} is the area of the circle that cuts the sphere at water level η_p.

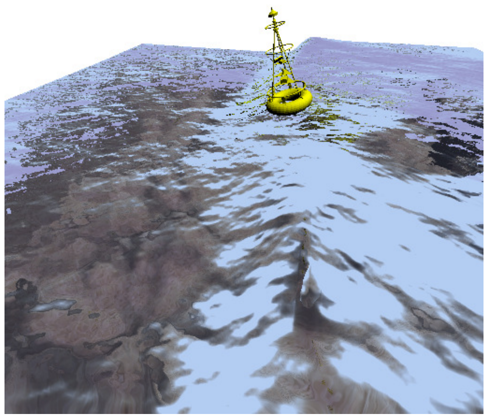


Figure 3.13: A buoy is being dragged by the fluid.

Finally, the three forces are added to the ith sphere. The rigid body simulator will take care of the evolution of the proxy model and will provide the corresponding transform to use in the render phase, as shown by the buoy example of Figure 3.13.

Obstacle to Fluid coupling

In this case, it is the obstacle who modifies the behaviour of the fluid. In order to do so, the fluid not only has to be displaced by the obstacle, but has to be accelerated with the velocity of the obstacle.

The following computations are done per sphere, as previously. To change the fluid correctly, we get the velocity of the obstacle for the ith sphere as $\mathbf{v}$ and the difference between the submerged height of the sphere and the fluid level as *depth*. We compute the following values

$$\begin{aligned} decay &= \exp(-depth), && (3.68)\\ h_o &= decay * C_{dis} * depth, && (3.69)\\ \mathbf{u}_o &= decay * C_{adp} * \mathbf{v}, && (3.70) \end{aligned}$$

where *decay* takes into account the depth the sphere is at and limits accordingly the effect it has over the fluid surface, C_{dis} and C_{adp} are parameters in the range $[0, 1]$ that dampen the effect of the coupling as it can cause instabilities in the LBM if *depth* is great or $\mathbf{v}$ exceeds the maximal wave velocity.

Then, we input the resulting h_o and $\mathbf{u}_o$ from Equations 3.69 and 3.70 into the LBM equilibrium distribution, Equation 3.58, updating the previous *df*s as

$$\begin{aligned} f_0 &= f_0 - h_o,\\ f_i &= f_i + f_i^{eq}(h_o, \mathbf{u}_o) + \frac{f_0^{eq}(h_o, \mathbf{u}_o)}{\mathsf{w}_o},\\ \text{where } \mathsf{w}_o &= \begin{cases} 5 & i = 1..4,\\ 20 & i = 5..8. \end{cases} && (3.71) \end{aligned}$$

The values of w_o are calculated from the original weights w_i from Equation 3.58. With this computation we effectively push the volume of fluid the obstacle displaces to the neighbour cells, taking into account in the process the obstacle velocity. Additionally, to avoid high differences between contiguous points of the fluid mesh, we distribute the h_o and $\mathbf{u}_o$ among the nearest cells using linear interpolation.

For the present examples we have used $C_{dis} = 0.8$ and $C_{adp} = 0.6$. The subcritical flow condition from Section 3.7.3 can also be used here to ensure that no instabilities are introduced from high velocities. Figure 3.14 shows the result of this coupling as some wake waves are generated by a boat.

3.8 Hybrid LBMSW-Particle Model

The model described so far can only simulate the surface of a fluid, it can not detect and simulate high discontinuities on the surface, e.g, breaking waves

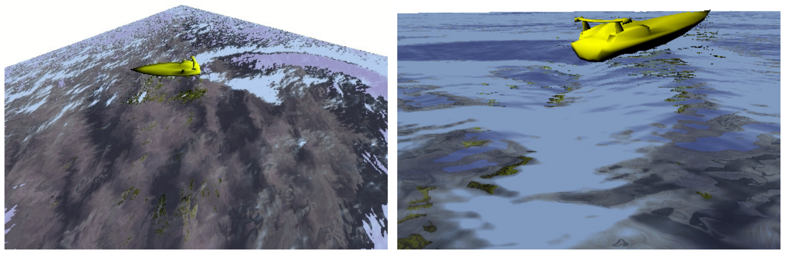


Figure 3.14: The boat responds to the fluid forces and introduces some new fluid waves at its tail as a result of the coupling.

or multiple level simulations. To solve this limitation, we have coupled the LBMSW with a particle system. When a breaking wave is detected, fluid from the shallow waters simulation is turned into particles, which are evolved independently until they fall back to the height field, when they are reintegrated.

Although we have used a ballistic particle system, our implementation, which will be explained thoroughly in Section 4.3, is generic, i.e., any type of particle system can be applied with minimal changes. This makes it more interesting for broad uses, as it can be coupled with more advanced particle systems like, for example, Smoothed Particle Hydrodynamics.

In this same direction, [OH95] already extended the [KM90] model, generating particles from splashes when objects impacted the surface of the fluid. Also, the coupling between mesh-based methods and particle methods provides the benefit of higher level of detail for lower scale phenomena like splashes and spray, as in, e.g., [TFK$^+$03, LTKF08].

However, the conditions for when to generate particles are still delicate. Studies of breaking waves had been performed in 2D by [CKZL99] and [MMS04] presented the treatment of breaking waves for full 3D Volume of Fluid simulations. [TMFSG07] introduced simple parameter-based conditions to generate new geometry patches for breaking waves, which where later improved by [CM10]; although more memory was needed for the new conditions, temporal coherence gave more robustness.

We have applied the detection conditions from [CM10] to the LBMSW algorithm, but the generation and reintegration of particles have been fully adapted with the LBM restrictions in mind. To be consistent, we use the same parametrization as for the adimensional LBM, so Δx and Δt are considered to be equal to 1. It is clear that for render purposes, a redimensionalization must be applied.

3.8.1 Detection

Particles should only be created when the fluid solver, in the present case the LBMSW, can not simulate certain effects, as breaking waves. From the variables used in the fluid simulation we have to be able to detect when these phenomena should occur.

A given cell (i, j) is considered to contain a breaking wave if it satisfies these three conditions:

$$\|\nabla\eta_{i,j}\| > \alpha g, \tag{3.72}$$
$$\eta_{i,j} - \eta_{i,j}^{prev} > \beta, \tag{3.73}$$
$$\nabla^2\eta_{i,j} < \gamma, \tag{3.74}$$

where $\eta_{i,j}^{prev}$ is the fluid height in the previous time step and α, β and γ are parameters, which should be tailored per scene, and more specifically by its scale. Equation 3.72 ensures the wave is steep enough to break. Equation 3.73 requires that the cell is part of the front of the wave and it is raising fast, introducing a comparison with the previous value of height. Finally, Equation 3.74 makes sure particles are only generated near the top of the wave.

As in [CM10], we compute $\nabla\eta_{i,j}$ using the maximum among the one-sided derivatives

$$\nabla\eta_{i,j} = \begin{bmatrix} \frac{max(|\eta_{i+1,j}-\eta_{i,j}|,|\eta_{i,j}-\eta_{i-1,j}|)}{\Delta x} \\ \frac{max(|\eta_{i,j+1}-\eta_{i,j}|,|\eta_{i,j}-\eta_{i,j-1}|)}{\Delta x} \end{bmatrix}. \tag{3.75}$$

Similarly, $\nabla^2\eta_{i,j}$ is computed using central differencing as

$$\nabla^2\eta_{i,j} = \frac{\eta_{i+1,j} + \eta_{i-1,j} + \eta_{i,j+1} + \eta_{i,j-1} - 4\eta_{i,j}}{\Delta x^2}. \tag{3.76}$$

If all three conditions are met, the next step will generate and initialize particles for the given cell. The total volume V_{total} the added particles will subtract from the LBMSW is proportional to $\|\nabla\eta_{i,j}\| - \alpha g$ and can be tweaked introducing a new parameter θ, as

$$V_{total} = \theta(\|\nabla\eta_{i,j}\| - \alpha g), \tag{3.77}$$

here θ just acts as a multiplier and enables a finer control on how much fluid volume will be converted to particles from the excess generated with Equation 3.72.

3.8.2 Generation

For each cell detected in the previous step, a number of particles will be generated for the volume computed in Equation 3.77.

For a particle of radius r, its volume is $V_p = \frac{4}{3}\pi r^3$. We can, however, amplify it by modifying artificially the radius without repercussion to the visual aspect, i.e., the radius of the visual representation may be different from the simulated one. This allows to control the number of active particles; the higher the radius, the less particles needed to supply the computed V_{total}.

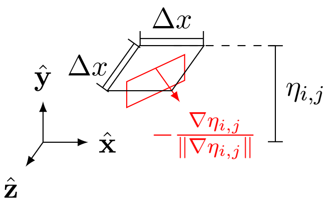


Figure 3.15: Particle placement in the generation step. They are initialized within the red rectangle.

The particles are positioned within a cell-centered rectangle of width equal to the LBMSW cell width ($\Delta x = 1$) and height V_{total} as shown in Figure 3.15. This rectangle is oriented with the opposite direction of the gradient computed in Equation 3.75.

The particle velocities in the xz plane are defined by the wave speed as in [TMFSG07] like $\mathbf{v}_{xz} = \frac{-\nabla\eta_{i,j}\sqrt{gh}}{\|\nabla\eta_{i,j}\|}$. The y component can be defined as a fraction of the height differences from Equation 3.73 as $\lambda_y(\eta_{i,j} - \eta_{i,j}^{prev})$. We have used $\lambda_y = 0.1$ for the present examples with satisfactory results as shown in Figure 3.16.

Furthermore, we lightly perturb the velocity of each particle and jitter their initial positions between $[\frac{-\Delta x}{2}, \frac{\Delta x}{2}]$ in the gradient direction. We also move the particles a random little fraction of a time step in their final velocity direction. These little perturbations add variation and result in less uniform, more chaotic particle movement.

The total volume the particles supply must be subtracted from the LBMSW, as well as the momentum they get. We can easily do this by computing the equilibrium distribution function from Equation 3.58; using as input values V_{total} and the xz velocity components from the particle velocities, prior to the perturbations we apply. These newly computed equilibrium dfs will be subtracted from the cell's original df set as

$$f_i = f_i - f_i^{eq}\left(\frac{V_{total}}{\Delta x^2}, \mathbf{v}_{xz}\right). \tag{3.78}$$

From another point of view, particles are not restricted to be generated only from the detected breaking waves of the previous step. We can generate and initialize particles with other requirements in mind, like a faucet pouring fluid into a basin or a heavy rain column, as demonstrated by Figure 3.17.

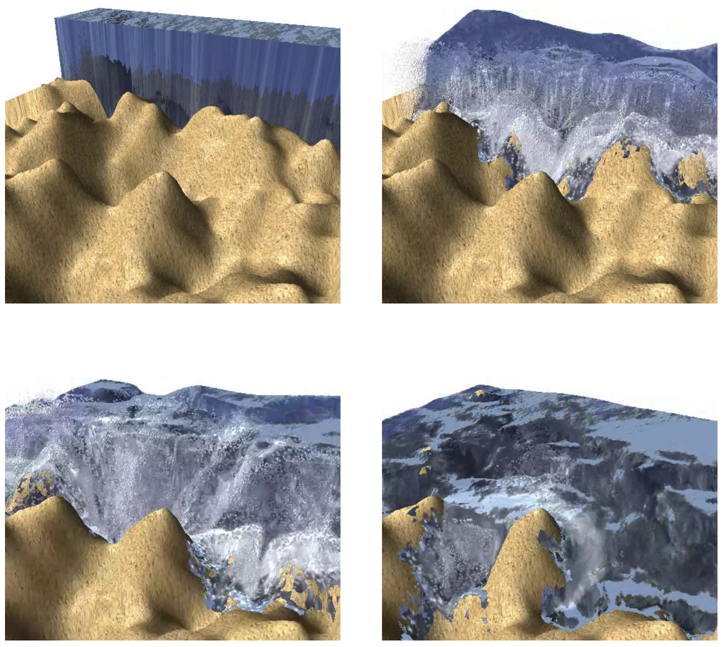


Figure 3.16: Particles generated from the wave of a breaking dam over a dry region.

3.8.3 Reintegration

The final step in the coupling of LBMSW and the particle system is the reintegration of the particles when they hit the surface of the fluid, i.e., $p_y \leq \eta_{i,j}$. The volume the particles carry, as well as their momentum, must be absorbed by the cell they fall on.

As the LBMSW has no explicit method to input vertical velocities, we introduce an interpolation for the absorption of the volume of the particle among the cell's dfs. This interpolation is based on the terminal speed the particle could achieve. Assuming that particles are simulated as spheres as we do, their terminal speed can be defined as

$$v_T = \sqrt{\frac{8rg}{3C_D}}, \tag{3.79}$$

where C_D is the drag coefficient. We normalize the particle's vertical speed with v_T and clamp the result to the range $[0,1]$, as $\chi = min(max(v_y/v_T, 0), 1)$.

Taking into account the previous consideration, we calculate $f_0^{eq\chi}$ as

$$f_0^{eq\chi} = f_0^{eq}\left(\frac{V_p}{\Delta x^2}, \mathbf{v}_{xz}\right), \tag{3.80}$$

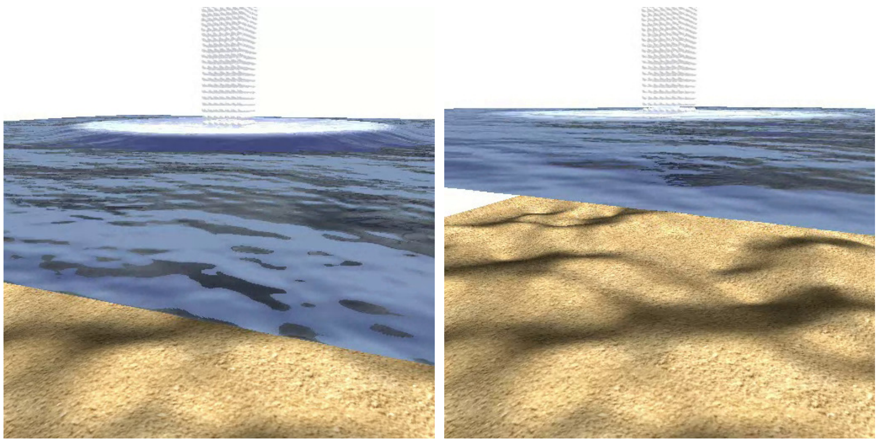


Figure 3.17: Particles generated like a heavy rain column, integrated afterwards to the bulk of the fluid. After a few seconds, the height of the surface of the LBMSW is effectively raised.

and we can finally update the *df*s of the cell using the following computations

$$f_0 = f_0 + (1-\chi)\cdot f_0^{eq\chi}, \tag{3.81}$$

$$f_i = f_i + f_i^{eq}\left(\frac{V_p}{\Delta x^2} + \chi\cdot f_0^{eq\chi}, \mathbf{v}_x z\right). \tag{3.82}$$

Similarly to the obstacle to fluid coupling from Section 3.7.4, using the interpolation with the terminal speed, the added volume is pushed from the cell's center to its neighbours with more energy, the faster the particle drops. Figure 3.17 shows how the water level is effectively raised from the dropped particles.

Chapter 4

CUDA Implementation

The computing scientist's main challenge is not to get confused by the complexities of his own making.

E. W. Dijkstra

Graphical Processor Units (GPU) have seen their computational power dramatically increased in last years. Additionally, they are no more restricted to use a fixed function pipeline; general programmability allows the developer the use of advanced and different techniques for the graphics visualization.

This general programmability has also enabled the use of GPUs as co-processors. Their specialization for high data-parallel functions for, e.g., per-pixel operations, can be leveraged in other situations. Problems which involve high parallel code sections can use GPUs, given a proper mapping of the problem domain to the graphics API and shading language of choice, i.e., the problem must be reformulated to use textures as basic data containers and execute multiple render passes on off-screen framebuffers. This use of the GPUs for general computation is commonly referred as GPGPU.

Many libraries or frameworks have been created to simplify the access to this functionality, like BrookGPU [Lab13], but the predominant ones are Nvidia's CUDA [Nvi13] and the open standard from Khronos Group, OpenCL [Gro13].

Taking into account the locality of the operations of the LBM, we have used Nvidia's solution for the computation of the fluid and OpenGL for the visualization as explained in Chapter 5 to reach real-time fluid simulations.

4.1 CUDA Architecture

CUDA was first released in November 2006, starting with the G80-based family products. It is a general purpose computing architecture that provides the access to the scalability of parallelism inherent to GPUs with minimal extensions to the C programming language, although wrappers for other languages do exist.

Its strengths are based on three pillars: a hierarchy of threads, shared memories and barrier synchronization. They guide the programmer to map the problem in hand into coarse sub-problems suited for the parallel computation in independent order by groups of threads. This decomposition is beneficial as the CUDA programs can be executed by different number of processors cores, scaling automatically to those available by the system.

4.1.1 Programming Model

CUDA functions, called *kernels*, are executed in parallel by a set number of N threads. The number of needed threads is specified in the kernel launch, as well as their configuration in a thread hierarchy.

This hierarchy of threads is provided by CUDA in order to group threads by processor core. Threads are distributed in *block*s, and blocks get distributed in *grid*s. While blocks can be one, two or three-dimensional, grids can only be two-dimensional. Their size, be them blocks or grids, is limited by the hardware used. On the GPU we have used, a GTX280, the size of a thread block is limited to 512 threads, as all threads are expected to be resident on the same processor core and must share a limited memory and register space on that core. Figure 4.1 shows an example of how threads are distributed within this hierarchy.

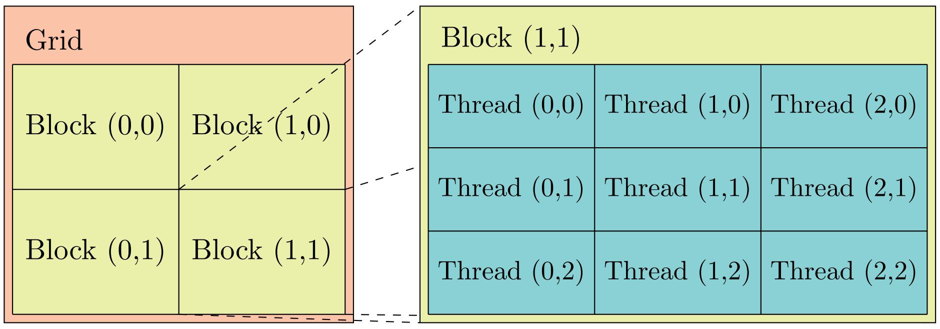


Figure 4.1: Grid of Blocks of Threads.

Thread blocks are required to be executed independently, without any special order. This requirement allows to schedule them across any number of available cores. Threads within these blocks can use some limited *shared memory* and synchronize their execution to coordinate memory accesses through an intrinsic function that acts as a barrier. A fundamental key to optimize execution is to maximize occupancy; the configuration of blocks and grids should be done depending on the per-core resources a kernel needs, as the registers and the shared memory are limited.

The memory the threads can access is also distributed in various spaces. Each thread has private local memory, represented as registers (if available). There is, also, a limited amount of shared memory per core that all threads in a block can access; it is meant to be low-latency memory used for local operations between threads in the block. The bulk of the memory, the global memory, can be accessed by all threads and is persistent across kernel calls by the same application.

Additionally, there are two more read-only memory spaces accessible by all threads: constant and texture memory spaces. Texture memory provides different addressing modes and data filtering. These spaces are also persistent, as the global memory.

How the memory is accessed is another important feature to keep in mind when programming with CUDA. Each space has its own restrictions and best practices, e.g., for global memory the common ground is to use data types with appropriate size and alignments, as well as make the threads access words in sequence to increase coalescing. These restrictions are also dependent on the hardware used, classified by their Compute Capability. [Nvi11] gives the detailed description of these limitations and how to maximize performance.

4.1.2 Hardware Implementation

From a hardware standpoint, thread blocks are enumerated and distributed to available multiprocessors (or processor cores) when the kernels are executed. The threads within a block are executed concurrently on one multiprocessor, and new blocks are launched when older ones terminate execution.

To manage that kind of simultaneous execution, multiprocessors are designed to use the *SIMT* architecture; Single-Instruction, Multiple Thread. A multiprocessor manages the threads from its assigned block and schedules them for execution in groups of 32 parallel threads, called *warps*. Threads composing a warp start simultaneously at the same program address and execute one common instruction at a time. They, however, maintain their own instruction address counter and register space so they can branch freely. This branching behaviour comes at a cost; if some of the threads branch on a data-dependent basis, the warp serially executes each branch path, thus, full efficiency is only achieved when all threads in a warp execute the same set of instructions, i.e., there is no divergence in the code execution. This branch divergence is another key point to take into account when designing algorithms for CUDA.

4.2 LBM in CUDA

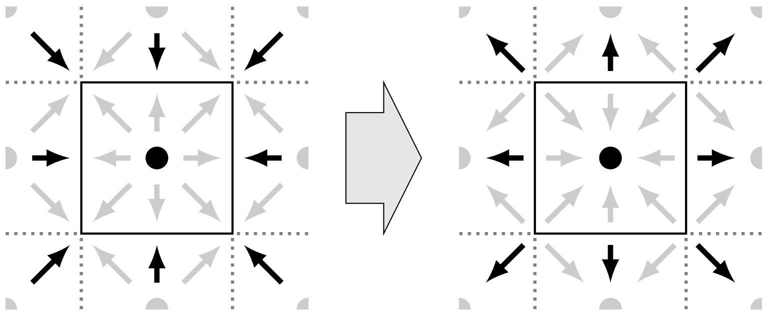
In the Lattice Boltzmann Method, only the streaming step needs neighbour data; the collision step is completely local. And even the streaming step can be formulated with a pull strategy (instead of a push one) to minimize data dependency; the cells gather the information needed from their neighbours instead of pushing the data to them in order to make it available. These features render the LBM a highly parallelizable algorithm.

Indeed, it has already been implemented in parallel architectures, e.g., [Thü07, Poh08, GRGT10], and even in GPUs like in [LWK03, WLMK04]. It has also been ported to CUDA, where it gets quite a reduced computational time in comparison to CPU implementations; [OS09, Tö10, OKTR11] give details about how it can adapted to CUDA. In contrast to [OS09], where a block size equal to the width of the simulation was used, we use here a fixed block size of 128 threads to improve occupancy, independently of the simulation domain size. To achieve maximum performance, however, this block size should vary from kernel to kernel depending on their resource demands (basically register and shared memory usage), as well as the GPU at hand.

LBM is a memory consuming algorithm; for example, the D3Q19 model needs 19 floating point values per cell only for the *df*s. In order to reduce the memory used by the algorithm, techniques like grid compression can be used as in [PKW$^+$03], although it can not be easily adapted to CUDA for real-time simulations. In order to reduce memory consumption in a GPU environment, [BMW$^+$09] proposed an alternative implementation of the streaming and collision steps with two different kernel functions, reducing the required memory to nearly a half. They refer to this memory layout as an A-A memory access pattern; only one set of *df*s is needed but how they are accessed depends on the parity of the iteration and, thus, the kernel used. Figure 4.2 shows how the two kernels access memory.

The most common approach that needs two sets of *df*s is referred as an A-B memory access pattern, shown in Figure 4.3. Only one kernel is needed for the full simulation of the LBM and for a given iteration it reads values from one of the *df* sets and writes to the other. The two sets are swapped after the kernel is done in a ping-pong setting. We have favored this methodology in the present work, as the additional tasks done per iteration, both in the LBM3D-VOF and the LBMSW, would have imposed us to double the number of kernels to accommodate to the varying directions of the *df*s in an A-A memory setting. The use of the A-A memory access pattern in those cases would have also introduced race conditions as in the non-deterministic LBM3D-VOF implementation of [Sch10].

Kernel AA:1 Stream - Collision - Stream



Kernel AA:2 Collision

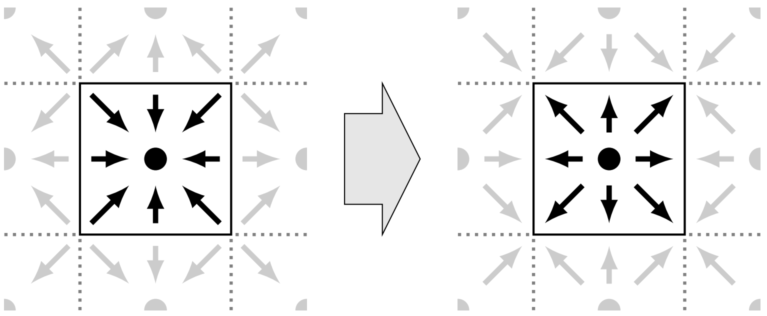


Figure 4.2: A-A memory access pattern from [BMW+09]. After each collision step, the direction of the *df*s is switched in place, taken into account in the successive streaming steps. Only black *df*s are accessed by the thread of the present cell.

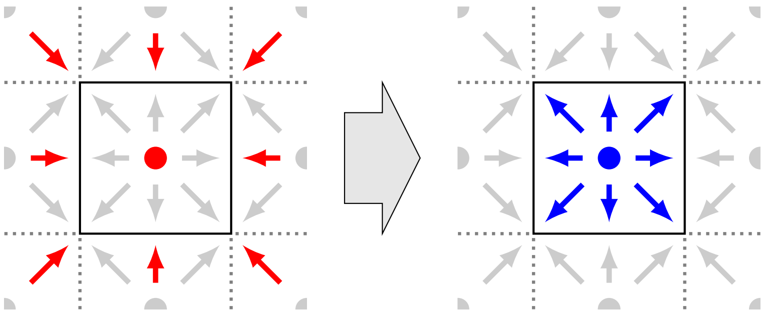


Figure 4.3: A-B memory access pattern. For each iteration, the thread reads from the red *df* set and writes to the blue one. The two *df* sets are swapped after the streaming and collision steps are done.

4.2.1 LBM3D-VOF

As we have seen, there are already implementations for the basic LBM algorithm in CUDA that could be used. However, problems arise when extending this basic algorithm implementation, as certain properties must be retained like the mass conservation in a free surface simulation. These problems take the form of race conditions resulting in, e.g., mass loss, and reduced performance caused by non-coalesced accesses to memory or divergent code paths in a warp.

The implementation of the full 3D free-surface LBM described in Section 3.6.2 requires a set of variables described in Table 4.1. A high level algorithm is shown in Algorithm 4.1.

density functions *df*	2 x 19 floats
macroscopic velocity	3 floats
macroscopic density	1 float
mass	1 float
fluid fraction φ	1 float
surface normal	3 floats
cell flag	1 char

Table 4.1: Values needed per cell for the LBM3D-VOF implementation. The cell flag variable is used as a bit field to describe the type of the cell (Fluid, Interface, Empty, etc.).

From the LBM parametrization, the time step of the simulation is quite smaller compared with that of the frame render time; as much iterations of the LBM algorithm should be done as needed to fill the frame time.

As CUDA can not ensure any specific order of execution of thread blocks and it is clear they are not executed instantaneously for the whole domain, we have to serialize the cell type changes of emptied or filled cells to ensure no race conditions can affect the final result, i.e., we achieve deterministic results. In order to do so, we introduce a new temporal flag type *tobeChanged* and change the filled and emptied cells conservatively, that is, emptied cells that may be needed in the next iteration because they are in the neighbourhood of fluid cells are not converted, they remain as Interface. Additionally, kernels targeted to a concrete set of cells provide an early exit condition for those not to be changed.

The LBM basic simulation is executed in the *LBM_stream_collision* kernel. It also provides the mass exchange tracking of Equation 3.42 as well as the reconstruction of the needed incoming *df*s of Equation 3.44.

The next kernel, *preflag*, checks the condition from Equation 3.47 for Interface cells and adds, if needed, the tobeChanged flag, as well as the type they should convert to. Therefore, if the cell has emptied it will be flagged as `Interface|tobeChanged|Empty`, and if the cell has filled it will be flagged

Algorithm 4.1 High-level LBM-VOF algorithm.

```
$\Delta t$ = frame time step (16ms)
$\Delta t'$ = LBM dimensional time step
foreach(frame) {
  foreach($\Delta t$) {
    //CUDA kernels
    LBM_stream_collision();
    preflag();
    NH_filled();
    NH_filled_MassDist();
    updMassFrac();
    NH_emptied();
    NH_emptied_MassDist();
    updMassFrac();
    //
    swap_DFs();
  }
  //Render
}
```

as `Interface|tobeChanged|Fluid`.

First, we deal with the filled cells. *NH_filled* updates the neighbourhood of the filled cells; every cell being Empty or Interface to be emptied is re-flagged as `tobeChanged|Interface`. Then, *NH_filled_MassDist* distributes the excess of mass as in Equation 3.48 and re-flags the current filled cell as Fluid. Finally, the *updMassFrac* initializes correctly the *df*s and updates the mass and fluid fraction of the new cells converted by *NH_filled* and flags them as Interface.

Afterwards, the same process is done for emptied cells. The main difference resides in that *NH_emptied* updates the neighbourhood in an opposite way: only Fluid cells are re-flagged to `tobeChanged|Interface`; there are no remaining Interface cells to be filled. *NH_emptied_MassDist* also distributes mass excess, but in this case is a negative one.

After all steps are done, *swap_DFs* just swaps the pointers to memory for the *df* sets, changing their role in the next iteration according the A-B access pattern: the set which we have just written to will become the one which we will read from, and vice versa.

Finally, after the needed iterations are done, the scene should be rendered with appropriate techniques, as those used in this thesis, explained in Section 5.1.

Although we get deterministic results and the race condition problems are solved by serializing the filled/emptied cell conversion, there are other

Algorithm 4.2 High-level LBMSW algorithm.

```
$\Delta t$ = frame time step (16ms)
$\Delta t'$ = LBM dimensional time step
foreach(frame) {
  //CPU
  ObstacleSimulation();
  ObstacleFluidCoupling();
  //CUDA
  for(i=0; i<$\Delta t$; i+=$\Delta t'$) {
    LBM_stream_collision();
    LBM_applyForce();
    upd_CellTags_pre();
    upd_CellTags_Fluid();
    upd_CellTags_Empty();

    swap_DFs();
  }
  //Render
}
```

problems which we haven't been able to solve. The LBM is a memory intensive algorithm, which impacts heavily in the performance, although we have enforced the memory accesses are coalesced. Additionally, a lot of these accesses are data dependent, e.g., the access to the next cell *df*s' depends on the type of the cell, causing further problems: there may be code divergence which forces the processor to serialize the execution of the threads for every code branch.

[Thü07] also described the rare possibility of single Interface cells that are left behind the free-surface layer; cells that could not be filled or emptied and remain as artifacts. Albeit we have not encountered this issue in our tests, we have added a condition in the *LBM_stream_collision* kernel for Interface cells to take into account this problem. If such a cell is found we treat it as if it was filled or emptied and proceed to change it accordingly; the mass loss or increment should be unnoticeable for real-time simulations as those cells should already be close to be filled or emptied.

4.2.2 LBMSW

Like the previous section, we present here the implementation of the LBMSW algorithm from Section 3.7, summarized in Algorithm 4.2. We use the same A-B memory access pattern, as well.

In this case, as there is not a free boundary to track and the simulation is only two-dimensional, the number of needed variables is more reduced as shown in Table 4.2. We can set an additional scalar field with the same domain size as the simulation to account for the underlying bed elevation z_b, although as it is constant, it will be stored as a texture in CUDA memory.

density functions *df*	2 x 9 floats
macroscopic velocity	2 floats
macroscopic density	1 float
cell flag	1 char

Table 4.2: Values needed per cell for the LBMSW implementation. The cell flag variable is used as a bit field to describe the type of the cell (Fluid, Empty, etc.).

In each frame iteration, apart from the fluid itself, we have to simulate also the coupled rigid bodies as explained in Section 3.7.4. The simulation of the rigid bodies can be done with any standard package, as the Bullet Physics Library [Sim13] which we have used here. The dynamic objects are set up prior to the simulation with the proxy model previously explained and their physical simulation is done entirely by the physics library. In our case, this step is done in CPU inside the *ObstacleSimulation* function. The coupling with the LBM is done in the function *ObstacleFluidCoupling*; the needed CUDA memory is mapped to the CPU memory space and modified accordingly. Data must inevitably travel through the bus from CPU to GPU space, being one of the current bottlenecks.

The stream and collision steps of the LBM are done, as before, in the *LBM_stream_collision* kernel. But in this case, only the basic LBM algorithm is executed, as no mass needs to be tracked or other operations have to be performed at this stage.

In order to apply the force computations from Equation 3.59, the gradient from the fluid height field must be computed, which in turn needs for all the cells to have their height defined at time t. This forces us to serialize this force computation in a separate kernel named *LBM_applyForce*.

Afterwards, the *upd_CellTags_** kernels are responsible of the dry-wet region tracking and according cell conversion. Similarly to the VOF cell conversion, we have to serialize the implementation in different kernels and do this re-flag operation conservatively; we try to maintain the Fluid cells, being them marked to be emptied or not, if they are likely to be needed again in the next iteration. First, *upd_CellTags_pre* checks the height of the cells against the threshold and pre-flags them with an additional type if necessary: for Fluid cells above the threshold, their neighbourhood is marked as `tobeFluid`, and the Fluid cells below the threshold are marked as `tobeEmpty`. Next, *upd_CellTags_Fluid* re-flags the marked `tobeFluid` cells to the final Fluid cell type. Lastly, *upd_CellTags_Empty* does the same but with a difference: the `tobeEmpty` cells are re-flagged to Empty but their excess of fluid, the threshold remaining, is distributed among the Fluid neighbours taking into account the gradient of the underlying terrain as in Equation 3.63.

Finally, the *df* sets are swapped with *swap_DFs* as in the previous section.

In order to render the scene with the common reflection and refraction effects one would expect from a fluid, the normals of the surface should be computed. They can easily be calculated using finite differences on η, being $\eta = h + z_b$. Additionally, as the render is done by OpenGL, the fluid heights have to be available; this is done by having them mapped as a *pixel buffer object* (PBO) used in the surface mesh rendering.

Similar problems to the LBM3D-VOF can be found in this implementation as those for divergent code, at least when not the whole domain is Fluid, i.e., there are dry regions to track, which require data dependent code execution and memory accesses. As the LBMSW is a two-dimensional simulation, the memory requirements are lower, only 9 floats per cell for the *df*s, but that is still higher than other more traditional methods as in, e.g., [CM10].

4.3 Hybrid Particle-LBMSW

To implement the hybrid particle-LBMSW algorithm from Section 3.8, we have to extend the LBMSW CUDA implementation, as can be seen in Algorithm 4.3 and modeled by the class diagram of Figure 4.4. There are basically two added code blocks: particle reintegration before the main LBMSW code and particle detection and generation afterwards. Particle reintegration could, instead, be done as the last thing in the loop, but moving it to the front allows us to render all newly generated particles. So we will explain below the algorithm from detection to reintegration, following the original supposed order in the particle simulation. The sort_*, remove_* and prefix_sum_* functions are provided by the Thrust library [HB13] and executed entirely in the GPU.

We have used a simple ballistic particle system, where position and velocity are evolved in time without further constraints, i.e., there is no inter-particle collision detection; however, other particle systems as a full CUDA SPH implementation like [GSSP10] could be applied with minimal changes. As there is no simple way to maintain a dynamic data structure for the particles, i.e., particles should be created and destroyed on the fly; we have resorted to a fixed number of particles from the beginning of the simulation. In addition to the usual particle properties as position and velocity, we add two more: a TTL (time-to-live) value and an active (ACTIVE/INACTIVE) flag. The TTL variable allows us to track particles depending on the time

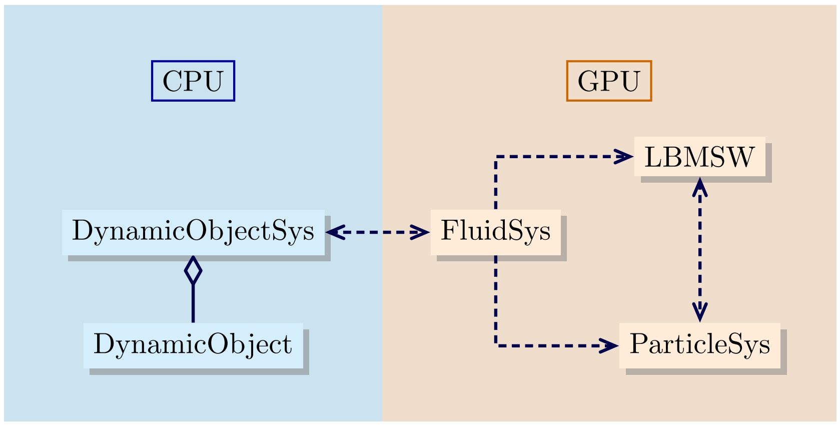


Figure 4.4: Class diagram of the modules of the simulation. DynamicObj represents the Proxy objects as defined in Section 3.7.4 which are managed by the DynamicObjSys (using the Bullet library). LBMSW and ParticleSys are the main modules that carry the fluid simulation behind the FluidSys one, which encapsulates them. FluidSys is, then, the face between the CUDA operations of LBMSW and ParticleSys and the CPU computations of the coupled dynamic objects represented by DynamicObjSys. In practice, as LBMSW and ParticleSys are tightly coupled in code (by the generation and reintegration phases) they could be both assimilated by FluidSys directly.

they have been alive in the simulation, prioritizing those that have existed longer (lesser TTL) to be reused. The active flag, on the other hand, is used to control the particles that can take mass from the fluid surface in their generation step; only INACTIVE particles can, ACTIVE particles won't, e.g., dead particles (TTL has become 0) but not reintegrated remain ACTIVE to not subtract more fluid next time they are initialized.

After the LBMSW core simulation, we compute the gradient and laplacian of the surface height field in *computeLBM_GradLaplacian* as in Equations 3.75 and 3.76. The values are stored for future use in the detection step.

Prior to the particle simulation per se, the particles are sorted by their TTL in ascending order. The particle simulation is advanced in *stepParticles*. For a ballistic particle system, the velocity and position of the particles are updated. The particles' TTLs are also updated, subtracting the current Δt. Their status is also updated to ACTIVE. If a particle has died ($TTL \leq 0$) before being reintegrated, we let them be ACTIVE but out of view. We can use this same behaviour for particles exiting the LBMSW domain if it was integrated in a much larger fluid surface; only the LBMSW is simulated and the rest is just a planar mesh. This ensures we don't lose mass because of dead particles not reintegrated in time. This step should

Algorithm 4.3 High-level Hybrid Particle-LBMSW algorithm.

```
$\Delta t$ = frame time step (16ms)
$\Delta t'$ = LBM dimensional time step
foreach(frame) {
  //CPU
  ObstacleSimulation();
  ObstacleFluidCoupling();
  //CUDA
  ReintegrateParts_S1();
  sort_tuples();
  remove_nonValidTuples();
  prefix_sum_tuples();
  ReintegrateParts_S2();

  for(i=0; i<$\Delta t$; i+=$\Delta t'$) {
    LBM_stream_collision();
    LBM_applyForce();
    upd_CellTags_pre();
    upd_CellTags_Fluid();
    upd_CellTags_Empty();

    swap_DFs();
  }
  computeLBM_GradLaplacian();

  sort_particlesByTTL();
  stepParticles();
  detectBreakingWaveCells();
  prefix_sum_NeededPartsPerCell();
  initParticles();

  //Render
}
```

also be the one being replaced if a different particle system was used, only the TTL and active flag should be maintained as in this case.

Afterwards, the possible breaking waves are detected in *detectBreaking-WaveCells* by applying the conditions of Equations 3.72 to 3.74 using the previously stored surface gradient and laplacian. Each cell will output the needed particle count that it needs. Then, with a prefix sum operation we can obtain an accumulated sum of the needed particles. We can use the result of this accumulated sum as the index at the particle array for which each cell will take their needed particle count. As particles have been sorted by TTL, we ensure the particles first taken in this step are those who had a lower TTL. Unfortunately, if more particles are needed than those with a $TTL = 0$, the next with lower TTL will be taken. With a bad parametrization this can lead to artifacts, as disappearing particles from frame to frame as they are needed. *initParticles* will, then, initialize the particles needed for each cell as explained in Section 3.8.2, marking them as ACTIVE2. As the reintegration only checks for ACTIVE particles, and the active flag is always updated to ACTIVE in the *stepParticles* kernel, it is ensured that they are alive at least for a frame. Their TTL is also set up as the maximum allowed time to live for a particle, which is a user-defined parameter. For particles that were previously marked as ACTIVE, i.e., those who were dead before reintegration, no fluid will be subtracted from the LBMSW, ensuring no mass loss; thus, only INACTIVE particles will take fluid from the LBMSW. Needless to say, these steps for the detection and initialization of particles can be changed or improved to take into account more situations as those described in [CM10].

Finally, for ACTIVE particles that have fallen again in the fluid, that is, their y coordinate is lower than the fluid height at the xz position, the reintegration should be as easy as the explanation from Section 3.8.3 but, as we can not be sure how many particles can fall in a cell at once, we should use atomic operations in the update of the cell's dfs. As our hardware, a GTX280, does not support these operations for float variables, we had to solve it from another perspective: *ReintegrateParts_S1* relates which particles have fallen in which cells and how many there are for each cell; from the cell point of view, *ReintegrateParts_S2* will gather the fallen particles and update the local dfs. In order to do so, for each particle, *S1* will write a tuple associating the cell id (the cell's position in a linear memory array) with the particle id, as well as the particle count for each cell (using integer atomics). Particles not to be reintegrated are associated to a fake cell, in this case we use the cell 0 that we ensure is a Boundary cell for all examples. Sorting the tuples by the cell id, removing those with the fake cell id and doing a prefix sum on the particle-in-cell count will lead us to the cells having the index where their particle count starts in the tuple array. *S2* will, for each cell, take their counted fallen particles and reintegrate them, marking them as INACTIVE with $TTL = 0$.

In order to reduce the number of arrays needed we have grouped the particle data in two float4 arrays: particle position and TTL for the first array, particle velocity and active flag for the second array. The reason to do that is because they are used also by OpenGL in the render phase. Then, the access to memory for each particle implies always the retrieval of 4 floats at once, being the size of each float 4 bytes, that is 16 bytes for each access: the maximum word size in CUDA memory transactions. This fact hits performance, even more in the sort operation where multiple accesses are done[1]. A possible solution for this problem could be the separation of the float4 arrays in multiple simple float arrays, so the accesses would also require only the minimum word size for the memory transaction; this would involve the addition of an id array used in the sorting operation and later required for further accesses in the particle data arrays, apart from complicating the data sharing with OpenGL. We instead have opted for the use of the original float4 arrays plus the id array used in the sorting operation. The id array allows for a faster sorting (smaller memory transactions), and while we lose memory coalescing because particle data is accessed by an indirection with this id array, the total time is reduced in comparison to the float4 array solution by itself because of that lower time used in the sort operation.

One key thing that penalizes the implementation is the hardware itself. By not supporting float atomic operations, we had to implement the reintegration of the particles in a multi-step fashion. The two additional kernels plus the needed sorting of the tuples, removing and prefix sum operations increase heavily the time needed for frame. Newer hardware will allow to solve this problem, aside from increase overall performance due to a greater number of processor cores and lower latency memory.

[1]In Section 6.3 we will show some results corroborating this statement.

Chapter 5

Fluid visualization

If real is what you can feel,
smell, taste and see, then 'real'
is simply electrical signals
interpreted by your brain.

Morpheus
The Matrix

So far we have presented how the fluids are simulated and implemented in CUDA. In this chapter, we will introduce how they are visualized.

For each of the models presented in this thesis, LBM3D-VOF and Particle-LBMSW, we need two different rendering algorithms due to their different nature. First, we will explain how the 3D fluid simulation is visualized through volume raycasting, which allows for an easy integration of effects such as reflection and refraction from an environment map, also included in our implementation. Next, we will show the render process for the hybrid particle-surface model, breaking it into the different aspects that contribute to the final image; apart from the much needed reflection and refraction solved with screen space techniques, this includes caustics, the inclusion of lower scale detail through FFT fluid simulation, as well as the particle rendering and the advection of surface foam.

Although we have used OpenGL plus its shading language, GLSL, in their 3.3 version in core profile, the explanation will remain at a higher level for an API independent understanding.

5.1 Raycasting the LBM3D-VOF volume

The full 3D simulation of the LBM-VOF algorithm results in a volumetric model of the fluid fractions φ stored in a 3D floating-point texture.

The common slicing techniques first depicted by [HS89] with more modern implementations as, e.g., [IKLH04], are not advantageous in this case:

we require only the first[1] intersection at a defined isovalue but they accumulate the different slices in a back to front blending operation. Moreover, the slices are created (or updated) dynamically as view-aligned planes.

As we are talking about the intersection at a defined isovalue, the isosurface, we could have used a mesh extracting algorithm as, e.g., *marching cubes* [LC87, NY06], and use the final mesh with the typical rasterization of the GPU. We discarded this kind of method because it required the dynamic generation of geometry, although possible with actual GPUs, but which can cause popping artifacts if there are no additional smoothing operations done.

Alternatively, due to GPU programmability, we can raycast the volume [RGW$^+$03, KW03], allowing us to find the required isovalue intersections. Our implementation, summarized in Figure 5.1, requires multiple render passes and is described for the first intersection as follows:

1. Render a proxy cube model, only back faces. This proxy cube represents where the 3D volume is located. For each fragment, the output values will be the distance of each fragment to the camera. These values can be stored in a single RGBA texture.

2. Render the front faces of the proxy cube model to the same previous framebuffer using a subtraction operation. In this case the distance to the camera of each fragment is needed but also the texture coordinates in the 3D texture. The subtraction operation results in the maximum distance each ray will have to traverse.

3. Render a screen-sized quadrilateral textured with the result of the previous step. With the texture coordinates and the maximum distance for the ray we can proceed to the actual raycasting in search of the intersection ray-volume. Using a combination of linear and binary search as in [POC05, ABB$^+$07] we can first get closer to the intersection and refine its location afterwards. Other techniques as the Secant method [RSP07] or the Newton-Raphson method [SKP07] could be used instead to achieve the same result. This step outputs a depth map of the fluid and for fragments out of the 3D volume, the depth will be set to the maximum value.

4. (Optionally) We can smooth the depth map for posterior usage applying a Gaussian blur. This step would account for the sharp jumps between the isovalues at contiguous voxels. In our case, we have implemented this filter as a separable binomial filter which requires two passes; one for the horizontal direction and one for the vertical. To avoid blurring on edges and keep them sharp we detect high jumps in

[1]Not only the first intersection should be required for a correct refraction implementation, but we will restrict ourselves to this case for simplicity.

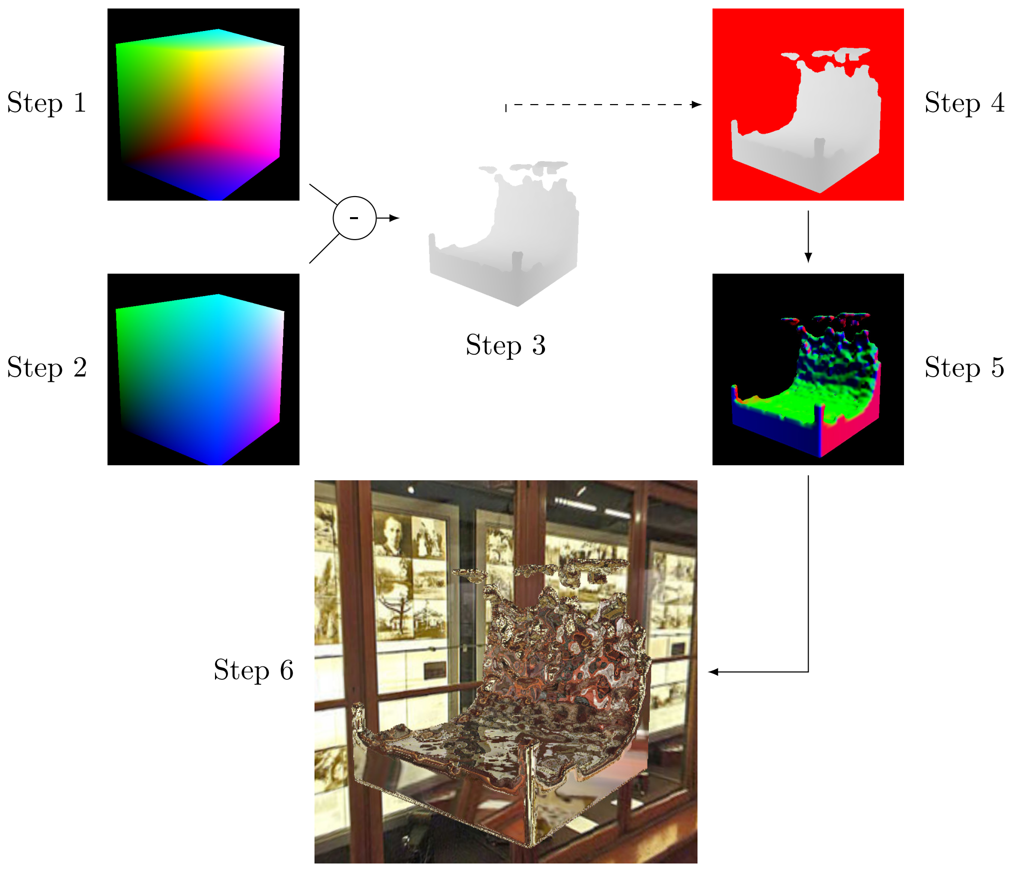


Figure 5.1: Multiple render passes of our raycasting implementation. Steps 1 and 2 render a proxy model and use a subtraction operation on the result. Step 3 raycasts the 3D texture from the previous result. Step 4 is a smoothing step and is optional (denoted by a dashed line). In Step 5, normals are computed in screen-space. Finally, the result is composed with the rest of the scene in Step 6.

the depth map and restrict the application of the filter in those cases.

5. From the depth map obtained previously, we compute screen-space normals at the surface of the fluid.

6. With depth and normals, the final scene is composed with the fluid. Additional effects as reflection and refraction are applied using Schlick's approximation [Sch94] to Fresnel terms with environment maps, where the view direction is modified (reflected/refracted along the fluid's surface normal) and used to look up at the environment map used.

The results of this volume raycasting implementation can be seen in Figures 5.2 and 6.4.

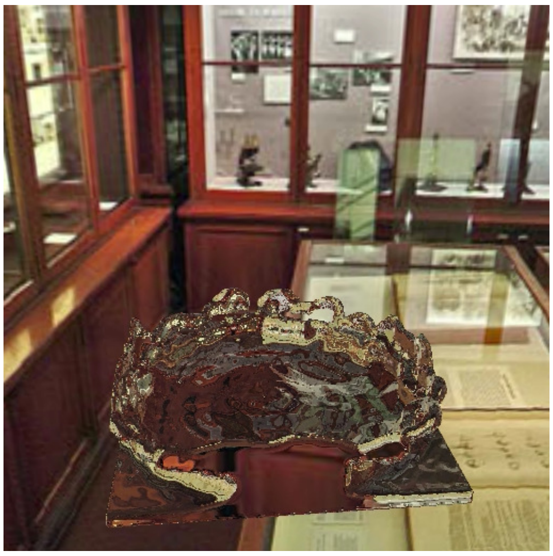


Figure 5.2: A drop has fallen into an existing body of water creating a splash. 64^3 grid resolution used.

As the volume is dynamic and we should maintain interactive rates, we could not add typical optimizations of raycasting techniques as those described in, e.g., [Lev90], but as the resolution of the fluid is quite lower than the one of the screen, we could improve the performance of this algorithm in a similar fashion to the checks for aliased edges of [Lot11]: the fluid is rendered with a lower resolution up to Step 4 and applying a Sobel filter (implemented as a separable filter) we find the edges of the fluid; these edges will be rendered again at full viewport resolution to account for the alias, and the final composition remains the same.

5.2 Rendering of the LBMSW-Particle model

In the surface-only simulation, whether only the LBMSW implementation is used or the full hybrid method with particles, we render the fluid surface as a triangle mesh where the height of the vertices is obtained from the simulation. The particles are then rendered separately.

Here we will present, apart form the particle rendering per se, some additional effects expected to be found in a fluid simulation which improve the appeal of the overall simulation as caustics, reflection and refraction or the advection of surface foam.

In some cases we will be referring to raycasting; in these situations we only mean the raymarching, the combination of linear and binary search through a texture, namely a depth map, to find some final position.

5.2.1 Lower scale detail

Although the LBMSW already provides appropriate normals (using finite differences over the resulting height field), the resolution of the simulation lattice imposes a limit on the detail that can be achieved.

To improve the situation, we can use some additional technique to increase the detail although it won't be physically correct. For example, [CM10] use a normal mapped texture generated from the FFT ocean simulation from [Tes01] and advect it with the velocity from the fluid simulation as in [MB96]. In the same spirit of increasing the detail through normal mapping, we have tried two alternatives: a gradient noise solution and the full FFT ocean simulation. In both cases, a normal texture is obtained from finite differences from the height fields obtained.

Gradient noise

Perlin noise [Per85, Per02] is probably the most famous noise algorithm. It is an implementation of gradient noise, which consists in the creation of a lattice of random gradients at integer locations, interpolated to obtain values in between the lattices.

The gradients are precomputed in a table G and accessed with a hashed value from the space location $\mathbf{x} = (i, j, k)$ applying successively a pseudo-random permutation, stored also in a table P, with the different coordinates combined. In CUDA, in order to reduce the number of accesses to memory, we have changed how the gradient is computed: instead of having only one permutation table and make multiple accesses to it with accumulated values from the coordinates as $grad = G[P[P[P[i] + j] + k]]$, we have created three different pseudo-random permutation tables (one for each dimension) and combine them with a XOR operation as $grad = G[P_x[i] \oplus P_y[j] \oplus P_z[k]]$. Except this unique change, the algorithm is the same as the last version of Perlin noise [Per02], that is, the *noise* function computes the gradients *grad* at the integer positions in space that surround the looked up point $\mathbf{x}$ (as the vertices of a surrounding cube) and interpolates them properly.

Although we only need a 2D texture (we generate a normal texture from a heightfield), we use the 3D version of the noise function. If we fix 2 of the coordinates and move through the other one, as if a plane was slicing a 3D volume, we get the illusion of animation, which is what we are aiming for with this technique: to increase lower scale detail.

Now, it is not clear how we have to tune the noise function to obtain a water-like noise; for example, applying a periodical function as *cos* creates banding (stripes) while applying the absolute value function *abs* makes sharp edges (discontinuities). We have used a fractal sum, which is usually employed to make cloud textures, and is defined as

$$N_f = \sum_{i=1}^{octaves} \frac{1}{i} noise(i\mathbf{x}), \tag{5.1}$$

where *octaves* is the number of layers of noise combined in the operation at different frequencies.

Furthermore, to be able to seamlessly repeat the texture, we wrap the coordinates with a module operation, although this repetition is easily visible afterwards, as shown in Figure 5.3.

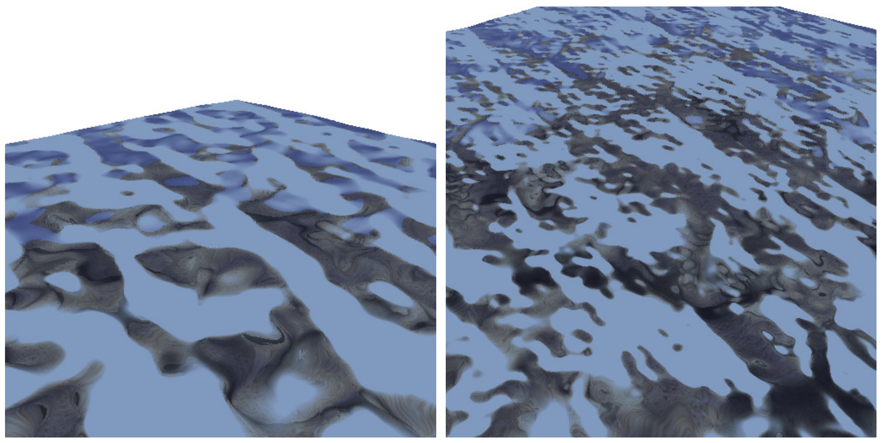


Figure 5.3: Two octaves of gradient noise on the left, four octaves of gradient noise on the right; applied to the LBMSW surface with normal mapping.

This solution has some problems: it is not clear which type of operation over the noise function should give the adequate look and feel to the fluid and the pattern repetitions are quite obvious. The resulting animation of moving through one of the axes isn't, neither, quite realistic.

FFT ocean simulation

To solve the problems from the previous approach we can use the simulation of ocean water which is based on statistical methods [Tes01].

This technique is based in the computation of the Fourier amplitudes of a wave field. These amplitudes can be produced as

$$\tilde{h}_0(k) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})}, \tag{5.2}$$

where ξ_r and ξ_i are independent draws from a random number generator. $P_h(\mathbf{k})$ is the wave spectrum and several empirical models can be used. A known model for wind-driven waves is the *Phillips* spectrum

$$P_h(\mathbf{k}) = A\frac{\exp\frac{-1}{(kL)^2}}{k^4}|\hat{\mathbf{k}} \cdot \hat{w}|^2, \tag{5.3}$$

where L is the largest possible wave, $\hat{w}$ is the direction of the wind and A is a numeric constant.

Given a dispersion relation $\omega(k) = \sqrt{gk}$ for deep waters, where g is the gravitational constant and k is the magnitude of the wavevector, which points in the direction of travel of the wave and is related to the length of the wave; the Fourier amplitudes of the wave field at time t are

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) \exp\left(i\omega(k)t\right) + \tilde{h}_0^*(-\mathbf{k}) \exp\left(-i\omega(k)t\right), \tag{5.4}$$

where $\tilde{h}^*(\mathbf{k}, t) = \tilde{h}(-\mathbf{k}, t)$.

Finally, at discrete points $\mathbf{x} = (nL_x/N, mL_y/M)$, the wave height of the fluid $h(\mathbf{x}, t)$ is obtained from the inverse FFT to get back to spatial domain

$$h(\mathbf{x}, t) = \sum_k \tilde{h}(\mathbf{k}, t) \exp\left(i\mathbf{k} \cdot \mathbf{x}\right), \tag{5.5}$$

where t is the time and $\mathbf{k}$ is the wave vector, a two dimensional vector with components $k_x = 2\pi n/L_x$ and $k_y = 2\pi m/L_y$, with n and m are integers in the ranges $[-N/2, N/2]$ and $[-M/2, M/2]$, respectively.

With this newly generated height field we compute a normal texture using finite differences and apply it to the LBMSW triangle mesh using normal mapping as shown in Figure 5.4. Instead of advecting the texture with the fluid as [CM10] do with the [MB96] technique, we compute the FFT each frame and apply the normal map directly.

As the FFT is periodic by itself, we can repeat the texture multiple times without noticeable seams nor repetitions but increasing frequency.

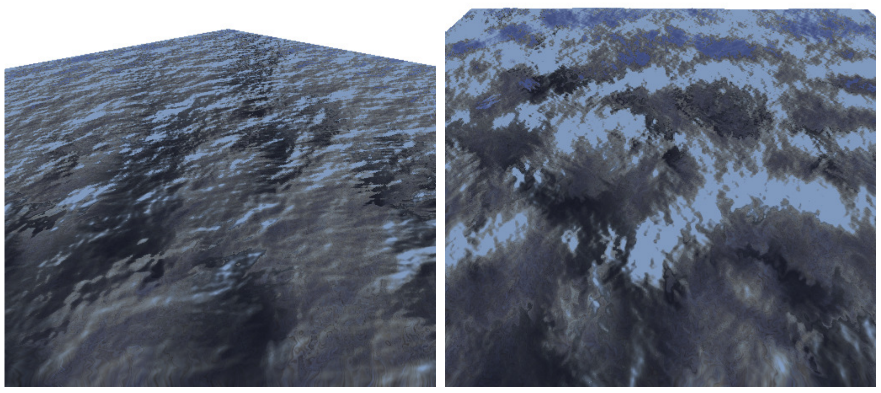


Figure 5.4: FFT ocean simulation applied to the LBMSW surface with normal mapping. The right image is a close-up version of the left one with a more zenital angle of view.

5.2.2 Caustics

Caustics are a very distinguishable effect from any refractive or reflective object. They are generated from the concentration of photons in the same spot in space, incrementing the intensity that location is lit.

The simulation of caustics is traditionally accomplished using techniques of global illumination [PH10] like pathtracing [Kaj86], the metropolis light transport method [VG97] or the more recent *photon mapping* [JCKS02], in which a number of photons are traced through the scene and gathered in their final positions to proceed with the pixel lighting. These techniques are costly, as there needs to be a high count of traced rays or photons to achieve soft caustics.

[Sta96] was the first to explore real-time caustics; although inaccurate, they were visually compelling, using synthetic texture maps. Nevertheless, to achieve physically realistic results, the more recent techniques are based on raymarching techniques:

- Rendering from light, using caustic maps[2] like, e.g., [SKALP05, WD06, SKP07].
- Rendering from the inverse side, [GSC04, YK09]: from the caustics receiver object (which they require to be planar as a simplification), the ground surface, photons are tracked back to the light through a limited area in the refractive surface, the fluid.

In order to provide higher realism, we have also added caustics to our LBMSW fluid simulations. We have inspired in [SKP07] and extended their work. They raycast a grid of photons, as points, through the scene in an orthographic space defined at the light source, which also allows them to easily add shadow mapping. They require the depth map used in the ray cast phase to be continuous or, at least, with no great jumps. This technique, however, has the limitations of image-based rendering, namely, the results depend on the resolution of the textures used, although the resolution used in the light space also limits where the photons can end.

Our contributions to their algorithm imply extending the raycast of photons out of the light space to screen space, splatting them oriented with the surface of the receiving mesh. In contrast to [SKP07], we do not generate a caustics map; the splats are blended with the scene, varying their intensity depending on the orientation of the caustic generating fluid pixel (if it is facing the light source or not), as well as the distance the photon has travelled. We also restrict our approach to refracted photons and ignore reflected ones.

Summarized in Figure 5.5, our algorithm is also composed with multiple render passes and is described as follows:

[2]Similar to photon maps, but are used like shadow maps are: reprojected in camera space in order to lit the visible pixels.

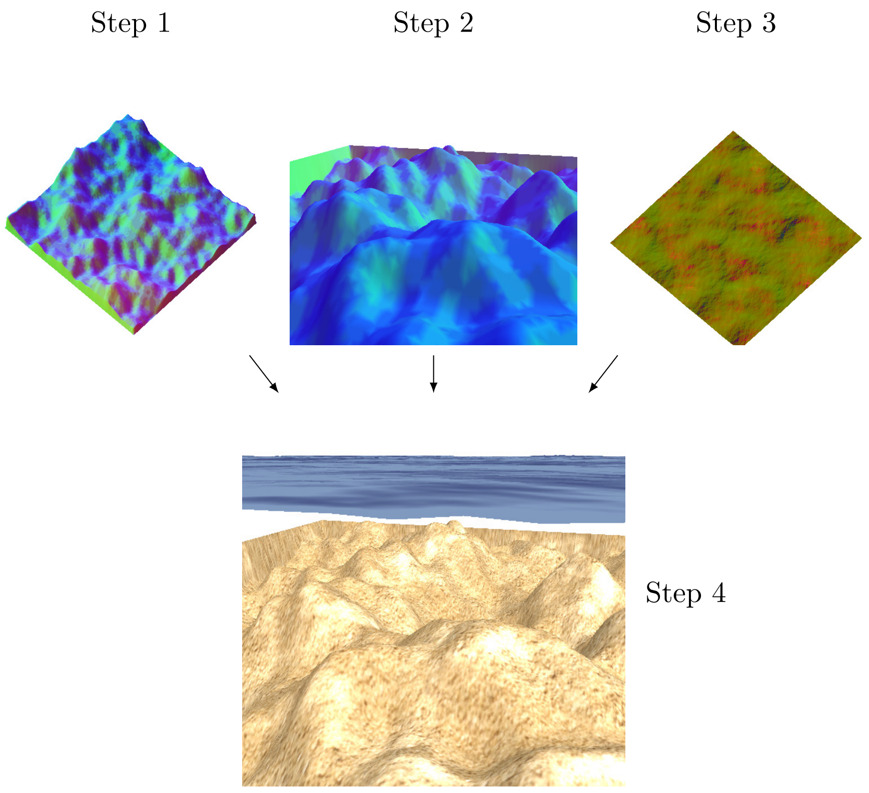


Figure 5.5: Multi-step caustics simulation. Step 1 renders everything (storing color, depth and normals) except the fluid from light, as does Step 2 from camera. Step 3 renders the fluid surface and stores space position and refracted vectors. Finally, Step 4 raytraces the photons as points using the data from Step 3 and expands them as quads when they hit geometry from Steps 1 and 2, blending them with the framebuffer contents.

1. Render the objects of the scene (excluding the fluid) from the camera and store the depth and normal maps.

2. Render the objects of the scene (excluding the fluid) from light with an orthographic projection and store the depth map.

3. Render the fluid from light with the same orthographic projection and store the space positions and refracted directions at each pixel.

4. Render the grid of photons. The primitives used are points which will be expanded to quadrilaterals when a final position is found.

   This grid of vertices has the same resolution as the orthographic projection used previously. In a vertex shader, the vertices will be raycast first in light space using the depth map from Step 2. If there is no intersection found (the photon exited through a wall of the frustum), the raycasting will be repeated in camera space. If there is not yet an intersection, the point is discarded (rendered out of frustum). Otherwise, if an intersection is found at light space, it is transformed to camera space and checked if it is correct:

- If the point is occluded in camera space, it is discarded.
- If the point depth does not match between light and camera space, the raycasting continues from the actual point position in camera space.
- If the point is correct, that is, the depths match between light and camera space, the point is final.

When a final point is found, from the previous condition or from the camera raycasting, the normal is looked up in the normal map from Step 1. In a geometry shader, the final points are expanded to quads oriented with their associated normal. Finally, in a fragment shader, the photons are textured with a Gaussian splat, and their intensity is regulated depending on how they are facing the light and the distance they have travelled through the fluid until finally hit the receiving surface. At last, they are blended to the contents of the framebuffer.

As shown in Table 5.1 and concluded in [SKP07], the performance of this algorithm depends primarily on the size of the grid of photons, but also on the direction of the light, which can cause more photons to miss the light space raycast and require the second camera space one, thus increasing the number of computations and texture fetches needed to try to find a final position for them. The fact that the raycast is done in the vertex shader also impacts the final performance because of the increased penalty of texture fetches in that shader stage.

With our double raycasting method, the photons are not limited to light space only and are finally oriented with the surface they fall over. However, as we directly blend the photons with the previous contents of the framebuffer, we do not have a correct radiance computation, but the results are good enough for real-time fluid simulations as can be seen in Figure 5.6. We have not included shadow mapping computations to keep the algorithm simple enough but, as we have stored the depth map from light in Step 2, it can be implemented as commented in [SKP07].

5.2.3 Refraction and Reflection

For the caustics, photons are refracted when they arrive to the fluid surface and concentrate in a ground mesh below. From the physical sense of light, some photons should bounce off the ground and travel to the camera point,

Viewport Size	Caustics Resolution	Caustics	Refraction & Reflection
512^2	128^2	1.0058	2.74639
	256^2	2.35144	2.79409
	512^2	7.92134	3.01034
	1024^2	51.3408	3.17178
1024^2	128^2	1.2585	5.79334
	256^2	3.33537	5.98238
	512^2	12.8643	6.21948
	1024^2	70.9195	6.53224

Table 5.1: Averaged timings in milliseconds for frame for the caustics and refraction/reflection algorithms. The Viewport column indicates the viewport resolution. Similarly, the Caustics Resolution column indicates the size of the viewport used for the orthographic camera, and thus, the number of photons traced.

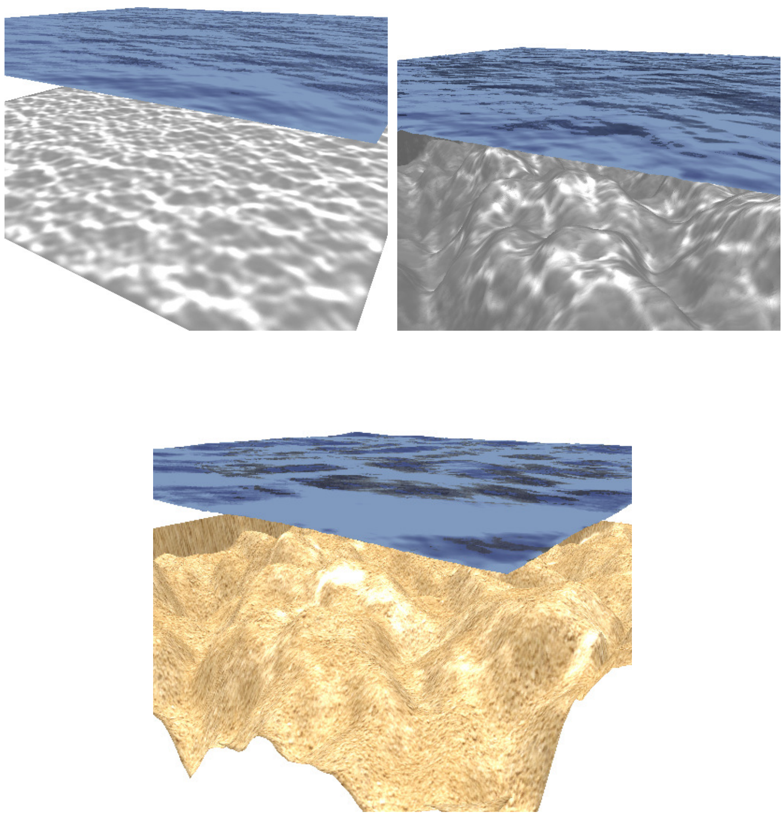


Figure 5.6: Caustics are projected onto the ground below the fluid surface. Planar ground at the top left image, Noisy ground at the top right and same ground with texture at the lower image.

crossing again the fluid surface and being refracted again. This is quite complex, more so in a real-time domain as we are targeting.

As rasterization in GPUs output pixels in screen, the common strategy is to render first the background geometry and storing the results to textures, maybe creating environment maps, and then render the refractive/reflective object using the previously created textures. In this direction, [Sou05] presented a simple method to simulate refraction, although not from a physically based standpoint. [SKALP05] relies only on environment maps as distance impostors to achieve approximate refraction. Other more complete techniques involve tracing rays through depth maps searching for intersections as, e.g., [Wym05, DW07, HQ07].

We approach the refraction and reflection view effects of the fluid in the same spirit of [DW07], like a backwards raytracing implementation; rays are cast through a depth map from each pixel of the camera viewport, being refracted or reflected (or both) when they hit the fluid.

As with caustics, our algorithm is composed with multiple render passes, summarized in Figure 5.7. It can be explained as follows:

1. Render the fluid mesh and store the depth buffer.

2. Render the objects of the scene (obstacles and ground) and compare the depth with the stored in the previous pass. If the depth is greater, store that fragment (color and depth) in one framebuffer (framebuffer A), if it is lower, store it in another (framebuffer B). To accomplish this we make use of multiple render targets. This pass can be thought of as a stencil test.

3. Render the fluid again, using the previous step result. For each fragment of the fluid two rays are cast: one for refraction (using results on framebuffer A), one for reflection (using results on framebuffer B). As the rays start from the camera, if the reflected/refracted rays should come back, they are discarded. The results of both raycasts are combined using Schlick's approximation [Sch94] to Fresnel terms.

4. Finally, to avoid the repeated render of the other objects of the scene, we use a screen-sized quadrilateral textured with the results of Step 2, more specifically from the "over" framebuffer (objects nearer than the fluid, framebuffer B). The textured quad is combined with the already rendered fluid.

To reduce somewhat the obligation of the double raycasting, we compute the Fresnel term from [Sch94] prior to the raycastings at Step 3 as

$$F = F_0 + (1 - F_0)(1 - \theta)^5, \tag{5.6}$$

being θ half the angle between the ingoing and outgoing light directions and F_0 the known value of F when $\theta = 0$, the reflectance at normal incidence.

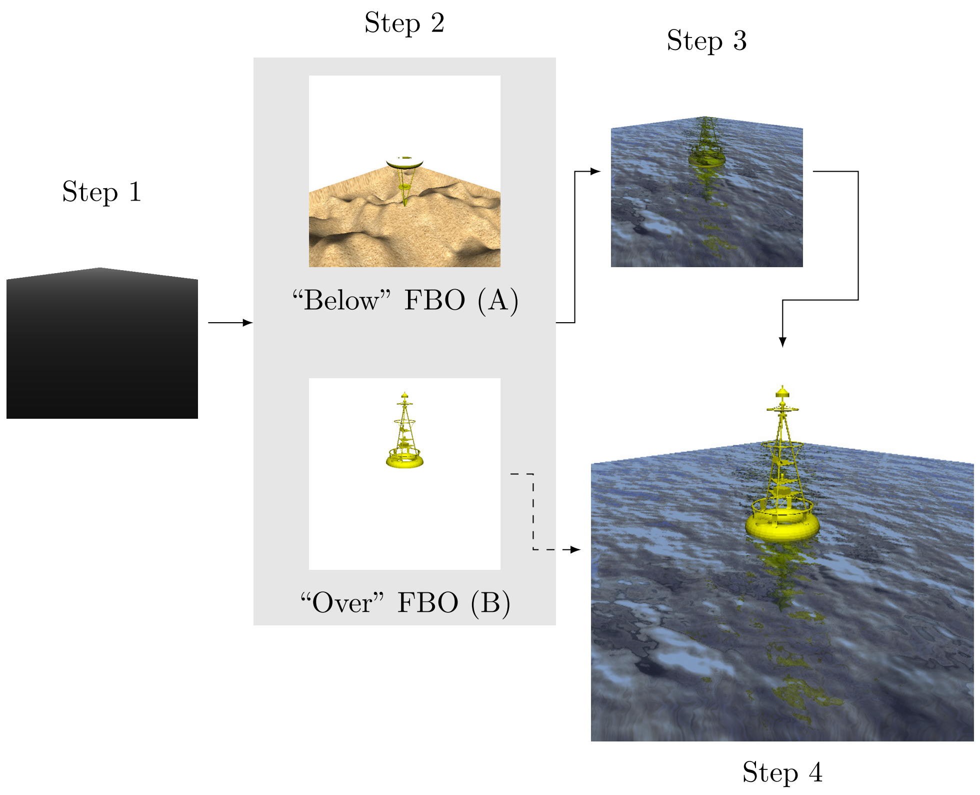


Figure 5.7: Refraction and reflection in multiple render passes. Step 1 stores the depth buffer from rendering only the fluid. In Step 2, objects above and below the fluid get separated in two framebuffers, which are used in Step 3 to raycast reflections and refractions in the fluid. Finally, Step 4 composites the stored "over" framebuffer from Step 2 into the scene to avoid rendering the objects again.

As we use the value of F for a linear interpolation between the refracted and reflected colors, we can impose a threshold ε were:

- If $F < \varepsilon$, only the refraction raycasting is done.
- If $1 - F < \varepsilon$, only the reflection raycasting is done.
- Otherwise, both raycastings are done.

Additionally, for zones where the height is quite low, i.e., cells where the fluid has just entered using the dry-wet algorithm of Section 3.7.3, we get the color from the direct view ray and interpolate from it to the final combined color from the raycastings using the depth difference between the fluid and the ground below. This alleviates the artifacts caused by the triangular mesh used in the fluid rendering as shown in Figure 5.8.

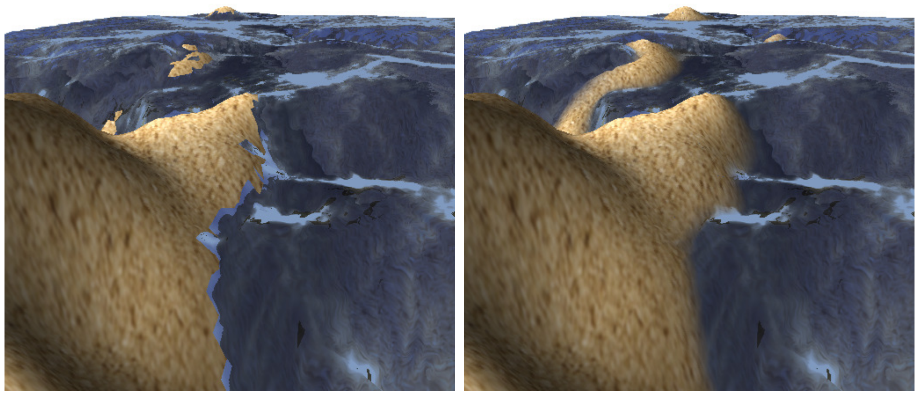


Figure 5.8: Artifacts from the fluid's triangular mesh on the left, alleviated on the right interpolating the color value between the fluid's color and ground color depending on the view distance from surface to ground.

Everything is done in screen-space, so there may be zones where there is not enough information, i.e., a ray should hit a point in space not visible; in those cases we detect the jump in the depth map and make use of the last pixel with information in the texture. Although this is an artifact, combining the fluid with the animated lower scale detail techniques described in Section 5.2.1, it remains greatly unnoticed.

The performance of this algorithm is quite variable, it depends on the size of the viewport as well as the coverage of the fluid in screen: the more visible pixels, the more rays are cast. For fair comparison, the results in Table 5.1 were captured with the fluid covering the whole viewport, and even in this case, the whole algorithm does not cost more than 10ms for a reasonable sized viewport. Nonetheless, the results are quite convincing, being a sample Figure 5.9.

5.2.4 Particle Rendering

For the hybrid Particle-LBMSW, we already have explained how the LBMSW is rendered but a different technique should be applied to the particles.

There are various alternatives which have been extensively used to render particle systems, like marching cubes [LC87] or metaballs [WH94, vKvdBT08], although these methods are costly and, thus, not often used in real-time applications.

However, for interactive environments, particles have been traditionally rendered using primitives as points, lines, or even making use of billboards. Nowadays, as particle fluid simulations become more common in real-time applications, new techniques have arisen to increase the appeal of their vi-

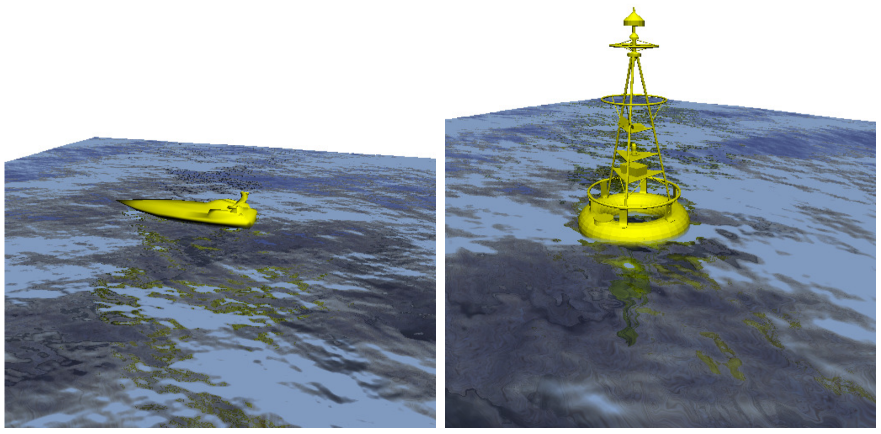


Figure 5.9: Reflection and refraction on the fluid surface. The FFT ocean simulation is used to increase normal disturbance.

sualization. [MSD07] renders an SPH fluid simulation as a mesh in screen-space, generating the surface only where it is visible using a marching squares algorithm. [CM10] use for its particle rendering the algorithm devised in [vdLGS09]; the particles are rendered as spheres and the depth buffer is used to retrieve a screen-space surface using in the process some smoothing methods to relax the surface of the fluid, computing afterwards normals to the surface as a screen-space technique. To avoid complex geometry, the spheres of [vdLGS09] are rendered as point sprites (or screen oriented quads) with depth replacement, quite related to the Nailboards technique found in [Sch97].

As we would like to maintain the crisp appearance from the particles, as they are splashes and not a big body of fluid as in [vdLGS09], we have chosen to use the depth replacement technique to render each particle but we also provide a normal texture for them. The particles are sent to the pipeline as points, which are then expanded to quadrilaterals in the geometry shader and later textured in the fragment shader with the normal and depth maps. We haven't used the same refraction as for the LBMSW simulation because its raycasting is quite costly, instead we have used the cheaper method from [Sou05]: the final color is looked up in the framebuffer[3] using an offset on the refracted vector. The particles are finally blended with the framebuffer, but updating accurately the depth buffer, too. Additionally, we take into account the TTL (time-to-live) of the particles, fading them as their lifes approach the end before being reintegrated.

In Figure 5.10 we can see the result. Still, at a very near sight of the particles, it is obvious their spherical nature. This could be alleviated by applying noise to the depth and normal maps we are using for them, which

[3]It has to be stored from the previous pass, so particles will be the last thing to render.

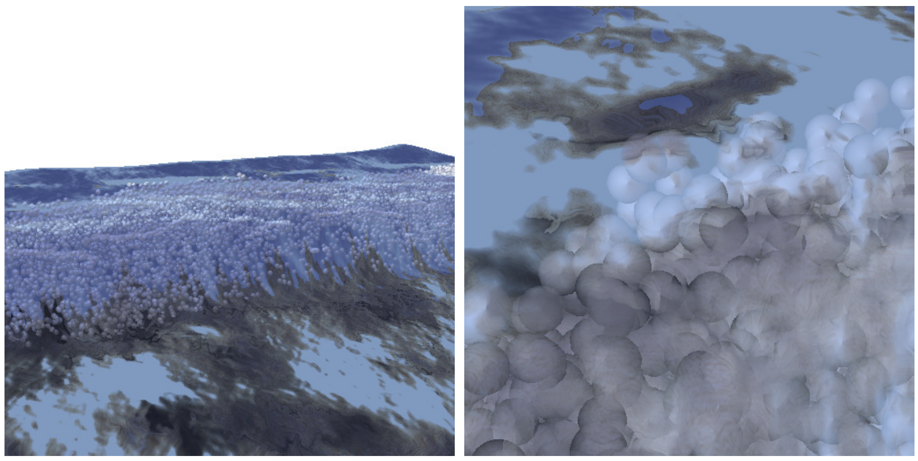


Figure 5.10: Particles are rendered as Nailboards. At a near point of view (right) it is still clear the spherical nature of the particles.

would result in a noisier refraction.

5.2.5 Surface Foam

As a final effect to improve the render of the hybrid Particle-LBMSW is the added surface foam generated from particles being reintegrated in the bulk of fluid.

For this effect [CM10] generate diffuse disks oriented with the surface fluid and advected by it. Although they don't explicitly explain how foam is simulated, this would require an additional number of particles dedicated for the effect, or changing the type of actual particle in use, limiting the global number of splash particles for the next frames, if they use a similar implementation of particle integration as ours (Section 4.3).

In contrast, we have resorted to a simpler but effective representation: using a float-point single component texture mapped to the whole fluid surface, we detect where a particle falls and initialize that texel to the maximum time-to-live (TTL) time of the foam. This texture is advected using the fluid's velocity field, tracing the value back as in [Sta99]. Each frame, the values of the texture are decreased Δt until they hit 0. For the visualization, we just apply the resulting texture to the LBMSW triangle mesh at the same time the refraction and reflection are computed. Figure 5.11 shows the result.

Although the resulting effect is acceptable, this method has a constraint: the texture resolution. Depending on the size of the particles to texel size ratio, it could happen that not only one texel should be initialized when a particle hits the fluid surface. It could also happen the inverse problem: a texel could be too big in relation to a particle size. Although we are

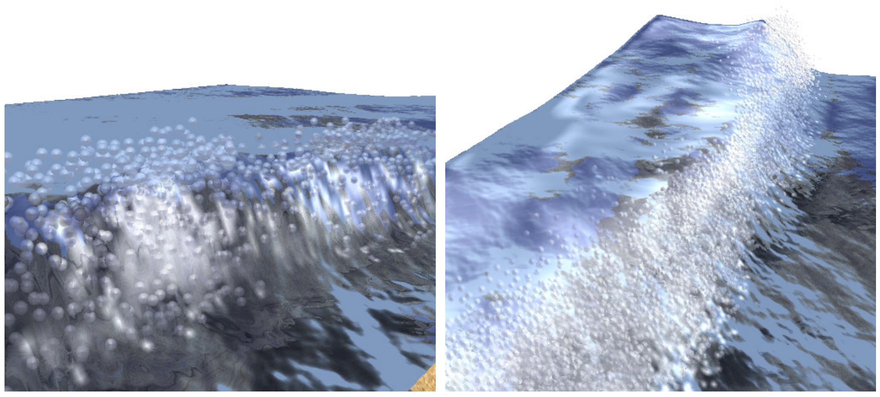


Figure 5.11: Foam is generated at particle-surface points and advected in successive frames using the fluid's velocity.

using a fixed sized texture in this case, this problem could be solved using a pyramidal texture approach; the particles initialize the correct level of the pyramid, being the other levels initialized extrapolating from that one.

Chapter 6

Results and discussion

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

Our initial objective was to achieve real-time fluid simulations and complete realistic visualization. Although it was an ambitious goal, we have succeeded on building some implementations based on the Lattice Boltzmann Method with correspondent visualizations done in not-so up-to-date hardware which still achieves interactive framerates.

In this chapter, we will summarize all the work done in this thesis, in addition to present the results obtained and discussion upon them. We will also discuss the caveats and drawbacks worth to have in mind, as well as the possible lines of research for future extensions to the present algorithms.

As the requirements for each implementation are somewhat different, we have separated the discussion on various sections. First, we will introduce the initial results for the LBM3D simulation which made us decide for the LBM as a viable algorithm for GPU implementation; this was the first comparative of performance between a CPU and a CUDA implementation of the LBM. We continue then with the free surface implementation using the Volume of Fluid extension to the LBM and finish with the hybrid method that combines a surface fluid as LBMSW with particle simulation for breaking waves.

In all the cases, the hardware has remained the same: an Ubuntu Linux (versions ranging from 8.10 to 11.10) running on an Intel Core2Duo E8400 at 3Ghz with 4 Gb of RAM and a Nvidia GTX280 card. The version of CUDA used has also varied through the duration of this thesis, from the 2.1 version to CUDA 4.1.

6.1 LBM3D in CUDA

The cellular automata approach of the LBM with its simple operations, using the BGK collision operator, is quite appealing for a parallel implementation, as the computations remain local to each cell.

For this reason, our first tests were directed to find how efficient a GPU implementation of the Lattice Boltzmann Method was, compared to a CPU version. For a fair comparison, we also implemented a parallel CPU version using OpenMP [Boa13]. Although the CUDA implementation presented in [OS09] was not the most optimized possible, it already shown a great acceleration for the GPU version.

For that comparison we used the main hardware already presented and a MacBook Pro which contained an Intel Core2Duo T9400 at 2.53Ghz with 4Gb of RAM and two Nvidia graphics cards, an integrated 9400M and a dedicated 9600M GT. The whole specifications are shown in Table 6.1.

	Main Clock	**RAM**	**Mem. Clock**	**Mem. Bus Width**	**Cores**
9400M	1.16Ghz	256Mb	1066Mhz	128-bit	16
9600M GT	1.5Ghz	512Mb	900Mhz	256-bit	64
GTX280	1.296Ghz	1Gb	1107Mhz	512-bit	240
C2D T9400	2.53Ghz	4Gb	1066Mhz	64-bit	2
C2D E8400	3.0Ghz	4Gb	1333Mhz	64-bit	2

Table 6.1: GPU and CPU Specifications.

Usually, the performance of the Lattice Boltzmann simulations are measured in updates per cell per second (Lattice node Updates Per Second, LUPS) although nowadays, with the raw power of the computational units, it has been conveniently replaced by millions of updates (MLUPS). For comparison purposes however, one has to be aware of the LBM DnQm model used, as well as the precision used; although the metrics unit is the same, we can't compare a D3Q19 simple precision simulation with a D3Q27 double precision one, as the latter has more memory constraints.

With a D3Q19 model, we used a common example for fluid simulations, a lid-driven cavity; the whole domain is full of fluid and all walls are boundary, one of them introduces some acceleration to the fluid. It is shown in Figure 6.1.

For an execution of 10000 iterations, the time execution needed for each implementation and hardware is shown in Figures 6.2 and 6.3. The second Figure shows a GPU-only plot, as there is a great difference between the CPU and GPU results that makes almost indistinguishable the GPU results. From these timings we can compute the MLUPS for each processing unit, as shown in Table 6.2. The same timing operation was done for 100000 iterations, but the resulting MLUPS were the same.

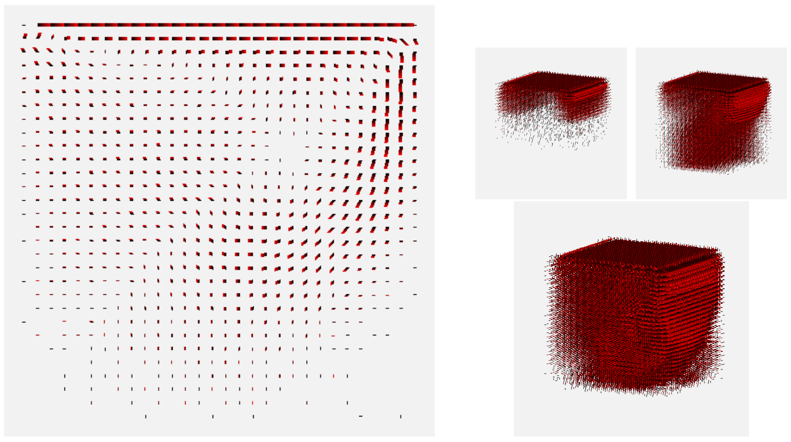


Figure 6.1: Lid-driven cavity velocity field. A 2D slice at the left, some frames to the right.

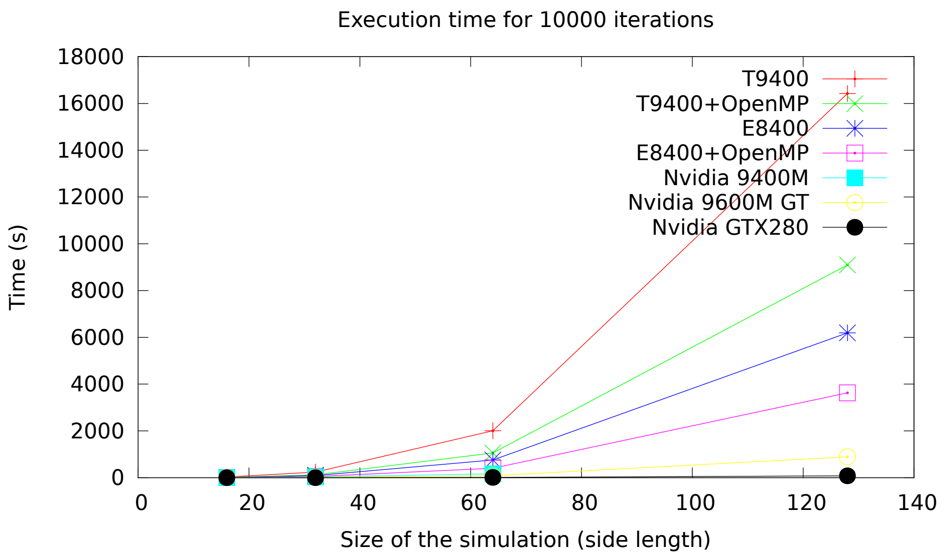


Figure 6.2: Execution times for 10000 iterations of the lid-driven cavity example. CPU and GPU results.

From those results in which the GPU had an up to 44.9x acceleration in comparison to the CPU parallel implementation, we decided to continue research of an LBM implementation in GPU with as many features as possible, within the real-time simulation and visualization realm as the final goal.

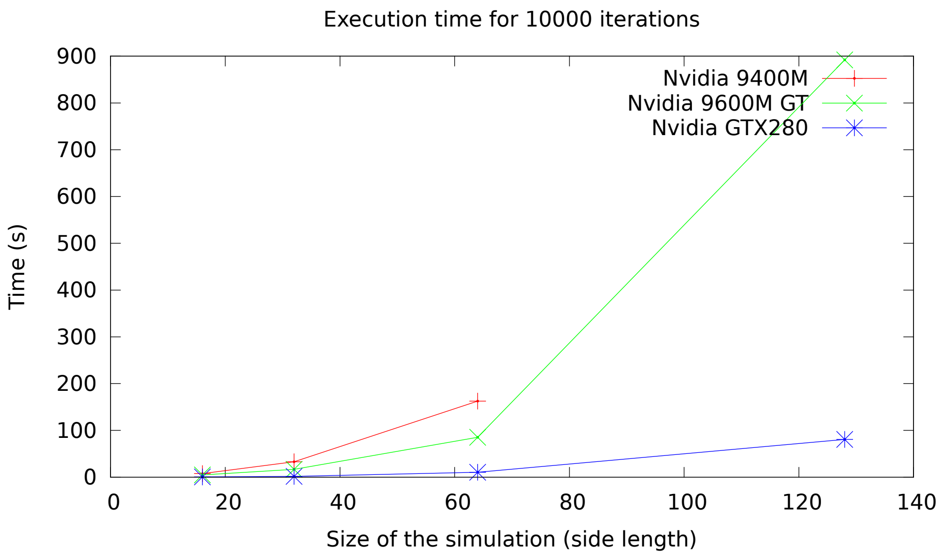


Figure 6.3: Execution times for 10000 iterations of the lid-driven cavity example. GPU only results. The 9400M GPU couldn't fit the 128^3 simulation because of its memory requirements.

	Domain size	**MLUPS**	**MLUPS with OpenMP**
T9400	16^3	1.48104	2.95092
	32^3	1.37848	2.79783
	64^3	1.30438	2.48282
	128^3	1.27699	2.30586
E8400	16^3	3,96587	7.36984
	32^3	3,68622	6.16978
	64^3	3.46614	6.48213
	128^3	3.38778	5.78497
9400M	16^3	5.31683	
	32^3	9.97503	
	64^3	16.13213	
	128^3	-	
9600M GT	16^3	8.60882	
	32^3	19.69716	
	64^3	30.70407	
	128^3	23.50601	
GTX280	16^3	88.32154	
	32^3	209.18237	
	64^3	248.02634	
	128^3	259.82635	

Table 6.2: MLUPS classified by simulation domain size and processing unit.

6.2 LBM3D - Volume of Fluid

In order to have a complete 3D fluid simulator, we extended the simpler LBM3D simulation with a free-surface algorithm. In fact, we pondered various methods, but the most important ones were the level-set approach and the volume-of-fluid method.

The level-set method has been heavily used, mainly for offline simulations. It tracks a narrow band of cells where the surface is located but needs to be reinitialized every some frames, and requires additional particles for a precise tracking. In comparison, the volume-of-fluid method can be implemented locally, which favors the GPU parallel architecture, and had already been adapted to the LBM in [Thü07], so we decided to create what has been its first CUDA implementation.

The algorithm has been presented in Section 3.6.2 and the CUDA implementation is shown in Section 4.2.1. With the raycasting visualization from Section 5.1 shown in Figure 6.4, we present some timings in Table 6.3.

	LBM3D-VOF
Breaking Dam (32)	4.275
Drop (32)	5.471
Breaking Dam (64)	37.539
Drop (64)	26.142

Table 6.3: Timings for various examples in milliseconds per frame ($\Delta t = 16ms$). Examples marked with (32) were executed in a 32^3 grid while examples with (64) were executed in a 64^3 grid.

For small domains the simulations can be done at interactive rates, but for greater ones it is evident that they can not achieve the same results. The hardware imposes heavy constraints which explains the timings. Although we improved the LBM implementation from [OS09] as explained in Section 4.2, the full LBM3D-VOF algorithm requires a lot of memory accesses which are the slowest operation in a CUDA application, although we ensure those accesses are coalesced. Furthermore, these memory accesses are data-dependent, they should be done (or not) according to certain conditions, e.g., if the next cell is empty or not, and this also introduces another problem: code divergence. Inside a warp, the lower unit of execution, some threads have to execute some code and others don't, the execution is serialized for each group of threads until the branching is done.

These limitations are inherent to the algorithm, but could be alleviated with newer hardware, as newer Nvidia cards have more cores, greater memory frequency and more features, e.g., all global memory is cached.

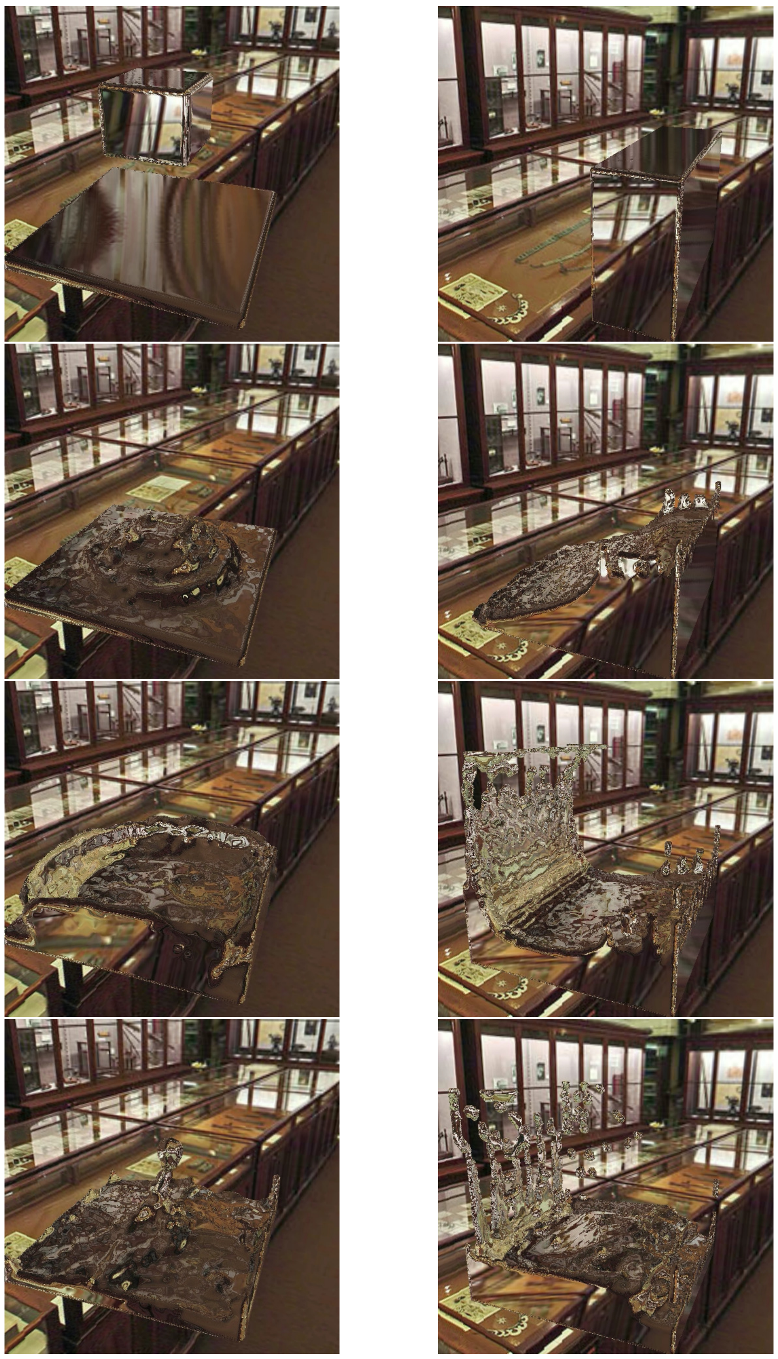


Figure 6.4: Image stills from the drop example in a 32^3 grid (left column) and the breaking dam example in a 64^3 grid (right column).

6.2.1 Comparing approaches

To put things in perspective, we should compare this method and its results with previous approaches trying to simulate and visualize fluids as physically-realistic as possible and in a real-time environment, preferably.

As the LBM3D-VOF simulates a full domain of liquid, it is fair to put it against other methods that work in the same sense of a full 3D simulation. We can see summarized the main differences in this comparison in Table 6.4.

Nowadays, a state of the art Navier-Stokes implementation uses the semi-Lagrangian advection from [Sta99], which enables to make arbitrary-sized time steps at the price of increased diffusion, an unwanted side-effect for liquids. Moreover, the solution to the pressure requires heavy operations, usually solved in an iterative manner. This is indeed how both [CLT08] and [CM11] generally work. In comparison, the LBM has a fixed time step, but its computational complexity is quite reduced because of the simpler arithmetic operations required for the collision step. Its major drawback comes from the VOF algorithm due to the free surface simulation; it requires several operations to be serialized and has many data-dependent code branches, which penalize severely its performance in the GPU, restricting the whole size of the simulated domain. [CM11] works upon the basic Navier-Stokes using additional tall cells for the underlying fluid; they allow to increase dramatically the simulation domain size because of this multiresolution-like approach at the expense of additional code complexity to deal with them. These tall cells could be somehow adapted to the present LBM3D-VOF solution, in order to allow for higher simulation domains.

An SPH approximation to the simulation of fluids clashes with the other solutions, as the SPH does not require a grid; the fluid is actually carried with the particles themselves. As the fluid is not restricted by a grid, the particles can go wherever they need in the full scene; the problem here comes from the required number of particles: for a similar result to the grid-based solutions, a high particle count is required. Additionally, SPH has an additional burden, so to say: its inherent computations require for each particle to check out the neighbour particles in a specified radius. For a GPU implementation, [GSSP10] propose the use of an auxiliary grid in order to implement this neighbour search efficiently, although it requires more memory than similar SPH CPU implementations and also needs of additional operations as sorting arrays.

It is hard to directly compare particle simulations with grid-based ones; particle simulations allow for sharper features as splashes because of its own nature, these sharp features are usually lost or smoothed in the grid simulations using the level set algorithm ([CLT08, CM11]). The Volume of Fluid algorithm, however, ensures mass conservation and keeps these sharp features, although makes them look more drop-like because of the iso-surface render methods used (being them mesh-generating as marching

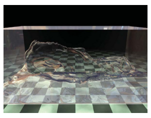
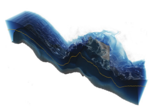
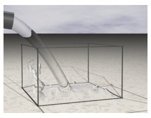
	LBM3D-VOF	 [CLT08]	 [CM11]	 [GSSP10]
Domain size	↓↓	↓	↑↑	↑
Comp. complex.	↓↓	↑	↑↑↑	↑↑
Smooth surface	✓	✓	✓	✗
Sharp features	✓	✗	✗	✓
Add. data struct.	No	No	Yes	Yes
Complex render	Rfl + Rfr	Simple Rfr	Add. maps	Rfl + Rfr

Table 6.4: Feature comparison between the LBM3D-VOF model, Navier-Stokes approximations ([CLT08, CM11]) and an SPH solution ([GSSP10]). In the table, Comp. complex. stands for computational complexity and Add. data struct. stands for additional data structures needed. In the Complex render row, values containing Rfl mean that the render does reflection, Rfr goes for refraction and Add. maps for additional texture maps to simulate alternative effects like foam advection.

cubes or direct raycasting). The same could be said in the other direction, level set fluid simulations provide naturally a very smooth surface, which can also be accomplished by the VOF algorithm. Particles in the other hand require more elaborate techniques, e.g., [vdLGS09], to allow the same level of smoothness.

As we want simulation and rendering at the same time, we should also compare how complex the visualization is, the effects it provides that enhances the look and feel. The main effects one should expect from a fluid are refractions and reflections, achieved easily with raycasting techniques as the one implemented in the present work, or the raycasting of distance fields used in [GSSP10]. In contrast, to provide faster results, although not as physically realistic, it is common to use simplifications, as the simplified refractions provided by [CLT08], which just use an arbitrary offset on the refracted vector to look up the background. To further simplify the render, these effects can be ignored altogether as in [CM11], although, in order to improve the overall quality of the visualization, they apply additional texture maps to simulate foam and to provide higher detail of a lower scale as well as use ballistic particles for aesthetic purposes.

A full numerical comparison is difficult because of the different hardware used in these approaches, but, as seen in Table 6.4, from this comparison we can conclude that the LBM3D-VOF is, maybe, the most balanced system for small domains, which can be further improved to enable higher simulations.

6.2.2 Future work

In the future, it should be interesting to apply multi-resolution methods to the LBM GPU simulation, which could enable for greater domains, as only the surface cells should remain at a higher level of detail and inner fluid cells could be simulated with more coarser levels as in the previously mentioned tall cells approach from [CM11]. An alternative could be the coupling with a Shallow Waters approach, requiring only a full 3D simulation for a given region of interest. Furthermore, as the LBM is a memory expensive algorithm it would also be important to achieve some method that reduced its requirements as in [PKW$^+$03], but in a GPU-friendly manner.

After an LBM implementation is efficient enough for average domain sizes, it should be added the coupling of external obstacles, e.g., rigid bodies. This would surely need some kind of object voxelization in realtime using techniques like [ED06, ED08], for example. Moreover, to increase the level of detail of the simulation, a coupling with a different fluid simulation method like SPH would be a great extension as in [LTKF08].

It should be also interesting to improve the raycasting technique used in the visualization to allow for multiple ray direction changes, thus empowering multiple refractions and reflections.

6.3 Hybrid Particle - LBMSW

As the hardware limited our ability to develop a full 3D Lattice Boltzmann simulation, we decided to simplify the problem: limiting the representation of a bulk of fluid to just the surface as a heightfield and then coupling it with additional methods to provide details lost in the simplification.

Evolving from the LBM implementation we already had, we created a Shallow Waters version and coupled it with obstacles. Later we added the particle simulation, generating particles with breaking wave conditions. Some timings on the final implementation with the main hardware are provided in Table 6.5. Here the boat example refers to that visible in Figure 3.14, the buoy corresponds to Figure 3.13, drop is Figure 3.17, the wave example is shown in Figures 5.10 and 5.11 and wavegr is visible in Figure 3.16.

In this case, the algorithm has some of the same problems as the LBM3D-VOF: high memory requirements and code divergence. The memory requirements are still high, but the fact that the domain of the LBM is in 2D instead of 3D allows us to simulate greater domains, although we require additional memory for the particles. Code divergence still is a problem when dry regions appear, which could be considered a similar case to the free-surface of the LBM3D-VOF. Particle simulation also introduces new problems; as we can not generate and destroy particles dynamically on the GPU, we have to limit a certain number and use them on-demand, which can cause arti-

		Total	LBMSW	Solids	Psort	PGen	PSim	PReint	PReint_s
CPU	boat	10.78	10.69	0.09	0.00	0.00	0.00	0.00	0.00
	buoy	10.91	10.85	0.06	0.00	0.00	0.00	0.00	0.00
	drop 32k	372.96	9.97	0.00	214.56	1.16	1.46	31.50	114.31
	drop 128k	1635.36	9.75	0.00	1001.35	2.09	2.79	114.92	504.46
	wave 32k	410.88	9.62	0.00	224.04	3.68	4.85	32.58	136.11
	wave 128k	1694.87	9.62	0.00	1007.30	3.37	10.71	117.51	546.36
GPU	boat	0.82	0.35	0.47	0.00	0.00	0.00	0.00	0.00
	buoy	0.87	0.35	0.52	0.00	0.00	0.00	0.00	0.00
	drop $32k$	14.30	0.35	0.00	7.03	0.45	0.10	1.24	5.13
	drop $128k$	23.50	0.34	0.00	10.71	0.41	0.14	1.75	10.15
	wave $32k$	15.28	0.36	0.00	7.26	0.99	0.12	1.32	5.23
	wave $128k$	25.71	0.36	0.00	11.21	1.42	0.32	2.01	10.39
	wave $256k$	34.39	0.37	0.00	16.45	1.18	0.54	2.55	13.30
	wave $512k$	54.84	0.36	0.00	27.76	1.04	0.81	2.97	21.90
	wavegr $64k$	18.79	0.39	0.00	9.48	1.18	0.18	1.64	5.92

Table 6.5: Timings per frame ($\Delta t = 16ms$) for various examples in milliseconds; the number in the name of the example indicates the total number of particles used, where $k = 2^{10}$. LBMSW includes the LBM simulation, as well as the dry-wet region tracking. Solids accounts for the 2-way coupling of dynamic rigid bodies. PGen, PSim and PReint are the timings for the generation and initialization, simulation and reintegration of the particles respectively. We have extracted the timings for the sort operation from the Thrust library in Psort and PReint_s, as they do not depend directly on the other steps but have a significant impact on the results, more heavily on the CPU version. Psort is for the sorting of particles by their TTL. PReint_s is the sort of the (cell_id, particle_id) tuples.

facts if the simulation is not well parametrized. Also, as the GTX280 does not have float atomic operations, the reintegration of the particles is more costly than it will be on newer hardware. With the same lattice size for the LBM, the timings only vary on the number of particles used, which could also improve with newer hardware as more computational power would be available. Still, the results are quite compelling (see Figure 6.5), enabling full interactive fluid simulations depending on the particle count.

6.3.1 Comparing approaches

As we previously did with the LBM3D-VOF, here we should also provide some context on the results of the Particle-LBMSW and how they fare with regard to alternative methods used in interactive scenarios.

At this point, the techniques which we have to compare should be using a heighfield representation of the fluid, as this condition itself restricts the type of simulation, but also provides higher performance because of the reduced

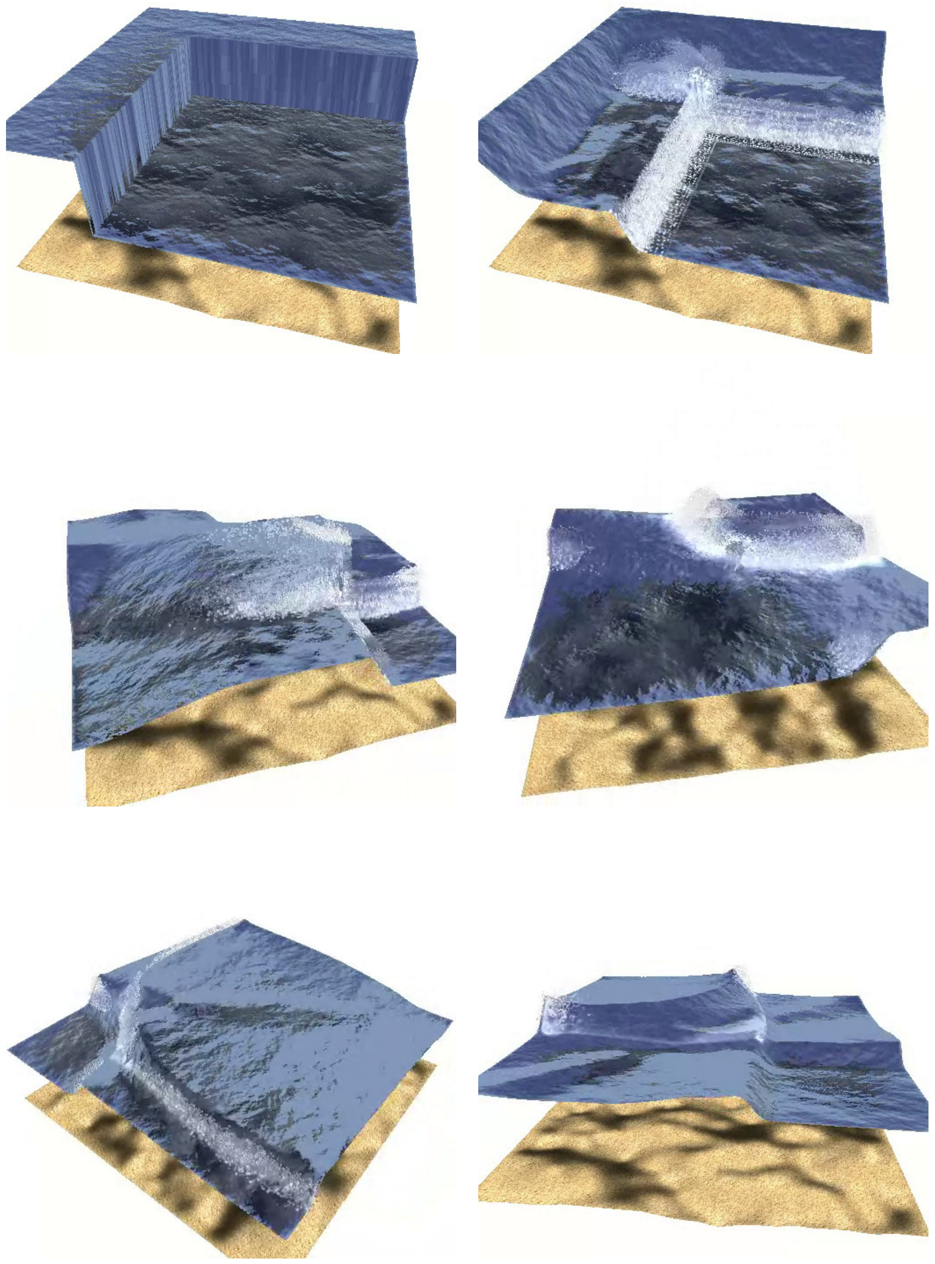


Figure 6.5: Image stills from another breaking dam example: two perpendicular columns of water break loose and generate expanding waves.

domain size as the simulation is done in a 2.5D fashion. This discussion is summarized in Table 6.6.

We are mainly interested in physically-based simulations with visualization in real-time, but [YHK07] has been successfully used in the Uncharted 3 videogame [GOH12], which relies in scenarios with great amounts of water in a realistic fashion. This shows us that it is more important the plausibility of a simulation rather than its physically accurate result, if it is as efficient as the application requires. The counter argument against such a procedural technique is that a lot of physical detail is lost, as horizontal flow, the inability to treat properly if the fluid should break or the appropriate consideration for the underlying terrain.

As shown, the physically-based simulations do not have those problems, and can respond adequately to the underlying terrain, as well as break apart (although [SBC$^+$11] do not provide a solution for, e.g., breaking waves, they mention that it should be studied in future work). For our Particle-LBMSW, we have followed the approach from [CM10] for the breaking wave detection and particle solution but, as they do not provide implementation details about their particle system, we have engineered our own, explained in Sections 3.8 and 4.3, adapting them to the hardware we had available, which has lower specifications than that used by the other approaches (except [YHK07]). It should be noted that although [CM10] provide multiple conditions for detecting when to generate particles and we have just used the breaking waves ones, the other conditions should be easily ported between methods. Moreover, we have shown that only the particle coupling (generation and reintegration phases) is to be adapted, leaving the particle system to use as a decision for the user of this method: here we have used a simple ballistic particle system for demonstration purposes and because of the hardware constraints, but a more sophisticated method as SPH could be used as well.

It is worth mentioning that all the methods compared have dynamic objects two-way coupling. Indeed, this should not come as a surprise because the three physically-based methods, Particle-LBMSW, [CM10] and [SBC$^+$11], use modified versions of the coupling introduced by [YHK07] due to its simplicity.

In contrast to the particle density-based computations for the heightfield from [SBC$^+$11], the Particle-LBMSW remains intact as in the LBM3D-VOF, where just the stream and collision steps of the basic LBM algorithm are needed. This simplicity comes at a cost: for greater sized domains, the time step has to be reduced, so more iterations have to be done in order to solve a single frame. On the other hand, in order to speed up the solving time for the simulation, [CM10] apply an explicit method (in contrast to the previously used semi-Lagrangian one) that may lead to instabilities during the simulation, which they handle specially.

Finally, not only the simulation has to be fast, we would like the vi-

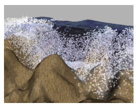
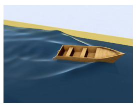
	 Particle-LBMSW	[CM10]	[SBC+11]	 [YHK07]
Physically-based	✓	✓	✓	✗
May break apart?	✓	✓	✗	✗
Object coupling	✓	✓	✓	✓
Under. terrain	✓	✓	✓	✗
Comp. compl.	↓	↑	↓	↓↓
Domain size	↓	↑	↑	↑
Complex render	Rfl + Rfr + Caust + alt.	Rfr + alt.	Rfl + Rfr	Rfl + Rfr + Caust

Table 6.6: Feature comparison between the Particle-LBMSW simulation, a Navier-Stokes shallow water solution ([CM10]), an SPH approximation ([SBC$^+$11]) and the Wave Particle method [YHK07]. In the table, Under. terrain stands for supporting arbitrary underlying terrains and Comp. compl. goes for computational complexity. For the Complex render row, Rfl and Rfr mean the same as in Table 6.4, reflection and refraction, Caust refers to caustics and alt. goes for alternative techniques that enhance the overall visualization as the advection of foam textures.

sualization to be believable. For that, the common light effects should be expected: refraction and reflection. Caustics are also a great addition to the final result. Although not every method explains which of this effects they implement, their images show some of them, so we can make educated guesses about what techniques they are using. [YHK07] provides results using reflections, refractions and caustics. [SBC$^+$11] also provide refractions and reflections although the latter seems to be generated only using static environment maps. [CM10] also use refraction and reflection which also seems to rely on environment maps and not dynamically generated from the scene, but they provide information about the use of an advected FFT ocean-based normal map to improve the low-scale details, as well as using a modification on the technique from [vdLGS09] for the rendering of the particles. It is hard to say whether the refractions are made using raycasting techniques or not, as simplified versions using just offsets over the refracted vectors result in plausible images. Our approach in this case provides both reflections and refractions raycast through the depth map of the scene, at the cost of reducing the available information to what is already visible in screen space. We also provide caustics, as well as other techniques to enable higher detail at the low scale like the FFT ocean-based normal map and the foam advection. In our case, we have used a modification on the Nailboard technique [Sch97], which is simpler than the one used by [CM10].

Overall, as seen in Table 6.6, although direct comparisons are hard to establish, the main restriction on our approach is the time step required by the LBMSW, as well as the constrained number of particles and how they are generated/reintegrated due to the hardware used. These restrictions should be loosened with newer GPUs and alleviated using multi-resolution and level-of-detail techniques. Furthermore, we have provided a more complete visualization module for this heightfield-based fluid simulation than the other approaches presented.

6.3.2 Future work

In order to improve this algorithm, there are some ideas worth mentioning. For example, it should be interesting to apply multi-resolution techniques, if possible, for the LBM, enabling higher domain sizes. Particles would also benefit from using Level-of-Detail techniques that allow to relax the artifacts created from a bad parametrization, i.e., more particles required than available; simply by clustering near particles would allow to reduce the needed count. Alternatively, it also would be worth investing in some technique that tries to prioritize the preservation of visible particles, i.e., those that fall in the actual view frustum.

With our hybrid algorithm, it is easy to change the ballistic particle system used here for a different one as long as the main features are maintained (generation and reintegration); a more complete SPH simulation would achieve scenes with more realistic physical results.

So far we have only implemented breaking wave conditions for the particle generation, so more particle generation conditions should be tested, like those in [CM10]. Ideally, it should be researched for a generalized condition in which particles should be generated to supply the needed details and effects the LBM simulation lacks.

For the visualization of the fluid, we have used FFT ocean simulation to increase the visual detail although it is not coupled in any way with the actual fluid simulation. It should be interesting to look for a tighter interaction, or at least integrate it using the technique from [MB96].

We have also implemented reflections and refractions using screen-space ray-marching techniques which limit the direction of the rays to what is directly visible from the camera. It would be worth to search how to allow the full range of vector directions after they are refracted/reflected. Seamlessly, caustics are also raycast from light and represented as single primitives which are blended with the scene. As the number of photons is limited, there are zones oversampled as well as other are undersampled, a hierarchical solution could solve the problem as shown in [Wym08, WN09]. It also would be interesting to extend these caustics to volumetric ones as those in [LD11].

At this moment, we have only taken into account the visualization of the surface of the fluid, in the future it should be also a key point to research under water rendering as in, e.g., [GSMA08], in order to provide a full featured visualization.

For particle rendering we just used an adaptation of the Nailboard technique; particles where rendered as billboards with depth and normal replacement. They still look like little spheres so, to improve their look and feel, some additional tweaking could be done as applying some noise to their normals or deforming them in the direction they are moving to simulate motion blur.

6.4 Closure

The central problem tackled by this thesis was the real-time simulation and visualization of fluids. For that, the Lattice Boltzmann Method was used in a GPU implementation in order to maximize the parallelization and achieve the so desired interactive framerates.

For the hardware used and as we have seen, a full 3D LBM implementation is heavily accelerated using CUDA with respect to CPU, but the inclusion of the free surface algorithm to provide rich graphical simulations restricts its applicability to very low domain sizes. This is due the serialization needed for some of the computations that must be done. With newer hardware however, several constraints of the current GPU will be lifted, naturally improving the final performance.

This realization led us to simplify our approach to a Shallow Waters simulation, which loses some of the physical benefits of the full 3D simulation as the breaking waves, but allows for better framerates. To enrich the scenes and allow direct user interaction, dynamic objects were coupled. Finally, to recover some of the lost functionality by the Shallow Waters, particle systems were coupled to supply breaking waves in this case, but open to implement more effects where splashes should burst.

Realistic visualization of the fluid has also been in mind when implementing the whole system and, as such, refractions, reflections and caustics are implemented in screen space to provide fast algorithms, suitable for the applications that can benefit from these techniques, namely, videogames and interactive physical software.

Although our solution is also greatly dependent on the hardware used, we have shown that the algorithms presented here are up to the challenge for real-time simulations although, as always as it is, there is room for improvement.

We can conclude by saying that fluid simulation is still a heavy problem that has a lot a possibilities for interactive applications, which makes it one of the most active research fields with a huge range of applications for the future.

Bibliography

[ABB$^+$07] C. Andújar, J. Boo, P. Brunet, M. Fairén, I. Navazo, P. Vázquez, and À. Vinacua. Omni-directional relief impostors. *Computer Graphics Forum*, 26(3):553–560, 2007.

[ATBG08] Roland Angst, Nils Thuerey, Mario Botsch, and Markus Gross. Robust and efficient wave simulations on deforming meshes. *Computer Graphics Forum*, 27(7):1895–1900, 2008.

[BGK54] P.L. Bhatnagar, E.P. Gross, and M. Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94:511–525, 1954.

[BMW$^+$09] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *International Conference on Parallel Processing*, pages 550 –557, 2009.

[Boa13] OpenMP Architecture Review Board. OpenMP.org. `http://openmp.org/wp/`, 2013. [Online as of February-2013].

[BR86] J U Brackbill and H M Ruppel. Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys.*, 65(2):314–343, August 1986.

[CCM92] Hudong Chen, Shiyi Chen, and William H. Matthaeus. Recovery of the Navier-Stokes equations using a lattice-gas boltzmann method. *Phys. Rev. A*, 45:R5339–R5342, Apr 1992.

[CD98] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

[CKZL99] Gang Chen, Christian Kharif, Stephane Zaleski, and Jie Li. Two-dimensional navier–stokes simulation of breaking waves. *Physics of Fluids*, 11(1):121–133, 1999.

[CLHM97] Jim X. Chen, Niels da Vitoria Lobo, Charles E. Hughes, and J. Michael Moshell. Real-time fluid simulation in a dynamic virtual environment. *IEEE Comput. Graph. Appl.*, 17(3):52–61, May 1997.

[CLT08] Keenan Crane, Ignacio Llamas, and Sarah Tariq. Real-time simulation and rendering of 3d fluids. In *GPU Gems 3*, pages 633–675. Addison-Wesley, 2008.

[CM10] Nuttapong Chentanez and Matthias Müller. Real-time simulation of large bodies of water with small scale details. In *Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA)*, pages 197–206, 2010.

[CM11] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, pages 82:1–82:10, New York, NY, USA, 2011. ACM.

[Coo12] Eben Cook. Creating the Flood Effects in Uncharted 3. In *Game Developers Conference*, 2012.

[Cor13] Exa Corp. PowerFLOW. `http://www.exa.com/powerflow.html`, 2013. [Online as of February-2013].

[Del02] Paul J. Dellar. Nonhydrodynamic modes and *a priori* construction of shallow water lattice boltzmann equations. *Phys. Rev. E*, 65:036309, Feb 2002.

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: a new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 61–76, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[DGK$^+$02] Dominique D'Humières, Irina Ginzburg, Manfred Krafczyk, Pierre Lallemand, and Li-Shi Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Phil. Trans. R. Soc. A*, 360:437–451, 2002.

[DW07] Scott T. Davis and Chris Wyman. Interactive refractions with total internal reflection. In *Proceedings of Graphics Interface 2007*, GI '07, pages 185–190, New York, NY, USA, 2007. ACM.

[ED06] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH, 2006.

[ED08] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization and applications. In *Proc. of Graphics Interface (GI)*, volume 322 of *ACM International Conference Proceeding Series*, pages 73–80. Canadian Information Processing Society, 2008.

[EFFM02] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, November 2002.

[FAMO99] Ronald P. Fedkiw, Tariq Aslam, Barry Merriman, and Stanley Osher. A non-oscillatory eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *J. Comput. Phys.*, 152(2):457–492, July 1999.

[FDH$^+$87] Uriel Frisch, Dominique D'Humières, Brosl Hasslacher, Pierre Lallemand, Yves Pomeau, and Jeam-pierre Rivet. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1(4):649–707, 1987.

[FF01] 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. ACM, 2001.

[FHP86] Uriel Frisch, Brosl Hasslacher, and Yves Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Phys. Rev. Lett.*, 56:1505–1508, Apr 1986.

[FM96] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471 – 483, 1996.

[FM97a] Nick Foster and Dimitris Metaxas. Controlling fluid animation. In *Proceedings of the 1997 Conference on Computer Graphics International*, CGI '97, pages 178–, Washington, DC, USA, 1997. IEEE Computer Society.

[FM97b] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GDN98] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics. A Practical Introduction.* SIAM, 1998.

[GKT^{+}06] Sebastian Geller, Manfred Krafczyk, Jonas Tölke, Stefan Turek, and Jaroslav Hron. Benchmark computations based on lattice-Boltzmann, finite element and finite volume methods for laminar flows. *Computers & Fluids*, 35(8–9):888 – 897, 2006.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Mon.Not.Roy.Astron.Soc.*, 181:375, 1977.

[GOH12] Carlos Gonzalez-Ochoa and Doug Holder. Water Technology of Uncharted. In *Game Developers Conference*, 2012.

[GPMC91] Massimo Germano, Ugo Piomelli, Parviz Moin, and William H. Cabot. A dynamic subgrid-scale eddy viscosity model. *Physics of Fluids A: Fluid Dynamics*, 3(7):1760–1765, 1991.

[GRGT10] Markus Geveler, Dirk Ribbrock, Dominik Göddeke, and Stefan Turek. Lattice-boltzmann simulation of the shallow-water equations with fluid-structure interaction on multi- and many-core processors. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the multicore-challenge*, pages 92–104. Springer-Verlag, 2010.

[Gro13] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2013. [Online as of February-2013].

[GRZZ91] Andrew K. Gunstensen, Daniel H. Rothman, Stéphane Zaleski, and Gianluigi Zanetti. Lattice boltzmann model of immiscible fluids. *Phys. Rev. A*, 43:4320–4327, Apr 1991.

[GS03] Irina Ginzburg and Konrad Steiner. Lattice boltzmann model for free-surface flow and its application to filling process in casting. *Journal of Computational Physics*, 185(1):61 – 99, 2003.

[GSC04] Juan Guardado and Daniel Sánchez-Crespo. Rendering water caustics. In *GPU Gems*, pages 31–44. Addison-Wesley, 2004.

[GSMA08] Diego Gutierrez, Francisco J. Seron, Adolfo Munoz, and Oscar Anson. Visualizing underwater ocean optics. *Computer Graphics Forum*, 27(2):547–556, 2008.

[GSSP10] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages

55–64, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[HA07] X.Y. Hu and N.A. Adams. An incompressible multi-phase sph method. *Journal of Computational Physics*, 227(1):264 – 278, 2007.

[Har04] Mark J. Harris. Fast fluid dynamics simulation on the gpu. In *GPU Gems*, pages 637–666. Addison-Wesley, 2004.

[HB13] Jared Hoberock and Nathan Bell. Thrust - Parallel Algorithms Library. `http://thrust.github.com/`, 2013. [Online as of February-2013].

[HdPP76] J. Hardy, O. de Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Phys. Rev. A*, 13:1949–1961, May 1976.

[HJ89] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9(7):663, 1989.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Proc. of Computer Graphics International*, CGI '07, pages 63–70, 2007.

[HL97a] Xiaoyi He and Li-Shi Luo. Lattice boltzmann model for the incompressible navier–stokes equation. *Journal of Statistical Physics*, 88:927–944, 1997.

[HL97b] Xiaoyi He and Li-Shi Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Phys. Rev. E*, 56:6811–6817, Dec 1997.

[HLD96] Xiaoyi He, Li-Shi Luo, and Micah Dembo. Some progress in lattice Boltzmann method. Part I. Nonuniform mesh grids. *Journal of Computational Physics*, 129(2):357 – 363, 1996.

[HN81] C.W Hirt and B.D Nichols. Volume of fluid (vof) method for the dynamics of free boundaries. *Journal of Computational Physics*, 39(1):201 – 225, 1981.

[HQ07] Wei Hu and Kaihuai Qin. Interactive approximate rendering of reflections, refractions, and caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):46–57, January 2007.

[HS89] William Hibbard and David Santek. Interactivity is the key. In *Proceedings of the 1989 Chapel Hill workshop on Volume* *visualization*, VVS '89, pages 39–43, New York, NY, USA, 1989. ACM.

[HSCD96] S. Hou, J. Sterling, S. Chen, and G. D. Doolen. A Lattice Boltzmann Subgrid Model for High Reynolds Number Flows. *Fields Institute Communications*, 6:151–166, 1996.

[HW65] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[IGLF06] Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 805–811, New York, NY, USA, 2006. ACM.

[IKLH04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. Volume rendering techniques. In *GPU Gems*, pages 667–692. Addison-Wesley, 2004.

[IYO95] Takaji Inamuro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Physics of Fluids*, 7(12):2928–2930, 1995.

[JCKS02] Henrik W. Jensen, Per H. Christensen, Toshiaki Kato, and Frank Suykens. A practical guide to global illumination using photon mapping. In *SIGGRAPH 2002 Course Notes*, 2002.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[KM90] 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, New York, NY, USA, 1990. ACM.

[KO96] S. Koshizuka and Y. Oka. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear Science and Engineering*, 123(3):421–434, 1996.

[KTL03] Manfred Krafczyk, Jonas Tölke, and Li-Shi Luo. Large-eddy simulations with a multiple-relaxation-time lbe model. *International Journal of Modern Physics B*, 17:33–39, 2003.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[Lab13] Stanford University Graphics Lab. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/`, 2013. [Online as of February-2013].

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[LCM$^+$08] Jonas Lätt, Bastien Chopard, Orestis Malaspinas, Michel Deville, and Andreas Michler. Straight velocity boundaries in the lattice Boltzmann method. *Phys. Rev. E*, 77:056703, May 2008.

[LD11] Gábor Liktor and Carsten Dachsbacher. Real-time volume caustics with adaptive beam tracing. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 47–54, New York, NY, USA, 2011. ACM.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, July 1990.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 457–462, New York, NY, USA, 2004. ACM.

[LH10] Hyokwang Lee and Soonhung Han. Solving the shallow water equations using 2d sph particles for interactive applications. *The Visual Computer*, 26:865–872, 2010.

[LL00] Pierre Lallemand and Li-Shi Luo. Theory of the lattice boltzmann method: Dispersion, dissipation, isotropy, galilean invariance, and stability. *Phys. Rev. E*, 61:6546–6562, Jun 2000.

[Lot11] Timothy Lottes. Fxaa. Technical report, Nvidia, 2011.

[LTKF08] Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, July 2008.

[Luc77] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024, December 1977.

[LvdP02] Anita T. Layton and Michiel van de Panne. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*, 18:41–53, 2002.

[LWK03] Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19:444–456, 2003.

[MB96] Nelson Max and Barry Becker. Flow Visualization Using Moving Textures. In David C. Banks, Tom W. Crockett, and K. Stacy, editors, *Proceedings of the ICAS/LaRC Symposium on Visualizing Time-Varying Data*, NASA Conference Publication 3321, pages 77–87, 1996.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[MLS99] Renwei Mei, Li-Shi Luo, and Wei Shyy. An accurate curved boundary treatment in the lattice Boltzmann method. *J. Comput. Phys.*, 155(2):307–330, November 1999.

[MMS04] Viorel Mihalef, Dimitris Metaxas, and Mark Sussman. Animation and control of breaking waves. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pages 315–324. Eurographics Association, 2004.

[MSD07] Matthias Müller, Simon Schirm, and Stephan Duthaler. Screen space meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '07, pages 9–15, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[MZ88] Guy R. McNamara and Gianluigi Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Phys. Rev. Lett.*, 61:2332–2335, Nov 1988.

[NCGB95] D. R. Noble, S. Chen, J. G. Georgiadis, and R. O. Buckius. A consistent hydrodynamic boundary condition for the lattice Boltzmann method. *Physics of Fluids*, 7(1):203–209, 1995.

[NS92] Francesca Nannelli and Sauro Succi. The lattice Boltzmann equation on irregular lattices. *Journal of Statistical Physics*, 68:401–407, 1992. 10.1007/BF01341755.

[Nvi11] Nvidia. *Nvidia CUDA C Programming Guide*, 4.1 edition, 2011.

[Nvi13] Nvidia. Parallel Programming and Computing Platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2013. [Online as of February-2013].

[NY06] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854 – 879, 2006.

[OH95] J. F. O'Brien and J. K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings of the Computer Animation*, CA '95, pages 198–. IEEE Computer Society, 1995.

[OKTR11] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. A new approach to the lattice boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628 – 3638, 2011.

[OS88] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, November 1988.

[OS09] Jesús Ojeda and Antonio Susín. Aceleración de simulaciones de fluidos Lattice Boltzmann utilizando CUDA. In *II Workshop de Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones*, 2009. isbn:978-84-692-7320-3.

[OS13a] Jesús Ojeda and Antonio Susín. Enhanced lattice boltzmann shallow waters for real-time fluid simulations. In *Eurographics 2013, accepted*, 2013.

[OS13b] Jesús Ojeda and Antonio Susín. Hybrid particle lattice boltzmann shallow water for interactive fluid simulations. In *Proc. of the 8th International Conference on Computer Graphics Theory and Applications (GRAPP), accepted*, 2013.

[OS13c] Jesús Ojeda and Antonio Susín. Real-time rendering of enhanced shallow water fluid simulations. *Preprint*, 2013.

[oVRPL13] Persistence of Vision Raytracer Pty. Ltd. Pov-ray. `http://www.povray.org/`, 2013. [Online as of February-2013].

[Per85] Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, pages 287–296, New York, NY, USA, 1985. ACM.

[Per02] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, 2002. ACM.

[PH10] Matt Pharr and Greg Humphreys. *Physically based Rendering.* Morgan Kaufman, 2010.

[Pix13] Pixar. Pixar's renderman. `http://renderman.pixar.com/view/renderman`, 2013. [Online as of February-2013].

[PKW$^+$03] Thomas Pohl, Markus Kowarschik, Jens Wilke, Klaus Iglberger, and Ulrich Rüde. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. *Parallel Processing Letters*, pages 549–560, 2003.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 155–162, New York, NY, USA, 2005. ACM.

[Poh08] Thomas Pohl. *High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method.* PhD thesis, Technischen Fakultät der Universität Erlangen-Nürnberg, 2008.

[PTB$^+$03] Simon Premžoe, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross T. Whitaker. Particle-based simulation of fluids. *Computer Graphics Forum*, 22(3):401–410, 2003.

[QDL92] Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479, 1992.

[RGW$^+$03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the symposium on Data visualisation 2003*, VISSYM '03, pages 231–238. Eurographics Association, 2003.

[RSP07] Eric Risser, Musawir Shah, and Sumanta Pattanaik. Faster relief mapping using the secant method. *Journal of Graphics, GPU, and Game Tools*, 12(3):17–24, 2007.

[Sal99] R. Salmon. The lattice boltzmann method as a basis for ocean circulation modeling. *Journal of Marine Research*, 57(3):503–535, 1999.

[SBC$^+$11] Barbara Solenthaler, Peter Bucher, Nuttapong Chentanez, Matthias Müller, and Markus Gross. SPH Based Shallow Water Simulation. In Jan Bender, Kenny Erleben, and Eric Galin, editors, *VRIPHYS 11: 8th Workshop on Virtual Reality Interactions and Physical Simulations*, pages 39–46, Lyon, France, 2011. Eurographics Association.

[Sch94] Christophe Schlick. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[Sch97] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 151–162, London, UK, UK, 1997. Springer-Verlag.

[Sch10] Martin Schreiber. GPU based simulation and visualization of fluids with free surfaces. Diploma thesis, Technische Universitat München, Fakultät für Informatik, June 2010.

[SF95] Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 129–136, New York, NY, USA, 1995. ACM.

[SFK$^+$08] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35(2-3):350–371, June 2008.

[Sim13] Game Physics Simulation. Game Physics Simulation. `http://bulletphysics.org/`, 2013. [Online as of February-2013].

[SKALP05] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3):695–704, 2005.

[Sko93] P. A. Skordos. Initial and boundary conditions for the lattice Boltzmann method. *Phys. Rev. E*, 48:4824–4842, 1993.

[SKP07] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, March 2007.

[SL03] Songdong Shao and Edmond Y.M. Lo. Incompressible sph method for simulating newtonian and non-newtonian flows with a free surface. *Advances in Water Resources*, 26(7):787 – 800, 2003.

[Sma63] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91:99–164, 1963.

[Sou05] Tiago Sousa. Generic refraction simulation. In *GPU Gems 2*, pages 295–305. Addison-Wesley, 2005.

[SP09] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 40:1–40:6, New York, NY, USA, 2009. ACM.

[Sta96] Jos Stam. Random caustics: natural textures and wave theory revisited. In *ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*, SIGGRAPH '96, pages 150–, New York, NY, USA, 1996. ACM.

[Sta99] 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, 1999. ACM Press/Addison-Wesley Publishing Co.

[Sta03] Jos Stam. Flows on surfaces of arbitrary topology. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 724–731, New York, NY, USA, 2003. ACM.

[Suá06] Núria Suárez. *Coupling Marker and Cell and Smoothed Particle Hydrodynamics for Fluid Animation.* PhD thesis, Universitat Politècnica de Catalunya, 2006.

[Sus03] Mark Sussman. A second order coupled level set and volume-of-fluid method for computing growth and collapse of vapor bubbles. *Journal of Computational Physics*, 187(1):110 – 136, 2003.

[SZ99] Ruben Scardovelli and Stéphane Zaleski. Direct numerical simulation of free-surface and interfacial flow. *Annual Review of Fluid Mechanics*, 31(1):567–603, 1999.

[Tec13] Next Limit Technologies. Realflow. `http://www.realflow.com/`, 2013. [Online as of February-2013].

[Tes01] Jerry Tessendorf. Simulating ocean water. In *SIGGRAPH 2001 Course Notes*, 2001.

[TFK$^+$03] Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki. Realistic animation of fluid with splash and foam. *Computer Graphics Forum*, 22(3):391–400, 2003.

[Thü07] Nils Thürey. *Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method.* PhD thesis, Technischen Fakultät der Universität Erlangen-Nürnberg, Mar 2007.

[TMFSG07] Nils Thürey, Matthias Muller-Fischer, Simon Schirm, and Markus Gross. Real-time breaking waves for shallow water simulations. In *Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on*, pages 39 –46, 2007.

[TR04] Nils Thürey and U. Rüde. Free surface lattice-boltzmann fluid simulations with and without level sets. *Proc. of Vision, Modelling, and Visualization VMV*, pages 199–207, 2004.

[TSB07] Guido Thömmes, Mohammed Seaïd, and Mapundi K. Banda. Lattice boltzmann methods for shallow water flow applications. *International Journal for Numerical Methods in Fluids*, 55(7):673–692, 2007.

[Tub10] Kevin Tubbs. *Lattice Boltzmann Modeling for Shallow Water Equations using high performance computing.* PhD thesis, Louisiana State University, 2010.

[Tö10] Jonas Tölke. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Computing and Visualization in Science*, 13:29–39, 2010.

[vdLGS09] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 91–98, New York, NY, USA, 2009. ACM.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[vKvdBT08] Kees van Kooten, Gino van den Bergen, and Alex Telea. Point-based visualization of metaballs on a gpu. In *GPU Gems 3*, pages 123–148. Addison-Wesley, 2008.

[Vla10] Alex Vlachos. Water Flow in Portal 2. In *SIGGRAPH Course on Advances in Real-Time Rendering in 3D Graphics and Games*, 2010.

[WD06] Chris Wyman and Scott Davis. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 153–160, New York, NY, USA, 2006. ACM.

[WG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction.* Springer, 2000.

[WH94] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 269–277, New York, NY, USA, 1994. ACM.

[WLMK04] Xiaoming Wei, Wei Li, K. Mueller, and A.E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *Visualization and Computer Graphics, IEEE Transactions on*, 10(2):164 –176, march-april 2004.

[WN09] Chris Wyman and Greg Nichols. Adaptive caustic maps using deferred shading. *Computer Graphics Forum*, 28(2):309–318, 2009.

[Wol86] Stephen Wolfram. Cellular automaton fluids 1: Basic theory. *Journal of Statistical Physics*, 45:471–526, 1986.

[Wym05] Chris Wyman. An approximate image-space approach for interactive refraction. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 1050–1053, New York, NY, USA, 2005. ACM.

[Wym08] Chris Wyman. Hierarchical caustic maps. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 163–171, New York, NY, USA, 2008. ACM.

[YHK07] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Trans. Graph.*, 26(3), July 2007.

[YK09] Cem Yuksel and John Keyser. Fast Real-time Caustics from Height Fields. *The Visual Computer (Proceedings of CGI 2009)*, 25(5-7):559–564, 2009.

[ZB05] Yongning Zhu and Robert Bridson. Animating sand as a fluid. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 965–972, New York, NY, USA, 2005. ACM.

[ZH97] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591–1598, 1997.

[Zho02] J.G. Zhou. A lattice boltzmann model for the shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 191(32):3527–3539, 2002.

[Zho04] J.G. Zhou. *Lattice Boltzmann Methods for Shallow Water Flows.* Springer, 2004.

[Zho11] J.G. Zhou. Enhancement of the labswe for shallow water flows. *Journal of Computational Physics*, 230(2):394 – 401, 2011.

[Zie93] Donald P. Ziegler. Boundary conditions for lattice Boltzmann simulations. *Journal of Statistical Physics*, 71:1171–1177, 1993.