Politecnico di Milano
Scuola di Ingegneria dell'Informazione
Corso di laurea specialistica in Ingegneria dell'Automazione

Accurate real-time fluid dynamics using Smoothed-Particle Hydrodynamics and CUDA

Relatore: Prof. Pier Luca LANZI
Correlatore: Prof. Davide MANCA

Tesi di Laurea di:
Michele PIROVANO Matr. 749324

Abstract

This thesis concerns the generation of physically and visually realistic simulations of fluids, to be used in interactive virtual reality environments.

The goals of real-time rendering and physically realistic simulation of fluids are not easy to achieve alone and providing both at once can be a great challenge, due to their contrasting nature.

In order to achieve a good trade-off between performance and accuracy, we extend a Smoothed-Particle Hydrodynamics approach for fluid modeling. An extensive review of many of the techniques proposed in the years by several authors is also provided in this thesis, producing a comprehensive state-of-the-art view of the method.

Our SPH model for the simulation of multiple fluids is implemented both as a serial program and as a parallel program, achieving real-time frame rates thanks to the power of modern day's Graphical Processing Units.

We provide solutions to the classic SPH formulation's weaknesses by extending known solutions. In addition, new solutions for tensile instability, free surface flow spurious surface tension and boundary handling are proposed.

The provided SPH model is validated by comparison with an analytical physically realistic model on a water jet test case. Our SPH implementation is fast, versatile and accurate and provides a realistic fluid behavior for many different situations.

Sommario

Lo scopo di questa tesi è ottenere una simulazione fisicamente e visivamente realistica di fluidi da usare per ambienti interattivi di realtà virtuale.

Gli obiettivi di visualizzazione in tempo reale e di simulazione realistica dei fluidi sono difficili da raggiungere e provvedere ad entrambi è una grande sfida per via della loro natura contrastante.

Per ottenere un buon compromesso tra accuratezza e velocità di calcolo, estendiamo un approccio basato su Smoothed-Particle Hydrodynamics per la formulazione del modello dei fluidi. Molte delle tecniche proposte negli anni da diversi autori sono presentate in questa tesi, proponendo una visione comprensiva dello stato dell'arte del metodo SPH.

Il nostro modello SPH per la simulazione di fluidi multipli è implementato attraverso un programma seriale ed uno parallelo, ottenendo grazie alla potenza delle schede grafiche moderne computazioni in tempo reale.

Presentiamo soluzioni per affrontare le debolezze della formulazione SPH classica estendendo soluzioni conosciute. Nuove soluzioni per i problemi di *tensile instability*, tensione superficiale fittizia e modellazione delle frontiere del dominio fisico sono proposte.

Il modello SPH è validato comparandone l'accuratezza con un modello analitico e fisicamente realistico attraverso l'analisi di un caso di test appropriato. La nostra implementazione SPH è veloce, versatile ed accurata e produce un comportamento realistico del fluido per molte situazioni differenti.

a mia madre e mio padre,
a cui devo tutto ciò che sono

Acknowledgments

This work would not have been the same without any of the people that were close to me or helped during my work.

Thanks to professor Pier Luca Lanzi for his ever-present support and interest in my work and for the many opportunities he promptly gave me, I did not know before that professors could be so kind.

Thanks to professor Davide Manca for his help, his questions and his observations which spurred my interest in this work and thanks to doctor Roberto Totaro for his rapidity in providing help when needed.

Thanks to D. K. Allister and M. Müller for their kind help in providing directions for my work.

Thanks to my family, because none of this would have been possible without their support.

Thanks to Federica for the many hours we spend together that always make me forget all the problems and the work that await.

Thanks to my friends, for they still let me talk endlessly without smacking me down.

Thanks to Priscilla, Eddie, Frattaglia, Kami and Sandy, because the endless hours working alone would not have been bearable without them jumping on the desk or running around my legs.

List of Figures

List of Tables

List of Symbols

t	Time	15
v	Velocity	15
x	Position	20
a	Acceleration	29
f	Force	22
g	Gravity acceleration	28
A	Generic fluid quantity	15
V	Total fluid volume	20
N	Total number of particles	21
n_{avg}	Average number of particles	79
M	Total fluid mass	23
m	Particle mass	21
ρ	Density	21
ρ_0	Rest density	25
$\tilde{\rho}$	Adapted density	59
δ	Number density	54
P	Pressure	22
k_P	Pressure stiffness	25
P^{near}	Near pressure	53
$\tilde{P}$	Adapted pressure	60
c	Speed of sound inside the fluid	25
μ	Viscosity coefficient	22
Π	Artificial viscosity	27
k_{el}	Elastic stiffness	29
dL_0	Spring rest length	29
C^V	Color value	61
$\vec{n}$	Surface normal	61
κ	Curvature	61
σ	Surface tension coefficient	61
K_{coll}	Collision stiffness	66
x_R	Rigid body position	87
v_R	Rigid body linear velocity	87

List of Source Codes

Contents

Chapter 1

Introduction

The computational simulation of fluids is a topic that has been gaining much interest in the recent years. From blood flowing in vessels and arteries to waves crashing on coasts, from the swirling of water around the hull of a ship to flames engulfing a wooden object, computational fluid simulation reveals to be useful and has been successfully put to good use for many problems in classic physics as well as in astrophysics, civil, coastal and aerospace engineering, weather prediction, surgery and graphics.

The study of fluid dynamics is one of the few branches of science in which many problems have yet to be addressed and many questions are still looking for answers. Suffice to say that the main unsolved mystery of applied mechanics is the problem of fluid turbulence, described as one of the seven Millennium Prize Problems by the Clay Mathematics Institute. In contrast to the classic rigid body mechanics or even to the dynamics of non-rigid solids, the comprehension of the motion of fluids is difficult to grasp, let alone master.

Another important problem regards speed, because fluids must be simulated as a continuum and as such the computational power required even for a simple simulation is way greater than, for example, in the case of rigid body simulation. However, due to the computational power provided by recent computers and to their fast-growing nature, what once was almost impossible to reproduce swiftly can nowadays be simulated even at real-time speeds with physically realistic results.

This achievement is most important in the graphic field where photo-realistic fluids are sought after for interactive applications, when up to a few years ago they were just the prerogative of special effects in movies. Video games, virtual realities and immersive and interactive applications for many purposes, such as chemical industrial training, military training or aerospace pilot training, can benefit from the increased speed and much research has

been recently done on these fields.

The pursue of the goal of reproducing *visually realistic* fluids was once performed from a graphics point of view by simulating fictitious fluids in order to achieve the desired visual effect. Nowadays, researchers agree that, in order to achieve a realistic visual effect, realistic physics must be simulated and as such the field of graphics fluid visualization has been converging towards the field of physically realistic fluid simulation, called Computational Fluid Dynamics (CFD).

1.1 Visually realistic fluid simulation

This thesis aims at simulating fluids in a three-dimensional virtual reality ambient in a physically and visually realistic way. In particular, the virtual environment we considered as an example is a life-sized reconstruction of a chemical plant used as a training ambient for field operators, inside of which incidents involving chemicals are simulated.

Usually, in this situation as well as in video games and other interactive applications, the simulation of fluids is usually implemented using simple particle systems, with no physical meaning underneath the visual effect. Smoke, water and flame simulation for real-time applications has historically been done with these particle systems, which are quite simple to implement and fast. For example, in all the current virtual environment, water is represented as a flow of particles falling to the ground and splashing on the surface, without any physical parameter involved. The aim of this work is to introduce realistic fluid animation in such environments and, because of this, a more physically realistic approach must be taken.

Accordingly, this work had three main objectives.

First, we aim to achieve a physically realistic and accurate simulation. As the simulator is used in a virtual reality environment focused on accidents occurring in chemical plants, it is important to pursue realistic behavior of the fluids in order to match what would happen in reality. This is the main focus of this work.

Second, real-time frame rates are needed. Due to the nature of interactive virtual environments, real-time simulation is a necessity, as the users are expected to react instantaneously to the simulated environment and watch the consequences of their actions.

Third, a visually realistic simulation is pursued. A perfect simulation of the fluid would not feel realistic at all if the exterior, *i.e.* the visual representation of it, would not live up to the expectations. In order to achieve a virtual reality effect, appropriate rendering methods should be chosen.

1.2 Computational Fluid Dynamics

CFD aims at accurately simulating the motion of fluids so that different tests can be performed on models and realistic data can be extrapolated from the simulations.

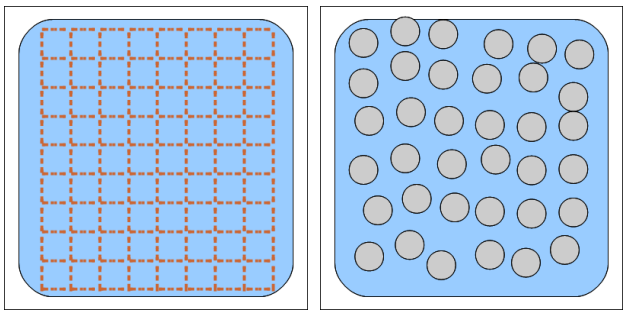
Since the early 1990s, this branch of physics has been growing and has seen spawning different methods for the simulation of fluids, based on the many advancements in physics theory. The most successful methods have been built upon forms of the *Navier-Stokes equations*, a mathematical model describing the motion of fluids created in 1822.

Historically, computational methods based on the Navier-Stokes equations have been divided in two major forms: Eulerian methods and Lagrangian methods. Eulerian methods are distinguished from the fact that the discretization of the fluid is done following a fixed spatial grid and all quantities inside the fluid are considered at fixed locations (see figure 1.1a). This family of methods has a longer history and has been thoroughly explored by scholars who have obtained many successes, such as the discovery of an universally stable method by Stam [1999]. Nonetheless, this method has its flaws, mainly because of its constrained nature, the difficulty to correctly conserve mass and the difficulty to simulate interactions with other bodies or with user input. Using a Lagrangian method, on the other hand, the fluid is discretized at points inside the fluid that move with it, following the velocity field by traveling with the flow. The fluid is in this case best described by moving particles (see figure 1.1b). This approach has several benefits. First of all, by abandoning the fixed grid, the method is unconstrained and is well suited for simulating fluids that need to move in free space. In addition, these methods allow for an easy implementation of interactions with other bodies and user interaction. Conservation of mass is also quite simple to implement, in contrast to Eulerian methods. At last, Lagrangian methods can be fast to compute and this quality has become more and more important. Lagrangian fluid simulation methods are, however, quite new and still present many open problems, among them are stability and visualization.

The difference between the two approaches can be viewed in mathematical terms by considering the rate of change of a quantity A inside the fluid during its evolution:

$$\frac{DA}{Dt} = \frac{\delta A}{\delta t} + A(\mathbf{v} \cdot \nabla) \tag{1.1}$$

In Eulerian methods, the right hand side is used, while in Lagrangian methods the left hand side is used, which is called *material derivative* taken along a path moving with velocity v. This difference simplifies the equations used



(a) Eulerian method (b) Lagrangian method

Figure 1.1: Computational Fluid Dynamics approaches

in Lagrangian methods enormously. In this work, the simple derivative of quantities dA/dt is used instead of the material derivative DA/Dt for simplicity of notation, as usually done in the literature.

1.3 Our approach to fluid modeling

In this work, we chose to implement a fully Lagrangian model so that no insuperable constraint on the motion of the fluid would have to be imposed and so that the fluid could interact with other bodies and with the user with ease.

Among Lagrangian methods we opted for Smoothed-Particle Hydrodynamics (SPH), that is the most widely used method and on which major research has been done in the last twenty years. The method was designed for compressible fluids, but it has been extended to behave correctly even for incompressible fluids. The method is still quite new and many problems have yet to be addressed, but its usefulness has been proven in different aspect of engineering.

Our SPH implementation fulfills all our three objectives: physical accuracy, real-time frame rates and visual realism. Since SPH is based on a discretization of the Navier-Stokes equations and the accuracy of the simulation is easily scaled by changing the number of particles of the simulation, our physical realism goal is fulfilled. Successful implementations of SPH for interactive applications have been reported in the last years and we provide our parallel version, running on a Graphics Processing Unit (GPU). The increase of speed granted by the parallelization of the SPH algorithms is huge and it allows us to create a real-time application with more than enough particles for most cases. At last, due to the popularity of SPH and other particle methods among graphics experts, many techniques for the visual-

ization of clouds of particles have been developed in the recent years, with different grades of complexity and realistic visual results. We choose to focus on simple point-based implementations for performance purposes.

As the SPH method is still young, many techniques have been and are still being proposed as this thesis is written. Due to this, literature is not always unanimous on the paths to follow in order to achieve a perfect simulation and thus it can be hard to find a state-of-the-art presentation of the method. We have thus conducted research on an SPH implementation suited to our needs, reviewing many of the current techniques proposed in the literature and adding new contributions to the international research.

1.4 Previous work

Previous work related to the subject of this thesis is here detailed, ranging from particle systems to SPH formulations and to meshless surface visualization.

The first formulation of particle systems is due to Reeves [1983] who published his work on a method for fuzzy objects modeling as an alternative to the classic vertex-based 3D modeling, in an attempt to improve movie special effects. Following this work, many people extended the simple formulation to achieve higher results in the graphics field, such as advanced particle systems (Lander [1998], Burg [2000], Ilmonen and Kontkanen [2003]), inter-particle forces (Miller [1989]), the creation of an API for particle systems (McAllister [2000]) and the goal of million-sized particle systems (Sims [1990], Latta [2004]). Other researchers applied the formulations to different fields, such as animal behavior (Reynolds [1987]), plant modeling (Reeves and Blau [1985]) and, shortly after, such as fluid simulation.

Particle systems and fluid rendering have been an important topic in the graphics field due to their usefulness in the making of movies and video games, with contributions by many different authors for the rendering of smoke (Foster and Metaxas [1997], Fedkiw et al. [2001]), flames (Beaudoin et al. [2001], Wei et al. [2002], Nguyen et al. [2002], James [2003]), clouds (Horng-Shyang et al. [2004]) and liquids (Bourke [1997], Müller et al. [2003], Keiser et al. [2005], Müller et al. [2007], Crane et al. [2007]). The famous marching cube algorithm has also been used as well (Lorensen and Cline [1987]).

After initial approaches using particle systems for physical modeling (Miller [1989]), Smoothed-Particle Hydrodynamics was born in the field of astrophysics as a method for simulating stars formation and matter aggregation,

thanks to the efforts of Lucy [1977] and Gingold and Monaghan [1977]. It was not until the early 1990s, however, that researches started considering the method for fluid simulation. In this field, the first formulation comes from Monaghan [1992] where he introduces the method and its applications on density, pressure and energy equations, as well as a fictitious viscosity force. Following this article, he later published work on the free surface flow of fluids (Monaghan et al. [1994]), with test cases that demonstrated how the method could be useful in the field of fluid simulation. Several publications followed, using the SPH method for modeling different fluids (Morris et al. [1997]), analyzing its weaknesses and strengths and addressing problems like tensile instability (Monaghan [2000]) or increasing efficiency with neighbor lists (Anderson [1993]).

In 2003, Müller published an article on the utilization of SPH in interactive applications (Müller et al. [2003]), which earned the method much fame due to the possibility to render realistic fluids in real-time. After this article, SPH had gained increasing interest among worldwide researchers and comparisons with more classical methods have been made (Agertz [2008]).

Spurious surface tension effects have been studied and controllable models have been proposed by different authors (Morris [1999], Grenier et al. [2008], Becker and Teschner [2007], Zhou et al. [2008]), with the first extensive formulation of the problem coming from Hoover [1998]. Solenthaler and Pajarola [2008] propose a new adapted SPH method that removes spurious tensions from multi-fluid simulations.

Methods for modeling the boundaries of SPH fluid simulations have been initially proposed by Monaghan [1992], but many advancements have been done on the subject with several approaches based on particle-based boundaries (Valizadeh et al. [2008], Crespo et al. [2007]), on direct or indirect forcing (Müller et al. [2004], Yildiz et al. [2009], Becker et al. [2009]) or on permeable boundaries (Lastiwka et al. [2008]).

In the last years, SPH's main problems have been addressed and the method has now evolved into a more mature state. SPH, born as a method for compressible flows, has been extended in the recent years for incompressible simulations thanks to the efforts of researchers like Khayyer et al. [2008], Bao et al. [2009], Solenthaler and Pajarola [2008] and Ihmsen et al. [2010]. Other phenomena have been simulated with SPH approaches, such as viscoelasticity and plasticity (Clavet et al. [2005]). Melting transitions have been implemented in the work of Losasso et al. [2006] and Keiser et al. [2005], while granular materials are simulated in the work of Bell et al. [2005] and ocean waves in the work of Capone [2009].

Still, many open problems persist and researches keep giving their answers, often different and sometimes incompatible, so that no state of the art

SPH can be yet agreed upon. The once standard formulations are now just a simple base on which authors have built different implementations, often mixing SPH with other methods, such as in the case of the moving particle semi-implicit method (Koshizuka and Oka [1996]), moving least-square extensions (Brownlee et al. [2007]) or level set hybrids (Chentanezg and Müller [2010]).

1.5 Thesis organization

This thesis is organized as follows.

In chapter 2, the classic SPH model is presented and its equations are explained. Different variations on the formalism of SPH are presented, the fluid dynamics equations are solved, integration algorithms are explained and compared and suitable kernel functions are detailed.

In chapter 3, we present a more complex SPH model, with extensions that aim to solve the shortcomings of the classic approach. The tensile instability, spurious surface tension and fluid compressibility problems are addressed and our solutions are proposed. Surface tension models and boundary methods are explained in detail. A rigid body coupling method is proposed.

In chapter 4, the implementation of the SPH model is addressed, detailing our sequential CPU code and parallel GPU code, the latter providing real-time frame rates. Algorithms for peculiar necessities are explained in detail and our simulator, able to sustain multiple different fluids, is presented. The rendering phase of the simulation is also addressed.

In chapter 5, the result we have obtained by making use of the implemented SPH model are presented. A test case involving a water jet is detailed and our SPH model is validated against a known analytical model. The performance of the parallel code is compared to the serial implementation. Different tests performed by making use of our SPH model are discussed.

In chapter 6, conclusions on the usefulness and versatility of our SPH model are presented and future contributions for research in this area of knowledge are proposed.

Chapter 2

Smoothed-Particle Hydrodynamics

Smoothed-Particle Hydrodynamics (SPH) is a meshless Lagrangian method for fluid simulation and its initial formulation for fluid dynamics is due to Monaghan [1992]. SPH is based on the *discretization* of the characteristics of the fluid as a continuum in discrete points, called *particles*, that represent infinitesimal volumes of the fluid and move according to the flow's velocity field. Particles carry individual properties along the fluid that are smoothed in the volume around the single particles, hence the name of the method.

This chapter illustrates the mathematical SPH model and briefly overviews the state-of-art of the classic model.

2.1 Basic formalism

Given a fluid, the value of a property A in a point x_a. $A(x_a)$, is written as convolution of the quantity itself $A(x)$, for all points x in the volume V, and Dirac's delta function $\delta_{dirac}(r)$ evaluated on the distance $r = |x_a - x|$ between the two points. The function $\delta_{dirac}(r)$ is zero everywhere except in the origin, where it is equal to one.

Thus, $A(x_a)$ can be expanded as:

$$A(x_a) = \int_V A(x)\delta_{dirac}(r)dx \tag{2.1}$$

This equation and the resulting spatial derivatives of $A(x_a)$ are computationally expansive and as such they must be approximated.

To compute equation 2.1, two approximations are therefore introduced.

First, the function $\delta_{dirac}(r)$ is approximated by a kernel function $W(r, h)$,

where h is called *limited support* and that means that $W(r,h)$ evaluates to zero when the distance r between the two points is greater than or equal to h:

$$A(x_a) \approx \int_V A(x)W(r,h)dx \tag{2.2}$$

The kernel function $W(r,h)$ converges to the function $\delta_{dirac}(r)$ when h tends to zero.

Second, the volume V in equation 2.2 is discretized into a limited number N of particles, where each particle b represents a small volume V_b of fluid, so that the integral can be approximated as a summation on all particles. Therefore, the quantity $A(x_a)$ in a point in space x_a is evaluated as an interpolation on all particles:

$$\begin{array}{rcl} A(x_a) & \approx & \sum_b^N A(x_b)V_bW(r_{ab},h) \\ & \approx & \sum_b^N A_bV_bW_{ab} \end{array} \tag{2.3}$$

Where r_{ab} represents $|x_a - x_b|$, A_b represents $A(x_b)$ and W_{ab} represents $W(r_{ab},h)$.

The error in approximating equation 2.1 with 2.3 is $O(h^2)$ and it depends on the particle disposition inside the fluid, with more orderly dispositions being more accurate, as quoted from Monaghan [1992]. The approximation in equation 2.3 provides two important benefits. First, since $A(x_a)$ depends only on the particles in its proximity a huge speed up can be achieved by considering in the summation only those particles whose distance from the point x_a is less than the smoothing length h. Second, the spatial derivatives of $A(x_a)$ only require the derivation of the kernel function $W(r,h)$, which is analytically known. In fact, for the first spatial derivative, the gradient, we obtain the following equation by considering A_b and V_b constant with respect to space:

$$\nabla A_a \approx \sum_b A_b \frac{m_b}{\rho_b} \nabla W(r_{ab},h) = \sum_b A_b \frac{m_b}{\rho_b} \nabla W_{ab} \tag{2.4}$$

Where A_a represents $A(x_a)$ and V_b has been written as $\frac{m_b}{\rho_b}$, with m_b being the mass of a particle and ρ_b its density. We refer to equation 2.4 as the *base gradient* formulation. Similarly, the second spatial derivative, the Laplacian, has the following form:

$$\nabla^2 A_a \approx \sum_b A_b \frac{m_b}{\rho_b} \nabla^2 W(r_{ab},h) = \sum_b A_b \frac{m_b}{\rho_b} \nabla^2 W_{ab} \tag{2.5}$$

Equation 2.4 and 2.5 introduce accuracy errors that can be avoided by rewriting the same equations with the density ρ placed inside the differential oper-

ators. This has the effect of achieving a higher-order approximation (Monaghan [1992]), this effect is referred as the *Second Golden Rule of SPH*.

In order to limit the aforementioned accuracy errors, new formulations for the computation of spatial derivatives of $A(x_a)$ have been proposed. A first different formulation for equation 2.4 can be achieved by rewriting ∇A_a as:

$$\nabla A_a \approx \frac{1}{\rho_a} \sum_b (A_b - A_a) m_b \nabla W_{ab} \tag{2.6}$$

Where ρ_a is the density of particle a. We refer to this equation as the *difference gradient*. Apart from reducing the accuracy error of the gradient computation (Monaghan [1992]), equation 2.6 has an additional advantage over the base gradient of equation 2.4, if A is constant, the gradient will be zero everywhere, which is not true for the base gradient form. This form tends to behave incorrectly, however, if the number of particles N is low.

Another formulation can be obtained by rewriting ∇A_a as:

$$\nabla A_a \approx \rho_a \sum_b (\frac{A_b}{\rho_b^2} + \frac{A_a}{\rho_a^2}) m_b \nabla W_{ab} \tag{2.7}$$

We refer to this equation as the *summation gradient*. This formulation still has the limitation of not imposing a zero gradient for constant A, but it has a higher accuracy than the original formulation 2.4 (Monaghan [1992]) while maintaining a good behavior for a small number of particles.

Both equation 2.6 and 2.7 work on the assumption that the density is not constant inside the fluid. This holds if the fluid that is simulated is not completely incompressible, which is the usual case for classic SPH.

2.2 Solving Navier-Stokes with SPH

A SPH approach makes it easy to solve the Navier-Stokes equations of fluid motion in order to provide realistic physics simulations.

The first equation that is needed is the *momentum equation*, which is basically Newton's second law for fluids:

$$\rho \frac{dv}{dt} = -\nabla P + \mu \nabla^2 v + f^{ext} = f^{press} + f^{viscos} + f^{ext} \tag{2.8}$$

Where P is the fluid's pressure and μ is its viscosity coefficient. This equation ties the rate of change of the fluid velocity $\frac{dv}{dt}$ to all the forces that act on it. These forces can be divided in pressure forces f^{press}, viscosity forces f^{viscos} and external forces f^{ext}. More accurately, the forces should be called *force*

densities, due to their dimension being $\frac{N}{m^3}$, but we refer to them simply as forces as usually done in the literature.

The second equation that will be used is the *continuity equation* and it imposes the conservation of mass in the fluid.

$$\frac{d\rho}{dt} + \rho(\nabla \cdot v) = 0 \tag{2.9}$$

By solving equation 2.8 and 2.9 at each time step for every particle using the SPH formulations of section 2.1, the accelerations of all particles are computed and the simulation can be evolved by integrating the resulting momentum equation and the simple relationship between each particle's velocity v and position x, usually by the application of a simple forward Euler algorithm. The quantities needed for the simulation must be computed in sequence, due to their interdependence. First, the density ρ of each particle must be computed, then its pressure P, then the pressure, viscosity and external forces can be computed.

2.2.1 Density computation

It has already been mentioned in section 2.1 how any quantity in a point inside the fluid, especially at particle positions, can be interpolated through the use of SPH. The same approximation can be used to compute the first needed quantity, the density ρ_a of each particle a at position x_a.

Historically, density has been computed simply by applying the SPH interpolant 2.3 for each particle.

$$\begin{aligned} \rho_a &= \sum_b^N \rho_b V_b W_{ab} \\ &= \sum_b^N m_b W_{ab} \end{aligned} \tag{2.10}$$

Where V_b is m_b/ρ_b.

This approach renders the continuity equation 2.9 unnecessary because, as a consequence:

$$\int_V \rho(x)dx = \sum_b m_b = M \tag{2.11}$$

Where M is the total mass of the fluid, thus achieving mass conservation. This simple expedient puts the SPH methods above many Eulerian approaches in terms of mass conservation, as it is often hard to maintain it for low resolutions in a fixed grid. We consider the mass of a single particle m_b to be constant for each particle in the same fluid.

Equation 2.10 highlights however one of SPH main problems: the incorrect approximation and thus inaccurate interpolation that appears at the

fluid interface. This effect is called *spurious surface tension* as it applies a fictitious force that tends to minimize the curvature of the whole fluid. Many authors do not address this problem as they pursue a semi-realistic reproduction of liquids, mainly water, which is known to have surface tension effects, and thus are content with the effect. In contrast, as the aim of this work is to have a more realistic and controllable simulation, a surface tension which is only the effect of an incorrect formulation must be eliminated. A detailed discussion of this problem and solutions are written in section 3.4.1.

Another approach for density computation that extends the classic SPH formulation in an attempt to solve the spurious surface tension problem is based on the analysis of the continuity equation (2.9). By applying the difference form of the gradient (equation 2.6) to the continuity equation (2.9), a new equation can be derived:

$$\frac{d\rho}{dt} = -\rho(\nabla \cdot v)$$

$$\frac{d\rho_a}{dt} = \sum_b m_b(v_a - v_b)\nabla W_{ab} \tag{2.12}$$

This equation is usually preferred to the density summation equation (2.10) both because it is thought to eliminate the particle deficiency at the interface and both because the fluid is more easily initialized, since all particles are created with their density equal to the nominal rest density of the fluid ρ_0.

However, Hoover [1998] has observed that this method is not capable of completely eliminating the spurious surface tension effect. In addition, although the initialization phase is more stable and easier to implement, the interactions between particles can be physically inconsistent as density is computed only when their velocities diverge. Thus, if a fluid is initialized incorrectly and left to evolve, the particles will not tend to change their initial situation, erroneously maintaining their initial density.

2.2.2 Equation of state for pressure

After computing the density of each particle, the corresponding pressure can be obtained from it. In order to do so, authors have been employing *equations of state* which are used to bind the properties of the fluid together, specifically to compute pressure as a function of density. As the equation of state is arbitrary, different choices can be made according to simulation needs.

For example, a first equation of state can be derived from the ideal gas state equation, which provides correct behavior for gases and compressible

fluids, by assuming a constant temperature:

$$P_a = k_P \ \rho_a \tag{2.13}$$

Where ρ_a is the particle's density and P_a its pressure. k_P is the pressure stiffness coefficient and has the dimension $[\frac{Nm}{kg}]$ or, equally, $[\frac{m^2}{s^2}]$.

A modification of equation 2.13 for liquids has been presented by Desbrun and Cani [1996] in order for particles to reach a constant non zero density ρ_0 when the fluid is at rest:

$$P_a = k_P(\rho_a - \rho_0) \tag{2.14}$$

The coefficient k_P can also be viewed as the squared speed of sound in the fluid c^2. For example, water has a speed of sound of around $c = 1500\ m/s$, which makes it a nearly-incompressible fluid. Since the coefficient k_P for water simulations is therefore in the order of millions, this can lead to instabilities during the integration phase. Accordingly, several authors have been obliged to use speed of sounds of orders of magnitudes lower than reality. This fact has always been one of the most discussed problems of SPH implementations that aim to achieve realism.

In order to simulate nearly-incompressible fluids, the use of Tait's equation 2.15 has been suggested as an alternative equation of state by Monaghan et al. [1994]:

$$P_a = \frac{\rho_0 k_P}{7}((\frac{\rho_a}{\rho_0})^7 - 1) \tag{2.15}$$

This equation allows for a greater degree of incompressibility as the pressure is more sensitive to density fluctuation. Integration algorithms do not suffer much from the stiffness of the equation and near-incompressibility can be reached even with a lower speed of sound.

As is further discussed in section 3.2, in order to achieve incompressibility for fluids with high speeds of sound different equations must be used, dropping the computation of pressure values with an equation of state.

As suggested by Monaghan et al. [1994], for free surface flows or for fierce impacts a numerical correction on particle velocities should be implemented:

$$\Delta x_a = \Delta t \sum_b \frac{\eta(v_b - v_a)}{\rho_b - \rho_a} m_b \nabla W_{ab} \tag{2.16}$$

Where η is an adimensional coefficient chosen in the range $[0, 1]$ and Δt is the simulation time step.

This correction is applied only to the final positions of the particles and it has the effect of averaging their velocities according to surrounding particles.

As such, the resulting disposition of particles is smoother and calculations become more accurate because of the orderly disposition.

2.2.3 Pressure term

The SPH approach is applied to the pressure term, obtaining an equation for the computation of pressure forces:

$$f_a^{press} = -\nabla P_a = -\sum_b P_b \frac{m_b}{\rho_b} \nabla W_{ab} \tag{2.17}$$

This term makes sure that the density fluctuations of the particles are small by adding repulsive and attractive forces between neighboring particles based on their distance and the difference between their densities. Therefore, if a particle's density ρ_a is lower than the designated rest density ρ_0, the particle will tend to attract the surrounding particles in an attempt to rise its density. *Vice versa*, if a particle's density is higher than the rest density, it will repulse the surrounding particles.

The standard formulation in equation 2.17 is problematic as if the pressure is constant in the whole volume of the fluid the gradient will be non zero. We apply the Second Golden Rule of SPH for spatial derivatives (see section 2.1) and use the summation form (equation 2.7) that is chosen over the difference form (equation 2.6) as it behaves better for a small number of particles.

$$f_a^{press} = -\rho_a \sum_b (\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2}) m_b \nabla W_{ab} \tag{2.18}$$

In addition, as proposed by Müller et al. [2003], another less computationally expansive formulation can be used by making the assumption that the density is roughly constant thoroughly the fluid (as in the case of a weakly compressible or incompressible fluid) and by using the mean of the two pressures:

$$f_a^{press} = -\sum_b (\frac{P_b + P_a}{2\rho_b}) m_b \nabla W_{ab} \tag{2.19}$$

As a consequence of applying the SPH formulation to pressure equations, a numerical artifact appears that is called in the literature *tensile instability*. This effect can be observed as particles tend to form clusters by strongly pulling a few close particles in order to reach their rest density instead of pulling more particles with less force, causing numerical instabilities. This problem has been addressed by several authors and different solutions can be found in literature. This work provides a solution to this problem in section 3.1.

2.2.4 Viscosity term

The fluid's viscosity term in the momentum equation (2.8) can be viewed as a dissipative term, smoothing the fluid's velocity field. This term has been formulated in the literature using the SPH approach in different ways, according to the degree of realism that was meant to be achieved.

In Monaghan's first formulation, the viscosity forces are created by means of an artificial viscosity term Π_{ab} added to the pressure computations (Monaghan [1992]). By extending equation 2.18, the following equation can be created:

$$f_a^{press} + f_a^{visc} = \rho_a \sum_b (\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2} + \Pi_{ab}) m_b \nabla W_{ab} \tag{2.20}$$

Where the artificial viscosity coefficient Π_{ab} is given by:

$$\Pi_{ab} = \frac{-\alpha(\bar{c}_{ab})\mu_{ab} + \beta\mu_{ab}^2}{\bar{\rho}_{ab}} \tag{2.21}$$

Where the constants α and β are respectively 1 and 2 and ϵ is a small constant used to prevent singularities, usually chosen as 0.01. $\bar{c}_{ab}$ and $\bar{\rho}_{ab}$ are the mean sound of speed and density of particles a and b. This computation of equation 2.21 is performed only if $(v_b - v_a) \cdot (x_b - x_a) < 0$, in order to make sure that the viscosity forces are dissipative. The viscosity coefficient μ_{ab} is given by:

$$\mu_{ab} = \frac{h(v_b - v_a) \cdot (x_b - x_a)}{(x_b - x_a)^2 + \epsilon^2} \tag{2.22}$$

Although controllable and providing conservation of angular momentum, this formulation is still considered artificial and thus more realistic formulations are preferred.

Morris et al. [1997] propose a new viscosity equation that is more realistic, based on a hybrid SPH and finite difference approximation:

$$f_a^{visc} = \sum_b \frac{m_b(\mu_a + \mu_b)}{\rho_b}(v_b - v_a)\nabla W_{ab} \tag{2.23}$$

As observed by Morris, this equation is best suited for low-speed flows.

Another interpretation is brought to us by Müller et al. [2003] by applying the SPH formulation to the laplacian of the velocity field in the momentum

equation 2.8.

$$f_a^{visc} = \mu_a \nabla^2 v = \mu_a \sum_b (v_b - v_a) \frac{m_b}{\rho_b} \nabla^2 W_{ab} \tag{2.24}$$

Although good enough for Müller's objectives, the Laplacian form is very sensitive to particle disorder and as such the accuracy can be lower than that of Morris' equation 2.23. However, the computation performance of this equation is higher as the Laplacian of the kernel function is usually simpler to compute than its gradient.

As suggested by several authors, the pressure and viscosity computations can be joined into a single step, since they do not depend on each other, thus allowing the performance to be improved.

2.2.5 External forces

External forces acting on the fluid do not need particular formulations as they can be added to particles as an additive term without trouble. This quality of the SPH approach allows gravity and other constant forces as well as different imposed forces such as wind or user-controlled interactions to be implemented with ease as follows:

$$f_a^{ext} = f^{grav} + f^{other}(t) = \rho_a g + f^{other}(t) \tag{2.25}$$

Where g is the constant gravity acceleration. User-controlled forces can thus be easily added as force fields, so that each particle can obtain the external force acting on it by checking its position.

2.2.6 Elasticity

Additional forces can be easily added to the classic Navier-Stokes momentum equation (2.8), such as elasticity forces. In order to achieve many different effects and to have a suitable simulation for any material, in the work related to this thesis the viscoelastic behavior as observed in some denser fluids such as blood or polymers has been implemented. The implementation is based on the work of Clavet et al. [2005]. Imaginary springs between each pair of close particles are first created when the fluid is initialized, setting the rest length dL_0 of the springs as their initial distance. At each time step, the elastic forces between all particles are then computed as a function of their distance $r_{ab} = x_a - x_b$, the fluid's elastic stiffness k_{el} and the individual spring's rest length dL_0 as in equation 2.26. Only half of the force is computed for particle

a as the other half is added to the other particle b.

$$f_a^{elastic} = -\frac{1}{2}(r_{ab} - dL_0)k_{el} \tag{2.26}$$

2.3 Integration

Since the simulation must be executed at discrete time steps, an integration algorithm is needed. The equations we need to integrate take the following form thanks to the SPH approach explained so far:

$$\begin{cases} \rho\frac{dv}{dt} = f(x)^{press} + f(x,v)^{visc} + f^{external} \\ \frac{dx}{dt} = v \end{cases} \tag{2.27}$$

Or, by explicating the acceleration $a(x, v)$, which is therefore known at each time step:

$$\begin{cases} \rho\frac{dv}{dt} = a(x,v) \\ \frac{dx}{dt} = v \end{cases} \tag{2.28}$$

The aim is to solve the equations in 2.28 by advancing the velocity and position of every particle at each time step, taking into account the computation cost of the chosen integration algorithm and its stability.

In the following comparison, we refer to some of the algorithms as *symplectic*. This term comes from molecular dynamics and it means that the integration algorithm conserves the energy of the system exactly. This property is very useful for the problem at hand because we do not want the energy of the fluid to increase or decrease due to numerical inconsistencies.

As noted by previous authors, the focus will be on lower order integration algorithms, with an approach similar to molecular dynamics, since the computational cost of evaluating higher order algorithms for many particles can become a serious bottleneck for simulation performance. Because of this, the widely used fourth order Runge-Kutta algorithm as well as Gears method or other higher order algorithm have not been considered. The choice to not use the fourth order Runge-Kutta algorithm could be regarded as a strange one, since it is still quite performant, however the method does not have a symplectic behavior and can thus be safely ignored.

In literature concerning SPH, a consensus on the integration algorithm best suited for the computation has not been reached and, in fact, several authors just use a simple forward Euler algorithm with small time steps. We are instead compelled to pursue interactivity and, as such, the time steps we work with must be larger. On the other hand, the fluids we are likely to simulate,

water *in primis*, have high speeds of sound and rest densities and this, by using an equation of state, results in an incredibly rigid behavior of the fluid which in turn requires smaller timesteps. A trade-off between performance and stability must thus be found. In order to address this problem, we have produced a comparison between the algorithms used in different publications and we have chosen the best suited for our needs. Since the law for position and velocity advancement, using the Navier-Stokes equations, can be viewed as a damped oscillator, comparisons are made by applying the algorithms to the integration of the motion of a suitable example system (see sections 2.3.2 and 2.3.3).

2.3.1 Algorithms

We first introduce all the algorithms that have been considered and detail their proprieties. In the following equations, we refer to x, v and a as the position, velocity and acceleration of a single particle during the integration from time t to time $t + \Delta t$, referred to as $t + 1$ for simplicity's sake. Acceleration is computed through the SPH approach described in section 2.2 as a function $f(x, v)$ of velocity and speed. The equations are solved for each particle in the fluid at each time step.

Forward Euler

The forward Euler algorithm is the simplest integration algorithm that can be considered. It has first order accuracy and an extremely low computational cost and as such it is the most performant algorithm. The algorithm takes the following form:

$$\begin{aligned} a_{t+1} &= f(x_t, v_t) \\ x_{t+1} &= x_t + v_t \Delta t \\ v_{t+1} &= v_t + a_{t+1} \Delta t \end{aligned} \tag{2.29}$$

Semi-implicit Euler

By simply swapping the velocity update with the position update a new method can be obtained that, although so similar to the forward Euler method, gains two important benefits: this method is symplectic and it provides a much greater stability than equation 2.29.

$$\begin{aligned} a_{t+1} &= f(x_t, v_t) \\ v_{t+1} &= v_t + a_{t+1} \Delta t \\ x_{t+1} &= x_t + v_{t+1} \Delta t \end{aligned} \tag{2.30}$$

Verlet

In molecular dynamics, the Verlet algorithm and its variations are often used due to their symplectic nature and their great stability. The particular form of the position update comes from the third order Taylor expansion of positions at times $t + \Delta t$ and $t - \Delta t$:

$$x_{t+1} = x_t + v_t dt + \frac{1}{2} a_t dt^2 + \frac{1}{6} b_t dt^3 + O(dt^4)$$

$$x_{t-1} = x_t - v_t dt + \frac{1}{2} a_t dt^2 - \frac{1}{6} b_t dt^3 + O(dt^4)$$

Where b_t is the *jerk*, the third time derivative of position. By adding the two expressions together, a new integration algorithm can be obtained:

$$\begin{aligned} a_{t+1} &= f(x_t, v_t) \\ x_{t+1} &= 2x_t - x_{t-1} + a_t \Delta t^2 \\ v_{t+1} &= (x_{t+1} - x_{t-1})/(2\Delta t) \end{aligned} \tag{2.31}$$

Where the velocity is actually computed with a delay. The algorithm has a global error of $O(\Delta t^2)$ both for position and velocity and a local error of $O(\Delta t^4)$ for position and of $O(\Delta t^2)$ for velocity. It is thus regarded as a second order accurate algorithm.

The drawbacks of the method lie in the need for the positions of the last two steps for the computation of the new position, which also means that the algorithm is not self-starting, and in the delay of the velocity.

Velocity Verlet

By modifying the Verlet method we can remove the method's first weakness, its not self-starting behavior, incorporating the velocity update in the computations. Accuracy remains of the second order for both position and velocity.

$$\begin{aligned} x_{t+1} &= x_t + v_t \Delta t + \tfrac{a_t}{2} \Delta t^2 \\ a_{t+1} &= f(x_{t+1}, v_t) \\ v_{t+1} &= v_t + \tfrac{a_{t+1} + a_{t-1}}{2} \Delta t \end{aligned} \tag{2.32}$$

Corrected Velocity Verlet

A modification of the Velocity Verlet reveals to be necessary for damped oscillation and uses a predictor-corrector approach. This adds a step and thus decreases performance, but the increase in stability that can be gained

makes up for it.

$$\begin{aligned} v_{t+1} &= v_t + \tfrac{a_t}{2}\Delta t \\ x_{t+1} &= x_t + v_{t+1}\Delta t \\ a_{t+1} &= f(x_{t+1}, v_{t+1})\Delta t \\ v_{t+1} &= v_t + \tfrac{a_{t+1}}{2}\Delta t \end{aligned} \tag{2.33}$$

Leapfrog

Leapfrog is an integration algorithm that is very similar to the Velocity Verlet algorithms, but updates velocity at half timesteps. The name comes from the fact that positions and velocities are computed at interleaved time points. The method is often used in SPH implementations due to its good accuracy and performance. The accuracy for position is the same as for the Verlet algorithms, but it is $O(\Delta t^3)$ for velocity.

$$\begin{aligned} x_{t+1} &= x_t + v_{t-1/2}\Delta t \\ a_{t+1} &= f(x_{t+1}, v_{t+1}) \\ v_{t+1/2} &= v_{t-1/2} + a_{t+1} \end{aligned} \tag{2.34}$$

The velocity at the current time can be computed, once again with a delay, as:

$$v_t = \frac{v_{t-1/2} + v_{t+1/2}}{2}$$

Clavet

The method explained in Clavet's paper (Clavet et al. [2005]) has much in common with the Leapfrog method, as written by the author himself. This method is however more costly, depending on the number of different forces acting on the fluid, but it provides greater stability due to a prediction-relaxation approach, close in concept to an implicit algorithm. A quality of this method is that it can easily support time-varying timesteps. In the algorithm, the steps are iterated for N forces, so that $n = 1, 2, \ldots, N$. We define x_* and a_* as the intermediate position and acceleration during the computations.

$$\begin{aligned} &x_*^0 = x_t + v_{t-1/2}\Delta t \\ &a_*^n = f(x_*^n, v_{t-1/2}) \\ &x_*^{n+1} = x_*^n + a_*^n \Delta t^2 \\ &x_{t+1} = x_*^N \\ &v_{t+1/2} = (x_{t+1} - x_t)/\Delta t \end{aligned} \tag{2.35}$$

Beeman

As presented by Capone [2009], Beeman's algorithm provides fourth order accuracy for position and third order accuracy for velocity.

$$\begin{aligned} x_{t+1} &= x_t + v_t\Delta t + (\tfrac{2}{3}a_t - \tfrac{1}{6}a_{t-1})\Delta t^2 \\ a_{t+1} &= f(x_{t+1}, v_t) \\ v_{t+1} &= v_t + (\tfrac{1}{3}a_{t+1} + \tfrac{5}{6}a_t + \tfrac{1}{6}a_{t-1})\Delta t \end{aligned} \tag{2.36}$$

Velocity Corrected Beeman

A new formulation of Beeman's algorithm can be found, again coming from Capone [2009], which assures fourth order accurate velocity. A velocity prediction-correction step is added. We refer to this version as the Velocity Corrected (VC) Beeman algorithm.

$$\begin{aligned} x_{t+1} &= x_t + v_t\Delta t + (\tfrac{2}{3}a_t - \tfrac{1}{6}a_{t-1})\Delta t^2 \\ v_{t+1} &= v_t + (\tfrac{3}{2}a_t - \tfrac{1}{2}a_{t-1})\Delta t \\ a_{t+1} &= f(x_{t+1}, v_{t+1}) \\ v_{t+1} &= v_t + (\tfrac{1}{3}a_{t+1} + \tfrac{5}{6}a_t - \tfrac{1}{6}a_{t-1})\Delta t \end{aligned} \tag{2.37}$$

Position Corrected Beeman

Capone also introduces a new version of the algorithm, with a prediction-correction approach for both velocity and position. We refer to this version as the Position Corrected (PC) Beeman algorithm. The accuracy is again of the fourth order for both position and velocity.

$$\begin{aligned} x_{t+1} &= x_t + v_t dt + (\tfrac{2}{3}a_t - \tfrac{1}{6}a_{t-1})\Delta t^2 \\ v_{t+1} &= v_t + (\tfrac{3}{2}a_t - \tfrac{1}{2}a_{t-1})\Delta t \\ a_{t+1} &= f(x_{t+1}, v_{t+1}) \\ x_{t+1} &= x_t + v_t\Delta t + (\tfrac{1}{6}a_{t+1} - \tfrac{1}{3}a_t)\Delta t^2 \\ v_{t+1} &= v_t + (\tfrac{5}{12}a_{t+1} + \tfrac{8}{12}a_t - \tfrac{1}{12}a_{t-1})\Delta t \end{aligned} \tag{2.38}$$

2.3.2 Undamped harmonic oscillator

Our objective is to compare the algorithms' stability and their symplectic behavior, as our focus is both on the conservation of energy inside the system and on a stable simulation. Clavet's algorithm 2.35 is not considered in the comparison and instead is assumed to behave like the Leapfrog algorithm 2.34, as mentioned by the author (Clavet et al. [2005]).

As a first test, the algorithms have been compared by integrating the motion of a well-known phenomena, the undamped harmonic oscillator. The

motion can be viewed as the movement of a point of mass M connected with a spring with stiffness K and rest length L_0 to a fixed ground. See figure 2.1.

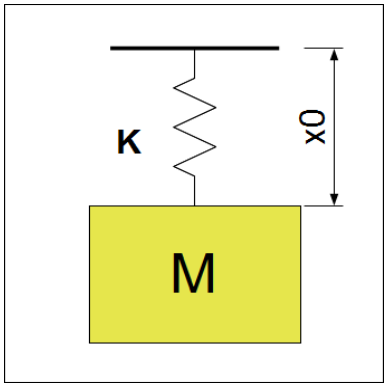


Figure 2.1: Undamped oscillator

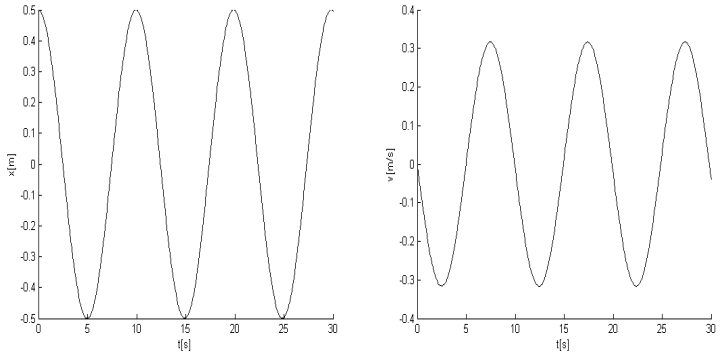
The position is initialized at point x_0 and the point of mass is known, in ideal conditions, to oscillate indefinitely with a frequency $\omega = \sqrt{\frac{K}{M}}$. The acceleration of this motion can be found as being $a(t) = -\omega^2 x(t)$ and the analytical expressions of the position and the velocity relative to time are known and are as follows.

$$\begin{aligned} x(t) &= A\cos(\omega t) \\ v(t) &= -A\omega\sin(\omega t) \end{aligned} \tag{2.39}$$

Where we have defined the amplitude $A = x_0$. For this comparison, the parameters have been chosen as $M = 250\,kg$, $x_0 = x(0) = 0.5\,m$, $\dot{x}(0) = 0\,m$. The stiffness of the system will be varied in order to compare the behavior of the algorithms.

By advancing the analytical expressions for 30 seconds with $K = 100\,\frac{N}{m}$, the periodic motion is obtained as can be seen in figure 2.2a (for the position of the motion) and 2.2b (for the velocity of the motion).

The different algorithms listed in section 2.3.1 for the integration of the motion have been used with a timestep of $h = 0.1s$ and $K = 100$, 10000 and 100000 $\frac{N}{m}$. For the position, the following behavior is obtained. As we can see in figure 2.3, simple Euler is not symplectic and diverges even for low stiffness, so we discard it. By increasing the stiffness of the spring to $K = 10000$, we see in figure 2.4 that many of the algorithms still behave correctly while the Position Corrected Beeman and the Leapfrog algorithms must be removed as they increase the energy of the motion. For $K = 100000$ the Verlet and Semi-implicit Euler start increasing the energy of the simulation and are thus removed. An additional increase of the stiffness over the current value



(a) Analytical position (b) Analytical velocity

Figure 2.2: Undamped oscillator for K=100: analytical solution

determines the instability of all the algorithms, decreeing the Velocity Verlet, Beeman and Position Corrected Beeman algorithms the best for position stability, as can be seen in figure 2.5 where the three solutions are overlying.

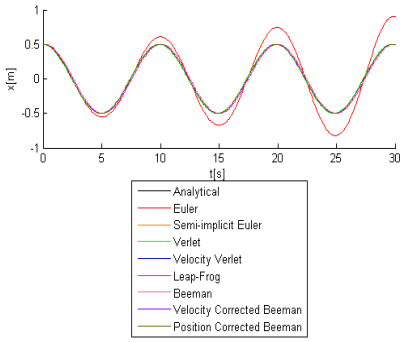


Figure 2.3: Position integration for K=100: undamped oscillator

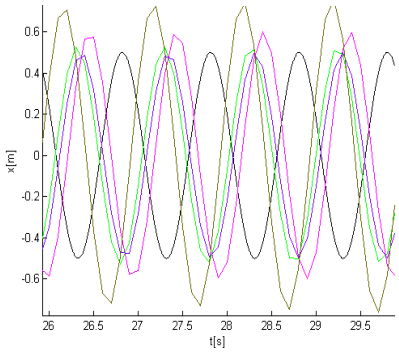


Figure 2.4: Position integration for K=10000: undamped oscillator

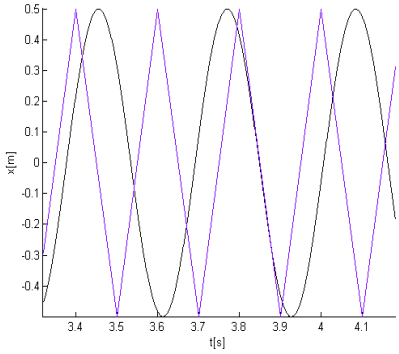


Figure 2.5: Position integration for K=100000: undamped oscillator

For velocity integration, we obtain the following behavior. Again, the Euler algorithm already behaves incorrectly at low stiffness, as can be seen in figure 2.6. Increasing the stiffness to $K = 10000$, once again the Position Corrected Beeman and the Leapfrog algorithms must be removed, as they are seen increasing the energy of the motion (see figure 2.7). At $K = 100000$ the Semi-Implicit Euler algorithms diverges, in contrast with its behavior for position, and it must be removed as well. The Velocity Verlet algorithm decreases the amount of energy of the system as the stiffness reaches $K = 100000$ and we are left with the Standard and Velocity Corrected Beeman algorithms alongside the Verlet algorithm, which however is seen slightly increasing the energy of the system. The results can be seen in figure 2.8.

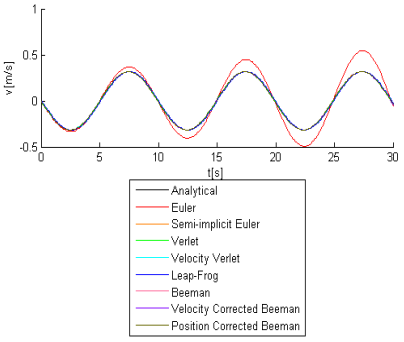


Figure 2.6: Velocity integration for K=100: undamped oscillator

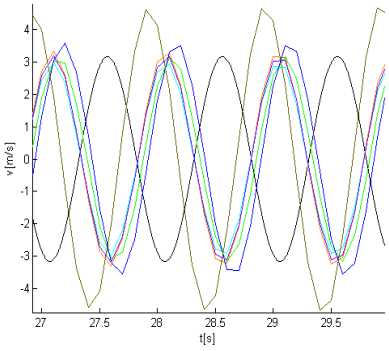


Figure 2.7: Velocity integration for K=10000: undamped oscillator

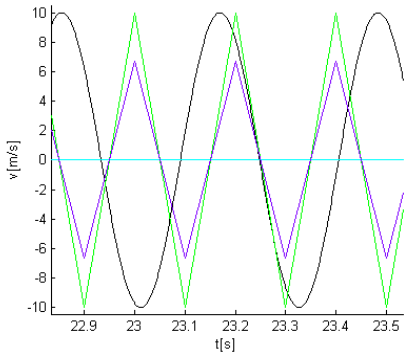


Figure 2.8: Velocity integration for K=100000: undamped oscillator

2.3.3 Damped oscillator

The SPH model can be more correctly viewed as a damped system since viscous forces act in order to decrease the energy of the system. Because of this, it is better to compare the algorithms based on their stability in integrating the motion of a dampened harmonic oscillator by adding to the simple undamped system of section 2.3.2 a damping coefficient $C = 2\sqrt{KM}$. See figure 2.9.

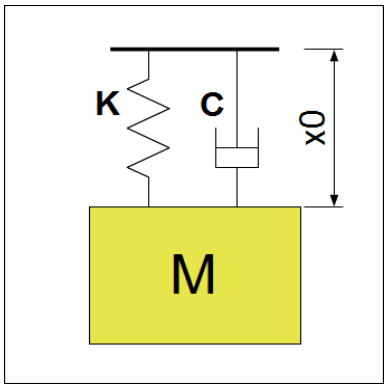


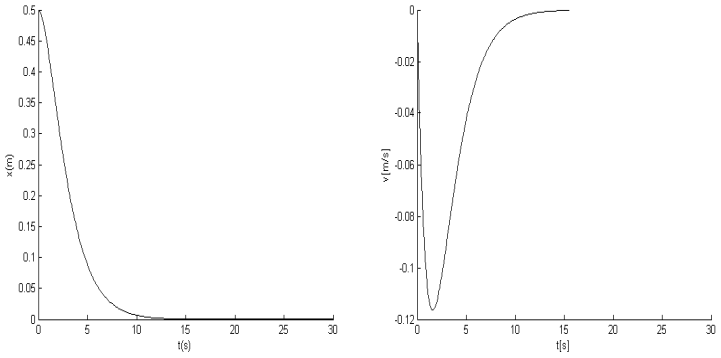
Figure 2.9: Damped oscillator

In this case, we consider the same parameters as for the undamped oscillator. The equations of the motion become as follows, with an acceleration of $a(t) = -\omega^2 x(t) - \frac{C}{M} v(t)$. We define $A = x_0$ and $B = v_0 + \omega x_0$.

$$\begin{aligned} x(t) &= (A + Bt)e^{-\omega t} \\ v(t) &= -(A\omega + B - B\omega t)e^{-\omega t} \end{aligned} \tag{2.40}$$

Once again the different algorithms of section 2.3.1 are compared for

multiple values of K. By advancing the analytical expressions for 30 seconds with $K = 100\ \frac{N}{m}$, the periodic motion is obtained as in figures 2.10a and 2.10b.



(a) Analytical position (b) Analytical velocity

Figure 2.10: Damped oscillator for K=100: analytical solution

The Modified Velocity Verlet algorithm is added in the comparison, as it reveals to be useful in the integration of this kind of motion.

With low stiffness, $K = 100$, all algorithms behave correctly, as seen in figure 2.11. As the stiffness is increased to $K = 10000$, all the Beeman variants become unstable and must be removed, leaving the algorithms as seen in figure 2.12. With $K = 15000$, the Verlet, Velocity Verlet and Leapfrog algorithms become unstable and are removed, the others remain stable as seen in figure 2.13. At $K = 20000$ even the Modified Velocity Verlet and the Semi-Implicit Euler algorithm become unstable, with the latter one diverging earlier as seen in figure 2.14, leaving only the simple Euler algorithm, which albeit being the worst for the undamped oscillation motion reveals to be the most stable for this comparison. The same behavior is obtained for the velocity comparison and as such it is not shown here.

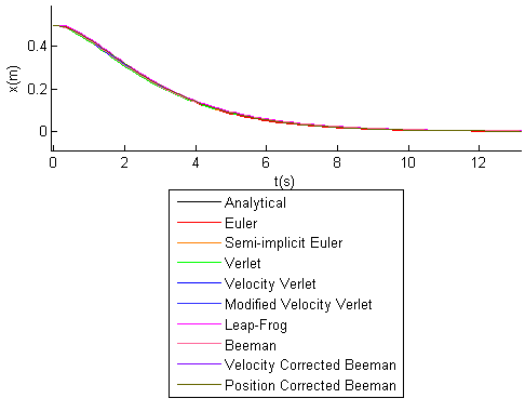


Figure 2.11: Position integration for K=100: damped oscillator

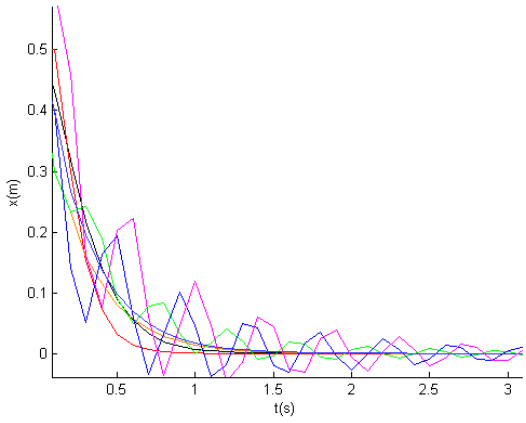


Figure 2.12: Position integration for K=10000: damped oscillator

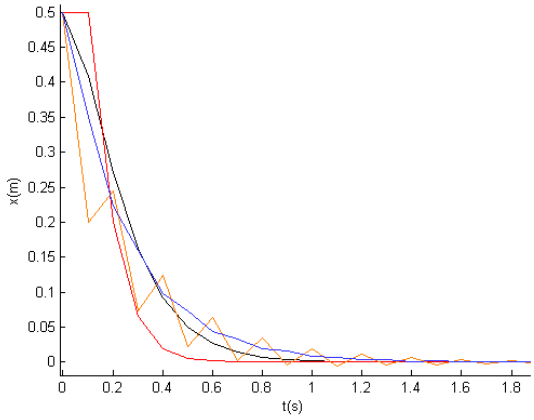


Figure 2.13: Position integration for K=15000: damped oscillator

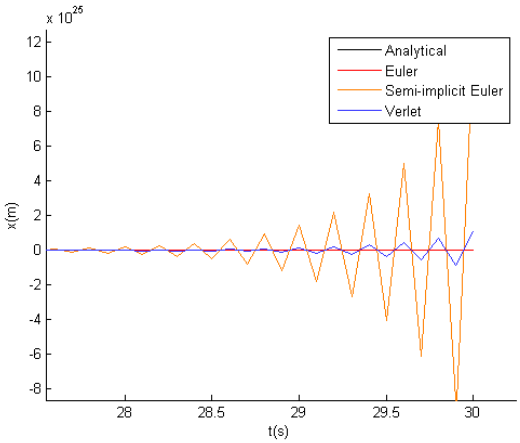


Figure 2.14: Position integration for K=20000: damped oscillator

2.3.4 Results

Retrieving the results of the last two sections and based on the knowledge on the proposed algorithms, they can be ordered in regards to their properties.

We order the algorithms in table 2.1 from worst to best in regards to their stability in the undamped position comparison (see section 2.3.2):

Euler
PC Beeman, Leapfrog
Verlet, Semi-Implicit Euler
Velocity Verlet, Beeman, VC Beeman

Table 2.1: Stability comparison: undamped position

We order the algorithms in table 2.2 from worst to best in regards to their stability in the undamped velocity comparison (see section 2.3.2):

Euler
PC Beeman, Leapfrog
Semi-Implicit Euler
Velocity Verlet
Verlet, Beeman, VC Beeman

Table 2.2: Stability comparison: damped velocity

We order the algorithms in table 2.3 from worst to best in regards to their stability in the damped position and velocity comparison (see section 2.3.3), taking into account that the behavior is identical for both position and velocity:

Beeman, PC Beeman, VC Beeman
Leapfrog, Verlet, Velocity Verlet
Semi-Implicit Euler
(Modified) Velocity Verlet
Euler

Table 2.3: Stability comparison: damped position and velocity

We also order the algorithms in table 2.4 from worst to best in regards to their accuracy (as detailed in section 2.3.1).

In conclusion, we have chosen to discard the Beeman algorithms due to their bad stability and non-symplectic behavior. The Euler and Semi-Implicit

1st order	Euler, Semi-Implicit Euler
4th order position, 2nd order velocity	Verlet, Velocity Verlet
4th order position, 3rd order velocity	Leapfrog, Beeman
4th order position, 4th order velocity	(Modified) Velocity Verlet

Table 2.4: Accuracy comparison

Euler are discarded as well because of their low accuracy order. With an eye on accuracy and performance alongside the stability requirement, in this thesis it has been decided to use the Modified Velocity Verlet algorithm 2.33 for most if not all simulations, as it is among the best algorithms in terms of stability, it is symplectic, self-starting, performant and it has a good order of accuracy.

In addition, when more accurate values are needed and performance can be allowed to decrease, which is the case of non-real-time simulations, the algorithm 2.35 proposed in the work of Clavet et al. [2005] is used due to its accuracy, similar to the Leapfrog method 2.34, and due to the increased stability given by its semi-implicit nature.

2.3.5 Adaptive timesteps

A larger time step allows the simulation to achieve higher performance at the expanse of accuracy and, depending on the integration algorithm, of stability. In order to achieve higher performance without renouncing to accuracy and stability, the time step can be in some cases modified as time advances.

A first method introducing variable time steps consists of fulfilling a *Courant-Friedrichs–Lewy* (CFL) condition, which is a necessary condition for convergence in solving certain partial differential equations numerically by discretization methods. The condition implies that the wavelength of a wave moving across a discrete grid must be less than the grid's length in order to observe the movement. In mathematical terms:

$$\frac{v \cdot \Delta t}{\Delta x} \leq C_{courant} \tag{2.41}$$

Where v is the velocity, Δt the time step, Δx the spatial grid length and $C_{courant}$ the Courant constant which depends on the particular problem.

In SPH, the length Δx is chosen as the smoothing length h of the system and the condition is usually considered as global, hence based on the maximum possible velocity of any particle during the simulation. This implies that the system dynamics must be known in advance or that the maximum velocity must be estimated at run time. Accordingly, the constant is often

paired with an adimensional factor in the range $[0, 1]$ in order to assure that more caution is taken. The time step is adapted, at each iteration, according to the condition:

$$\Delta t \leq C_{courant} \frac{h}{v_{max}}$$

Several authors add more constraints alongside the CFL condition, such as a modified CFL condition for acceleration:

$$\Delta t \leq C_{acc} \sqrt{\frac{h}{a_{max}}}$$

Both conditions have been implemented into our simulator for use with the Semi-Implicit Euler algorithm (2.30) and the Clavet algorithm (2.35). Whenever the CFL conditions are not fulfilled, the time step is lowered to the maximum possible time step that fulfills them. In addition, the user can choose that whenever this happens the simulation must be redone for the current step. If the condition is instead fulfilled, the time step is raised so that successive steps will be better performing. The time step is however limited by a lower and upper limit, hard coded into the simulator, to ensure a minimum degree of performance and stability.

2.4 Kernels

In this section, the properties of kernel functions and their uses are explained in detail.

As defined in section 2.1, the kernel function $W(r, h)$ depends on two parameters, the spatial coordinate r and the smoothing length h. The purpose of a kernel function is to approximate the Dirac's delta function δ_{dirac} in order to approximate the spatial distribution of a property inside the fluid. The first condition for a kernel function is thus:

$$W(r, h) \approx \delta_{dirac}(r)$$

Because of this, the kernel function $W(r, h)$ can be arbitrarily created as long as it fulfills three important conditions. In order to approximate the function δ_{dirac}, the kernel function must thus:

- Be normalized, so that its integral across all space sums up to 1.

$$\int_{-\infty}^{+\infty} W(r, h) = 1$$

- Converge to the Dirac's delta for $h \to 0$.

$$\lim_{h \to 0} W(r, h) = \delta_{dirac}(r)$$

- Be differentiable in space at least up to the second order, for the computation of the gradient and laplacian.

According to Monaghan [1992], a kernel with the latter characteristics will produce at least a second order accuracy in the solution.

In addition, two other conditions have to be considered:

- For physical accuracy, the function should be even, so that the principle of action-reaction is maintained.

$$W(r, h) = W(-r, h)$$

- The function should have limited support h, so that the interaction length can be cropped for a great increase in performance.

$$W(r, h) = 0 \quad for \quad |r| > h$$

It can be useful to know that the dimension of the kernel is in the order of $[\frac{1}{m^3}]$.

2.4.1 Kernel functions

In this section, several useful Kernel functions that have been used by authors in their SPH models are presented.

Gaussian function

Monaghan's first Golden Rule of SPH (Monaghan [1992]) says that if a physical explanation for the kernel must be found, then it should be considered as a Gaussian function:

$$W(r,h) = \frac{1}{h^3 \pi^{\frac{3}{2}}} e^{-\frac{r^2}{h^2}} \tag{2.42}$$

Where the prefix term is needed for normalization. The Gaussian function is differentiable infinite times, but it is computationally costly because of the exponential term and it has no limited support, which means that the function does not evaluate to zero for $r \geq h$. It is therefore only considered for its physical meaning.

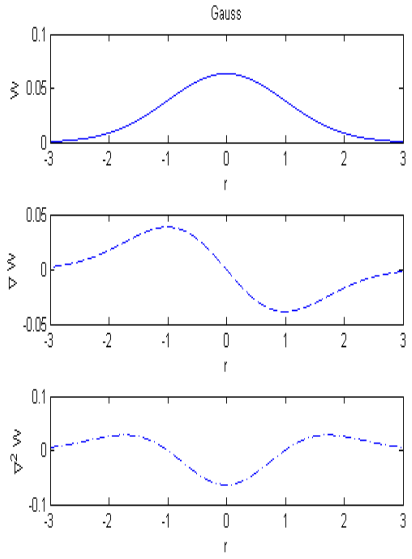


Figure 2.15: Gaussian kernel and its derivatives for h=1

Cubic spline

This function has been first proposed by Monaghan and Lattanzio [1985] as an approximation of the Gaussian function:

$$W(r,h) = \begin{cases} \frac{1}{\pi h^2}(1 - \frac{3}{2}\frac{r^2}{h} + \frac{3}{4}\frac{r^3}{h}) & \text{if } \frac{r}{h} \leq 1 \\ \frac{1}{\pi h^2}(\frac{1}{4}(2 - \frac{r}{h})^3) & \text{if } 1 < \frac{r}{h} \leq 2 \\ 0 & \text{if } \frac{r}{h} > 2 \end{cases} \tag{2.43}$$

This function has been reported to behave correctly in many situations and is thus a preferred kernel among many authors. However, it is a conditional function and this applies an additional cost in the computation. In addition, the kernel drops to zero at twice the smoothing length and this can create problems during the implementation, hence why we do not use this kernel function in our SPH model.

Poly6

Müller, in his attempt to reach interactivity, proposes a new approximating function which uses a sixth grade polynomial (Müller et al. [2003]):

$$\begin{aligned} W(r,h) &= \frac{315}{64h^9\pi}(h^2-r^2)^3 \\ \nabla W(r,h) &= -r\frac{945}{32h^9\pi}(h^2-r^2)^2 \\ \nabla^2 W(r,h) &= -r\frac{945}{8h^9\pi}(h^2-r^2)(r^2-\frac{3}{4}(h^2-r^2)) \end{aligned} \tag{2.44}$$

This new function, called Poly6, permits many performance improvements. It has limited support h, it is not conditional and the spatial coordinate r appears squared, which means that, remembering that r will be the distance between two particles, the costly operation of finding the root square of r^2 is avoided. On the other hand, the gradient of the kernel is physically inaccurate, resolving to 0 for $r = 0$.

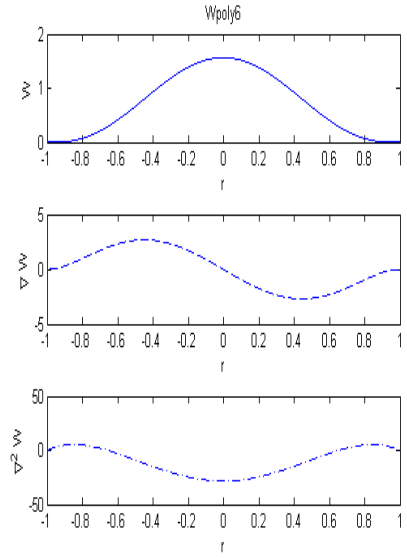


Figure 2.16: Poly6 kernel and its derivatives for h=1

Spiky

For the equations involving the gradient of the kernel function and in particular for the computation of the pressure forces (see section 2.2.3), Müller uses a different yet still not expansive kernel, called Spiky kernel because of its shape:

$$\begin{aligned} W(r,h) &= \frac{15}{h^6\pi}(h-|r|)^3 \\ \nabla W(r,h) &= \frac{45r}{h^6\pi|r|}(h-|r|)^2 \end{aligned} \tag{2.45}$$

This kernel's gradient reaches a large value when the spatial coordinate r tends to 0, solving the vanishing gradient problem of the Poly6 kernel.

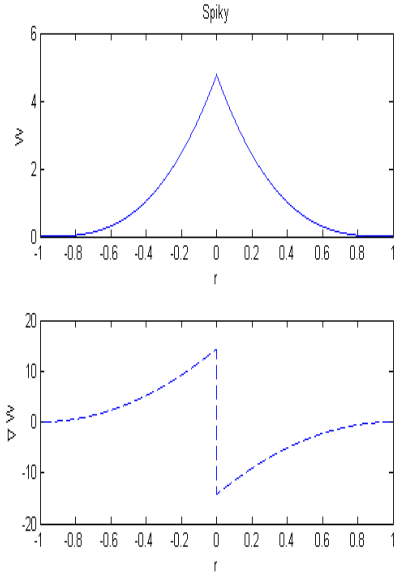


Figure 2.17: Spiky kernel and its first derivative for h=1

Viscosity

Both the Poly6 and the Spiky kernel (2.44 and 2.45) have a Laplacian with negative values. Since the Laplacian in Müller's work (and in this work as well) is only used for viscosity calculations, it has been chosen to use a kernel with an always positive Laplacian and thus modeling an always dissipative force. In addition, the final form of the Laplacian is computationally cheap.

$$\begin{aligned} W(r,h) &= \frac{15}{2h^3\pi}(-\frac{r^3}{2h^3}+\frac{r^2}{h^2}+\frac{h}{2r}-1) \\ \nabla W(r,h) &= \frac{15r}{2h^3\pi}(-\frac{3r}{2h^3}+\frac{2}{h^2}+\frac{h}{2r^3}) \\ \nabla^2 W(r,h) &= \frac{45}{h^6\pi}(h-|r|) \end{aligned} \tag{2.46}$$

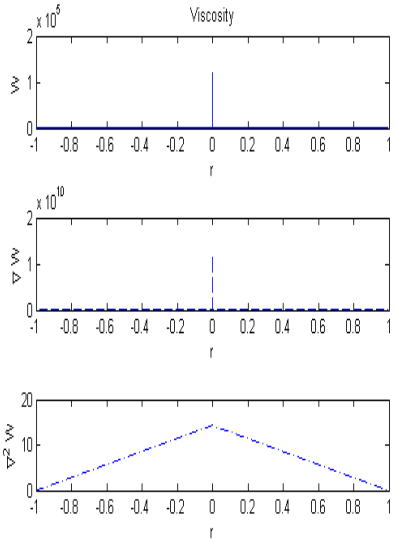


Figure 2.18: Viscosity kernel and its derivatives for h=1

2.4.2 Smoothing length

The smoothing length parameter h is of utmost importance in an SPH simulation as it is directly tied to the simulation's performance. Several authors have reported that the smoothing length is also closely related to the stability of the simulation and that it depends on the problem at hand. It is false to assume that by increasing the smoothing length infinitely the results will be perfect. This can be explained by the fact that with a high smoothing

length, due to the particles being sparse and due to inaccurate interfaces, the simulation is more likely to compute erroneous values. On the other hand, with a small smoothing length, not enough particles will take part in the computations. Because of this, it is most important to choose an appropriate smoothing length by trying to reach an agreement between performance and stability. Our solution is explained in section 4.1.

2.5 Conclusions

In this chapter, the basis of the SPH model has been discussed.

In section 2.1, the basic formalism, ideas and mathematical background of a SPH model are presented, with extensions on the classic formulation.

In section 2.2, the solution of the Navier-Stokes momentum equation is discussed by solving it using the SPH formalism, addressing density and pressure as well as cohesion, viscosity, external and elastic forces.

In section 2.3, the resulting dynamic system is analyzed and many integration algorithms are confronted and tested for its solution.

In section 2.4, kernel functions are explained in detail and different functions for different purposes are presented.

Chapter 3

Complex SPH model

In this chapter, the most important problems arising from a SPH approach to fluid modeling are discussed and they are addressed by extending known solutions. In addition, extensions to the classic model are presented in order to increase the scope and versatility of the simulator.

3.1 Tensile instability

As introduced in section 2.2.3, tensile instability is a numerical problem arising from the pressure force equations that can be seen as particles form clusters trying to pull a few particles very close in order to reach rest density instead of pulling more particles with less force.

Müller's Spiky kernel of equation 2.45, aims at reducing this phenomenon, as its gradient is created in such a way that the repulsion force between particles during pressure computations increases greatly as the particles get closer (Müller et al. [2003]). However, we have observed that the kernel shape alone is not enough to solve this problem.

In order to address the tensile instability as thoroughly explained by Monaghan [2000], we have implemented a method similar to Monaghan's and based on the work of Clavet et al. [2005], that introduces a method called *double density relaxation*. Both Monaghan and Clavet, in different ways, solve the clustering problem by the addition of an artificial pressure term.

Our implementation differs from Clavet's as his ideas are adapted to a more classic SPH model. The main difference lies in the fact that instead of using Clavet's fictitious *near density*, the normal density ρ_a of each particle is used. During the pressure computation, which is at default performed using Tait's equation of state (2.15), an additional *near pressure* P_a^{near} is computed as follows, actually resolving to the ideal state equation (2.14) for a fluid with

zero rest density:

$$P_a^{near} = k_P \rho_a$$

Therefore, near pressure is always higher than zero and this value is used to make sure that particles do not pull their neighbors too close.

During pressure force computation, Clavet's idea is followed and a different kernel function is used for near pressure. The Spiky kernel gradient ∇W_{ab} of equation 2.45 is used for standard pressure force computation. In addition, a higher order kernel is used for near pressure, which resolves to a kernel gradient ∇W_{ab}^{near} with a smaller area, as shown in figure 3.1. The sum of the two pressures give us the new pressure force, which can be observed to eliminate tensile instability altogether. The new equation, using Müller's pressure force formulation (2.19), takes the following form:

$$f_a^{press} = -\sum_b \frac{m_b}{2\rho_b}((P_b + P_a)\nabla W_{ab} + (P_b^{near} + P_a^{near})\nabla W_{ab}^{near}))$$

In addition, as observed by Clavet, the removal of the tensile instability highlights the spurious surface tension effect. This problem is further discussed in section 3.4.

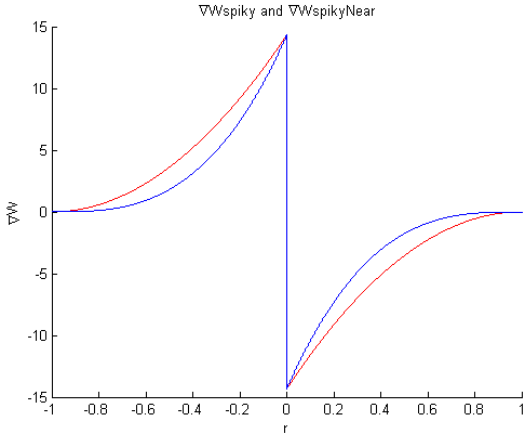


Figure 3.1: Double density relaxation kernel functions for h=1

3.2 Incompressible SPH

Born as a method for simulating compressible fluids, the basic SPH model is not suited for the simulation of incompressible or even nearly-incompressible flows. Since water can be viewed as a weakly compressible fluid, with its

speed of sound waves reaching 1500 meters per second, by using the standard equations of state (see section 2.2.2) the resulting pressure forces tend to be too large for a stable simulation and would require too small time steps.

Using the standard equation of state 2.14, the pressure stiffness coefficient k_P is in the order of a million and a small difference in density triggers a large difference in pressure, leading to an unstable simulation. The use of equation 2.15 eases the problem as a larger speed of sound can be sustained, achieving a less compressible fluid, but not yet incompressible. In both cases, a speed of sound of at least two orders of magnitude less than the real one must be taken for water-like simulations.

Because of this, an incompressible extension of SPH for the correct simulation of water flow has been one of the main goals in worldwide research, with contributions from several authors. In particular, two methods have been gaining visibility: Moving Particle Semi-Implicit and Pressure Corrected SPH.

3.2.1 Moving Particle Semi-Implicit

Published by Koshizuka and Oka [1996], the Moving Particle Semi-Implicit method has been introduced with the purpose to solve the stability issues of using classic SPH for the simulation of incompressible fluids. The method is quite similar to SPH, with the density and forces computations being the same apart from the pressure forces computation phase.

First, the *number density* δ_a is defined as a value that contains an estimate of the number of particles in the point x_a:

$$\delta_a = \sum_b W_{ab} \tag{3.1}$$

Then, instead of computing the pressure forces as usual, a Poisson equation of pressure is solved after discretizing it in a system of linear equations:

$$\nabla^2 P_a = -\frac{\rho_a}{dt}\frac{\delta_a - \delta^0}{\delta^0} \tag{3.2}$$

Where δ^0 is the preferred constant number density when incompressibility has been reached. Due to the computational cost of solving the Poisson equation, in the work pertaining to this thesis another method has been preferred.

3.2.2 Pressure Corrected SPH

Another approach for simulation of incompressible fluids has been proposed by Bao et al. [2009] with particular emphasis on the computational costs of the algorithms. This approach, called Pressure Corrected SPH, achieves incompressibility without resorting to a Poisson equation, which is quite time-consuming, and addresses the pressure disturbances and instabilities arising from the use of an equation of state.

According to this approach, the equations of SPH are solved as usual, but the pressure contribution is left out. Before the integration step begins, the pressure term is instead solved with a new equation, which is obtained by considering the continuity equation (2.9) alongside the ideal equation of state (2.14):

$$\frac{dP}{dt} + k_P \ \rho \ \nabla \cdot v = 0$$

An iterative algorithm can be obtained by writing the latter equation in SPH form and solving it for N steps, with $n = 1, 2, ...N$:

$$dP_a^{n+1} = -k_P \ dt \sum_b (v_b^n - v_a^n) m_b \nabla W_{ab}^n \tag{3.3}$$

At each step n, a velocity update is obtained by substituting the difference of pressure dP into the momentum equation 2.8:

$$\begin{aligned} dv_a^{n+1} &= -\nabla \tfrac{dt}{\rho_a} \ dP_a^n \\ &= -k_P \ \tfrac{dt}{\rho_b} \textstyle\sum_b (dP_b^n - dP_a^n) \nabla W_{ab}^n \end{aligned} \tag{3.4}$$

Equations 3.3 and 3.4 are supposed to be iterated alternatively until convergence, but for performance purposes Bao et al. [2009] observes that the iteration is convergent with very few steps and the pressure of every particle reaches rest density in a small time, providing incompressibility.

In the work this thesis is based on, a variation of equation 3.4 has been preferred based on the correct formula for the difference gradient variant (2.6) and has been observed to behave more correctly.

$$dv_a^{n+1} = -k_P \ \frac{dt}{\rho_a^2} \sum_b m_b (dP_b^n - dP_a^n) \nabla W_{ab}^n$$

Another expression for the last equation can be found by using the summation gradient variant 2.7, which we have observed behaves better for a limited

number of particles.

$$dv_a^{n+1} = -k_P \; dt \sum_b (\frac{dP_b^n}{\rho_b^2} + \frac{dP_a^n}{\rho_a^2}) m_b \nabla W_{ab}^n$$

Using the Pressure Corrected SPH method, the speed of sound can safely be assigned its real value without creating instabilities, thus achieving complete incompressibility at the expanse of performance.

3.3 Multiple fluids

One challenge for a more complex SPH model is the simulation of multiple fluids. SPH can be easily extended for multiple fluids as the equations presented thus far remain true for interactions between different fluids' particles, with additional physical phenomena emerging automatically. The different particle masses are already taken into account in our equations and as such densities, pressure and pressure forces are consistent, while a small modification of the viscosity forces 2.20 using the mean viscosity between the two interacting particles must be added.

Care must be taken when choosing a suitable smoothing length for the whole system, which should be selected as being the minimum among all the interacting fluids' individual smoothing lengths.

One of the most notorious consequences of the SPH approach for multi-fluid modeling is the automatic appearance of buoyancy effects. In the example of two immiscible fluids poured in the same container, the less dense fluid, that is with a lower rest density than the other, will tend to rise to the surface. This can be easily seen in reality in the behavior of common fluids like oil and water. As a consequence of the density summation approach, on the interface between the two fluids a pressure gradient will be forming, which will make the lower density particles rise and the higher density particles fall. However, due to the spurious interface effects that are discussed in section 3.4, buoyancy can be more correctly simulated with additional extensions.

3.4 Surface tension

Surface tension is another characteristic behavior of fluids, especially of liquids such as water. This phenomenon can be observed when placing a leaf on plain water or while watching certain insects walking on water. The water exerts a force that blocks its penetration by outer bodies. In mathematical

terms, the surface tension is defined as the tension that a liquid imposes in order to minimize its curvature. It is therefore of great interest to simulate the surface tension effect.

3.4.1 Spurious surface tension

Before being able to create a controllable surface tension, a first important problem must be addressed. This problem is the spurious surface tension effect that arises from the use of the density summation approach (using equation 2.10), which can however be observed even if using a continuity approach (using equation 2.12). A detailed explanation of this phenomenon can be found in the work of Hoover [1998].

At the fluid's interface, the lack of particles or the presence of other bodies' particles with different densities decrease the accuracy of the kernel interpolation, even at the density computation phase. This can be seen in figure 3.2. In the figure, since less particles are available at the free surface, the density of the surface particles will be lower than that of non-surface particles, this will therefore create a pressure gradient which will tend to push the surface particles inside the fluid, resulting in what is called a spurious surface tension.

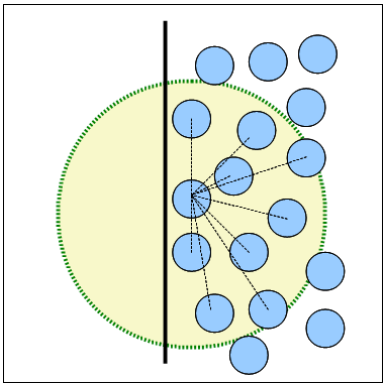


Figure 3.2: Cause of the spurious surface tension effect

The fake tension is uncontrollable and, even if some authors tolerate its presence since no additional surface tension force must be added for the simulation of water-like fluids, it is to be considered an undesired consequences and must be removed and substituted with a controllable, parametric and physically realistic surface force. The error given by the spurious surface tension is particularly problematic when the number of particles is small and the smoothing length is large, but it still persists for higher resolution simulations with many particles.

The spurious surface tension effect can be observed in three different cases. The most problematic case is the free surface flow, since no particles are placed outside the fluid. Fluid-fluid interfaces can also suffer from this problem when the rest densities of interacting fluids are different. At last, fluid-boundary interfaces suffer from spurious surface tension as well.

This work contains possible solutions to the three problems. A new method is proposed for the free surface flow problem in the next section, fluid-fluid interfaces are addressed in section 3.4.3 and boundaries which avoid the problem are presented in section 3.5.

3.4.2 Air particles generation

In this work, a new approach is presented for the solution of the spurious surface tension problem for free surface flows, since no other complete solutions have been found in literature. Most research on spurious surface tension is geared towards multi-fluid simulations and is not suited for simulations where fluids get often in contact with air.

This work expands the idea contained in the work of Müller et al. [2005] regarding the dynamic creation of air particles for additional graphical effects. Müller tries to achieve the formation of water bubbles inside water that is dropped into a container, but we think the method can be also used in order to solve the spurious surface tension problem.

The idea is that the volume around the free surface is not composed of empty space, but is actually filled with air. Air could be simulated as an additional SPH fluid and, with the multi-fluid extension of section 3.4.3, interface errors would disappear. This would however be quite computationally expansive and remove the benefits of a Lagrangian implementation, mainly the possibility to simulate only the needed portion of the simulation volume.

Air particles can however be added only around the fluid, creating a coat of air in which the fluid is surrounded, while the rest of the simulation volume is considered filled with still air and is not simulated. This option is allowed because kernel functions drop above their smoothing length, so by surrounding the fluid with air particles up to that distance the fluid finds enough particles for its computations and removes spurious surface tension effects. This idea is shown in figure 3.3.

The generation of particles is controlled by checking the magnitude of the outward normal of surface particles $\vec{n}$, which can be computed as shown in section 3.4.4. An air particle is generated if the magnitude is higher than a given threshold and it is removed when its density drops below a minimum density, that means that the air particle is too far from the fluid.

The evolution of air particles is addressed by making use of the usual

SPH equations, but we have reported that this still gives birth to spurious surface effects because air particles would still be in contact with the empty space, so other techniques should be investigated and used.

This idea has yet to be fully implemented and tested because of time constraints and as such it has not been used during the tests of chapter 5.

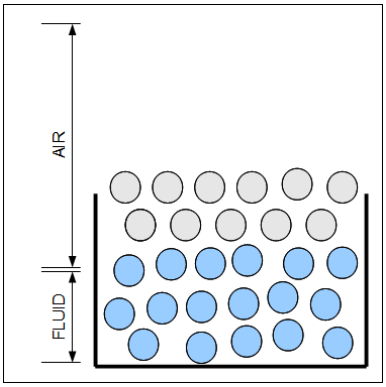


Figure 3.3: Particles generated around the free surface of the fluid

3.4.3 Multi-fluid adapted SPH

The phenomenon of spurious surface tension, as explained in section 3.4.1, still persists at the interface of two or more fluids when their rest densities are different, with a higher error the higher that difference is. Again, a fake tension arises between the interfaces, resulting in the fluids not correctly mixing together and effects such as the Rayleigh–Taylor instability to not arise in classic SPH simulations. The recent work of Solenthaler and Pajarola [2008] aims at the removal of this problem by modifying the base SPH equations. The basic idea is to have all particles treat neighboring particles as if they had its same rest density ρ and mass m. Solenthaler defines a new non-physical quantity which is used as an estimate of the number of particles in a point of the fluid, the number density δ, which is the same as in equation 3.1 where it is part of the Moving Particle Semi-Implicit method of section 3.2.1.

Using the number density δ as in equation 3.1 for the density computation, an adapted density $\tilde{\rho}$ can therefore be obtained:

$$\tilde{\rho} = m_a \delta_a \tag{3.5}$$

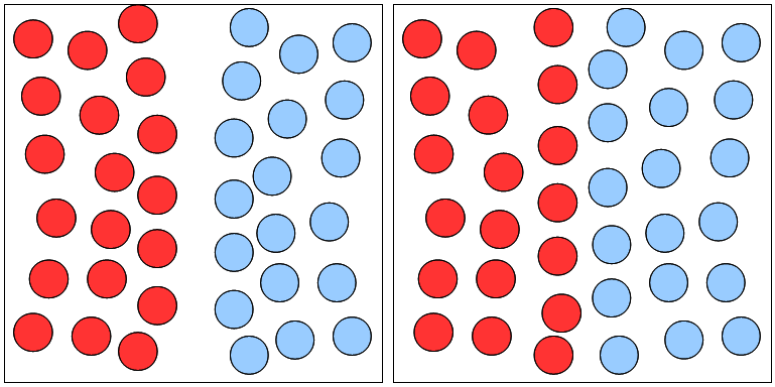
As a consequence, the rest of the physical equations are modified to account for this different density.

An adapted pressure $\tilde{P}$ can be computed from the adapted density $\tilde{\rho}$, either with Tait's equation 2.15 (as done by Solenthaler and Pajarola [2008]) or with a different equation of state. The force equations are then derived using the adapted values which, if using for example Müller's formulations, take the following forms:

$$f_a^{pressure} = -\frac{1}{\delta_a} \sum_b \frac{1}{\delta_b} \frac{\tilde{P}_a + \tilde{P}_b}{2} \nabla W_{ab} \tag{3.6}$$

$$f_a^{viscosity} = -\frac{1}{\delta_a} \sum_b \frac{1}{\delta_b} \frac{\tilde{P}_a + \tilde{P}_b}{2} \nabla W_{ab} \tag{3.7}$$

By using this method, the spurious surface tension arising at the interface of two fluids is removed, resulting in a behavior similar to what is shown in figures 3.4a and 3.4b.



(a) Without Adapted SPH (b) With Adapted SPH

Figure 3.4: Density contrast solution: effect on the interface

3.4.4 Controllable surface tension

Several authors have already proposed different solutions for surface tension modeling. We have investigated and compared the known methods and we present them here.

Curvature minimization

Proposed by Morris [1999], the method for surface tension modeling based on curvature minimization relies on the mathematical macroscopic definition

of the phenomenon. This is also the method extended by Müller et al. [2003] in their work.

First, surface particles must be located. In order to do so, we introduce a parameter called *color value* C^V, which is one at particle positions and zero everywhere else. Using the SPH formulation, the color value for a given particle a is:

$$C_a^V = \sum_b C_b^V \frac{m_b}{\rho_b} W_{ab} = \sum_b \frac{m_b}{\rho_b} W_{ab} \tag{3.8}$$

The inward surface normal $\vec{n}$ can then be computed as the gradient of the color field:

$$\vec{n}_a = \nabla C_a^V = \sum_b \frac{m_b}{\rho_b} \nabla W_{ab} \tag{3.9}$$

The surface normal $\vec{n}$ is computed for each particle and particles with the normal magnitude higher than a chosen threshold are considered surface particles. At last, the curvature κ_a of the fluid in a point x_a is computed as follows, inverting the value so to have positive curvature for convex surfaces:

$$\kappa_a = -\frac{\nabla^2 C_a^V}{|\vec{n}_a|} \tag{3.10}$$

The surface force for each surface particle can then be derived as follows:

$$f_a^{surface} = \sigma_a \kappa_a \vec{n}_a = -\sigma_a \nabla^2 C_a^V \frac{\vec{n}_a}{|\vec{n}_a|} \tag{3.11}$$

This method has been used successfully in the literature, especially in multi-fluid flows. In the latter case, the color value is taken into account only for the same-fluid particles. As observed by Becker and Teschner [2007], the curvature minimization approach has been intended for multi-phase simulations and performs poorly with a single phase since there are no particles at the free-flow interface or at the boundaries. Basically, this is the spurious surface tension problem.

Microscopic attraction

Proposed by Becker and Teschner [2007], this method for surface tension modeling ignores the global aspect of curvature minimization and instead focuses on the microscopic interactions between particles that make surface tension effects emerge. The idea behind the method is based on cohesion forces that spawn from Van Der Waals interactions between particles of the same fluid.

The surface tension force is thus computed as follows:

$$f_a^{surface} = -\frac{\sigma\rho_b}{m_b}(x_b - x_a)W_{ab} \tag{3.12}$$

The method is best suited for free surface flows since it performs better for a single phase (Becker and Teschner [2007]) and as such has been preferred in this work for most simulations.

3.5 Boundary methods

As Smoothed Particle Hydrodynamics was born to model astrophysical phenomena, where boundary conditions are scarce, there was at the beginning little to no focus on the matter of finding suitable boundary implementations. When the method became popular for fluid simulation, however, boundaries became an aspect that could not be passed over.

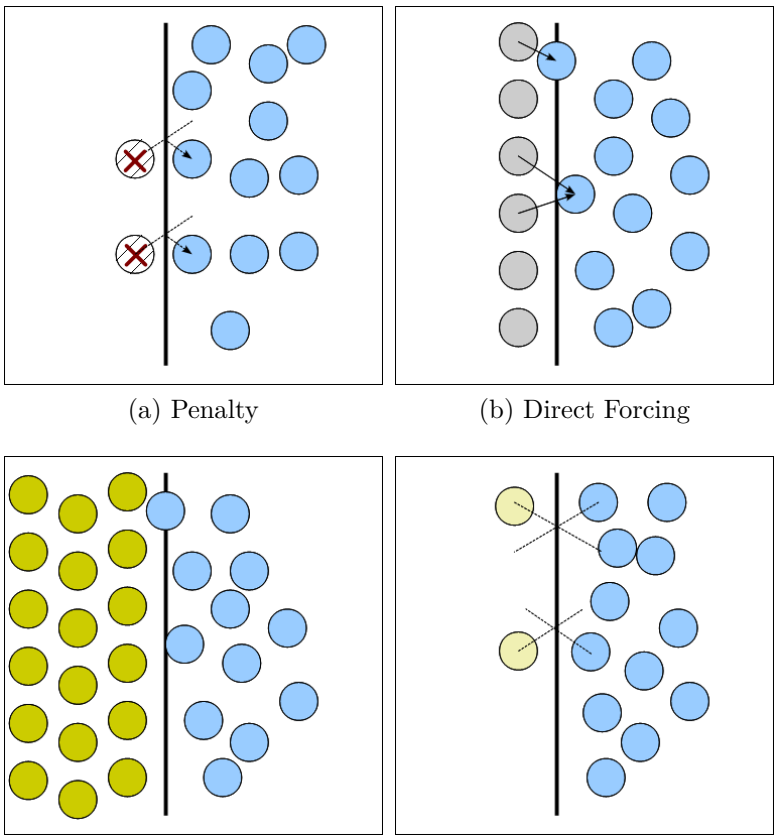
Boundary methods are used in order to model the interaction of fluid particles with solid boundaries, which could be walls, containers, or even an abstract *spatial end* of the simulation. The boundaries are usually supposed to be immobile, but every method can be extended for movable objects.

The methods for boundary collision modeling are divided in two major families: penalty-based methods and force-based methods.

The former methods achieve a no-penetration condition by directly imposing constraints on the fluid particle's velocities and positions, they are relatively simple to implement and can be cheap, but they are less stable due to the direct modification of particle's positions and velocities and thus of the interpolation points of the SPH approximation.

The latter methods, instead, impose new forces to the particles that reach the boundary, thus presenting a continuity with the SPH method and modifying positions and velocities according to the integration algorithms without ruining stability. Among these methods, we have investigated the direct-forcing approach, the quasi-fluid approach and the ghost-particles approach.

In our work, different boundary algorithms are implemented extending the known methods. This versatile approach has been taken in order to accommodate for different needs, so that the user can choose to focus on performance or accuracy. All methods have been extended to improve their stability and remove their shortcomings.



(a) Penalty (b) Direct Forcing

(c) Semi-fluid (d) Ghost particles

Figure 3.5: Boundary methods

3.5.1 Penalty approach

Born from classic particle system simulations, that have been used since the late eighties in computer graphics, the standard penalty approach is quite simple to implement and it is performant, but it can also give birth to instabilities as particle velocities and positions are directly modified, without taking into account the integration algorithms and SPH forces. In addition, the method is based on chosen boundary conditions (Müller et al. [2004]) and not on physical parameters, which renders the method useful for fast simulations without accuracy objectives in mind such as in computer graphics. The simplest penalty case consists of a box-shaped boundary placed at the system's origin. The source code 3.1 controls the collision of particles with the boundary on the negative x-axis:

Already a few problems can be pointed out, such as the possible modification of the particle's energy and the completely elastic reflection. Following

```
if (particle->pos.x < boundaryDimension.x) {
        particle->pos.x = - boundaryDimension.x; /// Push out
  particle->dir.y = -particle->dir.y; // Reflect Y velocity
  particle->dir.z = -particle->dir.z; // Reflect Z velocity
}
```

Source code 3.1: Simple penalty boundary approach

the classic method, the approach has been extended in this work with various optimizations in order to provide a more versatile and accurate method.

- **Shape-based reflection**

  In order to create different boundaries for different test cases, the penalty boundary method has been generalized by computing the surface normal B_N of each boundary during a collision and thus separating the normal velocity v_N from the tangent velocity v_T. The velocity of the colliding particle can thus be divided as seen in figure 3.6.

  $$v_N = B_N * (B_N * v)$$

  $$v_T = v - v_N$$

  Due to this, for a completely elastic collision with an arbitrary boundary, we just have to invert the normal velocity v_N.

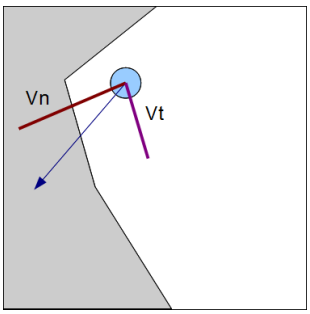


Figure 3.6: Separation of particle velocities using the improved penalty approach

- **Boundary bounce and slip**

  Simple penalty boundaries simulate a completely elastic reaction. With the use of bounce and slip coefficients, we can differentiate this behavior for each fluid. The bounce coefficient C_{bounce} is included in the range $[0, 1]$, from no-bounce effects to completely elastic reactions. The slip

coefficient C_{slip} is included in the range $[0, 1]$, with 0 simulating the no-slip condition, useful for water simulations, and 1 simulating a full slip.

$$v' = -v_N \; C_{bounce} + v_T \; C_{slip}$$

- **Particle collision checking and repositioning**

  After the collision, particles are repositioned according to the boundary normal B_N. For performance purposes, the actual collision point is computed at the collided particle position as shown in figure 3.7, which can be inaccurate. However, this does not visibly reduce the simulation's accuracy since time steps are small (Kelager [2006]).

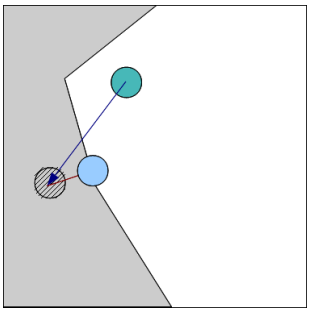


Figure 3.7: Repositioning of a particle using the improved penalty approach

- **Energy conservation**

  As explained by Kelager [2006], depending on the timestep and the actual collision point, kinetic energy may erroneously increase. Energy conservation is assured by reflecting only the actual velocity that was omitted in the collision, using the penetration distance d for the purpose as seen in figure 3.8

$$v' = -v_N \; C_{bounce} \frac{d}{|v| \, \Delta t} + v_T \; C_{slip}$$

- **Direction control**

  A direction control has been implemented for all boundaries which allows the boundary to block and reflect particles that travel in a chosen direction. This can be useful for the simulation of water flowing in a closed box by removing the collisions on the boundary ceiling.

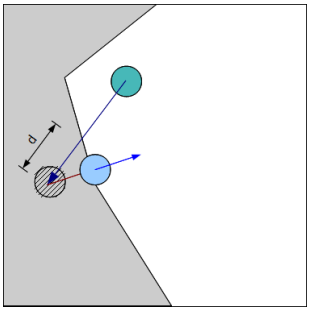


Figure 3.8: Energy conservation using the improved penalty approach

3.5.2 Direct forcing

A first force-based solution comes from Monaghan et al. [1994] who proposed to place fixed particles at the boundaries which would exert Lennard-Jones forces in the center of close-by fluid particles, thus pushing them out of the boundary. This expedient is based on observed inter-atom forces and tries to achieve the no-penetration condition by considering the microscopic interactions between the fluid and the boundary atoms, called *solid particles*.

The equation for the force between a fluid particle and a solid particle takes the following form, that is computed only for $\frac{h}{|r_{ab}|} > 0$ in order to have only repulsive forces, where $r_{ab} = x_a - x_b$:

$$f_{ab}^{bound} = K_{coll} \frac{r_{ab}}{|r_{ab}|^2} \left(\left(\frac{h}{|r_{ab}|} \right)^{k_1} - \left(\frac{h}{|r_{ab}|} \right)^{k_2} \right) \tag{3.13}$$

Where the constants are chosen as $k_1 = 12$ and $k_2 = 6$. The rigidness K_{coll} can be chosen according to the problem as hand and, as suggested by Monaghan et al. [1994], results are insensitive to its value, provided it is large enough.

Our algorithm iterates over all fluid particles and checks for the presence of neighboring solid particles, then uses the provided force to create a no-penetration condition. The no-slip condition is instead enforced by considering the solid particles in the viscosity computation.

This work also introduces a less expensive variant to Lennard-Jones forces, based simply on an SPH formulation of a direct force that we have observed as being good enough for simple simulations.

$$f_{ab}^{bound} = \frac{r_{ab}}{|r_{ab}|} K_{coll} \frac{m_b}{\rho_b} W_{ab}$$

In addition, a distinction is made between static and dynamic boundaries, the latter being rigid bodies, in order to reflect the force on the solid objects

as well. This is done even for solid-solid interactions, effectively creating a two-way coupling behavior. However, a more consistent method can be found for this purpose, as is discussed in the next section.

3.5.3 Quasi-fluid particles

In order to achieve a greater inter-connection between SPH fluids and rigid boundaries, new techniques have been proposed in the recent years which try to extend the SPH formulation to rigid bodies. Among them, the quasi-fluid particles method proposed by Crespo et al. [2007] is still simple to implement and provides good results.

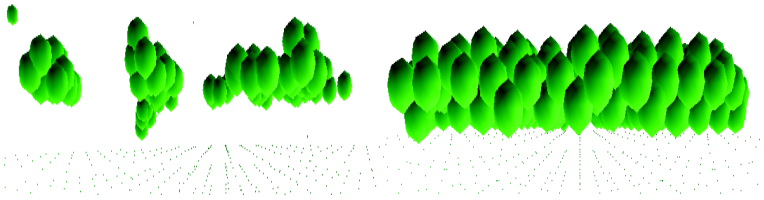
With this approach, layers of rigid particles are placed below the boundary surface that satisfy the same SPH equations of the fluids but that remain fixed in space. Three layers are usually placed so that there are enough particles for correct computations. By providing continuity with SPH, this method achieves greater stability and accuracy. A performance loss, however, is observed, as the method increases the number of particles of the simulation.

In addition, this method is a step towards a comprehensive approach for fluid and solid simulation, since rigid bodies are actually modeled as clouds of linked particles.

This method is still new and presents unsolved questions, such as the value of parameters to be chosen for rigid particles. In this work, it has been observed that density can be chosen arbitrarily and allows heavy objects to not be effected by the fluids or light objects to be moved. The pressure stiffness coefficient must be high in order to allow the no-penetration condition, so that particles do not get too close to the boundary. The viscosity coefficient can be chosen in order to provide a no-slip condition (high viscosity) or some degree of slip (low viscosity).

The quasi-fluid boundary method has been enhanced in this work by using the multi-fluid adapted SPH proposed by Solenthaler and Pajarola [2008], so that the spurious tension arising at the interface between the fluid and the boundary is removed, allowing for a complete adhesion of the fluid to the boundary surface.

As can be seen in figure 3.9, using the adapted quasi-fluid boundary method the particles of the fluid tend to get closer to the boundary particles, removing the spurious surface tension effect on the interface and increasing the simulation's stability.



(a) Not adapted (b) Adapted

Figure 3.9: Comparison of the classic quasi-fluid boundary and the adapted version

3.5.4 Ghost particles

This method is similar to the quasi-fluid particles boundary approach, but it aims at achieving higher performance. Boundaries are not filled with particles at their creation, instead ghost particles are created dynamically outside the boundary over the course of the simulation. Ghost particles are born when a fluid particle gets close enough to the boundary, mirroring the properties of the interacting particles, but with the normal velocity inverted, thus enforcing a repulsion.

Proposed initially by Randles and Libersky [1996], the method is also used by other authors, such as Valizadeh et al. [2008], in their research. It has been reported as being both efficient and accurate and it adapts easily do different boundaries, even dynamically.

Due to time constraints, a ghost particles boundary method has not been implemented in this work.

3.5.5 Rigid bodies

In addition to static boundaries, it is of great interest to simulate rigid bodies alongside fluids, in order to simulate the complexity that arises from their interactions. Examples are a boat floating on water, objects transported by the flow, liquid dripping from a cup and many more everyday situations.

Rigid body mechanics are well understood and can be simulated efficiently with simple methods. By extending these methods with a SPH approach, we obtain a complete two-way interaction between solids and fluids. Several methods have been proposed in literature, such as by Becker et al. [2009] or by Müller et al. [2004], in addition to more simple Verlet methods used for fast rigid body dynamics such as those present in video games.

In this work, a new method is developed, which is born from the choice of a quasi-fluid boundary approach. Much like a fixed boundary, each rigid

body contains a fixed number of fake particles called *rigid particles* inside its volume, placed underneath the body's surface. These particles fulfill the SPH equations exactly like fluid particles, directly affecting the fluids they enter in contact with, but their position will be dependent on the rigid body's movement.

By implementing an action-reaction approach on the interaction of the rigid particles with fluids' particles, or even with other bodies' rigid particles, two-way interaction is achieved. This allows us to simulate an even greater range of problems.

The algorithm can be found in section 4.2.4.

3.6 Conclusions

In this chapter, extensions to the classic SPH model are proposed and detailed in order to provide solutions to its shortcomings.

In section 3.1, the tensile instability problem is addressed and a solution is presented under the form of a variant on the double density relaxation algorithm.

In section 3.2, the compressibility issue is addressed and two methods, Moving Particle Semi-Implicit and Pressure Corrected SPH, are detailed for its solution.

In section 3.3, the issue of multiple-fluids simulation is addressed.

In section 3.4, the spurious surface tension problem is addressed and solutions are proposed for different cases. In addition, models for fully controllable surface forces are proposed.

In section 3.5, boundary methods are explained and improved upon. An extension for the two-way coupling of fluids and rigid body dynamics is also presented.

Chapter 4

Implementation

In this chapter, the implementation of the SPH model discussed in the previous chapters is presented. Two different softwares have been created, a serial implementation and a parallel implementation, and are explained in detail.

4.1 Simulator and guided creation

The simulator developed in order to implement the SPH model as discussed in the previous chapters is described in the following sections, with details on peculiar implementations and choices.

The code of the serial SPH implementation has been written in the C++ programming language and the visualization and interaction aspects have been developed by making use of the Simple DirectMedia Layer (SDL) libraries and the OpenGL graphics libraries. The code runs on a Windows machine and is compiled with Visual C++ 2005.

4.1.1 Simulator

As we delved inside the project, we quickly realized that due to the complex mathematics involved and their fragility, we would need a more structured simulator in order to achieve the correct simulation of different scenarios. A simulator based on our SPH model has thus been produced which allows multiple fluids with several different starting conditions that can be specified by the user.

The base of the simulator is the *SPHSystem* class that governs all the global aspects of the simulation. Using the simple API as explained in section 4.1, the user can define different fluids and rigid bodies and they are inserted into the system accordingly.

The simulator allows the definition of different emitters in the system by instantiating the class *SPHEmitter* and all emitters are accessible by the system trough a linked list. Each emitter, on the other hand, owns a dynamically linked list of particles and updates them at each time step. Emitters determine the creation and the destruction of individual particles and all the particles of a single emitter share its physical properties, its initialization procedures and its color properties.

Rigid bodies are accessed and created in much the same way as emitters, linked in a dynamic list in the *SPHSystem* class.

The individual properties of each particle are stored in the *SPHParticle* class and particles are organized in different linked lists according to their local index (inside the emitter), global index and neighbor cell index (see section 4.2.2 for details).

Utilities

Various utilities have been implemented for a greater ease of use of the simulator and for more efficient coding.

Positions, velocities and other vector values are managed by istances of the *tVector* class. This class also implements various useful functions for vector math, such as dot and cross product or normalization.

Similar to the *tVector* class, a *tColor* class has been created for ease of access to color properties, stored as RGB values.

Additional utilities are contained in the *Utils.cpp*, *DrawUtils.cpp* and *SPHUtils.cpp* files for ease of access to additional mathematical computations, drawing primitives and common SPH routines.

Since instabilities can occur when single particles get too close to each other and in fast flows simulations this can happen, a set of optional warnings have also been implemented in order to predict situations of potential instability, with the added option to reset or entirely removed the unstable particles.

4.1.2 Simulation choices

Before the simulation begins, the user can define which options to use for the many aspects of the simulation. Many different choices, implementations of the techniques illustrated in the previous chapters, are presented through the use of enumerations in order to let the user decide what approach would be best for their case or to compare them. The user can make choices on the kernels used in the computations, on integration algorithms, on rendering variables, on force equations and on the boundaries methods.

Many additional choices are instead imposed by the simulator, as we have been reporting that some algorithms become unstable when used together. Default choices are imposed based on our observations.

Integration

As discussed in section 2.3.1, all the different integration algorithms that have been compared have also been implemented. As previously stated, the default value is the Velocity Verlet algorithm.

```
enum IntegrationAlgorithm {EULER, VERLET, CLAVET, VEL_VERLET,
    LEAPFROG, BEEMAN, BEEMAN_PC, BEEMAN_PC_CAPONE};
```

Density equation

The two different density equations of section 2.2.1 have been implemented. In addition to the summation multi-fluid adapted method, the adaptive form has been extended for a continuity approach. The default approach uses the adapted density summation equation.

```
enum DensityEqType {DE_SUMMATION, DE_SUMM_ADAPTED, DE_CONTINUITY
    , DE_CONTINUITY_ADAPTED};
```

Pressure equation of state

The two proposed equations of state and the pressure correction approach as shown in sections 2.2.2 and 3.2.2 are made available. The default equation of state is the Tait's equation for weakly compressible fluids, found to be sufficient for most of our needs.

```
enum EquationOfState{EOS_IDEAL, EOS_TAIT, EOS_CORRECTION};
```

Pressure forces equation

The different pressure equations discussed in section 2.2.3 have been implemented, as well as multi-fluid adapted versions. The default approach uses Müller's adapted pressure equation.

```
enum PressureEqType {PR_MULLER, PR_MULLER_ADAPTED, PR_MONHAGHAN,
     PR_MONHAGHAN_ADAPTED};
```

Viscosity forces equation

Again, for viscosity, the different equations discussed in section 2.2.4 have been implemented, with the default being Müller's force with the multi-fluid adaptation.

```
enum ViscosEqType {VI_MULLER, VI_MULLER_ADAPTED, VI_MON92,
    VI_MORRIS};
```

Surface tension equation

For surface tension computation, we have implemented the methods listed in section 3.4.4. The default approach has been chosen as Becker's micro-interaction method because of its better behavior in free surface flow simulation.

```
enum TensionEqType {
        ST_CURV_MU,   // Müller 03
        ST_MICRO_BE   // Becker 07
};
```

Emission

For emission purposes, several different choices are provided. The fluid can be created as a continuous stream with a square section (continuous-square mode) or circular section (continuous-circle mode). It can be created as a volume of fluid, such as a box (one shot-box mode) or a sphere (one shot-sphere mode). The fluid can also be created as a spray of liquid originating from a single point (spray-cone mode) or as an empty emitter, to be filled dynamically (remote-empty mode, used in the air particles generation method of section 3.4.2).

```
enum EmissionType {EMIS_CONTINUOUS, EMIS_ONESHOT, EMIS_SPRAY,
    EMIS_REMOTE};
enum EmitterType {EMIT_BOX, EMIT_SPHERE, EMIT_SQUARE,
    EMIT_CIRCLE, EMIT_CONE, EMIT_EMPTY};
```

Rigid bodies

The rigid body simulator provides many different shapes:

```
enum RigidbodyShape {RS_BOX, RS_CUBE, RS_SPHERE, RS_PLANE,
    RS_CYLINDER, RS_CUP};
```

As discussed in section 3.5, the different boundary methods have been implemented, with the penalty-based approach chosen as the default for better performance).

```
enum RigidbodyType {
        RT_PENALTY, // Penalty based
        RT_FIXED_LEN, // Lennard-Jones forces
        RT_SEMIFLUID, // Quasi fluid particles
}
```

4.1.3 Reports

In order to analyze properties of the implemented SPH model, a number of functions have been created for reporting different data regarding the simulation. The user can specify what type of data is needed and get feedback on the simulation.

Time data

Data regarding simulation time can be gathered from the running program in order to estimate the computation time of each routine. This is quite useful for code optimization and as a mean of pinpointing bottlenecks in the application. It must be noticed that the values provided by the time estimation routines are in no way absolute and must only be viewed as relative to each other.

The time routines take advantage of Windows' clock functions for computations with one second resolution or, when available, of the performance counter functions, which allow resolutions of under one millisecond. The GPU routines instead use Unix's internal time functions.

In addition, by making use of Fish [2001]'s glFont routines, the simulation time is shown on the screen in real-time as a text in the bottom left part of the simulation screen.

Simulation statistics

A simulation report can be printed to the console or to a text file using a single function. This report shows various information on the current simulation, such as a list of emitters and rigid bodies with their parameters and system-wide parameters and options. A report example is given in figure 4.1.

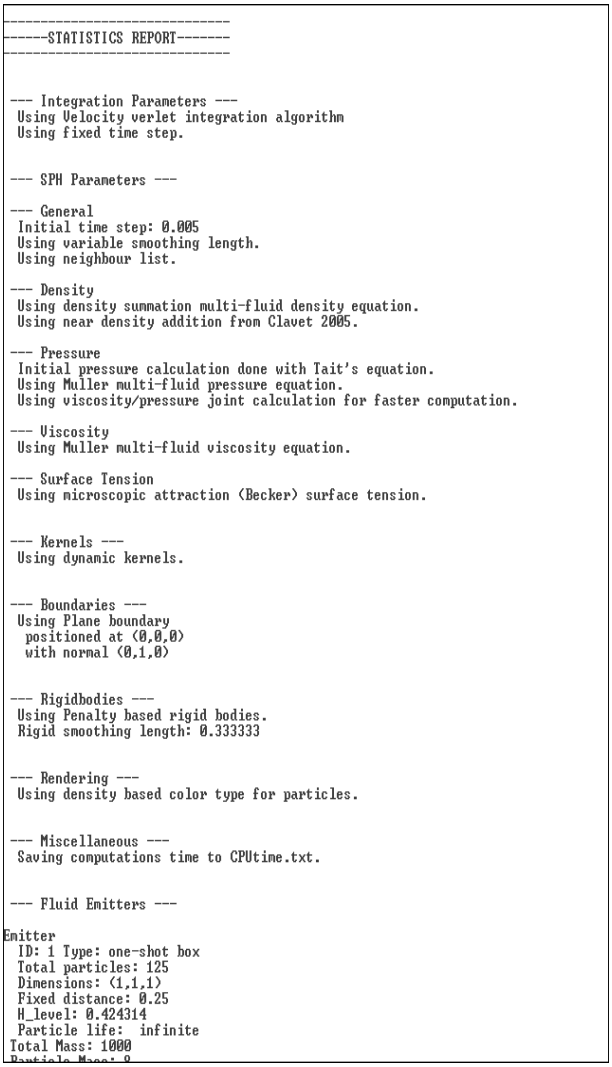


Figure 4.1: Simulation report example

Simulation data

Functions for data estimation of different characteristics have been implemented. The data gathered can be useful for comparison with analytic or different models' data. Routines are provided for the estimation of the trajectory of the fluid, the volume occupied and the section of streams and the position of the center of mass of the simulated fluid. The estimation is conducted differently for each needed data type.

Trajectory is estimated by dividing particles in groups according to their spawn time, with particles with close-by lifetimes belonging to the same group and assumed to travel together. As the simulation advances, the com-

putation of each group's barycenter is performed at timed intervals and the trajectory is then estimated for each group at that interval. At last, the positions of the barycenter of all groups at a given life-time are averaged to obtain a final trajectory.

Section is estimated by checking the maximum distance between particles of the same group. This estimation however suffers from the fact that particles in the same group can potentially diverge greatly and continue on very different paths and from fluid fragmentation artifacts.

A MATLAB program has been created for data processing and allows the user to visualize the data gathered from the simulation in graphical form, as shown in the results of chapter 5.

Image data

Image data concerns all images rendered by the simulator through the OpenGL context. All rendered images of a single simulation can be saved as automatically ordered files with a chosen format between PNG and BMP. The screen saving routine makes use of Windows's Graphic Device Interface Plus (GDI+).

4.1.4 Guided creation: User-imposed parameters

Due to the huge mathematical complexity of the model and its fragility when parameters are not chosen accordingly, especially when trying to simulate realistic fluid physics, a set of guided creation mechanisms and a simple API interface have been created for the user. All that is up to the user is to choose the main characteristics of the simulation, such as the initial conditions of the fluid and its physical properties, while all the rest is derived by the simulator.

The parameters that are for the user to choose are here listed and problems that may arise from incorrect values are commented.

Physical parameters

All physical parameters are imposed by the user. The parameters that can be assigned and the correspondent functions are listed below.

- **Rest density** $setRestDensity(float)$

  The parameter ρ_0 determines the density of each particle at rest. It influences the magnitude of pressure forces because a big difference between a particle's density and the fluid's rest density determines a high pressure. The dimension of this parameter is $[\frac{kg}{m^3}]$.

- **Speed of sound** $setSpeedOfSound(float)$

  The parameter c_0 determines the speed of sound inside the fluid. This value is used in pressure force computations and it can be the primary cause for instability. It is therefore advised to use a smaller than reality speed of sound for weakly compressible fluids, up to three orders of magnitude lower, or better use the incompressible SPH option. The dimension of this parameter is $[\frac{m}{s}]$.

- **Viscosity coefficient** $setViscosity(float)$

  The parameter μ determines the magnitude of viscous forces inside the fluid. The dimension of this parameter is $[\frac{Ns}{m^2}]$.

- **Surface tension** $setSurfaceTension(float)$

  The parameter σ determines the magnitude of surface forces at the interface of the fluid. The dimension of this parameter is $[\frac{N}{m}]$.

- **Elastic stiffness** $setElasticStiffness(float)$

  The parameter k_{el} determines the magnitude of elastic forces inside the fluid. The dimension of this parameter is $[\frac{N}{m}]$.

Emitter type and shape

Different emitter types and shapes have been implemented in order to provide the simulator with different starting conditions for the fluids, so that different test cases can be easily simulated.

- **Instant Cube** The most basic shape is the fluid cube, or box, in which the fluid is initialized in a 3D grid of chosen size in the x, y and z dimensions. The initial positions of the particles inside the volume of the emitter can be fixed or randomly generated. All particles are emitted at once.

- **Instant Sphere** Another basic shape is the fluid sphere. The fluid is initialized inside a sphere of chosen radius and with fixed or random positions. Again, all particles are emitted at once.

- **Square Flow** In order to simulate fluid flows, suitable emitters have been added. The square emitter spits particles alongside a chosen axis with a chosen velocity. The particles are initialized on a 2D grid of chosen size. The emitter keeps emitting particles, recycling particles that exceed their maximum lifetime.

- **Circle Flow** Similar to the square flow emitter, the circle emitter differs as it initializes the particles in a circular area. This emitter is very useful for the simulation of water jets.

Initial dimensions of the fluid

The total volume of the emitted fluid can be specified at start up by inserting the x,y and z dimensions of the emitter based on the emitter's shape.

Initial position of the fluid

The position of the emitter can be chosen by the user and defaults to the origin of the simulation area. Particles belonging to the emitter assume their origin to be the emitter's position.

Initial speed of the fluid

The speed of the emitter can be assigned by the user and it is added to all emitted particles at the start of their lifetime. This value is useful for the initialization of fluid flows.

Total number of particles

The total number of particles is not chosen directly. The user must decide on the number class $nClass$ of the simulation that is chosen as an integer in the range $[0, 10]$ and that sets the resolution of the emitter. Therefore, the number of particles for the simulation is chosen based on the number class value and the emitter's dimensions.

The number class determines N_l, the number of particles assigned to the minimum dimension of the initial volume of the fluid. The higher $nClass$ is, the more particles are used, with a consequent decrease in performance and increase in accuracy and resolution. The total number of particles is then chosen according to N_l by computing the relative size of the other two dimensions and setting their assigned number of particles as in the source code 4.1.

4.1.5 Guided creation: System-imposed parameters

The parameters that are automatically assigned by the system according to the chosen user parameters are now listed.

```
int getParticleNumber(tVector dim, int n, tVector* dimN) {
    float min = dim.x;
    if (dim.y < min) min = dim.y;
    if (dim.z < min) min = dim.z;
    dimN->x = (n + (n-1)*(dim.x/min -1));
    dimN->y = (n + (n-1)*(dim.y/min -1));
    dimN->z = (n + (n-1)*(dim.z/min -1));
    return dimN->x*dimN->y*dimN->z;
}
```

Source code 4.1: Particle number derived from number class and dimensions

Average number of particles

The average number of particles n_{avg} is chosen at startup by the simulator according to the number class $nClass$ of the emitter in order to increase the stability of the simulation. This parameter should be the smallest possible to increase performance, but big enough to provide accurate results. In the literature there has not yet been a general consensus on an appropriate number. In the performed tests, a value of $n_{avg} = 32$ has been proven successful for most simulations, although for better stability in the case of a small number of particles a different n_{avg} for each $nClass$ can be chosen. This value is still strongly dependent on the problem at hand at may be increased by the user in order to increase stability at the expanse of performance.

Kernel function smoothing length

With our approach, similar to what has been proposed by Kelager [2006], the smoothing length h used in the kernel functions is chosen to be the radius of a sphere so that the average number of particles n_{avg} contained inside that sphere is big enough. By defining the particle number density as N/V, where V is the total volume of the fluid and N is the total number of particles, we can find the particles inside a sphere of radius h with the following equation:

$$n_{avg} = \frac{N}{V}\frac{3}{4}\pi h_a^3$$

The smoothing length is then:

$$h_a = \sqrt[3]{\frac{3Vn_{avg}}{4\pi N}} \tag{4.1}$$

As mentioned previously, n_{avg} is the only user-chosen quantity and it depends on the problem at hand.

Variable smoothing length

As noted by several authors, the smoothing length can be varied at each time step in order to increase the accuracy of the simulation. For this purpose, the smoothing length is chosen dynamically for each particle so that the number of nearby particles is constant. This is achieved by varying the smoothing length of a particle h_a from the fluid's base smoothing length h according to the relation of the density of the particle ρ_a to its rest density ρ_0:

$$h_a = h\sqrt{\frac{\rho_0}{\rho_a}} \tag{4.2}$$

Particle mass

The mass of a single particle in an emitter is constant and chosen at the initialization phase. It is determined by the volume V, the rest density ρ_0 of the fluid and by the total number of particles N:

$$m_a = \frac{V\rho_0}{N}$$

As a consequence, the total mass of the fluid can be found by multiplying the singular mass of a particle for the total number of particles. Since the mass is always constant in a single emitter, it could be removed from the summations in the equations, but since different fluids with different masses can interact in this implementation, it must be left inside.

Life

The life of a particle, that is its individual time since birth in the simulation, is useful for determining when to dispose of it and for rendering purposes. As such, it is saved in memory for each particle. The maximum life of all particles in an emitter can be imposed by the user for any emitter, but since it is used as the threshold for recycling of particles in fluid flow emitters, it is automatically initialized in that case at the best value in order to limit the total number of particles in the scene. An emitter having a maximum life set to zero will not destroy emitted particles.

Fixed inter-particle initial distance

According to the chosen emitter type, the initial position of each particle in the emitter is imposed. A fixed position algorithm allows the simulator to determine a suitable initial fixed distance between each pair of particles. By

choosing this method instead of randomly placing the particles inside the volume, the accuracy and the stability of the simulation can greatly increase (Monaghan [1992]).

Since the total number of particles, the volume and the rest density of the fluid are user-chosen and the smoothing length is assigned according to their values, we impose a fixed start distance based on the geometry of the emitter and these parameters. Because of this, however, the particles may initially be too close and create strong pressures. It has been observed that if this happens the simulation can break.

In order to solve this problem, the user can still increase the average number of particles n_{avg}, thus increasing the smoothing length and damping the effect of initialization instabilities. However, this decreases performance.

Another option is to instead initialize the fixed distance as a factor of the smoothing length, thus allowing particles to reach rest density with ease. We have observed that a suitable factor is in the range $[0.8, 0.9]$ h, as this means that particles need to only pull a few other particles to reach rest density, resulting in a small compression of the initial fluid volume.

4.1.6 Guided creation: Extensions

Additional extensions to the creation phase have been implemented, based on our simulation needs.

Relaxation phase

The fluid can be initialized through a relaxation phase in which the fluid is allowed to move freely with greatly increased viscosity in order to reach a more stable situation.

Our algorithm updates the simulation without actually advancing the time step, as if the fluid was the only thing not frozen in time. No gravity force is added and the viscosity coefficient is greatly increased. The real simulation begins when the maximum velocity of any particle in the fluid is less than a given threshold for at least three simulation steps.

Inlet flow

Adapted from the work of Lastiwka et al. [2008], an inflow area has been implemented. This initialization area provides a more stable start for the simulation of inlet flows, such as in the case of our water jet simulation (see chapter 5). By providing a permeable interface with equally spaced particles, the particles are less affected by spurious surface tension and instabilities at

their birth, since the number of particles for the interaction would be too small otherwise.

This has been implemented by creating an inlet volume with section equal to the spawning section and length equal to four times the smoothing length of the fluid in order to provide enough particles. All particles are born inside this area and travel in the direction of the speed of the flow. All properties of the particles are computed but forces, so that the particle motion is completely kinematic. When any particle reaches its real birth position (that is, it has traveled for the length of the inlet area), standard computations are resumed for that particle.

4.2 Algorithms

The algorithms created for the simulator are here presented. The parallel code shares many solutions in its implementation with the serial code, so in this section the focus is on the CPU serial code.

4.2.1 Main algorithm

The main algorithm performs the initialization phase and the simulation loop.

Initialization

The first step is context initialization using the SDL routines. This phase creates the window in which the simulation takes form and manages the event listeners for user inputs.

Both CPU and GPU code make use of the OpenGL graphic libraries for rendering. The initialization phase takes also care of the OpenGL scene creation, assigning lights, enabling OpenGL states and configuring the viewport and camera matrices.

At last, the SPH system is initialized by adding emitters in the scene, setting non-default parameters such as a different equation of state or different integration algorithms, setting the chosen boundaries, setting display modes and choosing the physical time step.

Main loop

The main loop consists of two separately timed phases: the simulation step and the rendering step.

The rendering step is performed at each render time step, which is at default computed every 0.02 seconds of the physical simulation (drawing 50 frames per seconds), during which the program draws the scene. All the emitters, the rigid bodies and the boundary are drawn at once.

The simulation step is performed at each physical time step and consists of different phases, based on the options and parameters chosen at start up. All the following steps are performed for all emitters in the system:

1. All particles are loaded into their neighbor cells (see section 4.2.2).

2. Before-step integration updates are performed and particles are advanced to their partial positions.

3. Physical SPH computations are performed, resulting in forces for the current time step being evaluated. Based on each emitter's parameters, some of the computations can be avoided.

4. After-step integration updates are performed and particles are advanced to their final positions.

5. The collisions between fluid particles and boundaries and rigid bodies are computed and particles are repositioned accordingly.

6. Rigid body positions are updated.

4.2.2 Neighbor lists

Since each particle is supposed to interact with every other particle inside the fluid, the basic approach would be to use a brute force algorithm, which would be in the order of $O(N^2)$ where N is the total number of particles of the simulation. Since the number of particles is supposed to be high, the computational cost would become unsustainable, especially for an interactive environment.

Instead, by making sure that the kernels have finite support as in equation 2.4, we can only compute the interactions between each pair particles and its close neighbors. By taking an average number of neighbors n_{avg}, the algorithms become of the order $O(N \cdot n_{avg})$, which provides a great increase in performance.

In order to achieve a very high performance and to take advantage of the peculiarities of a SPH formulation, neighbor lists have thus been introduced. The idea behind a neighbor list is that if we divide the simulation volume in a grid of correctly-sized cells, when computing the properties of particle a, we have to check only the particles inside the particle's corresponding cell

and in the surrounding cells. In 2D, this means 9 cells and in 3D this means 27 cells. If the cell size is chosen accordingly, a given cell contains only a fraction of the total particles of the simulation, the neighboring n_{avg} particles at average.

Since all kernel functions we have chosen to implement drop off above the smoothing length h, The cell size is chosen as h itself. This allows any particle to find all neighboring particles in a radius of h.

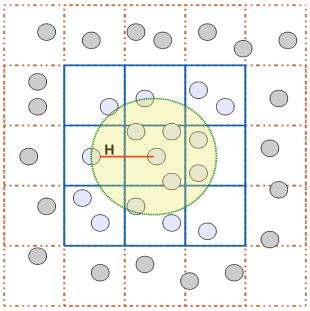


Figure 4.2: Neighboring particles and the cells that are checked for a 2D simulation

Algorithm

The algorithm we have developed for the neighbor list is based on three phases: the initialization, the loading and the finding phase. All of this is contained in a suitable class called *NeighbourList*.

In the initialization phase, performed once at the start of the simulation, the three-dimensional grid is created with a chosen cell size, dependent on the global smoothing length of the system, and with a chosen cell number. By increasing the cell number a higher performance can be achieved at the expanse of memory occupancy.

In the loading phase, performed at the start of each simulation step, the corresponding cell for all particles is found, arranging them with a linked list for each cell. The corresponding cell is found according to the particle's position thanks to the source code 4.2, which clamps the position of the particle around the grid size. Because of this, the simulation volume is virtually infinite. However, there is a drawback as particles in very distant positions could belong to the same cell. This is not a problem for the SPH computations as their distance would be way too big for the particles to interact, but it could impede performance if the cell number is not high enough. For our simulations, a cell number of 32^3 has been chosen, which has proved successful.

```
int NeighbourList::findCell(tParticle* pt){
    int posx = ((int)(pt->pos.x/(CELL_RES*h))% N_CELLS);
    if (pt->pos.x < 0) posx += N_CELLS-1;
    int posy = ((int)(pt->pos.y/(CELL_RES*h))% N_CELLS );
    if (pt->pos.y < 0) posy += N_CELLS-1;
    int posz = ((int)(pt->pos.z/(CELL_RES*h))% N_CELLS ) ;
    if (pt->pos.z < 0) posz += N_CELLS-1;
    return posx + posy * N_CELLS + posz* N_CELLS * N_CELLS;
}
```

Source code 4.2: Function for finding the appropriate cell given a particle

In the neighbor find phase, which is performed at each SPH computation for all particles, each particle checks which cell it belongs to according to the above code, then retrieves a list of all neighbor cells. During the following computations it then checks the interactions with the particles belonging to the 27 neighbor cells and no more. The functions for the neighbor find phase are shown in the source code 4.3.

4.2.3 Kernel functions

All kernels are contained in the *SPHKernel* class. This class provides an interface for the computation of kernel values according to the chosen kernel shape. All the kernel functions proposed in section 2.4 have been implemented apart from the cubic spline and Gaussian kernel, in order to impose a cut-off value equal to the smoothing length for all calls.

For the computation of kernels, two modes are available: pre-computed or real-time. By using the pre-computed kernel mode, all calculations for the various kernel functions are done in advance at the start of the simulation and the results are stored in arrays of fixed size. Whenever a kernel value is needed, a simple access is performed. The advantage of this approach is in performance, as additional computations are avoided during the simulation. The disadvantage lies in the greater memory needs and, most importantly, in the impossibility to change the smoothing length. Another implemented variant of the pre-computed mode uses hashed tables for the kernel value access phase. This avoids the necessity of computing the distance between particles at each look-up for those kernels, such as with the Poly6 kernel function (2.44) that needs just the squared distance. With the real-time kernel mode, on the contrary, all calculations are done at each kernel value look-up. This can hinder performance if smoothing kernel functions are not chosen accordingly. However, due to the possibility for variable smoothing lengths and how this can increase performance, in addition to the necessity of

```
void NeighbourList::findNeighbours(tParticle* pt, int neighs[]){
    int i = findCell(pt);
    int n1[9],n2[9],n3[9];
    findNeighboursZ(i-(N_CELLS*N_CELLS*(N_CELLS-1))>=0 ? i-(
        N_CELLS*N_CELLS*(N_CELLS-1)) : i+N_CELLS*N_CELLS,n1);
    findNeighboursZ(i,n2);
    findNeighboursZ(i-N_CELLS*N_CELLS<0 ? N_CELLS*N_CELLS*(
        N_CELLS-1)+i : i-N_CELLS*N_CELLS,n3);
    for (int ii = 0; ii < 9; ii++){
        neighs[ii] = n1[ii];
        neighs[ii+9] = n2[ii];
        neighs[ii+18] = n3[ii]; }
}
// Finds the neighbours at a constant y
void NeighbourList::findNeighboursY(int i, int neighs[]){
    neighs[0] = i%N_CELLS == 0 ? i-1+N_CELLS : i-1;
    neighs[1] = i;
    neighs[2] = i%N_CELLS == N_CELLS-1 ? i+1-N_CELLS : i+1;
}

// Finds the neighbours at a constant z
void NeighbourList::findNeighboursZ(int i, int neighs[]){
    int n1[3],n2[3],n3[3];
    findNeighboursY(i%(N_CELLS*N_CELLS)-(N_CELLS*(N_CELLS-1))>=0
        ? i-(N_CELLS*(N_CELLS-1)) : i+N_CELLS,n1);
    findNeighboursY(i,n2);
    findNeighboursY(i%(N_CELLS*N_CELLS)-N_CELLS<0 ? N_CELLS*(
        N_CELLS-1)+i : i-N_CELLS,n3);
    for (int ii = 0; ii < 3; ii++){
        neighs[ii] = n1[ii];
        neighs[ii+3] = n2[ii];
        neighs[ii+6] = n3[ii];
    }
}
```

Source code 4.3: Functions for finding the neighbor cells given a particle

simulating different fluids with different smoothing lengths, we have decided to settle for this solution.

An API for easy access to the kernel functions has been developed. All kernel values look-ups are performed through the use of three functions: *getKernel*, *getKernelGrad* and *getKernelLapl*, corresponding to the base kernel, its gradient and its Laplacian. The user needs only to call the corresponding function using as parameters the *KernelType* enumeration constant which determines the chosen kernel shape, the two particles interacting (or their distance) and the smoothing length.

In addition, default kernels are imposed by the simulator if none is specified. These kernels follow the considerations discussed in section 2.4.

4.2.4 Rigid bodies

In our implementation, all rigid bodies extend the *RigidBody* class and are created by defining their shape, position and mass. As discussed in section 3.5, the implementation is based on a quasi-fluid approach. The initialization phase creates the rigid particles inside the rigid body, positioning them on three layers as seen in figure 4.3. The parameters are then assigned to each rigid particle. The rigid body total inertia I is computed according to the rigid particles's position.

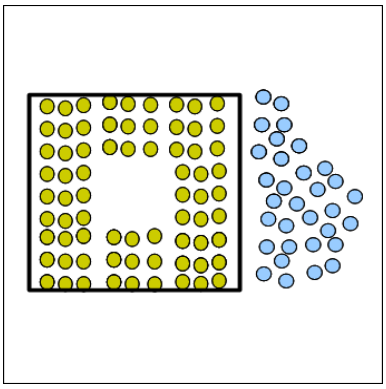


Figure 4.3: Quasi-fluid rigid bodies: particle position

The algorithm for the evolution of the rigid body dynamics is reported here as implemented in the work related to this thesis.

In the algorithm, v_R, x_R, ω_R and θ_R are respectively the linear velocity, position, angular velocity and angle of rotation of the center of mass of the rigid body. F_{ext} and T_{ext} are the total external force and torque acting on the body and M_R and I_R are its total mass and rotational inertia. N is the total number of particles inside the rigid body and b_i is the lever arm of particle i, which is basically $x_i - x_R$. The algorithm is as follows:

- Gather external forces (*i.e.* gravity)
- Apply forces to the whole body and derive temporary cinematic properties

$$v'_R = (g + \frac{F_{ext}}{M_R})\Delta t + v^t_R$$

$$x'_R = v'_R \Delta t + x^t_R$$

$$\omega'_R = \frac{T_{ext}}{I_R}\Delta t + \omega^t_R$$
$$\theta'_R = \omega'_R \Delta t + \theta^t_R$$

- Reflect the rigid motion on each rigid body particle i

$$v'_i = v'_R + b^t_i \times \omega'_R$$
$$b'_i = b^t_i + (b^t_i \times \omega'_R)\Delta t$$
$$x'_i = x'_R + b'_i$$

- Apply local forces to particles during the SPH computations and update the local velocities (with j indicating another particle)

$$v''_i = \sum_j \frac{f(i,j)}{\rho_i}$$

- Compute the linear and angular speed of the whole rigid body

$$v^{t+1}_R = \frac{\sum m_i v''_i}{M_R} = \frac{\bar{m}\sum v''_i}{M_R} = \frac{\sum v''_i}{N}$$
$$\omega^{t+1}_R = I_R^{-1} \sum b'_i \times m_i v''_i$$

- Compute the final position and angle of the whole rigid body

$$x^{t+1}_R = v^{t+1}_R \Delta t + x^t_R$$
$$\theta^{t+1}_R = \omega^{t+1}_R \Delta t + \theta^{t+1}_R$$

4.3 Rendering

The rendering aspect of the simulation has been considered in this work, although with a lower priority than the interactive and physical aspects. Our aim is achieving real-time and realistic rendering of different fluids with various degrees of verisimilitude.

4.3.1 Known methods

Particle systems have been used for many years in computer graphics and many techniques have been created for their visualization. Some useful techniques are reviewed here and their features are detailed. The problem of

particle system rendering lies in their meshless nature and thus the lack of a proper surface. Because of this, the rendering methods must usually find a way to reconstruct the surface.

Billboards

A first simple method, used even in the first particle systems in the work of Reeves [1983], places billboarded textures at particle positions. Billboarded textures are small two-dimensional images that are always oriented towards the viewer. This method is useful for smoke or flame rendering, especially when paired with a color value for each particle that varies with the heat of the particle, but results are poor for liquid rendering.

Marching cubes

Created by Lorensen and Cline [1987], the marching cube algorithm is a method for 3D surface reconstruction that creates a triangle-based mesh from a set of voxel data. This method is widely used in the rendering of Eulerian grid based fluid simulations, as the fixed spatial grid renders the method easy to implement. The basis of this algorithm is a list of graphical primitives which are selected by using the data around a single voxel. These primitives, when combined together, form a continuous surface.

With non-grid based methods the algorithm can still be used, such as in the case of meta balls rendering (Bourke [1997]). The algorithm can thus be used for meshless models, such as with SPH and particle systems, and renders a dense fluid-like material. However, this can be computationally costly in respect to more advanced methods.

Screen space meshes

A more recent and advanced method, useful for achieving real-time rendering, has been proposed by Müller et al. [2007]. The idea behind the screen space meshes method, as the name implies, is to render only the visible shape resulting from the cloud of points, thus limiting its use to single-view rendering.

Using this method, an initial depth map and silhouette is created in screen space, creating a 2D triangle mesh. The result is then transformed back in 3D space for additional rendering computations, such as refractions and lighting.

This method achieves a good effect for liquids without the costly operations involved in a marching cubes algorithm and it has been recently

extended in order to increase its realism in the work of van der Laan et al. [2009].

4.3.2 Implementation

Since our simulator has been created with testing in mind, particle rendering is given a *visual aid* purpose. However, additional methods could be implemented upon the existing system.

Render modes

Different render modes have been implemented. With the use of a keyboard shortcut the various modes can be toggled. All choices are contained in the *RenderType* enumerator.

- **Points**

  The particles are shown as small points. This display mode achieves great performance at the expanse of the loss of depth. This mode is not suited for realistic rendering.

- **Aliased lines**

  The particles are shown as small aliased lines, drawn in their direction of travel according to their speed. This mode is useful for the visualization of fast motion and for the visual rendering of sparks.

- **Vectors**

  All particles are shown as a point and a vector pointing in a chosen direction. This display mode is useful for the visualization of the properties of a fluid as vector fields. Different values can be chosen as vectors: speed, pressure force, viscosity force, surface tension and surface normal.

- **Quad**

  This fast display mode shows all particles as small squares. This mode retains most of the performance of point visualization and adds depth as the square size changes according to particles position.

- **Billboard**

  This display mode renders all particles as billboarded sprites. An image can be loaded into the simulator and is rendered as a small square with transparency. Color hue change is supported. This method is

widely used in particle systems. The visual effect is good for fire and smoke rendering, (Foster and Metaxas [1997]), but it looks unnatural for liquids.

- **Sphere**

  This display mode is the slowest among those implemented by us but it is also the most visually appealing. It renders all particles as small spheres. The GPU code improves this visualization mode by making use of a smooth sphere particle shader and virtual buffer objects (Green [2010]).

Particle colors

The color of individual particles in an emitter can be chosen by the user as fixed or as a function of a particular parameter. The different choices available are useful for the visual evaluation of the fluid's properties. The possible choices are available in the *ColorType* enumerator as follows:

- **Preset**

  A preset color can be assigned to all particles of an emitter. This is useful for the visualization of multiple fluid interactions.

- **Life**

  The color of the particles can be chosen as a function of their lifetime. This can be useful for the rendering of flames or smoke, with particle color fading to black as the particles reach their maximum life. Both a start and end color can be chosen and the color is interpolated between the two.

- **Density**

  Color can be based on density, with above rest-density particles having a strong red hue and below rest-density particles having a light blue color. This can be useful for the visualization of fluid compressibility.

- **Speed**

  Color can be based on speed, with faster particles having a yellow hue and slower ones having a blue shade. This can be useful for the visualization of the velocity gradient of the fluid.

- **Neighbor Cells**

  The color of the particles can be based on their assigned neighbor cell. This is useful for the visualization of neighbor cells position in the 3D grid.

- **Normal**

  The color of a particle is chosen as red if it is on the surface of the fluid, blue otherwise. This is useful for surface reconstruction.

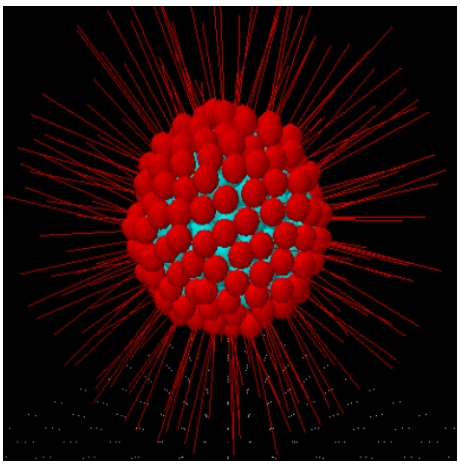


Figure 4.4: Fluid rendered with its surface normals and surface particles marked as red

Rigid bodies and boundary rendering

In this work, various display modes have been developed for rigid body and boundary rendering, again with a focus on performance and testing. The different choices can be found in the *RigidRenderType* enumerator.

- **Solid**

  This display mode renders the body shape with a solid user-defined color, with computed normals for lighting. It is a simple and effective method for realistic rendering. Transparency is also supported.

- **Wireframe**

  Useful for boundary rendering, the wireframe display mode shows only the edges of the body, allowing the user to see the fluid inside or other elements behind it.

- **Point**

  All rigid bodies are rendered as clouds of points, corresponding to the fixed body particles. This mode is useful for analyzing the interactions between solid and fluid particles. The center of mass is given a red hue.

- **Velocity**

  All particles are shown as lines pointing towards their direction of travel, with magnitude tied to their speed. This display mode is useful for analyzing solid and fluid interactions.

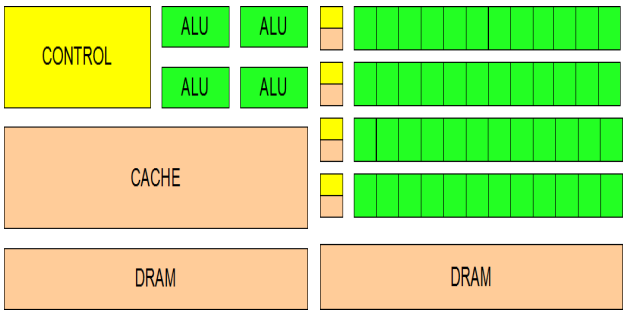
4.4 Parallelization

The SPH model proposed and implemented thus far can be used to achieve good performance, with a simple SPH model containing a few hundreds particles achieving real-time frame rates. However, for more complex and accurate simulations, a few hundreds of particles are not enough and, as has been already discussed, a more complex model is needed for realistic simulations, which requires additional computations.

In order to achieve interactive frame rates for a complex model, the trend in the recent years has been to focus on parallelization, that is the use of a Graphic Processor Unit (GPU) for the implementation of the model.

A modern GPU has a very different architecture compared to Central Processing Units (CPU). Since the purpose of a CPU is the serial computation of simple binary operations, its classic architecture sports many registers (for memory usage), a Control Unit, a few Arithmetic and Logical Units (ALU), several buses for data transfer and a fast cache for data memorization. On contrast, a GPU consists of many parallel ALUs, each with its own register, while the control and local memory parts are given less importance, because it is designed for highly parallel computation, exactly what graphics rendering is about.

Thanks to its peculiar architecture, the GPU can and has been used as a parallel machine, with hundreds or thousands of threads running at the same time on the chip, while a CPU can only sustain one at a time (or, with recent multi-core CPUs, up to four). A great range of problems can be solved with a parallel approach instead of a serial one with a terrific increase in performance and this is why interest on GPU computing has grown increasingly for many different fields, not only in the graphic industry, but even for chemistry, mathematics and physics, spanning a few new programming environments and extensions to classic programming languages. Among these environments, the Compute Unified Device Architecture (CUDA), property



(a) CPU (b) GPU

Figure 4.5: Architecture differences as explained in the NVIDIA CUDA Programming Guide

of NVIDIA, is at this moment the most used and most supported architecture.

4.4.1 CUDA basics

CUDA allows a programmer to use extensions for common programming languages, originally for C, in order to take advantage of the GPU's processing power. Due to the fast-growing power of GPUs, CUDA has been designed with a scalable programming model in mind, which allows parallel programs to scale their performance with the number of processors the assigned device is equipped with. The model that allows all of this is based on thread grouping. A fixed number of threads are grouped into a block and many blocks are allocated to a single processor in the device. The hardware underneath makes sure to distribute the blocks according to its architecture, while the programmer only has to designate the dimensions of the grid of blocks and of the single blocks.

Both the grid and block dimensions can be defined in one, two or three dimensions, in order to mimic the domain of the problem at hand. For our 3D simulation problem, this has been taken into account.

Kernels

Using CUDA, the programmer is able to write GPU code alongside CPU code, mixing serial and parallel code effectively. Due to this, the parallel (GPU) code can be used as an extension to already existing serial (CPU) code. Serial code is contained in host functions, while parallel code must be executed in kernel functions. Kernel functions are defined by the declaration specifier **__global__** and by the dimensions of the grid and blocks, which

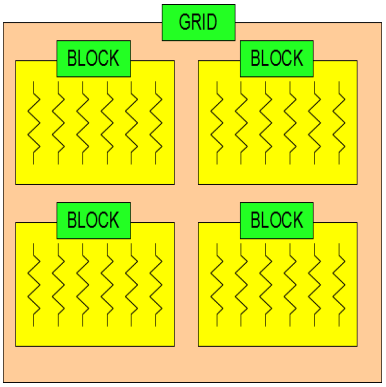


Figure 4.6: CUDA execution model

determine the number of parallel threads, as shown in the example source code 4.4. Kernel functions are imposed to have a void return type.

```
__global__ void kernelFunction(int param){
        // Parallel code
}

int main(){
        // Serial code
        ...
        // Kernel invocation
        kernelFunction<<< gridDim, blockDim >>>(param);
}
```

Source code 4.4: Example CUDA kernel launch

The programmer can also specify routines that can only be called by parallel code with the specifier **__device__**.

Threads

When a kernel is invoked, all threads execute in parallel the instructions included in the kernel function. All threads can access the dimensions of the grid and the block as well as the position of the block inside the grid and the position of the thread inside the block. This information can be accessed thanks to the variables *gridDim*, *blockDim*, *blockIdx* and *threadIdx*. As such, an unique thread identifier can be computed as seen in the source code 4.5.

```
__device__ getThreadUniqueId(){
        return threadIdx.x + blockDim.x*threadIdx.y + blockDim.x
            *blockDim.y*threadIdx.z.
}
```

Source code 4.5: Unique thread identifier in CUDA

Memory

Memory hierarchy must be taken into account carefully when working with parallel code, since the memory available on the GPU device is often limited and subject to additional restrictions.

NVIDIA devices have different memories available with different sizes and scopes. The CUDA programmer must fully understand the differences among the different memory types in order to take advantage of parallel computation effectively, as memory usage can be a potential bottleneck if not properly approached.

- **Global Memory**

  Global memory resides on the device and can be accessed by all kernel threads. It is slower but has a greater capacity than other memories. Global memory must be allocated prior to the kernel launch using the dedicated CUDA allocation functions. Since global memory is linear, particular care must be taken as threads computing in parallel should access different and subsequent memory addresses in groups of 16, 32 or 64 bytes in order to assure a *coalescing* behavior. If this does not happen, memory accesses is automatically serialized and this can greatly impede performance, neglecting the advantages of parallelism. Note that newer devices of computational capability 2.0 or higher, such as the GTX480 we use for our simulations which sports a NVIDIA Fermi architecture, greatly reduce these requirements.

- **Shared Memory**

  Shared memory is accessed by all the threads in a single block. This memory is smaller but has lower latency than global memory and the opportunity to replace global memory with shared memory should be taken where possible.

- **Local Memory**

  Each thread has a private fast local memory area which is quite small. Care must be taken as memory exceeding the local memory constraints overflows to global memory.

- **Constant Memory**

  Constant memory is really fast but it can be written only by host code and therefore parallel threads can only read it. It is useful for storing parameters that must be accessed by all threads and that do not vary during kernel launches. However, constant memory still presents some issues, such as its context scope which obliges the programmer to put all calls to constant memory in the same file.

- **Texture Memory**

  Texture memory works similarly to constant memory, but it is optimized for two-dimensional access and thus is better used for 2D domains.

Warps

In addition to the global memory's constraints, special care must be taken on per-thread memory access and execution. Threads are grouped in warps of 32 and each warp executes independently in a block. Because of this, all 32 threads execute in parallel inside a single multiprocessor and their execution should be convergent. If threads diverge, as in the case of improperly balanced conditionals, a performance loss could be noticed.

Compiling

CUDA code must be put into .cu files, which are extensions of simple .c files in which CUDA routines can be called and the extension's identifiers are supported. The CUDA Toolkit comes with a compiler called NVCC that can compile .cu files. The compiler however calls the C or C++ compilers when used on .c or .cpp files, allowing the programmer to mix the different extensions. In order to take advantage of the compute capability 2.0 features, the option $-gencodearch = compute\_20, code = sm\_20$ must be used when compiling.

4.4.2 Parallel SPH implementation

A parallel version of our SPH model has been implemented in order to take advantage of the huge performance benefits that a GPU can provide. The model is based on a simple version of what has been discussed in the previous chapters, but it could be extended to incorporate all solutions that have been provided. In the simple model only the variations that have been observed to behave more correctly for the system have been implemented.

For the parallel code, the implementation of the SPH system has been rewritten in C++, C and CUDA. The window and input management is provided by Xlib while the graphical aspects are still coded using OpenGL. We also took advantage of the CUDA-OpenGL interoperability.

This work is based on the Particles demo available in the CUDA Software Development Kit (SDK). The implementation follows the work of Green [2010] and expands on the ideas presented in the document.

Initialization

The initialization phase is similar to the CPU code (see section 4.2.1), but an additional step is done in order to initialize the CUDA context and the interoperability between CUDA and OpenGL.

In addition, during the initialization phase of the SPH system, arrays are allocated for particle positions, velocities and their physical parameters. Much like Green [2010] does, the position and velocity values for each particle are saved in a groups of four floats, which means 32 bytes, in order to assure coalescing (see 4.4.1). The parameters of the simulation are allocated in an array of floats both in global host and device memory.

For better performance, constant memory should be used, but the constraints on the coding of constant memory declarations contrast with the need for different classes and files in order to increase the usability of the code. As such, we have preferred to use global memory.

Block dimensions are imposed as 8*8*8, with a total of 512 threads per block, half of the maximum possible but chosen as to allow symmetry alongside the three axes. The grid dimensions are based on the total number of particles and their initial position, dividing the number of particles per length by the block's length, effectively taking advantage of the three-dimensional domain of the problem.

Two of the emitter types discussed in the previous chapters have been implemented: the fluid box and the square fluid flow. Both take advantage of the three-dimensional block and grid dimensions. The fluid box emitter is initialized with the source code 4.6.

```
        pos.x = threadId.x * d;
        pos.y = threadId.y * d;
        pos.z = threadId.z * d;
```

Source code 4.6: CUDA fluid box initialization

The fluid flow emitter is initialized with the modified algorithm, as can be seen in the source code 4.7. In this case, each particle is given a lifetime

value and is emitted at periodic times in a two-dimensional section.

```
        pos.x = threadId.x * d;
        pos.y = threadId.y * d;
        pos.z = 0;
```

Source code 4.7: CUDA square fluid flow initialization

Main algorithm

The main loop of section 4.2.1 is extended with parallel code. At the beginning of each step the parameters are updated in device memory with a *memory set* call, then the update routine is launched.

The update routine launches a subsequent kernel for each needed SPH step, with one thread per particle. The usage of many different kernels is needed for thread synchronization, as the kernel functions contain loops regarding neighboring particles, whose number is variable for each particle. This gives birth to thread divergence and even calls to the **__synchthreads()** routine, which should synchronize all threads in a kernel, does not suffice.

After each kernel launch, errors are checked with a suitable function and the serial and parallel code is synchronized. This allows the program to wait for the kernels to finish before advancing to the next step.

All particle kernels share the same base implementation, which can be summarized in the example kernel launch in the source code 4.8. C wrappers are used so that CUDA functions can be called by C++ code and our utility functions.

The following kernels are therefore launched in succession:

- **Emission**

  Particles are emitted at each time step according to the period of emission and the shape of the emitter. In the case of the fluid box emitter, this happens once at the first step. In the case of the fluid flow emitter, the particles are emitted periodically.

- **Neighbor search**

  The neighbor search is based on an uniform grid implementation and provides each particle with an array of neighboring particle indexes to be used in the following steps.

```
__global__ void doSomethingForEachParticle(
                        float* posArray,
                        float* velArray,
                        devParams){

        uint index = getGlobalIndex();
        if (checkEmission(index,devParams)) return; // Check if
            the particle is emitted

        // Get particle values from the global arrays
        float4 pos = posArray[index];

        // Do something to the particle
        ...

        // Write back to the global arrays
        posArray[index] = pos;

}

extern "C"{
        void cuDoSomething(
                        float* pos,
                        float* vel,
                        float* hostParams,
                        float* devParams){

                dim3 block = getBlockSize(hostParams);
                dim3 grid = getGridSize(hostParams);

                doSomethingForEachParticle<<<grid,block>>>(
                        (float4*)pos,
                        (float4*)vel,
                        devParams);

                cuCheckLastError("DoSomething");
                cuSync(); // For host-kernel synchronization
        }
}
```

Source code 4.8: Example complete CUDA kernel launch

- **Integration - Before Step**

  The before-step update of the integration scheme is advanced using the Modified Velocity Verlet integration algorithm, chosen as explained in section 2.3.1.

- **Density and Pressure**

  For each particle, its density is computed using the standard density summation approach (equation 2.10), using the neighboring indexes provided by the neighbor search step. Since pressure is independent for each particle, we also compute it for each particle according to Tait's equation of state (2.15). Density and pressure values are stored in arrays in global memory. The near-pressure correction of tensile instability as explained in section 3.1 has been also implemented.

- **Viscosity and Pressure Forces**

  Viscosity and pressure forces are computed in a single pass using Müller's SPH equations. Forces are stored in global memory.

- **Integration - After Step**

  The after-step update of the integration scheme is advanced using the Modified Velocity Verlet integration algorithm, chosen as explained in section 2.3.1.

- **Boundary Collisions**

  Boundary collisions with the boundary box are checked. For the simple system, the penalty approach as discussed in section 3.5.1 has been chosen due to the presence of a single phase and since the high number of particles diminishes interface problems. Dimensions and bounce and slip factors can be assigned.

- **Life Check**

  At last, in the case of timed particles, those particles whose life has ended are removed or respawned if needed.

Parallel neighbor list

The neighbor list is created with an uniform grid implementation as suggested by Green [2010]. A two-dimensional array of grid cells $gridCells$, that contain the particles, and an array of grid cell counters $gridCellCounters$, that store the number of particles for each cell, are saved in global memory. A kernel for parallel neighbor list creation is launched. Each particle finds the index

of the cell it belongs to with the same algorithm used in CPU code, then places itself in the cell's list and increases the cell counter. The uniform grid approach is extended in this work with an infinite grid.

By taking advantage of CUDA's atomic operations, available on devices of compute capability 2.0, the cell counter can be increased independently from other particles, although this could lead to serialization when many particles are added to the same cell. However, since each cell usually contains a small number of particles, the uniform grid has been observed as being good enough for our purpose.

The big difference from the CPU code (see section 4.2.2) lies in the static nature of this implementation since dynamic linked lists are replaced by static arrays. As such, the maximum number of cells and the maximum number of particles for each cell must be correctly assigned.

Rendering

CUDA provides the programmer with a series of functions that guarantee the interoperability with OpenGL on the same GPU device. Since both environments operate on the graphics processor, if care is not taken, their interaction could lead to improper memory accesses.

A Virtual Buffer Object (VBO) can be mapped to either the CUDA or OpenGL context. This allows the programmer to switch between the parallel operations and rendering with ease. Because of this, the particle position array is stored as a VBO and we map it to CUDA when performing parallel computations and to OpenGL when rendering directly from the VBO.

In addition, a VBO is mapped for the color property of each particle in order to render the color values directly from it. Different color gradients can be chosen for each simulation, based on the life, density or velocity magnitude of particles, and are assigned with a suitable kernel launch.

The example shader contained in the CUDA SDK's particles demo is used (Green [2010]). Rendering is possible as points or as shaded spheres.

Once again, a function for screen saving has been added, that uses the Cairo libraries for image manipulation and saving. (see Cairo [2010]).

Utilities

For ease of use, a set of function wrappers written in C has been implemented so that the use of CUDA routines from C++ code is made possible. The utility functions allow the programmer to allocate, set, copy, print and free device memory, map and unmap VBOs, check for memory info usage and handle errors. In addition, our utilities include vector operations as exten-

sions of the CUDA SDK examples and of our own *tVector* class used in the serial code.

4.5 Conclusions

In this chapter, the implementation of our SPH model is presented. Extensions and optimizations created in order to increase the performance and versatility of our SPH simulator are detailed.

In section 4.1, the implemented SPH simulator and the guided creation phase, which allows the system to automatically compute most of the critical parameters of the simulation, are explained in detail.

In section 4.2, the algorithms used for the creation or optimization of our SPH simulator are explained.

In section 4.3, the final rendering of the fluid is addressed, discussing known approaches and presenting the rendering options of our SPH simulator.

In section 4.4, a parallel implementation of the SPH model on a GPU device is presented, in order to greatly increase the performance of the simulation.

Chapter 5

Experimental results

The proposed SPH model has been validated on a test case that has been provided by domain experts: the water jet test case. In this chapter, the results of this validation are presented. In addition, several tests are performed in order to highlight the versatility of the SPH model. The performance gain of the GPU implementation over the CPU implementation is reported.

5.1 The water jet test

The water jet test case involves the formation of a small hole in a tank filled with water, a possible fault appearing in chemical plants. At the beginning of the test, the tank is filled with water up to a height h_{max}, then, a hole of chosen radius r_{hole} is created at an assigned height h_{hole}. Water flows at constant speed v out of the hole until there is no more liquid, creating a jet that travels up to a maximum range x_{range}, where it touches the ground. A representation of the test is shown in figure 5.1.

For this test, we are interested in simulating with our SPH model the water jet trajectory and its section with accuracy. In order to validate our SPH model, we compared its results to the data gathered from the AXIM model, that is an analytical and physically realistic model of a water jet (Brambilla and Manca [2009a], Brambilla and Manca [2009b]).

5.1.1 AXIM fluid jet model

Data regarding the water jet test case has been gathered using the AXIM fluid jet model. The model is adapted from the work of Clark [1988] and it is based on a mathematical representation of the physics of a jet of liquid, providing accurate analytical solutions for trajectory, velocity and jet radius evolution.

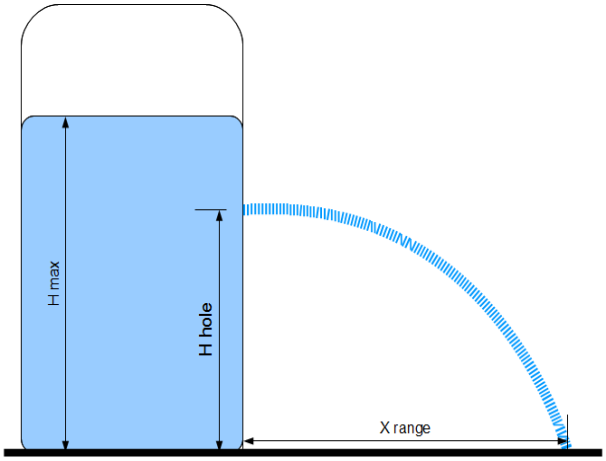


Figure 5.1: Water jet test case

The model has also been validated by Brambilla and Manca [2009a] with experimental data.

Using the AXIM model, the liquid has been initialized with the parameters of pure water at a temperature of 25 °C, with a density of 1000.0 kg/m^3, a dynamic viscosity coefficient of $8.9 \cdot 10^{-4}$ $Pa \cdot s$ and a surface tension coefficient of 72.8 mN/m. The water tank is placed on the ground and filled to a height of $h_{max} = 6.5$ m, with a hole of radius $r_{hole} = 0.1$ m positioned at $h_{hole} = 3.5$ m from the ground. The axis of the hole is parallel to the ground. The water is subject on the free surface and at the hole to the atmospheric pressure (101325 Pa), while the coefficient of discharge of the hole is set to 0.61.

Using these parameters, the speed of the water flow from the hole, computed using the AXIM model, is 4.76 m/s. The range of the water jet is $x_{range} = 3.97$ m, that is the distance alongside the x-axis from the hole to the point at which the water is assumed to hit the ground.

The model is used to compute the analytical trajectory of the water jet, as shown in figure 5.2. In the plot, y is the jet height from the ground and x is the range of the jet, both measured in meters.

The model is also used to compute the analytical section of the water jet alongside its whole trajectory, with a methodology derived from the work of Mashayek et al. [2008]. The data is shown in figure 5.3. In the plot, y is the dimension of the water jet section (or, more accurately, half of the section, because it is assumed to be symmetric) alongside the y-axis measured in meters and t is the time in seconds at which the section is evaluated alongside the fluid's trajectory.

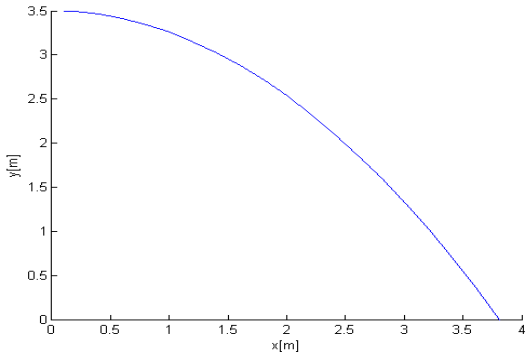


Figure 5.2: Water jet trajectory computed using the AXIM model

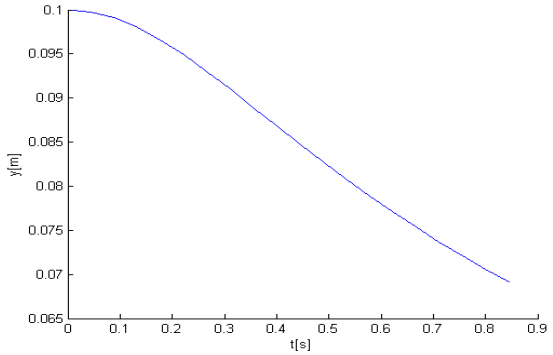


Figure 5.3: Water jet section computed using the AXIM model

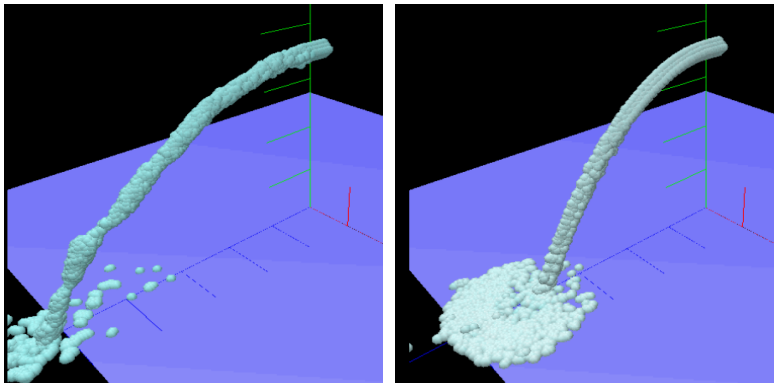
5.1.2 SPH simulation of the water jet test

Our SPH simulator has been applied to reproduce the water jet test case and the results are compared with the data gathered from the AXIM model. The test is performed using the CPU code.

A circle flow emitter is created at the position $(0.0, 3.5, 0.0)$ using the inlet flow extension (see section 4.1.6). Just like with the AXIM model, the liquid is initialized with a rest density of 1000.0 kg/m^3, a viscosity coefficient of $8.9 \cdot 10^{-4}$ $Pa \cdot s$ and a surface tension coefficient of 72.8 mN/m. The number class of the simulation, that defines the number of particles, is 3, thus a total of 32768 particles are used, so to have enough particles for an accurate simulation. The hole is again created with a radius of 0.1 meters and its axis parallel to the ground. The initial speed is chosen as obtained from the Axim model: 4.76 m/s. The SPH integration is performed using a physical time step of 0.001 s, using the Modified Velocity Verlet algorithm (see section 2.3). In order to use the Tait's equation of state for pressure computation

and thus achieve better performance (equation 2.15), the speed of sound of the water is defined as 1.5 m/s, that is three orders of magnitude lower than the actual water's speed of sound. This has however the effect of increasing the compressibility of the fluid. The SPH equations and kernels are based on Müller's variants (section 2.2).

The continuity approach is used for the computation of density because the jet has a large free surface and we thus diminish surface tension problems. If a density summation approach were to be used, a large spurious surface tension would arise. The difference between the two approaches can be seen in figures 5.4a and 5.4b. Our modified double density relaxation extension is also enabled (see section 3.1). The boundary that represents the ground is a flat plane placed at the origin, with no bounce or slip coefficient, and uses the simple and performant penalty-based approach (see section 3.5.1) since our aim is to model the jet, not the formation of the pool on the ground.



(a) Summation (b) Continuity

Figure 5.4: Comparison of the density computation approaches for the water jet test

The trajectory of the water jet simulated with the SPH model is estimated and the comparison with the data retrieved from the AXIM model is shown in figure 5.5. In the plot, y is the height of the water jet from the ground and x is the range of the jet, both measured in meters. The trajectory simulated with SPH is consistent with the AXIM model data, apart from an error that is due to jet fragmentation as the jet travels. The error plot can be seen in figure 5.6.

The section of the water jet simulated with SPH is retrieved as well and a comparison is made (see figure 5.7). In the plot, y is the dimension of the section alongside the y-axis, measured in meters, while t is the time in seconds at which the section is evaluated alongside the fluid's trajectory.

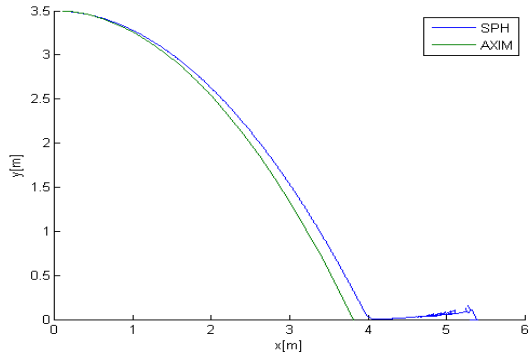


Figure 5.5: Comparison of the trajectories computed with the AXIM and SPH models

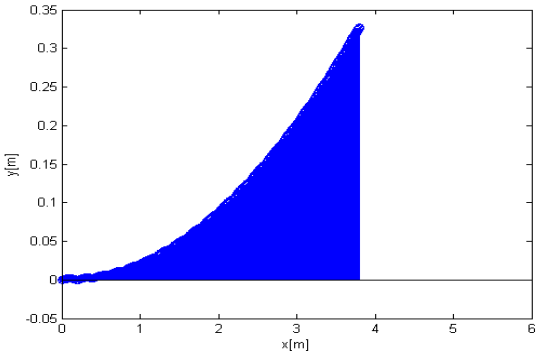


Figure 5.6: Error of the trajectory computed with the SPH model compared to the AXIM model

The error between the two models can be seen in figure 5.8 and it highlights a problem with the SPH model. The error appears because the SPH simulation is subject to the compressibility problems arising from the choice of Tait's equation of state, which has been chosen for performance reasons. In addition, using a continuity approach for the density computations an error is added when the particles' velocities are too similar, as in this case.

We think that by simulating the air around the jet, in order to remove the spurious surface tension, and using a density summation approach the simulation will behave more correctly. Because of this, either the whole volume should be modeled with SPH resulting in a huge number of particles (and in this case it would be better to choose an Eulerian method), or a new extension should be added, such as the air particle generation idea of section 3.4.2.

Alternatively, as can be seen in the GPU water jet video (at `http://`

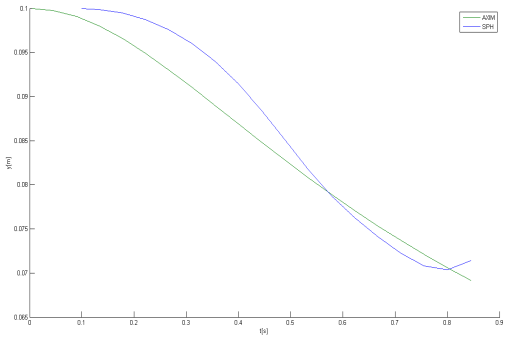


Figure 5.7: Comparison of the sections computed with the AXIM and SPH models

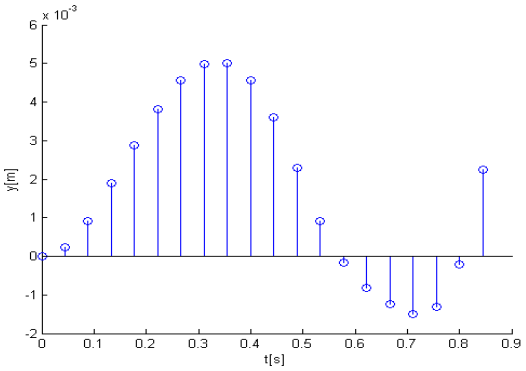


Figure 5.8: Error of the section computed with the SPH model compared to the AXIM model

`vimeo.com/32460374`), that uses the density summation approach, by using a large number of particles the section correctly diminishes as the jet velocity increases due to gravity pulling the fluid downwards (as also shown in figure 5.9). Using a large number of particles while maintaining the same average number of particles n_{avg} results in a smaller smoothing length and this, as a consequence, inhibits the effect of the spurious surface tension.

5.2 Parallel code performance

We have compared the performance of the CPU and GPU versions when used on the same example test case that has been created for the comparison. In order to focus on the physical simulation throughput, rendering time has not been taken in consideration.

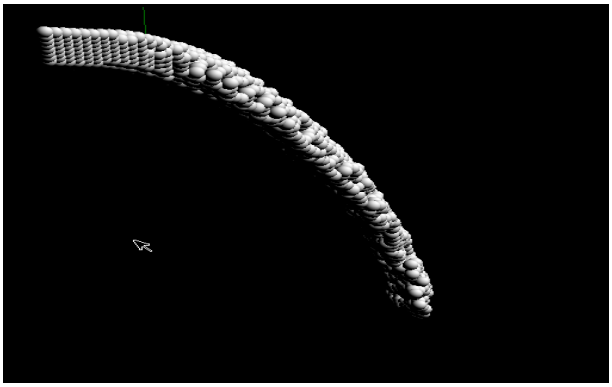


Figure 5.9: Water jet test case simulated with CUDA

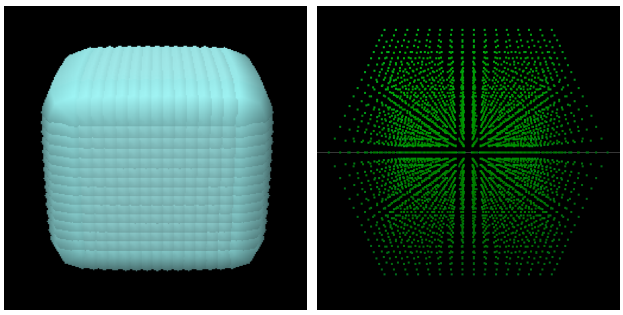
The CPU code runs in a Windows 7 environment on a 64 bit AMD Phenom 8650 triple-core processor at 2.30 GHz, with 4.00 GB of RAM. The GPU code runs in an OpenSUSE Linux environment on a 64 bit quad-core processor with 4.00 GB of RAM and a NVIDIA Fermi GTX480 GPU. Although the comparison is made on two different machines due to external requirements, the performance gain of the GPU code over the CPU code, that is as we will see in the order of 10^2, is so great that the differences between the two machines are negligible.

The test case is performed with $16^3 = 4096$ particles, the average number of particles n_{avg} is chosen as 27 and the timestep is chosen as $0.001s$. The emitter is initialized as a cube of fluid placed at the system's origin. Müller's equations, Tait's equation of state and the double density relaxation extension are used (see sections 2.2 and 3.1). No gravity force is added and no boundary is created.

The physical parameters are set as follows: $\rho_0 = 1000.0\ kg/m^3$, $c_0 = 10.0\ m/s$, $\mu = 10.0\ Pa \cdot s$, $\sigma = 0.0\ N/m$ $k_{el} = 0.0\ N/m$. In both implementations, the neighbor list is initialized with 32 cells per side.

We compare the performance results by checking the execution times when the simulation time reaches $0.005s$.

The CPU code is slow and the density and pressure computations take almost 150 ms for a single step. In addition, the surface tension step, which also computes the color values and normals of all particles (and this is done even for $\sigma = 0\ N/m$), takes almost 90 ms. The total time of the computation is 277 ms just to simulate a 1 ms step and, removing the rendering time of 36 ms, we can say that the update took roughly 241 ms. This low performance is due to serialization. Since the average number of neighbors for a single particle is 27 and since each particle must iterate over all the



(a) CPU (b) GPU

Figure 5.10: Test case for performance comparison

neighbors, we get at average $4096 * 27 = 110592$ kernel function evaluations at each time step, counting only the density computations.

Performing the example test using the parallel code, the computation time takes 3 ms for the simulation of a single step of 1 ms.

Comparing the computation time of the GPU code to that of the CPU code, we get a performance multiplier of almost 100. This can be explained as the parallel code limits severely the impact of the number of particles, since all computations are assumed to be performed in parallel with one thread for each particle.

Repeating the test with $32^3 = 32768$ particles, we observe that the CPU code takes 2677 ms. This is roughly ten times the amount it took to simulate the same period with 4096 particles.

The GPU code, on the other hand, takes only $18ms$, six times more than with $4,096$ particles. The performance in respect to the CPU code gains thus a multiplier of 148.

In these test, the GPU implementation still does not reach a real-time performance, but a slow-motion effect is achieved. It must be noted that due to the stability of the Modified Velocity Verlet integration algorithm and to the optimizations implemented in our SPH model, timesteps up to 15 ms have been taken during some of our simulations, resulting in real-time performance for up to 30000 particles at once.

In addition, more optimizations could be implemented in the parallel code in order to take advantage of constant, shared and texture memory which have not been implemented in the work pertaining to this thesis due to time constraints and thus achieve even higher performance.

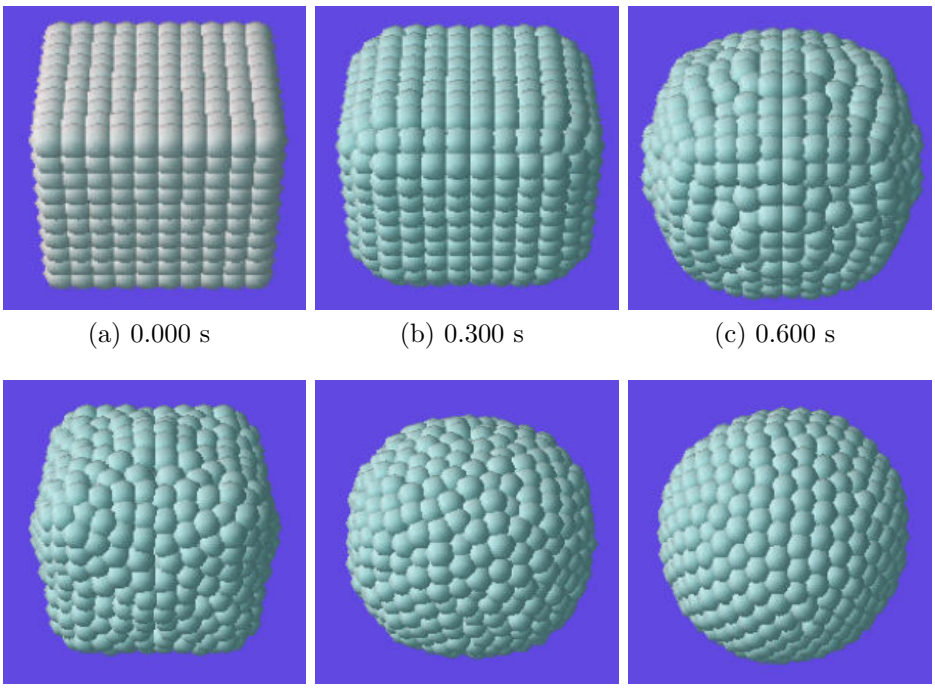
Videos showcasing the capabilities of the GPU implementation can be viewed at `http://vimeo.com/32460374` (water jet test case) and `http://vimeo.com/32460409` (water splash).

5.3 Additional results

In order to test the capabilities and the versatility of our SPH simulator, additional tests have been performed.

5.3.1 Test 1: double density relaxation

Using our modified double density relaxation algorithm as in section 3.1, an initial volume of water, shaped as a cube, is left to evolve with gravity set to zero. As can be seen in the frames in figure 5.11. the fluid tends to minimize the curvature and form a sphere, resulting in a spurious surface tension. As can be seen in the last frame, taken after five seconds, the result is stable and tensile instability is avoided due to the double density relaxation. The spurious surface effect, although uncontrollable, is usually considered beneficial for water simulations. A complete video can be found at `http://vimeo.com/32459971`.



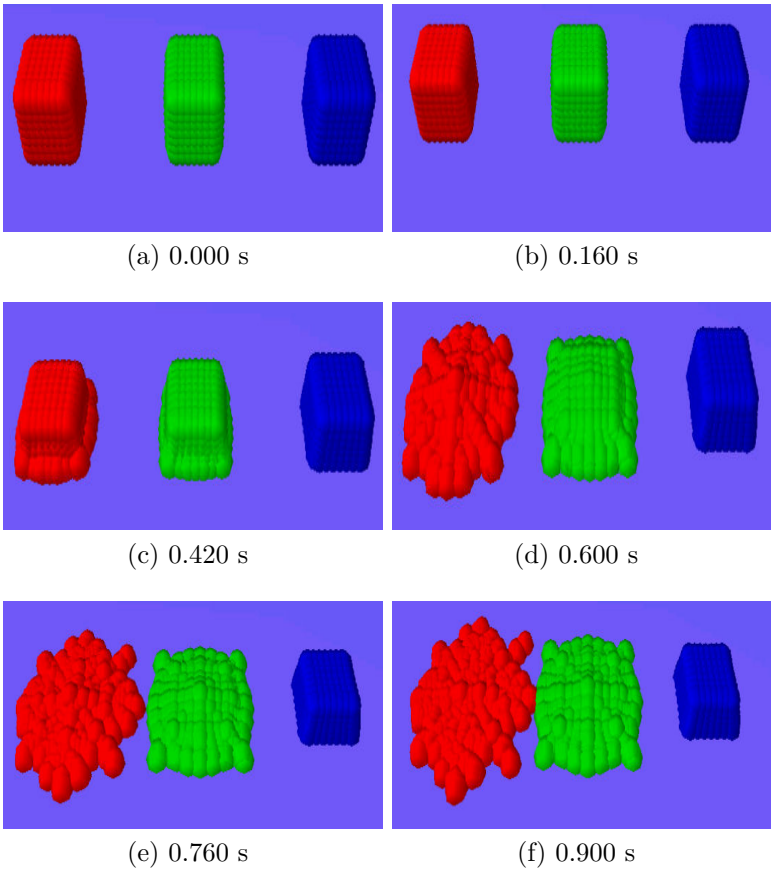
(a) 0.000 s (b) 0.300 s (c) 0.600 s

(d) 0.900 s (e) 1.200 s (f) 5.000 s

Figure 5.11: First test: double density relaxation

5.3.2 Test 2: Viscosity

Different cubes of fluid with different viscosity coefficients, lowest on the left and highest on the right, are seen landing on a flat plane. The fluids behave correctly as the more viscous fluid tends to maintain its shape for a longer time. See the frames in figure 5.12. A complete video can be found at `http://vimeo.com/32460051`.



(a) 0.000 s (b) 0.160 s

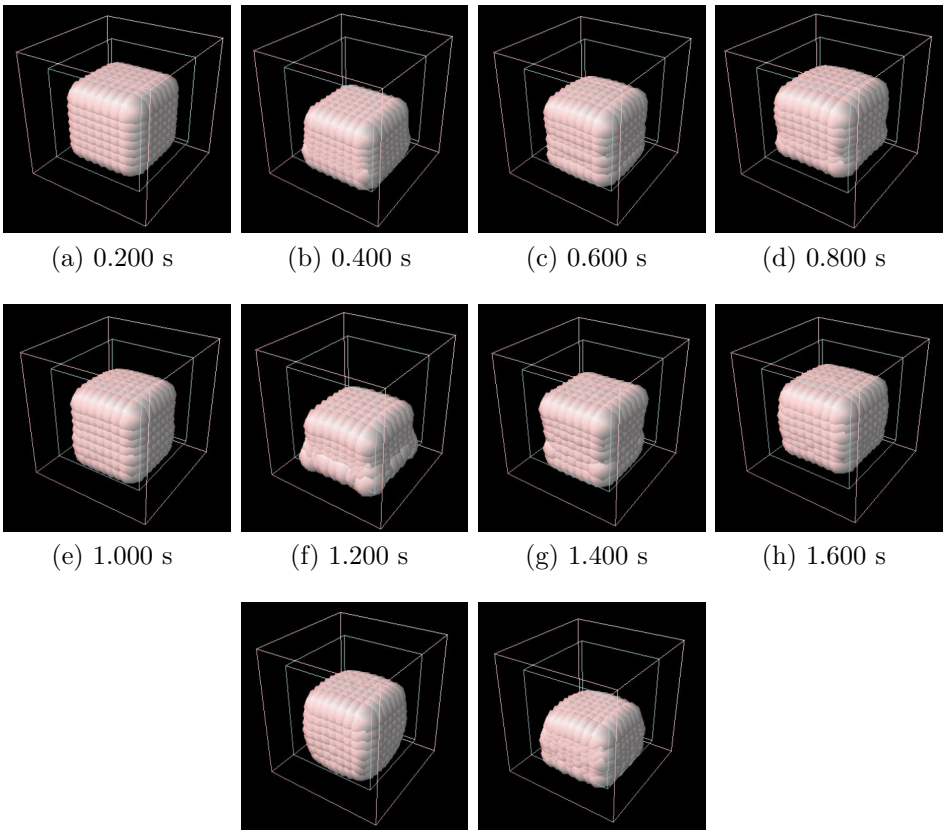
(c) 0.420 s (d) 0.600 s

(e) 0.760 s (f) 0.900 s

Figure 5.12: Second test: different viscosities

5.3.3 Test 3: Elasticity

A cube of fluid is given a high elasticity stiffness and is dropped in a box boundary. As can be seen, following equation 2.26, a jelly-like elastic behavior can be easily obtained. See the frames in figure 5.13. A complete video can be found at `http://vimeo.com/32460167`.



(a) 0.200 s (b) 0.400 s (c) 0.600 s (d) 0.800 s

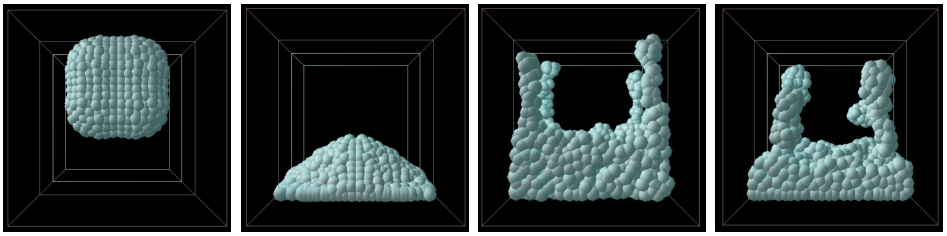
(e) 1.000 s (f) 1.200 s (g) 1.400 s (h) 1.600 s

(i) 1.800 s (j) 2.000 s

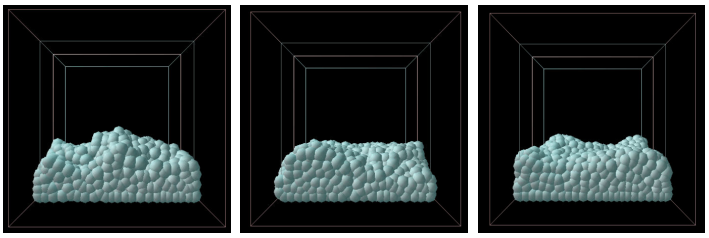
Figure 5.13: Third test: elastic behavior

5.3.4 Test 4: Quasi-fluid adapted boundary

A mass of liquid is dropped in a cube box, using the quasi-fluid boundary method of section 3.5.3, improved with the adapted extension. The fluid can be seen to adhere to the boundary surface without forming erroneous surface tensions at the interface, solving the spurious surface tension problem. See the frames in figure 5.14. A complete video can be found at `http://vimeo.com/32460260`.



(a) 0.340 s (b) 0.680 s (c) 1.060 s (d) 1.400 s

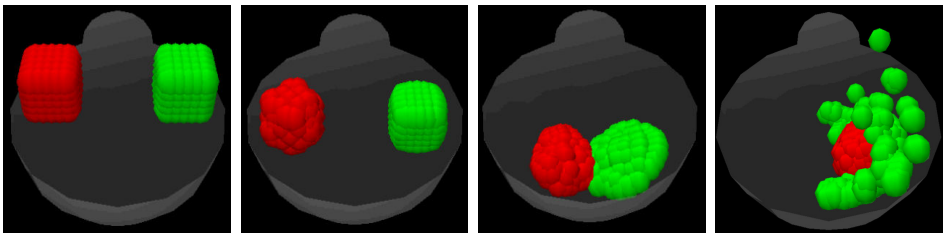


(e) 1.780 s (f) 2.150 s (g) 2.500 s

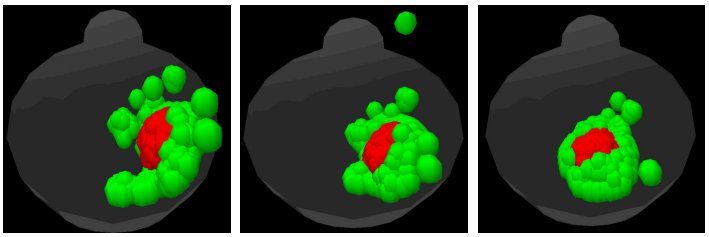
Figure 5.14: Fourth test: quasi-fluid adapted boundary

5.3.5 Test 5: Two fluids with different densities

Two fluids with different densities are dropped in the same spherical container. At the collision, the less dense fluid cannot resist the mass of the denser fluid and is scattered around. See the frames in figure 5.15. A complete video can be found at `http://vimeo.com/32460294`.



(a) 0.000 s (b) 0.500 s (c) 1.000 s (d) 1.500 s

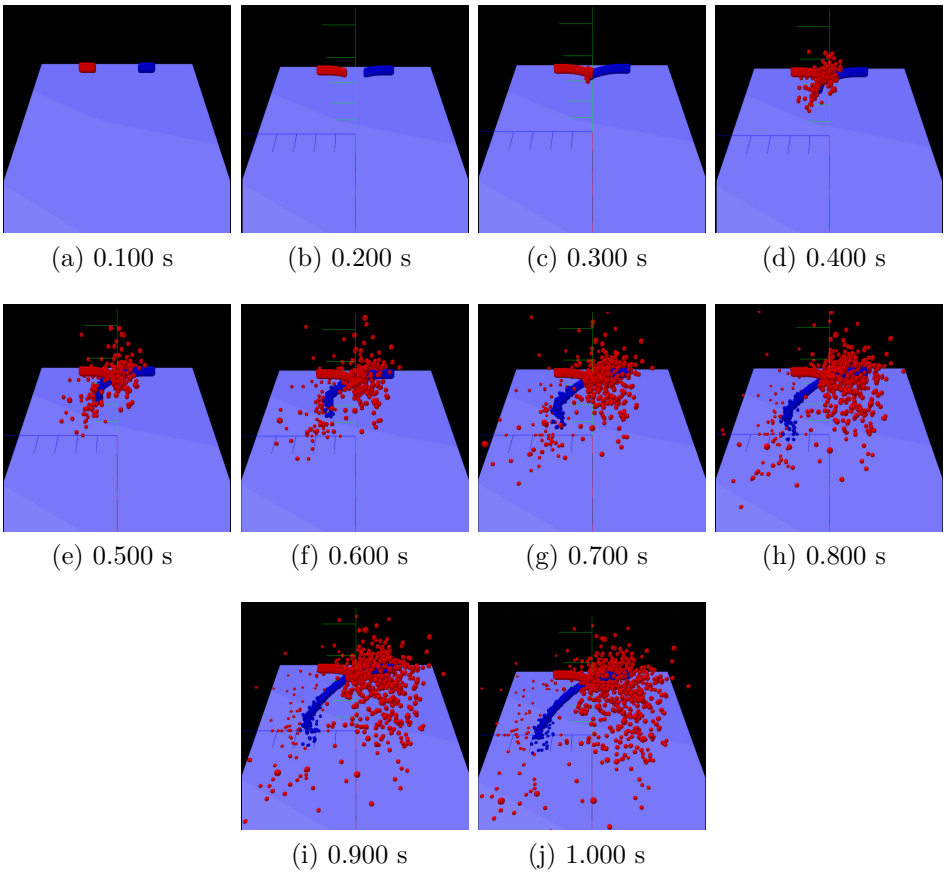


(e) 2.000 s (f) 2.500 s (g) 3.000 s

Figure 5.15: Fifth test: two fluids with different densities

5.3.6 Test 6: Two jets with different densities

Two fluid jets with different densities collide at mid-air. The denser jet's motion continues almost unaltered, while the less dense fluid is scattered around. See the frames in figure 5.16. A complete video can be found at `http://vimeo.com/32460345`.



(a) 0.100 s (b) 0.200 s (c) 0.300 s (d) 0.400 s

(e) 0.500 s (f) 0.600 s (g) 0.700 s (h) 0.800 s

(i) 0.900 s (j) 1.000 s

Figure 5.16: Sixth test: two fluid jets with different densities

5.4 Conclusions

In this chapter, we have discussed the results that can be achieved by making use of our implemented SPH simulator.

In section 5.1, the physical realism of a SPH model is compared to the AXIM model, known to be realistic and accurate. The results are commented and explained.

In section 5.2, a performance comparison between the serial CPU implementation and the parallel GPU implementation is performed.

In section 5.3, additional visual results for different test cases are shown, highlighting the versatility of the SPH simulator.

Chapter 6

Conclusions and future work

Using our SPH approach, a performant and accurate model has been created which is also versatile, allowing us to simulate viscous, slow moving fluids as well as jets of water with ease and at the same time.

By extending the classic method with the solutions provided in this thesis, a greater degree of accuracy and stability can be achieved, allowing the SPH model to rival with older established computational fluid dynamics approaches.

Due to its Lagrangian nature and its performance, SPH allows us to simulate the real-time interactions with different fluids, bodies or even the user and does not constrain the simulation volume, providing a good model to be used with virtual reality simulators, video games or other interactive applications.

At last, particle methods such as SPH are good candidates for a GPU implementation and the performance gain that we achieved permits real-time frame rates, resulting in an interactive, realistic, multi-purpose and extensive fluid simulator.

To this day, due to its young nature, SPH is still open to new solutions on some of its shortcomings, mainly on stability and the spurious surface tension issue. Much work can be done in this direction by extending known methods or producing new approaches. In addition, more phenomena apart from those mentioned in this work could be formulated with an SPH approach, such as plasticity (Clavet et al. [2005]) and thermal and magnetic equations (Monaghan [1992]).

Thanks to the increase of parallel computation power of current GPUs and following the trend of Moore's law, in the near future SPH models might easily achieve real-time performance even for millions of particles. This, coupled with the fact that even a few thousands of particles can give good visual and even physically realistic results, could lead to SPH being widely

used in real-time applications such as virtual reality environments.

SPH can already easily simulate the change of phase of a fluid from liquid to gas and backwards. This can be done by adding a temperature value to each particle, computed again with an SPH equation, and changing its properties based on it. In addition, since two-way coupling with rigid body dynamics methods can be easily achieved and rendered consistent with the SPH model with methods such as the quasi-fluid boundary approach, a complete phase change model could be implemented even for solid to liquid, solid to gas and backwards transformations. Alternatively, by focusing on adding SPH formulations of solid's effects such as elasticity and plasticity, even solids could be modeled using the standard SPH equations. Some recent works, such as the work of Solenthaler [2010], are already aiming at creating an unique, consistent, complete, realistic and real-time model for solid and fluid simulation.

Bibliography

Oscar Agertz. Fundamental differences between sph and grid methods. Technical report, Institute for Theoretical Physics, University of Zürich, 2008.

Richard Anderson. Nearest neighbour trees and n-body simulation. Technical report, Department of Computer Science and Engineering - University of Washington, 1993.

Kai Bao, Hui Zhang, Lili Zheng, and Enhua Wu. Pressure corrected sph for fluid animation. In *Computer Animation and Virtual Worlds*, volume 20, pages 311–320, 2009.

Philippe Beaudoin, Sébastien Paquet, and Pierre Poulin. Realistic and controllable fire simulation, 2001.

Markus Becker and Matthias Teschner. Weakly compressible sph for free surface flows. *ACM SIGGRAPH Symposium on Computer Animation*, pages 1–8, 2007.

Markus Becker, Hendrik Tessendorf, and Matthias Teschner. Direct forcing for lagrangian rigid-fluid coupling. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):493–503, May 2009.

Nathan Bell, Yizhou Yu, and Peter J. Mucha. Particle-based simulation of granular materials. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation (2005)*, 2005.

Paul Bourke. Implicit surfaces (metaballs). `http://paulbourke.net/miscellaneous/implicitsurf/`, June 1997.

Sara Brambilla and Davide Manca. Accidents involving liquids: a step ahead in modeling pool spreading, evaporation and burning. *Journal of Hazardous Materials*, 161:1265–1280, 2009a.

Sara Brambilla and Davide Manca. Dynamic process and accident simulations as tools to prevent industrial accidents. *Chemical Product and Process Modeling*, 4:1–15, 2009b.

R. A. Brownlee, P. Houston, J. Levesley, and S. Rosswog. Enhancing sph using moving least-squares and radial basis functions. In *Algorithms for Approximation: Proceedings of the 5th International Conference*, pages 103–112, 2007.

John Van Der Burg. Building an advanced particle system. `http://www.gamasutra.com/view/feature/3157/building_an_advanced_particle_.php`, June 2000.

Cairo. Cairo library 1.10.2. `http://cairographics.org/`, 2010.

Tatiana Capone. Sph numerical modeling of impulse water waves generated by landslides. Master's thesis, Sapienza University of Rome, 2009.

Nuttapong Chentanezg and Matthias Müller. Real-time simulation of large bodies of water with small scale details. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 197–206, 2010.

M. M. Clark. Drop breakup in a turbulent flow – i. conceptual and modeling considerations. *Chemical Engineering Science*, 43:671–679, 1988.

Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2005.

Keenan Crane, Ignacio Llamas, and Sarah Tariq. Real-time simulation and rendering of 3d fluids. *GPU Gems 3*, 2007.

A.J.C. Crespo, M. Gomez-Gesteria, and R.A. Dalrymple. Dynamic boundary particles in sph. In *2nd International Workshop on SPHERIC-Smoothed Particle Hydrodynamics*, pages 152–155, 2007.

M. Desbrun and M. P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation '96 - Proceedings of EG Workshop on Animation and Simulation)*, pages 61–76, 1996.

R. Fedkiw, J. Stam, and H.W. Jensen. Visual simulation of smoke. In *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, 2001.

Brad Fish. glfont library. `http://students.cs.byu.edu/~bfish/glfont.php`, 2001.

Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. Technical report, University of Pennsylvania, Philadelphia, 1997.

R. A. Gingold and J.J. Monaghan. A numerical approach to the testing of the fission hypothesis. *Monthly Notices of the Royal Astronomical Society*, 181:375, 1977.

Simon Green. *Particle Simulation Using Cuda*. NVIDIA Corporation, 1.3 edition, 2010.

N. Grenier, D. Le Touzé, M. Antuono, and A. Colagrossi. An improved sph formulation for multi-phase flow simulations. In *Proc. of 8th Int. Conf. on Hydrodynamics (ICHD 2008)*, 2008.

Wm. G. Hoover. Isomorphism linking smooth particles and embedded atoms. *Physica A*, 260:244–254, 1998.

Liao Horng-Shyang, Chuang Jung-Hong, and Lin Cheng-Chung. Efficient rendering of dynamic clouds. In *Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pages 19–25, 2004.

M. Ihmsen, N. Akinci, M. Gissler, and M. Teschner. Boundary handling and adaptive time-stepping for pcisph. In *Proc. VRIPHYS*, pages 79 – 88, 2010.

Tommi Ilmonen and Janne Kontkanen. The second order particle system. *Journal of WSCG*, 11(1):13–19, 2003.

Bryan Feldman James. Animating suspended particle explosions, 2003.

Richard Keiser, Bart Adams, Dominique Gasser, Paolo Bazzi, Philip Dutré, and Markus Gross. A unified lagrangian approach to solid-fluid animation. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 125–133, 2005.

Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. *Science*, 2006.

A. Khayyer, H. Gotoh, and S. Shao. Corrected incompressible sph method for accurate water surface tracking in breaking waves. *Coastal Engineering (2008)*, 55(3):236 – 250, 2008.

S. Koshizuka and Y. Oka. Moving-particle semi-implicit method for fragmentation of incompressible fluids. *Nucl. Sci. Eng.*, 123:421–434, 1996.

Jeff Lander. The ocean spray in your face. *Game Developer*, pages 13–19, July 1998.

Martin Lastiwka, Mihai Basa, and Nathan J. Quinlan. Permeable and non-reflecting boundary conditions in sph. Technical report, Department of Mechanical and Biomedical Engineering, National University of Ireland, Galway, Ireland, 2008.

Lutz Latta. Building a million-particle system. `http://www.gamasutra.com/view/feature/2122/building_a_millionparticle_system.php`, June 2004.

William Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (SIGGRAPH 87 Proceedings)*, volume 21, pages 163–170, 1987.

F. Losasso, G. Irving, E. Guendelman, and R. Fedkiw. Melting and burning solids into liquids and gases. In *IEEE Transactions on Visualization and Computer Graphics*, pages 343–352, 2006.

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

A. Mashayek, A. Jafari, and N. Ashgriz. Improved model for the penetration of liquid jets in subsonic crossflow. *AIAA Journal*, 46:2674–2686, 2008.

David K. McAllister. The design of an api for particle systems. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

G. Miller. Globular dynamics: A connected particle system for animating viscous fluids. *Comput. and Graphics*, 13(3):305–309, 1989.

Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

Matthias Müller, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus Gross. Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds - Special Issue: The Very Best Papers from CASA 2004*, 15(3-4), 2004.

Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 1–7, 2005.

Matthias Müller, Simon Schirm, and Stephan Duthaler. Screen space meshes. *ACM SIGGRAPH Symposium on Computer Animation*, page 9–15, 2007.

Joseph J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

Joseph J. Monaghan. Sph without a tensile instability. *Journal of Computational Physics*, 159:290–311, 2000.

Joseph J. Monaghan and J. C. Lattanzio. A refined particle method for astrophysical problems. *Astronomy and Astrophysics (ISSN 0004-6361)*, 149(1):135–143, 1985.

Joseph J. Monaghan, M.C. Thompson, and K. Hourigan. Simulation of free surface flows with sph. *Journal of Computational Physics*, 110(2), 1994.

Joseph P. Morris. Simulating surface tension with smoothed particle hydrodynamics. *Int. J. Numer. Meth. Fluids*, 33:333–353, 1999.

Joseph P. Morris, Patrick J. Fox, , and Yi Zhu. Modeling low reynolds number incompressible flows using sph. Technical report, School of Civil Engineering, Purdue University, West Lafayette, Indiana 47907, 1997.

Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire, 2002.

NVIDIA. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 3.1.1 edition.

P. W. Randles and L. D. Libersky. Smoothed particle hydrodynamics - some recent improvements and applications. *Comput. Methods Applied Mech. Eng.*, 139:375–408, 1996.

William T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, 19:313–322, July 1985.

Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 21(4):23–34, 1987.

Karl Sims. Particle animation and rendering using data parallel computation. In *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 405 – 413, 1990.

Barbara Solenthaler. *Incompressible fluid simulation and advanced surface handling with SPH*. PhD thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich, 2010.

Barbara Solenthaler and R. Pajarola. Density contrast sph interfaces. *ACM SIGGRAPH / EG Symposium on Computer Animation*, pages 211–218, 2008.

Jos Stam. Stable fluids. In *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, volume 8, pages 121–128, August 1999.

A. Valizadeh, M. Shafieefar, J.J. Monaghan, and S.A.A. Salehi Neyshaboori. Modeling two-phase flows using sph method. *Journal of Applied Sciences*, 8:3817–3826, 2008.

Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. *I3D '09 Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 2009.

Xiaoming Wei, Wei Li, Klaus Mueller, and Arie Kaufman. Simulating fire with texture splats, 2002.

M. Yildiz, R. A. Rook, and A. Suleman. Sph with the multiple boundary tangent method. *International journal for numerical methods in engineering*, 77:1416–1438, 2009.

Guangzheng Zhou, Wei Ge, and Jinghai Li. A revised surface tension model for macro-scale particle methods. *Powder Technology*, 183(1):21 – 26, 2008.