FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Smoothed Particle Hydrodynamics Real-Time Fluid Simulation Approach

David Staubach

Bachelor Thesis

Smoothed Particle Hydrodynamics
Real-Time Fluid Simulation Approach

David Staubach

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dr.-Ing. Klaus Iglberger
Bearbeitungszeitraum:	28.04.2010 – 22.12.2010

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 22.12.2010 .....

Abstract

This bachelor thesis covers an extensive introduction into the smoothed particle hydrodynamics method as well as a thorough implementation guideline and a discussion about its real-time performance. We provide insight into the fluid dynamics of this particle-based approach and its comparison to the yet commonly applied mesh-based methods. Next to the SPH fluid mechanics, the handling of collisions between the fluid particles and rigid bodies poses a rather interesting application field for this thesis. Derived from the introduced SPH terms, a well depicted and conceptional implementation tutorial is presented. The implementation section comprises specific data structures and algorithms to reduce the computational complexity of the particle-particle interactions. Besides the applied optimizations, the implementation part is completed with an approach into parallel programming using Open Multi-Processing to achieve shorter simulation runtimes. Finally, the results of different simulation scenarios and the performance impact of multi-threaded execution with regards to real-time capabilities are discussed.

Zusammenfassung

Diese Bachelorarbeit umfasst eine ausführliche Einleitung in die Smoothed Particle Hydrodynamics Methode, eine detaillierte Implementierungsbeschreibung sowie die Untersuchung der Echtzeitfähigkeit der Methode. Sie liefert außerdem einen Einblick in die Fluidmechanik dieser partikelbasierten Herangehensweise und einen Vergleich mit den bisher überwiegend angewandten gitterbasierten Methoden. Neben der SPH Fluiddynamik stellt die Kollisionslösung zwischen Partikeln und Starrkörpern ein interessantes Anwendungsfeld dieser Arbeit dar. Abgeleitet von den vorgestellten SPH Größen wird ein anschauliches und verständliches Programmiertutorial beschrieben. Das Implementierungskapitel als solches beinhaltet Algorithmen und Datenstrukturen zur Reduzierung des rechnerischen Aufwands der Partikel-Partikel Interaktionen. Abgesehen von den angewandten Optimierungstechniken wird zum Abschluss dieser Rubrik eine Möglichkeit der Parallelisierung des Simulationsprogramms mit Hilfe von Open Multi-Processing vorgestellt. Schließlich werden die Ergebnisse verschiedener Simulationsszenarien sowie detaillierte Laufzeitmessungen unterschiedlicher Optimierungsgrade im Hinblick auf Echtzeitfähigkeit diskutiert.

Contents

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

API ..... Application Programming Interface

CFD ..... Computational Fluid Dynamics

CPU ..... Central Processing Unit

LC ..... Linked Cell

LCS ..... Local Coordinate System

MPI ..... Message Passing Interface

OBB ..... Oriented Bounding Box

OpenMP Open Multi-processing

POV-Ray Persistence of Vision Raytracer

SPH ..... Smoothed Particle Hydrodynamics

STL ..... Standard Template Library

WCS .... World Coordinate System

Chapter 1

Introduction

The graphical representation of fluid dynamics is a topic of great demand in computer science and particularly for computational engineering. The application areas of fluid simulations are of a very diverse manner and cover science fields like astrophysics, meteorology or medical engineering. However, even for mundane scenarios happening in our environment, the research of fluid dynamics is worthwhile for the progress in technology and science. The different kinds of fluid simulation problems vary in the rate of computational complexity and demand for physical correctness or rather real-time interactivity. Obviously the requirement for the physical correctness of the fluid flow is in most cases obligatory for any medical or mechanical engineering application, while a computer game graphics developer has to focus on real-time capabilites.

Generally, a method combining physical correctness with real-time interactivity does not exist, but there is always a trade-off between both properties. Therefore, it is indispensible for any computational engineer to examine the various fluid dynamic methods and evaluate their qualification for the simulation requirements of a certain application field.

1.1 Computational Fluid Dynamics

The numerical methods brought up in the introduction are included in the science field of Computational Fluid Dynamics and are used to solve certain fluid mechanics equations. Typically these equations are the Navier-Stokes (B.1) or the Euler equations (B.3). As already mentioned, there are lots of CFD methods in use and each of them has different properties when it comes to performance and accuracy. It is a common way to distinguish between mesh-based and particle-based methods or, from the fluid's point of view, to differ between Eulerian and Lagrangian fluids. While the Eulerian method corresponds to mesh-based solvers, an implementation of a Lagrangian fluid can be either mesh-based or particle-based. In the Eulerian view the simulation

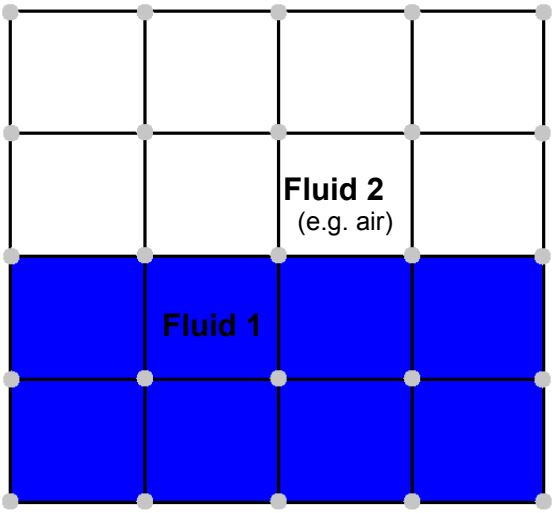

Figure 1.1: The Eulerian fluid description in two dimensions. The fluid's properties are evaluated at the grid points. The problem of the grid itself gets even clearer with such a coarse grid example. The fluid is constrained to exist only at the discrete grid cells, a smooth transition between both fluids is not possible.

domain is discretized by a fixed mesh and the fluid's properties affecting its dynamics can only be calculated at the discrete grid points of the domain.

The Lagrangian view describes the fluid as a discrete number of particles that model the fluid flow as such and that are carrying along all of the fluid's quantities. The particles are able to move freely through the whole domain dependent on the acting forces in the scenario and hence, the fluid is able to exist at any location in the simulation domain. Figure 1.1 and 1.2 depict the basic description of the two different fluid views in a very simplified manner. Even though the Eulerian fluid solvers provide more accurate results for some fluid quantities like the mass-density or the pressure field, their worst drawback is the mesh itself. Namely, the fluid properties are constrained to exist only at the discrete grid points leading to unrealistic fluid flows in some occasions, e.g. free surface flows. If this problem is faced by refining the mesh more and more, the memory overhead of the grid as well as the growing computational costs aroused by the larger number of grid points, inhibit any ambitions for simulations at real-time rates. Famous examples for mesh-based solvers are the finite difference method, the finite element method, the finite volume method and the Lattice Boltzmann methods. Further information on theoretical details and implementation examples of Eulerian fluids are given in [ESHD05].

The performance of particle-based solvers strongly depends on the num-

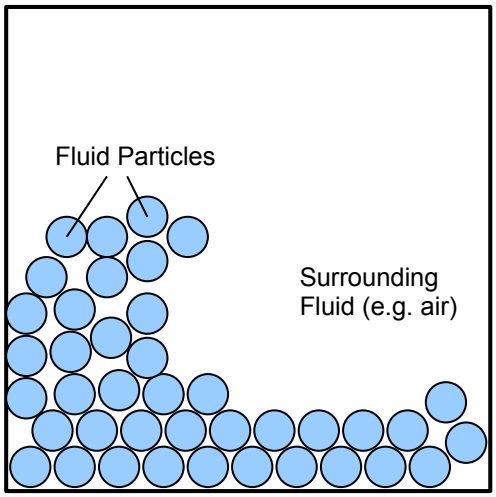

Figure 1.2: The Lagrangian fluid description in two dimensions. The particles are defining the fluid as such and are able to float freely through the simulation domain. Each of the particle carries the necessary fluid mechanical quantities.

ber of simulated particles and the algorithms used to optimize the execution process of the simulation. However, particle-based methods generally provide an advantage over the mesh-based methods in terms of real-time interactivity. Hence, the examination of particle-based fluid flow solvers, particularly with regard to the exploit of computational complexity and efficiency, seems to pose an interesting modern research field.

1.2 Goals / Preview

This bachelor thesis is supposed to provide an insight into the computational simulation of fluid dynamics with the application of the smoothed particle hydrodynamics method. The main focus of this approach lies in the real-time interactivity of the implemented solver and the interaction between the fluid particles and static rigid bodies, modeling obstacles or boundary layers. Next to the real-time achievement, the physical accuracy of the simulation parameters and accordingly the fluid dynamics is of significant importance for the results of this work.

Chapter 2 gives an introduction to the origin of the SPH method as well as a theoretical background to and the derivation of the fluid mechanics equations and SPH formalisms. Moreover, the applied numerical time integration scheme and a detailed collision handling routine is presented. Chapter 3 comprises the practical part of this work with a thorough description of algorithms and

certain techniques to optimize the computational costs of the developed step-by-step implementation guideline. Furthermore, an approach for the parallel execution on multi-core architectures, applying the OpenMP API, is presented as part of the implementation chapter. Although our developed implementation is only intended to simulation scenarios with the fluid water, the SPH method provides all the common basics for other fluids, like mucus or steam, as well. At last, Chapter 4 collects the most important results of this work and completes with a discussion about precedent expectations and implementation issues.

Chapter 2

Smoothed Particle Hydrodynamics

The first appearance of the smoothed particle hydrodynamics method goes back to 1977, where it was used for computational research in astrophysics. The originators of the method are Bob Gingold and Joe Monaghan [GiMo77] as well as Leon Lucy [Luc77] in personal research. SPH was actually designed for compressible flow problems [Mon92] and since it has been applied in various fields of scientific research including astrophysics, ballistics, volcanology, and oceanography. The SPH method got its name from the term fluid dynamics, also known as hydrodynamics and was adapted into the Computer Graphics community in 1995 [StFi95].

The following sections 2.1 and 2.2 deal with the basic derivation of the SPH terms and depict the necessary characteristics of an exemplary smoothing kernel function that is decisive for the quality of the method. Generally, this chapter is based on the documentation of [Kel06].

2.1 SPH Essentials

As already mentioned, SPH is a particle-based Lagrangian method and thus, it introduces particles to define the simulated fluid and to evaluate the arising dynamics properly. These so called smoothed particles, represented as point masses in the simulation domain Ω, carry information like position and velocity as well as certain densities and force fields that describe the fluid's properties and impacts at the respected particle's location in the domain. As the name of the method already reveals, the properties of a fluid particle are smoothed out over the neighbouring particles within a support region with radius h.

The integral interpolant of any continuous field quantity $\boldsymbol{A}$ is,

$$\boldsymbol{A}_I(\boldsymbol{x}) = \int_\Omega \boldsymbol{A}(\boldsymbol{x}')\, W(\boldsymbol{x} - \boldsymbol{x}', h)\, d\boldsymbol{x}', \tag{2.1}$$

where $\boldsymbol{x}$ is the position vector in Ω and W is a certain smoothing kernel function with the parameter h as smoothing length.

The numerical equivalent to (2.1) yields

$$\boldsymbol{A}_S(\boldsymbol{x}) = \sum_{\forall j} \boldsymbol{A}_j V_j W(\boldsymbol{x} - \boldsymbol{x}_j, h), \tag{2.2}$$

where the integral is approximated by a summation interpolant and j iterates over every single particle. V_j represents the theoretical volume of a single particle j. Considering the general relation between volume V, mass m and mass-density ρ

$$V = \frac{m}{\rho}, \tag{2.3}$$

the basic SPH formulation is derived by substituting the volume term in (2.2) by the right hand side of (2.3)

$$\boldsymbol{A}_S(\boldsymbol{x}) = \sum_{\forall j} \boldsymbol{A}_j \frac{m_j}{\rho_j} W(\boldsymbol{x} - \boldsymbol{x}_j, h). \tag{2.4}$$

The gradient (A.3) and the Laplacian (A.4) of (2.4) can be set up as follows. We can take advantage of the fact that $A_j \frac{m_j}{\rho_j}$ does not depend on $\boldsymbol{x}$ and hence, it is not affected by derivation. It turns out that only the kernel function W is decisive for the gradient and the Laplacian.

The gradient of (2.4) is

$$\nabla \boldsymbol{A}_S(\boldsymbol{x}) = \sum_{\forall j} \boldsymbol{A}_j \frac{m_j}{\rho_j} \nabla W(\boldsymbol{x} - \boldsymbol{x}_j, h). \tag{2.5}$$

Furthermore, there is another version of the gradient, introduced and derived in [Kel06], that is supposed to obtain more accurate results

$$\nabla \boldsymbol{A}_S(\boldsymbol{x}) = \rho \sum_{\forall j} \left(\frac{\boldsymbol{A}_j}{\rho_j^2} + \frac{\boldsymbol{A}}{\rho^2} \right) m_j \nabla W(\boldsymbol{x} - \boldsymbol{x}_j, h). \tag{2.6}$$

The Laplacian of (2.4) finally yields

$$\nabla^2 \boldsymbol{A}_S(\boldsymbol{x}) = \sum_{\forall j} \boldsymbol{A}_j \frac{m_j}{\rho_j} \nabla^2 W(\boldsymbol{x} - \boldsymbol{x}_j, h). \tag{2.7}$$

2.2 The Smoothing Kernel

The main requirement of the kernel function W is to smooth out a particle's contribution to an arbitrary SPH quantity field $\boldsymbol{A}(\boldsymbol{x})$ dependent on the distance of the respected particle. The farther two particles are located from each other, the smaller the mutual influence between the two particles has to be.

We have already mentioned the importance of the smoothing kernel for the SPH formulation and especially its derivatives. Which kernel function provides the most accurate and realistic results for a certain quantity field depends very much on the function's characteristics. However, generally a smoothing kernel function has to fulfill the following properties [Mon92]

$$\int_\Omega W(\boldsymbol{r}, h) d\boldsymbol{r} = 1 \tag{2.8}$$

as well as

$$\lim_{h \to 0} W(\boldsymbol{r}, h) = \delta(\boldsymbol{r}) \tag{2.9}$$

with $\delta(\boldsymbol{r})$ as the Dirac delta function (A.1).

Other defined characteristics are

$$W(\boldsymbol{r}, h) \geq 0 \tag{2.10}$$

and

$$W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h). \tag{2.11}$$

To combine the requirements in a few words, a suitable kernel function has to be normalized (2.8), positive (2.10) and axial symmetric (2.11).

The parameter h, already introduced as the smoothing length or support radius, is of essential importance when it comes to computational complexity and physical accuracy of any SPH fluid quantity. The smoothing length specifies the radius around an observed particle i within which another particle j

needs to be located to interact with particle i. If the distance between particle i and particle j is larger than the support radius, the kernel is supposed to return a zero value, meaning

$$W(\boldsymbol{r}, h) = 0, \ \ \|\boldsymbol{r}\| > h. \tag{2.12}$$

The support radius h has to be chosen in such a way that on the one hand it is large enough to let a suitable number of particles interact with each other and on the other hand it is small enough to limit the computational costs of the SPH contributions.

2.2.1 Example: Isotropic Gaussian kernel

In order to visualize some of the aforementioned characteristics of a smoothing kernel, the isotropic Gaussian kernel is depicted here as an exemplary kernel function, given by

$$W_{Gaussian}\left(\boldsymbol{r}, h\right) = \frac{1}{\left(2\pi h^2\right)^{\frac{3}{2}}} \exp\left(-\left(\frac{\|\boldsymbol{r}\|^2}{2h^2}\right)\right), \ \ h > 0. \tag{2.13}$$

Figure 2.1 shows the kernel's curve in one dimension for $h = 1$.

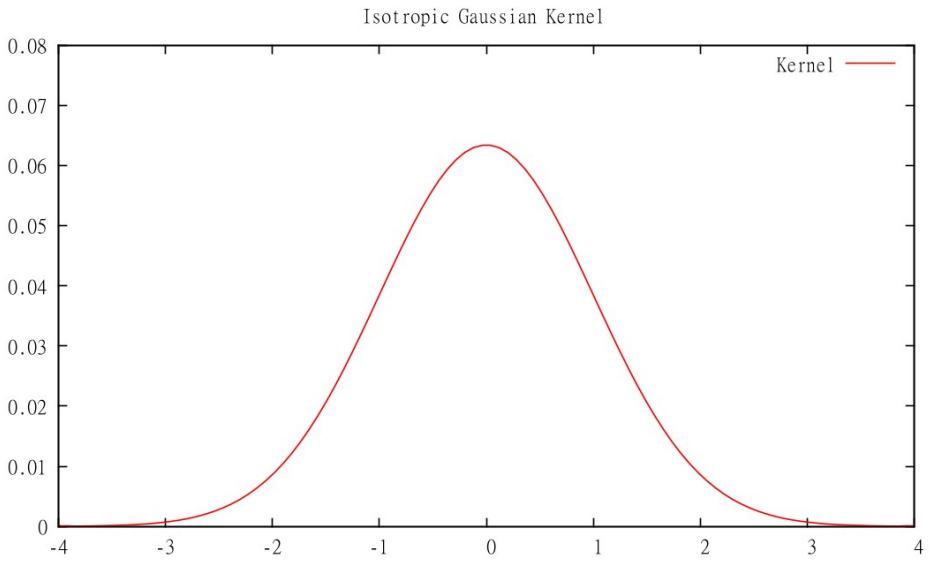

Figure 2.1: Isotropic Gaussian kernel in one dimension, $h = 1$

However, the isotropic Gaussian kernel does not fulfill the constraint (2.12) because of its asymptotic curve. Furthermore, the computational costs of the exponential term evaluation does not meet our ambition of real-time interactivity. Due to the unsuitable properties of the isotropic Gaussian kernel, it is not applied as the default kernel throughout this thesis, but the 6th degree polynomial kernel (C.1) suggested by [MCGr03].

2.3 Fluid Dynamics

This section is about the fluid mechanical theory and the necessary quantity fields, calculated with the help of the SPH method, to obtain the properties and dynamics of the simulated fluid. Based on the Navier-Stokes equations for incompressible fluids and the continuity equation (B.2) we can setup the general fluid dynamics equation for our simulation purpose. By using the Lagrangian fluid view we are able to simplify the standard equations in the following way. Due to the fact that our particles have a fixed mass, the continuity equation, defining mass conservation, is always satisfied as long as we do not change the number of particles in our closed simulation system. Moreover, we can neglect the spatial derivative of the velocity term on the left hand side because the particles are floating with the fluid and hence the carried quantities are independent of $\boldsymbol{x}$. Thus, the remaining derivative on the left hand side is the time dependent component only and the mass-density ρ described in subsection 2.3.1. We will not examine the fluid mechanical background in more detail, but we want to focus on the resulting Lagrangian formulation for an incompressible fluid

$$\rho \frac{d\boldsymbol{v}}{dt} = -\nabla p + \mu \nabla^2 \boldsymbol{v} + \boldsymbol{f}, \tag{2.14}$$

where, on the right hand side, the pressure term $-\nabla p$ and the viscosity term $\mu \nabla^2 \boldsymbol{v}$ model the internal force fields, while $\boldsymbol{f}$ represents the external force field contributions. We can retrieve the total resulting force field acting on a particle i by the summation of its internal and external force fields,

$$\boldsymbol{F}_i = \boldsymbol{f}_i^{internal} + \boldsymbol{f}_i^{external}. \tag{2.15}$$

An equivalent reformulation of Equation (2.14) leads to the definition of the acceleration $\boldsymbol{a}_i$ of an arbitrary particle i,

$$\boldsymbol{a}_i = \frac{d\boldsymbol{v}_i}{dt} = \frac{\boldsymbol{F}_i}{\rho_i}. \tag{2.16}$$

The next subsections cover the description and calculation of the individual mandatory quantities and force fields needed to solve Equation (2.16).

2.3.1 Mass-Density

Generally, the mass-density is defined as mass per unit volume $\rho = \frac{m}{V}$. Regarding water, the mass-density would be about $1000\,\frac{kg}{m^3}$. However, in this subsection we are not interested in the mass-density of a certain fluid as such, but we need to determine the mass-density acting at the specific location of a respected particle i. In this form, the mass-density gives information about the concentration of adjacent particles j around the position of particle i. The more particles occupy the same area, the higher the respected mass-density will turn out in this region.

Using the general SPH formulation (2.4) the mass-density of particle i yields

$$\rho_i(\boldsymbol{x}_i) = \sum_{\forall j} \rho_j \frac{m_j}{\rho_j} W(\boldsymbol{x}_i - \boldsymbol{x}_j, h) = \sum_{\forall j} m_j W(\boldsymbol{x}_i - \boldsymbol{x}_j, h), \tag{2.17}$$

where W is our default kernel. The calculation of the mass-density needs to be applied before further SPH computations because each of the following quantity fields depend on the adjacent particles' mass-density.

2.3.2 Internal Force Fields

The internal force fields are the pressure and viscosity force introduced in (2.14). The origin of these forces lies within the fluid itself and hence it arises only from the interaction between the corresponding particles. Both internal components are obtained by the SPH formalism, however. In contrast to the mass-density calculation the default kernel does not provide realistic results. Consequently, there are two different kernel functions applied to model the pressure and viscosity forces properly.

2.3.2.1 Pressure

The pressure force is the main trigger for repulsive and attractive force impacts between adjacent particles. Within a location where lots of particles are positioned densely, the pressure field rises and consequently the pressure force generates repulsive forces to tear the particles apart from each other. Contrarily, the particles experience an attractive force towards each other in a deserted region. Figure 2.2 depicts the different particle constellations and the resulting repulsion and attraction between the particles modeled by the pressure force.

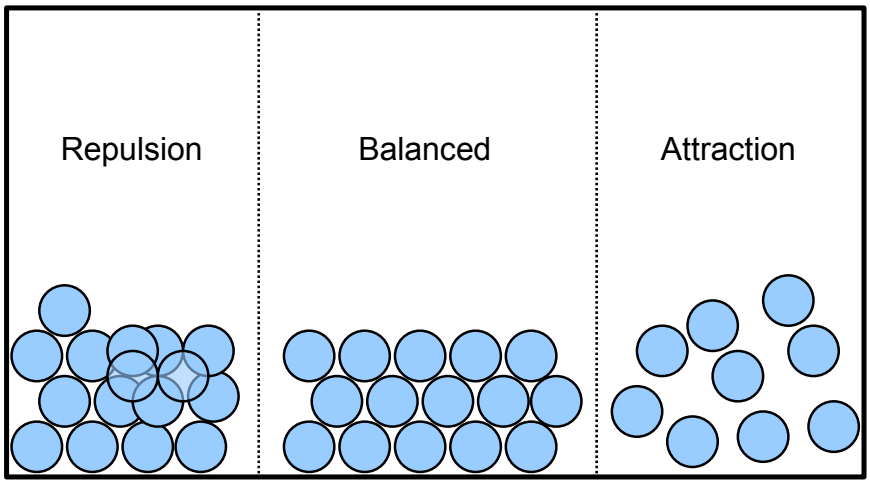

Figure 2.2: Repulsive forces in narrow particle regions (left), attractive forces in deserted regions (right) and balanced disposition in between.

At first, we have to determine the pressure field p. The ideal gas law, obtaining particle i's pressure field as

$$p_i = k\rho_i, \tag{2.18}$$

is a simple way to get the pressure field out of the gas stiffness constant k and the mass-density ρ_i. However, the standard ideal gas law does not produce attractive forces because the pressure field is always positive. In fact, this might be adequate for gaseous fluids, that tend to diffuse into the simulation domain, but it is not applicable for liquids, e.g. water. We want to model negative pressure fields as well, therefore [DeCa96] introduces a modified version of Equation (2.18), that is

$$p_i = k\left(\rho_i - \rho_0\right), \tag{2.19}$$

where ρ_0 is another parameter given in table 3.2 and known as the rest density of the fluid.

Now that we have the pressure field, the pressure force can be easily set up using the basic gradient formulation (2.5) with $\boldsymbol{A}(\boldsymbol{x})$ as the pressure term $-\nabla p(\boldsymbol{x})$

$$\boldsymbol{f}_i^{pressure}(\boldsymbol{x}_i) = -\nabla p(\boldsymbol{x}_i) = -\sum_{\forall j \neq i} p_j \frac{m_j}{\rho_j} \nabla W(\boldsymbol{x}_i - \boldsymbol{x}_j, h). \tag{2.20}$$

In [Kel06] it is stated that this form of the pressure force calculation actually does not produce symmetric force impacts due to the fact that the corresponding particles' contributions do not fulfill Newton's third law of motion. In other words, since p_j and p_i as well as ρ_j and ρ_i are generally not equal, the action-reaction law is not conserved by (2.20). Using the reformulated gradient equation (2.6) instead of the basic form (2.5), the SPH equation for the pressure force becomes

$$\boldsymbol{f}_i^{pressure}(\boldsymbol{x}_i) = -\rho_i \sum_{\forall j \neq i} \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) m_j \nabla W(\boldsymbol{x}_i - \boldsymbol{x}_j, h), \tag{2.21}$$

providing the conservation of symmetry and momentum scaled by particle i's mass-density. Another symmetric equation to obtain particle i's pressure force is introduced by [MCGr03] and employs the standard arithmetic mean of the respective particles' pressure fields p_i and p_j

$$\boldsymbol{f}_i^{pressure}(\boldsymbol{x}_i) = -\sum_{\forall j \neq i} \frac{p_i + p_j}{2} \frac{m_j}{\rho_j} \nabla W(\boldsymbol{x}_i - \boldsymbol{x}_j, h). \tag{2.22}$$

As already mentioned, our default kernel is not suitable for the pressure force calculation because the gradient of the default kernel tends to zero when the adjacent particles are floating very close to each other. Hence, the particles would not experience the desired repulsive forces in this situation leading on to a possible coalescence of particles. To prevent clustering, we choose the spiky kernel (C.4) from [MCGr03] to smooth out the pressure force contributions. In contrast to the 6th degree polynomial kernel, the spiky kernel's gradient preserves the impact of the repulsive forces when two nearby particles approach each other. Equation (2.23) describes the spiky kernel's characteristics for small distances.

$$|\nabla W_{spiky}(\boldsymbol{r}, h)| \rightarrow \frac{45}{\pi h^6}, \quad \|\boldsymbol{r}\| \rightarrow 0 \tag{2.23}$$

2.3.2.2 Viscosity

The viscosity term $\mu \nabla^2 v$ in Equation (2.14) is defined as the fluid's internal resistance to flow. While a mucous fluid provides a high viscosity, water, for example, is very thin and has a relatively low viscosity. In SPH manner, the viscosity force acting at a specific location can be calculated with the standard interpolation scheme

$$\boldsymbol{f}_i^{viscosity}(\boldsymbol{x}_i) = \mu\nabla^2\mathbf{v}(\boldsymbol{x}_i) = \mu\sum_{\forall j\neq i}\boldsymbol{v}_j\frac{m_j}{\rho_j}\nabla^2 W(\boldsymbol{x}_i - \boldsymbol{x}_j, h), \tag{2.24}$$

where μ is a given viscosity coefficient in Table (3.2) describing the viscosity of the fluid as such. However, similar to Equation (2.20), the standard version does not provide symmetric results due to the fact that particle i and particle j do not usually have identical velocities. Via the relative difference between both velocities we can easily symmetrize Equation (2.24) and end up with

$$\boldsymbol{f}_i^{viscosity}(\boldsymbol{x}_i) = \mu\sum_{\forall j\neq i}(\boldsymbol{v}_j - \boldsymbol{v}_i)\frac{m_j}{\rho_j}\nabla^2 W(\boldsymbol{x}_i - \boldsymbol{x}_j, h) \tag{2.25}$$

as introduced in [MCGr03] as well.

It is necessary for the stability of the simulation that the Laplacian of the smoothing kernel W is positive for $\|\boldsymbol{r}\| < h$. Without this condition we could not assure that the viscosity force has a dampened effect on the velocity only and hence it would be possible to introduce energy and instability into the simulation. Therefore, neither the default kernel nor the spiky kernel are an adequate W-function for the viscosity force evaluation. To solve these stability issues, [MCGr03] presents the viscosity kernel function (C.7) as the applied smoothing kernel for the viscosity force calculation.

2.3.3 External Force Fields

External force fields do not arise from within the fluid but they are caused by the physical laws of the fluid's environment. There are two external forces that are explicitly necessary to simulate a liquid like water. On the one hand there is the gravity force and on the other hand the surface tension force. Considering gaseous materials, there would be no impact of the surface tension force, but a buoyancy force term instead. As already mentioned, the external forces are introduced as $\boldsymbol{f}$ on the right hand side of Equation (2.14).

2.3.3.1 Gravity

The gravity force field is not really a part of the SPH quantities as it is not obtained by the basic SPH equations. However, the impact of gravity is necessary for realistic results of our simulation. The applied equation to obtain the gravitational force acting equally on any particle i yields

$$\boldsymbol{f}_i^{gravity}(\boldsymbol{x}_i) = \rho_0\boldsymbol{g}, \tag{2.26}$$

with the gravitational acceleration $\boldsymbol{g}$ defined in Table 3.1 and the already

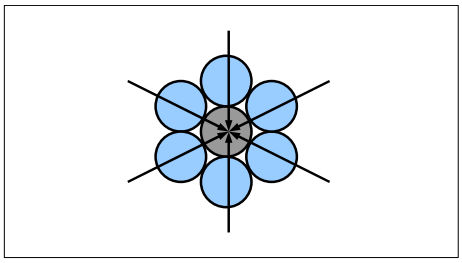

Figure 2.3: The particles on the surface are equally forced in inward direction forming a circle in 2D and a sphere in 3D.

introduced rest density ρ_0.

2.3.3.2 Surface Tension

The surface tension force of a liquid, e.g. water, has several visible effects like drop clustering on smooth waxy surfaces, floating of a water strider or the formation of interface tension forces that separate two different liquids. In our water simulation, the most suitable picture of the surface tension force is the effect of drop clustering. Generally, surface tension forces are caused by the unbalanced molecular dynamic forces at the free surface, e.g. the region between two different fluids like water and air. In these regions the water molecules, represented by our particles, are forced to shift in the direction of the surface normal towards the liquid itself causing a minimization of the liquid's curvature at the surface, visualized in Figure 2.3. Therefore, the resulting geometry forming a liquid's surface is always approximating a spherical body, because a sphere provides the smallest surface area compared to its volume. In a very similar manner it is possible to explain why water drops roll off smooth surfaces like for example glass panels or front lids of automobiles.

However, let us now come back to our SPH equations and to the force contribution of the surface tension. Usually, the surface tension force is not directly present in the Navier-Stokes equations but does only effect the boundary conditions. In our case, the particles are suited for the moulding of the boundary conditions because the particles themselves define the liquid as such and therefore they form its boundaries as well.

Obiously the surface tension force is only present in regions near the liquid's surface, so first of all, we need to determine if a certain particle is lying on the surface of the simulated liquid or not. The color field $c(\boldsymbol{x})$ is supposed to interpolate the value 1, smoothed out by the well-known SPH formula

$$c_i(\boldsymbol{x}_i) = \sum_{\forall j} m_j \frac{1}{\rho_j} W(\boldsymbol{x}_i - \boldsymbol{x}_j, h) = \sum_{\forall j} \frac{m_j}{\rho_j} W(\boldsymbol{x}_i - \boldsymbol{x}_j, h). \tag{2.27}$$

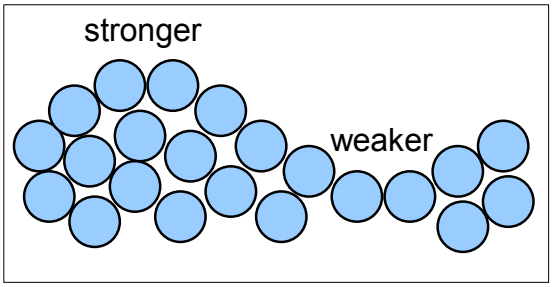

Figure 2.4: The effect of the direction of the surface's curvature on the impact of the surface tension force. The positive curvature on the left side induces a stronger surface tension force than the negative surface curvature.

The resulting surface normal, introduced as a vector pointing towards the liquid, is actually the gradient of the color field (2.27) and hence

$$\boldsymbol{n}_i = \nabla c_i \left(\boldsymbol{x}_i\right) = \sum_{\forall j} \frac{m_j}{\rho_j} \nabla W \left(\boldsymbol{x}_i - \boldsymbol{x}_j, h\right). \tag{2.28}$$

The condition$\|\boldsymbol{n}_i\| > 0$ verifies that the particle i is located near or on the liquid's surface.

The intensity of the surface tension force depends on the direction of the curvature on the surface. Figure 2.4 shows a very descriptive example for the characteristic that the influence of the surface tension force is stronger on parts of the surface with a positive curvature than on parts with a negative curvature. The quantity that measures curvature of the surface and its effect on the impact of the force is given by

$$\kappa_i = -\frac{\nabla^2 c_i}{\|\boldsymbol{n}_i\|} = -\frac{\nabla \boldsymbol{n}_i}{\|\boldsymbol{n}_i\|}. \tag{2.29}$$

Now we have already collected the necessary information to set up our surface tension force equation as presented in [MCGr03] as follows,

$$\boldsymbol{f}_i^{surface} \left(\boldsymbol{x}_i\right) = \sigma \kappa_i \boldsymbol{n}_i = -\sigma \nabla^2 c_i \frac{\boldsymbol{n}_i}{\|\boldsymbol{n}_i\|}, \tag{2.30}$$

where σ is the tension coefficient dependent on the simulated fluids sharing the free surface region.

The calculation of the surface tension force does only produce adequate results and is therefore only applied, if

$$\|\boldsymbol{n}_i\| \geq l, \tag{2.31}$$

where l is a threshold parameter given in Table 3.2. Otherwise, the evaluation of the term $\frac{\boldsymbol{n}_i}{\|\boldsymbol{n}_i\|}$ is not stable and may lead to numerical problems.

2.4 Leap-Frog Time Integration Scheme

After the calculation of each of the necessary SPH quantities and force fields, the total acting force on particle i yields

$$\boldsymbol{F}_i = \boldsymbol{f}_i^{pressure} + \boldsymbol{f}_i^{viscosity} + \boldsymbol{f}_i^{gravity} + \boldsymbol{f}_i^{surface}. \tag{2.32}$$

Remembering the original equation (2.16) we wanted to solve with the SPH method, we have now summoned all terms to calculate each particle's current acceleration. The obtained acceleration $\boldsymbol{a}_i$ is then integrated numerically (2.16) to determine the particle's advanced position and velocity. There are three integration schemes presented in [Kel06], two of them are the implicit Euler and the Verlet method, but in this thesis we decided to apply the leap-frog scheme [Ebe03] only.

Generally, numerical time integration schemes are responsible for various phenomena in particle simulations, e.g. motion damping, complications with collision handling or stability issues. We have chosen the leap-frog scheme due to its superiority concerning performance and stability and furthermore it provides the most realistic results for shorter time steps Δt. The numerical integration step for a particle's velocity update using the leap-frog scheme yields

$$\boldsymbol{v}_{t+0.5\Delta t} = \boldsymbol{v}_{t-0.5\Delta t} + \Delta t \boldsymbol{a}_t \tag{2.33}$$

and for the position update it is

$$\boldsymbol{x}_{t+\Delta t} = \boldsymbol{x}_t + \Delta t \boldsymbol{v}_{t+0.5\Delta t}, \tag{2.34}$$

initialized once with the leap velocity on negative time scale

$$\boldsymbol{v}_{-0.5\Delta t} = \boldsymbol{v}_0 - 0.5\Delta t \boldsymbol{a}_0. \tag{2.35}$$

The current particle's velocity at time t, that is necessary for the calculation of the fluid dynamical forces and the collision handling procedure in the next

section 2.5, can be approximated by the arithmetic mean of the leap velocities

$$\boldsymbol{v}_t \approx \frac{\boldsymbol{v}_{t-0.5\Delta t} + \boldsymbol{v}_{t+0.5\Delta t}}{2}. \tag{2.36}$$

2.5 Collision Handling with Rigid Bodies

Concerning collision handling, we want to focus on collisions between the fluid particles and static rigid bodies only. The collision handling routine is split up in two different parts, namely the collision detection and the collision response. For reasons of simplicity and the reduction of computational costs, the application of standard implicit primitives, introduced in [Kel06], in favor of the more advanced tetrahedra meshes is recommended to model the collision objects. As aforementioned, the use of implicit primitives in our simulation is limited to rigid bodies. The two implemented primitives to choose from in the developed simulation are a sphere and a rectangular box of arbitrary size. While the box geometry is only implemented as a boundary primitive, also known as oriented bounding box, the spherical body can be applied in both ways, as a boundary object and as an arbitrarily positioned obstacle inside the simulation domain. Figure 2.5 is supposed to clearify the main difference between a boundary and an obstacle object and why it is necessary to distinguish between the two types. For our purpose, we will assume the collision objects to be impermeable and stationary.

2.5.1 Collision Detection

Basically, the only information needed to decide whether a particle has collided with a collision object or not is its position $\boldsymbol{x}$. However, the position alone is not sufficient to handle a collision adequately because it does not reveal anything about the intensity of the penetration or the colliding particle's original direction of flow. Hence, the following three properties are necessary for a satisfactory analysis of any collision:

- Contact point $\boldsymbol{cp}$
- Depth of penetration d
- Surface normal $\boldsymbol{n}$, pointing away from the collision object.

The contact point $\boldsymbol{cp}$ is supposed to be the location where a particle has penetrated the implicit primitive. The penetration depth gives information

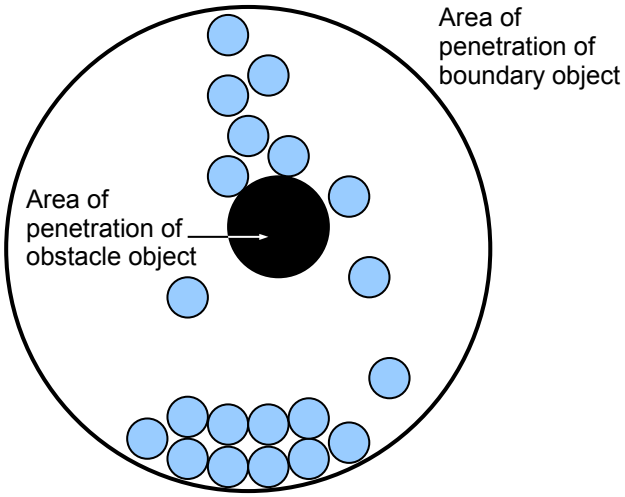

Figure 2.5: Illustration of the main difference of boundary and obstacle primitives. The "forbidden" area of penetration of a boundary object is outside of the sphere, while it is the other way round for the obstacle object.

about the distance a particle has already travelled inside an obstacle or outside of the simulation domain boundaries. The surface normal $\boldsymbol{n}$ is an unit vector that orthogonally points away from the collision object.

As already mentioned, with the use of implicit primitives we can determine any possible collision with the help of the respected particle's current position only. A simple detection function $F : \mathbb{R}^3 \to \mathbb{R}$ for any implicit primitive yields

$$F(\boldsymbol{x}) \quad \begin{cases} < 0 & \boldsymbol{x}\, is\, inside\, primitive \\ = 0 & \boldsymbol{x}\, is\, on\, the\, primitive's\, surface \\ > 0 & \boldsymbol{x}\, is\, outside\, the\, primitive. \end{cases} \tag{2.37}$$

From now on, we appoint that the second condition of (2.37) will not be evaluated as an actual collision, so as long as the observed particle has not yet penetrated the implicit primitive, no collision will be detected. At that point of time, we again need to think about the difference of a container and obstacle object. If the primitive is simulated as a container, condition three of (2.37) will lead to an actual detection of a collision, because the penetration area will be outside of the primitive. For an obstacle object it is the opposite case, because there the particles will trigger a collision if the first condition of (2.37) is evaluated as true. Hence, the collision detection routine needs to be capable to distinguish between containers and obstacles and to generate valid detection results for both cases.

Both of the presented implicit primitives apply different detection functions and different analytical ways to detect potential collisions properly and are therefore treated in separate subsections.

2.5.1.1 Sphere

The sphere has a smooth, continuous structure and is a very suitable object for collision handling purposes. With the following definition of the spherical implicit primitive, we are able to implement the sphere as a container or an obstacle. The detection function is defined by

$$F_{sphere}(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{c}\|^2 - r^2, \tag{2.38}$$

where $\boldsymbol{c}$ is the center of the sphere and r is its radius. If $F_{sphere}(\boldsymbol{x})$ has been evaluated accordingly and a collision has taken place, the mandatory properties will be calculated as follows:

$$\boldsymbol{cp}_{sphere} = \boldsymbol{c} + r\frac{\boldsymbol{x} - \boldsymbol{c}}{\|\boldsymbol{x} - \boldsymbol{c}\|} \tag{2.39}$$

$$d_{sphere} = |\|\boldsymbol{c} - \boldsymbol{x}\| - r| \tag{2.40}$$

$$\boldsymbol{n}_{sphere} = sgn\left(F_{sphere}(\boldsymbol{x})\right)\frac{\boldsymbol{c} - \boldsymbol{x}}{\|\boldsymbol{c} - \boldsymbol{x}\|}, \tag{2.41}$$

where $sgn(x)$ is the signum function defined in (A.5) and in this case, it is responsible for the valid direction of the surface normal vector $\boldsymbol{n}_{sphere}$.

2.5.1.2 Box

Using a box as collision object is not as straightforward as using a sphere due to its discontinuous and square-cut geometry. For the sake of simplicity it is here presented as a container primitive only. However, an OBB geometry still provides a lot of practical application scenarios, e.g. pouring a fluid into an open vessel, simulating water movement inside a basin or a combination of both scenarios. First of all, we need to introduce a local coordinate system as a transformation from the usual world coordinate system. The local particle position $\boldsymbol{x}_{local}$ seen from the box's point of view is the relative position of the particle compared to the center of the box container. Figure 2.6 shows an

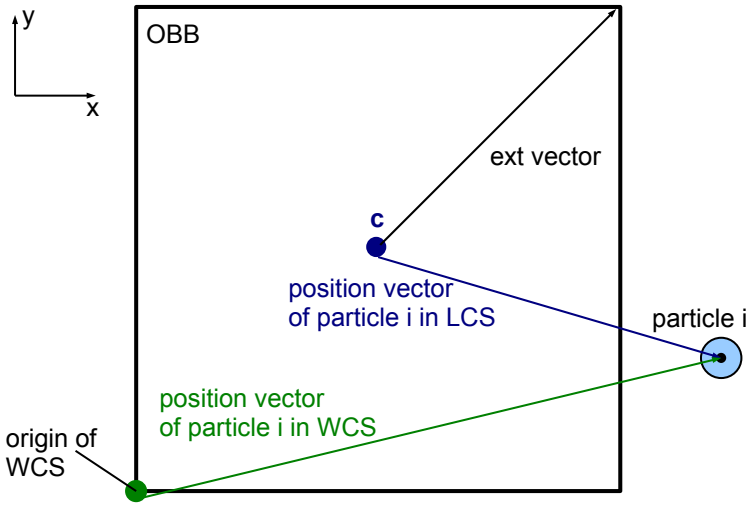

Figure 2.6: Exemplary description of particle i's position in the two different coordinate systems WCS (green) and LCS (blue).

example for the different descriptions of a particle's position in both coordinate systems.

The transformation from a regular oriented world to a local coordinate is calculated by

$$\boldsymbol{x}_{local} = \boldsymbol{x} - \boldsymbol{c}, \tag{2.42}$$

where $\boldsymbol{c}$ defines the center of the box. Regular oriented means that the axis orientation, illustrated in Figure 2.6, is of standard form and neither rotated nor skew.

With the help of the respected particle's local position, the corresponding detection function of the OBB yields

$$F_{box}\left(\boldsymbol{x}\right) = \left[|\boldsymbol{x}_{local}| - \mathrm{ext}\right]_{\max}, \tag{2.43}$$

with vector ext as the axis extent in each direction measured from the center of the box. The local contact point is

$$\boldsymbol{cp}_{local} = \min\left[\mathrm{ext}, \max\left[-\mathrm{ext}, \boldsymbol{x}_{local}\right]\right] \tag{2.44}$$

and the corresponding WCS contact point, necessary for the collision response, is given by the transformation

$$\boldsymbol{cp}_{box} = \boldsymbol{c} + \boldsymbol{cp}_{local}. \tag{2.45}$$

The further missing mandatory collision information for the box becomes

$$d_{box} = \|\boldsymbol{cp}_{box} - \boldsymbol{x}\| \tag{2.46}$$

and

$$\boldsymbol{n}_{box} = \frac{sgn\left(\boldsymbol{cp}_{local} - \boldsymbol{x}_{local}\right)}{\|sgn\left(\boldsymbol{cp}_{local} - \boldsymbol{x}_{local}\right)\|}. \tag{2.47}$$

2.5.2 Collision Response

After the mandatory information from the collision detection is summoned, we need to apply a realistic and adequate response to the corresponding collision. In [Kel06] there are several different collision responses defined, e.g. the acceleration-based, impulse-based or projection-based method. An impulse-based collision response is an event driven method modifying a particle's velocity at the very moment of the collision to avoid the penetration of the collision object. A projection-based collision response modifies the position of a particle, that is already penetrating a collision object, by projecting it back on the surface of the implicit primitive. In our implementation we do not explicitly avoid penetrations but react to them in the same time step as they appear. For this reason, a collision is not actually visible in the simulation but only the response can be observed. As the procedure of the applied collision response routine involves modifying the penetrating particle's position as well as its velocity, it can be imagined as an hybrid version of an impulse-based and a projection-based method.

2.5.2.1 Hybrid Impulse-Projection Method

As already mentioned, a projection-based collision response method projects a penetrating particle's position back on the surface of the collision object. In our implementation with implicit primitives the position projection simply yields

$$\boldsymbol{x}_i = \boldsymbol{cp}, \tag{2.48}$$

where we just set the colliding particle back to the assumed contact point on the collision object's surface.

The velocity $\boldsymbol{v}_i$ of the respected particle can be reflected using the standard vector reflection method

$$\boldsymbol{v}_i = \boldsymbol{v}_i - 2\left(\boldsymbol{v}_i \cdot \boldsymbol{n}\right)\boldsymbol{n}, \tag{2.49}$$

that results in a perfect elastic collision implying a conservation of kinetic energy. However, such a collision behaviour does not seem appropriate for fluid particles because generally they are not bouncing materials. Thus, we need to introduce a restitution parameter into (2.49) to control the conservation of kinetic energy as follows

$$\boldsymbol{v}_i = \boldsymbol{v}_i - (1 + c_R)\left(\boldsymbol{v}_i \cdot \boldsymbol{n}\right)\boldsymbol{n}, \tag{2.50}$$

where $0 \leq c_R \leq 1$ is the coefficient of restitution. With $c_R = 0$ we are able to model the usual no-slip boundary conditions suitable for any simulated liquid, while $c_R = 1$ again leads to (2.49).

Another more sophisticated reflection method comprises the penetration depth as well. Equation (2.50) does not constraint the kinetic energy for $c_R > 0$. To ensure that there is no energy introduced by the collision, we need to take care that only the velocity relative to the penetration depth is implied in the reflection equation

$$\boldsymbol{v}_i = \boldsymbol{v}_i - \left(1 + c_R \frac{d}{\Delta t \left\|\boldsymbol{v}_i\right\|}\right)\left(\boldsymbol{v}_i \cdot \boldsymbol{n}\right)\boldsymbol{n}. \tag{2.51}$$

However, if one is only interested in simulations of low viscosity fluids, the restitution coefficient will consequently be set to zero ($c_R = 0$). For this case, Equation (2.50) is equal to (2.51) and thus the simplified impulse reflection equation for brisk fluid collisions with stationary rigid bodies yields

$$\boldsymbol{v}_i = \boldsymbol{v}_i - \left(\boldsymbol{v}_i \cdot \boldsymbol{n}\right)\boldsymbol{n}. \tag{2.52}$$

2.5.3 Discussion

The main focus of the applied collision handling routine lies on the low computational cost of the collision solver. We are not interested in expensive algorithms, e.g. intersection tests between colliding geometries, to handle our rigid body collisions in the most perfect possible way. However, the simple and rapid way we solve our collision problems with the presented routines and primitives brings some irregularities into the fluid simulation. Figure 2.7

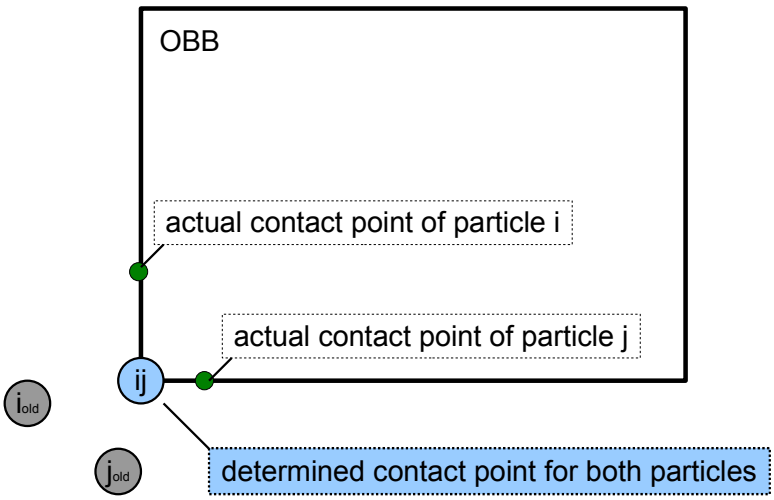

Figure 2.7: Depiction of an exemplary situation where the collision handling routine would advance two colliding particles to the exact same position at the edge of the OBB. Once two particles have merged, they float superposed

shows an exemplary situation where two particles that penetrated the OBB at different positions are set to the exact same position by our collision handling method. Unfortunately, there is no simple and affordable way to separate coalesced fluid particles. Once two particles share the same position, they will experience the same forces and will therefore superpose each other for the rest of the simulation.

Furthermore, the contact point $\boldsymbol{cp}$, determined by the collision detection routine, is usually not the entrance point of the particle into the collision object but this point on the implicit primitive's surface that lies nearest to the partricle's current position and therefore normal to the surface. Finally, one needs to decide whether accuracy or the computational rate of a certain method is more important. For the sake of real-time achievements of our SPH solver, the presented collision handling method meets our needs sufficiently anyhow. Moreover, the smaller we choose the time step Δt, the better the results of the collision handling routine will turn out. Therewith, a situation like it is depicted in Figure 2.7 would be avoidable by simply adapting Δt.

Chapter 3

Implementation

The most challenging part of the thesis, along with the study of the fluid dynamics theory and the determination of the SPH quantities, is the implementation of the particle-based solver and the setup of the whole simulation program. This chapter provides a thorough introduction of optimization techniques, including algorithms and data types to make the SPH simulation work efficiently as well as the necessary physical parameters to make it work realistically. Beyond that, it is supposed to support less experienced readers with a detailed implementation guideline and is finally presenting an approach into the field of parallel computing using the OpenMP API. The following semantics of exemplary algorithms and the introduced data types are based on the C++ programming language. As already mentioned in Section 1.2 in the introductory chapter, the main application field of this implementation is the simulation of various flow scenarios of water. However, the readers who are interested in the simulation of other liquids or even gases should be aware of the fact that the only difference lies in the measurement of the physical parameters in Section 3.2. The optimization techniques and programming guidelines presented here are therefore still applicable and helpful for other real-time simulation approaches of further fluid materials.

3.1 Optimization Techniques

3.1.1 Linked Cell Method

As already mentioned, an important goal of this thesis is to achieve a real-time fluid dynamics solver. Therefore, we first have to gain insight into the complexity of the whole simulation. Even though particle-based methods like the SPH method are generally qualified for efficient simulation programs, there is always a necessity to apply certain optimization techniques and algorithms to improve the computational complexity of a method. The calculation of any

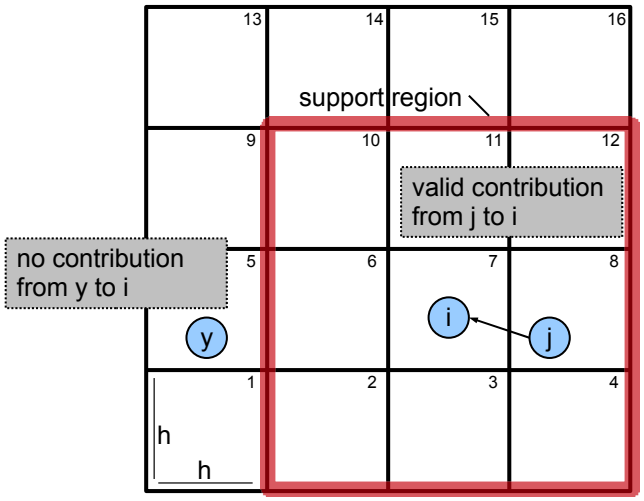

Figure 3.1: Main idea of the Linked Cell method. Equilateral cells are put under the simulation domain and allow for a simple determination of the support region of a certain cell (here 7) and the contained particles. Further particles outside of the support region are left untouched.

fluid quantity field requires the iteration through all particles in the simulation. Moreover, the calculation of each particle's SPH terms, except for the directly applied fields, needs the contribution of every other particle in the simulation as well. Altogether the complete evaluation of any SPH term with N simulated particles would have the computational complexity of $\mathcal{O}(N^2)$, making it irrelevant to talk about real-time capabilities.

However, the property (2.12) of the kernel functions states that a certain particle's contribution to any quantity is zero, if the particle is located outside the smoothing radius h. Hence, it is sufficient to iterate only over adjacent particles that lie within the support radius and leave the further particles untouched. The Linked Cell method, also used for short-range potentials in molecular dynamics simulations [GCKn03], is quite a straightforward way to reduce the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(M \cdot N)$, where M is the average number of particles located in a certain support region. Figure 3.1 gives an overview of the main idea of the Linked Cell method in two dimensions. Each depicted cell is implemented as a particle container that holds a copy of or a reference to every contained particle. If each cell's side length is $\geq h$, the method will guarantee that for the computation of any SPH term, it is enough to iterate through the respected particle's cell and its neighbouring cells only. Depending on the dimension of the simulation domain the Linked Cell algorithm limits the maximum number of involved cells per

Algorithm 3.1 C++ - definition of the particle and cell structure as well as each vector containers.

```
struct Particle{
    real x[DIM];
    real v[DIM];
    real f_pressure[DIM];
    real f_viscosity[DIM];
    ...
    real mass_density;
    ...
};

struct Cell{
    std::list<Particle*> p_list;
};
...
std::vector<Particle> p_vec;
std::vector<Cell> c_vec;
```

SPH computation to 27 in three dimensions and nine in two dimensions.

Obvious drawbacks of the Linked Cell method are the memory overhead of the cell structure and an additional routine to update the content of the cells after each time step to transfer all particles that moved to an adjacent cell to their new container cell. However, the gain of the reduction of the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(M \cdot N)$ supersedes any drawbacks considering higher memory consumption and additional methods to update the cells' contents. After the pros and cons of the Linked Cell method have been discussed, we now would like to deal with the implementation of the method as such, talking about applied data types and structure in memory.

With regards to our Linked Cell variant, a single cell is implemented as a structure carrying a linked list of references to those particles that are positioned inside the corresponding cell. The numerous cell structures covering the domain are organized in a standard vector container as shown in algorithm 3.1. Due to the fact that the cell structs are pushed into the vector in lexicographic order, the indices likewise will be arranged lexicographically. In our case, the lexicographic ordering means that the cells are enumerated at first in x-direction, then in y- and at last in z-direction. There is also an exemplary depiction of the cell enumeration in Figure 3.1 in the two dimensional case. With such an arrangement, it is always possible to determine the respective cell's direct neighbours without searching through the vector. Hence, we can guarantee that each cell is accessible in $\mathcal{O}(1)$. Algorithm 3.2 describes how an arbitrary cell is easily addressed by its index position in the cell vector.

Algorithm 3.2 Definition of the function *index* that returns the index position of any addressed cell in the vector. Integer array *ic* contains the three dimensional coordinates, expressed in cell sizes, of the respective cell, while *nc* provides the total number of cells in each direction.

```
int index(int* ic, int* nc){
    return( ic[0] + nc[0] * ( ic[1] + nc[1] * ic[2] ) );
}
...
list<Particle*> &icList = c_vec[ index( ic, nc ) ].p_list;
```

Simultaneously, we now have access to the whole particle reference list *p_list* of the cell. A brief discussion about the reasons why we decided to use the here presented data types and containers can be found in subsection 3.1.3.

3.1.2 Optimization of Memory Access

The main issue of numerous computational simulations today is not the lack of computing power of the CPU but the performance bottleneck of the main memory. The computer engineers try to counteract this problem with the integration of hierarchical cache architectures. Those caches are small but very fast memory chips holding copies of data elements from the main memory to provide a much faster access to these data values. However, if the implementation of a certain computer program does not account for the functionality of the cache architecture, the developer of the program will not benefit of the possible advantages and the performance of the code will rather suffer from the cache operations.

The general idea of caches is to store these data elements that are most possibly used next or in the near future by the executed program. In fact, data elements are not transfered separately but in so-called cache lines, which contain a certain number of data elements. Consequently, the code developer should use the already cached data elements whenever it is possible to exploit the performance of cheap data accesses, also called cache hits. Otherwise, cache misses will most likely occur, which limit the performance by the speed of main memory latency and bandwidth.

Coming back to our fluid simulation, we are suffering from incoherent memory accesses as well. While the fluid particles themselves are properly stored sequentially in a vector container, their ability to float through the whole domain and consequently enter different cells of our Linked Cell domain leads on to a more or less unstructured memory access to the particles when we iterate over the cells as illustrated in Figure 3.3. However, with the application of the Linked Cell method it is generally not possible to access the particles sequentially in any case without the necessity of expensive additional effort,

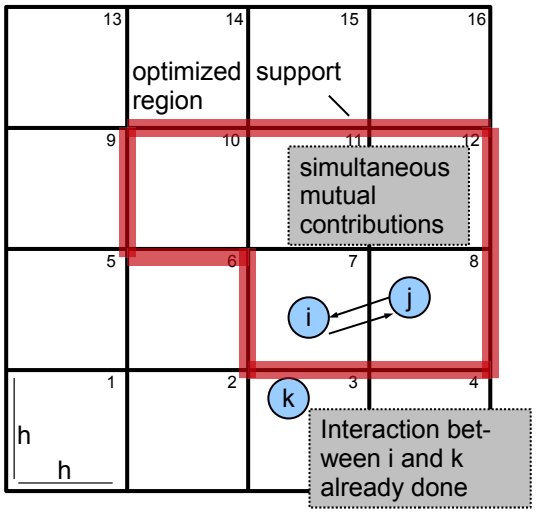

Figure 3.2: Optimization of the Linked Cell support region by making use of symmetric particle-particle contributions. Cell 7 is the respective cell again.

e.g. the permanent sorting of the contained particles. Nevertheless, there is a possibility to minimize cache misses within the Linked Cell-iterations. For example, while we calculate a certain force field, e.g. the pressure force of particle i with regards to the respective contribution of particle j with the help of the arithmetic mean variant (2.22)

$$\boldsymbol{f}_{ij}^{pressure} = -\frac{p_i + p_j}{2}\frac{m_j}{\rho_j}\nabla W\left(\boldsymbol{x}_i - \boldsymbol{x}_j, h\right), \tag{3.1}$$

we should take advantage of the temporal locality property of cache elements. Hence, we calculate particle i's contribution to particle j's pressure force right after expression (3.1) as well

$$\boldsymbol{f}_{ji}^{pressure} = -\frac{p_j + p_i}{2}\frac{m_i}{\rho_i}\nabla W\left(\boldsymbol{x}_j - \boldsymbol{x}_i, h\right) \tag{3.2}$$

and may experience a performance gain because particle i's and particle j's fluid quantities have already been cached for the calculation of equation (3.1). The term *temporal locality* describes the profitable reuse of previously cached data elements and the involved increase of the cache hit rate.

Using this scheme of successive computations, we do not only benefit from the possible cheap data accesses alone but we also handle two contributions in one iteration step at once. In principle, this optimization technique should also be helpful for other SPH implementations without the application of the

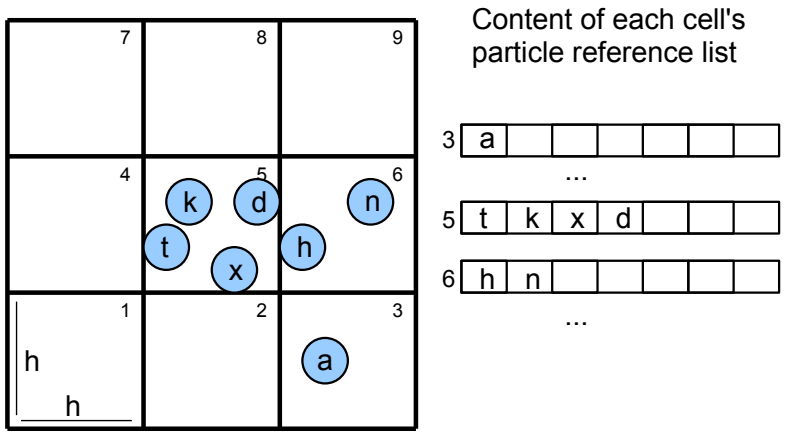

Figure 3.3: Arbitrary distribution of particles in the simulation domain. Each cell stores a reference to the comprised particles in a list.

Linked Cell method. However, in combination with this method, we can reduce the number of necessary cell lookups per SPH term evaluation per particle again. Considering the determination of a certain particle's quantity field, it is now sufficient to touch the considered particle's own cell and its neighbouring cells with a higher lexicographic number only. Figure 3.2 depicts the further limited support region caused by this order of computations in two dimensions. The support region needs to cover the higher numbered neighbour cells only because the contributions from the particles in the lower ordered neighbour cells have already been calculated in the previous iteration step. Moreover, concerning particle i's own cell, we only need to loop over adjacent particles that are listed in p_list after particle i. Expressed in terms of complexity, the maximum number of considered cells per SPH computation is reduced to 14 in three dimensions and to five in two dimensions, while there is no additional algorithmic or memory overhead necessary. Altogether, this optimization technique approximately has halved the computational complexity of the Linked Cell method alone and moreover, we exploit the temporal locality of cached data elements providing a further speed-up of our SPH solver.

3.1.3 Discussion and Further Improvements

In the previous two subsections, we have already introduced the optimization techniques that influence our solver's complexity most intensely. However, as we already suggested in the last subsection, the cell-iterations needed to evaluate the SPH terms for each of the particles are leading to very unstruc-

tured and non-sequential memory accesses. Figure 3.3 shows a composition of exemplary simulation states and the content of the corresponding particle reference lists. Considering the particle constellations, we have to iterate over each cell to determine a certain fluid quantity field of each involved particle. Coming across cell 3 we need all particles from cell five and six to completely calculate the smoothed field values for particle a. Obviously, we need to jump through the particle vector in a more or less arbitrary way causing our simulation program to mess around with memory accesses and to lose a lot of time by demanding particle elements from the main memory. The question is, if we can do any better by not storing particle references inside the cell but the particle elements themselves and if the implementation of the list container is the best choice for our purpose.

Let us think about the main requirements for a proper particle container within each cell. We actually need the possibility to efficiently enumerate, insert and delete the particles in the cells. Considering the container type, we restrict our performance examination to the application of the C++ STL containers *list* and *vector* for the reason of simplicity. First of all, we want to introduce the abilities of both container types based on [Jos01]. The vector container shows good performance for appending and deleting elements at its end but poor results for insertion and deletion at middle positions. The reason is that the elements of a vector are always stored successively causing the reassignment of every element behind the deleted or inserted one. Moreover, the standard implementation of the vector container does not automatically provide the reduction of the vector's size and the deallocation of unused memory after the deletion of a certain element. Due to the fact that we generally are not able to predict the definite amount of memory required for each cell's container, the memory management of the vector type seems unfavourable for our purpose. In contrast to the vector, a list container does not provide random access leading to slow performance when accessing a particular element. However, the implementation of the standard list allows for efficient insertion and deletion at any position because the internal structure of the list does only operate on pointers. Generally, the list elements are not stored in a structured and coherent way causing the program to jump through the memory when iterating over the contained data elements.

We already mentioned that we have a major problem with unstructured memory accesses due to the fact that the particles are flowing freely around the cell domain. One way to possibly avoid the expensive arbitrary accesses to the particle vector p_vec would be to store not the references but the particles themselves in each cell's container. This way, we would not need an external particle storage anymore. However, this would only make sense in combination with a vector container that stores its elements successively. The list container would not exploit the possible advantages because of its property to store its elements in an unstructured manner anyway. The vector container

helps us to organize our memory accesses to the adjacent particles in a better way. Unfortunately, the deletion of particles from an arbitrary position in the vector is a very often performed operation in our simulation and especially for this, the vector container shows a bad performance. Furthermore, the larger size of a whole particle element in contrast to the reference would even lead to a worse performance when it comes to shifting elements inside the vector container. Finally, we have to face the fact that there does not seem to be an ideal way to guarantee coherent data access in combination with minimum complexity for the cell updates. In order to make a decision, we preferred storing particle references instead of whole particles and accept unstructured memory accesses in favor of an expensive and memory inefficient cell update routine.

After we decided to store particle references in the cell containers and try to live with expensive and incoherent memory accesses, the list container provides the most suitable and efficient properties. Considering our implementation, the main advantage in contrast to the standard vector container is the better performance for the deletion of elements at arbitrary positions that is constantly necessary for the cell update procedure.

While we introduced the fluid mechanical force and density fields in Section 2.3 we already came across the symmetry conservation, regarding Newton's action-reaction law, of force and density field contributions. In our case, it states that particle j's smoothed quantity contribution towards particle i is the negation of the contribution particle i exerts on particle j, generally meaning $\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$. Considering this symmetry of particle-particle interactions, we are able to skip certain redundant computations that slow down our SPH solver for nothing. In fact, the action-reaction law cannot be applied for our SPH terms as it was originally set up. For example, if we take the evaluation of the viscosity force, we will observe the following two equations for the mutual contributions of particle i and j

$$\boldsymbol{f}_{ij}^{viscosity} = \mu \left(\boldsymbol{v}_j - \boldsymbol{v}_i \right) \frac{m_j}{\rho_j} \nabla^2 W \left(\boldsymbol{x}_i - \boldsymbol{x}_j, h \right) \tag{3.3}$$

$$\boldsymbol{f}_{ji}^{viscosity} = \mu \left(\boldsymbol{v}_i - \boldsymbol{v}_j \right) \frac{m_i}{\rho_i} \nabla^2 W \left(\boldsymbol{x}_j - \boldsymbol{x}_i, h \right). \tag{3.4}$$

Considering our simulation, we employ an identical mass for each particle $m_i = m_j = m$. Hence, Equation (3.3) and (3.4) can be simplified furthermore. If we take a closer look on each of the terms, we will observe that they only differ in the sign and the mass density, making it possible to reduce the computational effort as presented in Algorithm 3.3.

Algorithm 3.3 Code example for the exploit of symmetrical force contributions. The negative sign in the last assignment arises from the interchanged velocity difference in the first term.

```
...
tmp = mu * (v_j - v_i) * m * v_kernel_lap( x_i - x_j, h );

f_viscosity_ij = tmp / mass_density_j;
f_viscosity_ji = (- tmp ) / mass_density_i;
...
```

Obiously, we do not fully agree with Newton's third law of motion because of the scaling factor comprising the respective mass densities. However, the action-reaction law helps us to understand the particle-particle interactions more clearly and to exploit the possible mathematical simplifications more easily. Regarding this short example for the viscosity force, we actually save three floating point multiplications, two additions and one kernel function evaluation per dimension. Such a splitting method is applicable for any SPH based continuous quantity field in our fluid simulation and is a minor but effective optimization technique to further reduce the computational complexity of the solver.

3.2 Simulation Parameters

The following parameters are mostly based on the calculations and experiments from [Kel06]. For the sake of integrity of our own implementation, all of the necessary parameters applied for the SPH water simulation solver are listed here as well. While some of the parameters are based on physical properties, other values are determined by simulation experiments improving the stability of the solver and the realistic behaviour of the particles.

Description	**Symbol**	**Value**
time step	Δt	$0.01\,s$
gravitational acceleration	g	$\begin{pmatrix} 0 \\ -9.82 \\ 0 \end{pmatrix} \frac{m}{s^2}$

Table 3.1: General parameters independent of the simulated fluid.

Description	Symbol	Value
rest density	ρ_0	$998.29\,\frac{kg}{m^3}$
mass	m	$0.02\,kg$
viscosity	μ	$3.5\,Pa \cdot s$
surface tension	σ	$0.0728\,\frac{N}{m}$
threshold	l	7.065
gas stiffness	k	$3\,J$
restitution	c_R	0
support radius	h	$0.0457\,m$

Table 3.2: Physical parameters and their values used for a realistic simulation of water dynamics.

3.3 Implementation Guideline

Throughout this section, we focus on a thorough and understandable guideline how to implement the SPH fluid solver in combination with the presented collision handling and time integration scheme as well as the above introduced optimization techniques. Especially the use of the Linked Cell method is mandatory for the following implementation tutorial because each of the particle neighbourhood iteration is based on the cell structure. In order to simplify our guideline, we make use of different listing formats to emphasize the order of the necessary steps. The operations done in the indented lines depend on the corresponding parental line and therefore need to be executed successively. In the same way, the numerically and alphabetically enumerated steps have to be performed in the exact order they are listed. Otherwise, the arbitrarily marked list elements sharing the same indentation depth can be executed in a user-defined sequence.

3.3.1 Initialization of the Simulation

1. Set up the necessary information for the implicit primitive container and possible obstacle objects.
2. Create the fluid particles and set their positions, velocities and other quantities to their according initial values. Insert each of them in the particle vector container p_vec with suitable size N.
3. Initialize the cells for the Linked Cell method with equidistant side length of h and order them lexicographically in another vector structure denoted in algorithm 3.1. Assign the reference to each single particle to the correspondent cell's particle list according to the particle's initial position $\boldsymbol{x}$.

4. If necessary, initialize the leap velocity $\boldsymbol{v}_{-0.5\Delta t}$ for the leap-frog integrator as introduced in (2.35) for each particle.

3.3.2 Evaluation of the Mass Density

Iterate over each cell ic in the simulation domain Ω in lexicographic order

- Iterate over each particle i in ic
  1. Iterate over each particle $j > i$ in ic
     - Calculate the mutual mass density contributions for particle i and j using (2.17).
  2. Iterate over each cell $kc > ic$ in the neighbourhood of cell ic
     - Iterate over each particle k in kc
       - ▷ Calculate the mutual mass density contributions for particle i and k using (2.17).

3.3.3 Evaluation of the Pressure Field

Iterate over each particle i in the particle vector p_vec

- Calculate the pressure field using (2.19).

3.3.4 Evaluation of the Internal and External Forces

Iterate over each cell ic in the simulation domain Ω in lexicographic order

- Iterate over each particle i in ic
  1. Calculate the gravity force of particle i using (2.26)
  2. Iterate over each particle $j > i$ in ic
     - Calculate the mutual pressure force contributions for particle i and j using (2.22)
     - Calculate the mutual viscosity force contributions for particle i and j using (2.25)
     - Calculate the surface normal $\boldsymbol{n}$ and the Laplacian of the smoothed color field c of particle i and j using (2.28) and the Laplacian of Equation (2.27)
  3. Iterate over each cell $kc > ic$ in the neighbourhood of cell ic
     - Iterate over each particle k in kc

- ▷ Calculate the mutual pressure force contributions for particle i and k using (2.22)
- ▷ Calculate the mutual viscosity force contributions for particle i and k using (2.25)
- ▷ Calculate the inward surface normal $\boldsymbol{n}$ and the Laplacian of the smoothed color field c of particle i and k using (2.28) and the Laplacian of equation (2.27)

4. Calculate the norm $\|\boldsymbol{n}\|$ particle i's inward surface normal
5. If $\|\boldsymbol{n}\| \geq l$ then
   - Calculate the surface tension force of particle i using (2.30)

Keep in mind to simplify the particle-particle contributions for the SPH-based force terms as well as for the mass density field as denoted in subsection 3.1.3 to reduce the number of necessary operations. The order of the calculations applied in this implementation guideline is not mandatory at all. We just tried to keep the computational effort of the domain-covering Linked Cell iterations as small as possible. Hence, we calculate each of the different force fields in the same iteration of the respective loop causing our code to look unorganized at first sight. For the sake of a more structured implementation, there is the possibility to split the computation of the internal and external force fields into separate routines due to their mutual independence. However, the computational cost of an additional domain-covering cell-iteration is not compatible to our ambition to achieve real-time performance.

3.3.5 Leap-Frog Scheme and Collision Handling

Iterate over each particle i in vector p_vec

1. Calculate the total force $\boldsymbol{f}$ acting on particle i using (2.32)
2. Calculate the acceleration of particle i with (2.16)
3. Update particle i's leap velocity $\boldsymbol{v}_{t+0.5\Delta t}$ using (2.33)
4. Update particle i's position using (2.34)
5. Approximate the current velocity $\boldsymbol{v}$ with (2.36)
6. Check for a collision with an implicit primitive boundary or obstacle object evaluating the detection function (2.37)
   If a collision is detected
   (a) Summon the mandatory collision information listed in subsection 2.5.1

(b) Project particle i's position back on the contact point $\boldsymbol{cp}$

(c) Modify its velocity with (2.51)

3.3.6 Linked Cell Update

Iterate over each cell ic in the simulation domain Ω in lexicographic order

- Iterate over each particle i in ic

  1. Compare particle i's current position with the extent of the cell ic
  2. If particle i moved to different cell kc
     (a) Add particle i to cell kc
     (b) Delete particle i from cell ic

3.3.7 Visualization

In order to visualize our simulation results, we either used the open-source visualization software *ParaView* [Kit10] or the *POV-Ray* tool [Pov10]. Both simulation tools require a specific file format and different information to visualize the particle-based simulation satisfactory. In each case, we need to perform the following steps:

- Every n-th SPH simulation step

  1. Open a new visualization output file in the desired format
  2. If necessary, write out the information about the simulation container and possible obstacles
  3. Iterate over each particle i in the particle vector p_vec
     – Write out the file format dependent information for particle i
     – If necessary, use

$$r = \sqrt[3]{\frac{3m}{4\pi\rho_0}} \tag{3.5}$$

     as the radius to render the particles as equally sized spheres
  4. Close the output file

3.4 Parallel Programming using OpenMP

There are different ways to parallelize an existing sequential program code like the one we produced with the recently presented implementation guideline. The design of the respective computer system is a very decisive element for the actual implementation of parallel algorithms. Just to mention a few different computer architectures, there are cluster systems combining several computing machines, single computing machines with multiple processors or single computing machines with a single processor comprising multiple cores. In addition, we have to distinguish between shared and distributed memory architectures. For our parallelization purpose, we want to focus on the architecture that is implemented in the majority of today's personal computers, a single processor multicore shared memory system.

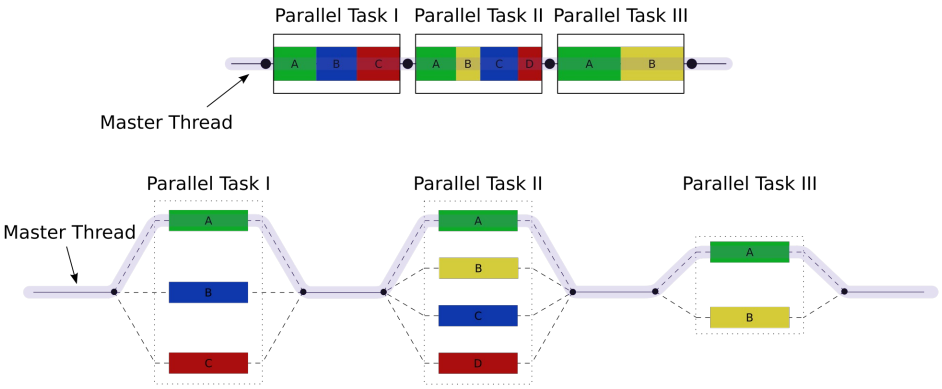

Figure 3.4: The task fork mechanism of OpenMP.
Source: http://en.wikipedia.org/wiki/File:Fork_join.svg

The OpenMP API [Omp10] is primarily designed to enforce parallel programming in combination with shared memory systems. It is applicable on multiple platforms and can be implemented in a rather simple way by adding specific compiler directives to an existing program written in C, C++, or Fortran. Figure 3.4 depicts the main idea of the parallelization process using OpenMP. Generally, the program is executed in the usual sequential manner and processed by the master thread only. If a certain OpenMP parallelization directive is called from inside the code, a specified number of tasks will share the upcoming workload until the end of the parallel section. There, the parallel tasks merge into the master thread again. In combination with certain extensions or MPI, OpenMP can be used for the parallelization of distributed memory systems as well. However, in the following examples we restrict to the use of OpenMP only and therefore stick to shared memory systems.

Algorithm 3.4 General construct of the OpenMP *for*-directive.

```
#include <omp.h>
...
#pragma omp parallel for
```

Algorithm 3.5 Parallelization of a particle-loop using the OpenMP API *for*-construct.

```
#pragma omp parallel for shared( p_vec )
for( int i = 0; i < p_vec.size(); ++i ) {
    Particle &p = p_vec[ i ];
    ...
}
```

Obviously, the code parts mainly considered to be parallelized in our simulation are the expensive particle- and cell-loops. The OpenMP API provides a particular construct to parallelize *for*-loops. In C and C++, any directive is specified using the *#pragma* mechanism. In Algorithm 3.4, there is a simple example how to parallelize the workload of the directly following *for*-loop. In fact, there are lots of different directives and constructs to parallelize a sequential piece of code with OpenMP, but for the sake of simplicity and usability we restrict our implementation to the use of the loop-directive presented in algorithm 3.4. For further details and possibilities concerning the OpenMP API, one can find the specification in [Omp10]. In addition to the simple *#pragma* construct, we need to declare *private* and *shared* variables to ensure that any parallel task does not mess around with another task's variables causing segmentation faults or irregular work flow. The *private clause* declares the contained data elements to be private for each parallel task. This means that for each data element contained in the *private* list, a separate element will be automatically allocated for each task. Hence, each task will be referencing another element guaranteeing the unique access to this variable only from within this particular task. Contrarily, the *shared clause* declares the contained data elements to be shared by all parallel tasks. If any task modifies the content of a shared variable, every other task will possibly work with the modified value as well. The clauses cannot be applied arbitrarily but one has to think about the way a certain data element needs to be declared within the parallel region. However, we will examine the different clauses by introducing our way of applying the OpenMP constructs.

Starting with the simpler particle-loop necessary for the evaluation of the pressure field (2.19) and for the leap-frog scheme presented in Subsection 3.3.5, we just need to apply the code construct depicted in Algorithm 3.5 to parallelize each of the particle-loops. The particle container *p_vec* is explicitly declared as a shared variable because the parallel tasks are all accessing the

Algorithm 3.6 Parallelization of the cell-loop.

```
int ic [ DIM ];
int kc [ DIM ];

#pragma omp parallel for shared( nc, c_vec ) private( ic, kc )
for( int i = 0; i < nc[2]; ++i ) {
    ic[2] = i;
    for( ic[1] = 0; ic[1] < nc[1]; ++ic[1] ) {
        for( ic[0] = 0; ic[0] < nc[0]; ++ic[0] ) {
            ...
        }
    }
}
```

same data element. The particle alias p is automatically declared private because it is defined inside the parallel region. Regarding the *for*-construct, the advantage of the parallelism lies in the subdivision of the number range of the loop variable i in order to speed up the execution of the loop. While in the sequential case the program needs to iterate *p_ vec.size()* times in a row through the construct, the same is now done by several tasks processing certain chunk sizes of the whole number range of i in parallel. However, we have to take care on our own that the calculations and assignments inside the parallel region are actually adequate for parallelization. For example, mutual dependencies of data elements or input/output operations are somehow difficult to program in parallel or at least difficult to control and are therefore mostly not worth the additional effort. Generally, this is the reason why we waive the parallelization of subsection 3.3.7. For the same reason, we do not parallelize the Linked Cell update presented in Subsection 3.3.6. If we need to move a certain particle from one cell to another, we will mess around with the iterator of the cell's particle container by deleting the particle element. Without additional serialization effort, we cannot guarantee that each of the other threads is still working properly.

Concerning the more delicate cell-iterations in our SPH solver, the basic appliance is similar to Algorithm 3.5. As we already discussed throughout Subsection 3.1.1, we have ordered our Linked Cells lexicographically in the vector container *c_ vec*. Hence, we iterate through our cell-domain in the same manner. Algorithm 3.6 shows an exemplary possibility to parallelize our cell-loop. The array *ic* is used to define the currently respected cell's position in the cell-domain, while *kc* is the iteration element for the neighbourhood of cell *ic*. Both are declared as private variables to ensure that each parallel task is only modifying its own versions and is able to step independently from the other tasks through the cell-neighbourhood. One restriction of the OpenMP

directives is that variables, that are defined outside the parallel region and declared *private* in the construct, have to be initialized inside the parallel section anyway. Afterwards, the value of these predefined elements is unknown. Imagining our cell-domain as a three dimensional OBB, we can see clearly that the presented cell-loop divides the space in its basic coordinates. In our example, only the number of cells in the third direction, the z-direction, is subdivided into parallel processed chunks.

Finally, we have not yet declared how many tasks we actually want to comprise in the parallel work flow. In the OpenMP API there are different ways to define the desired number of threads for the parallel regions. For this, we have chosen to use the OpenMP function

$$void\ omp\_set\_num\_threads(\ int\ num\_threads\ ),$$

where *num_threads* is supposed to be a positive integer. The function is comprised in the *omp.h* header file and has to be called by the master thread to affect the subsequent parallel regions.

Chapter 4

Discussion of Results

As already mentioned repeatedly, the goal of this thesis is to achieve a proper C++ implementation of the SPH method with particular focus on the reduction of the computational costs and stability. In this chapter, we want to deal with the outcome of our particle-based water simulator in terms of performance improvements using the introduced optimization techniques, particular simulation issues coming across as well as a short conclusion about the general qualification of the SPH method.

To provide a clearer understanding of the visible results of the fluid simulation and the actual fluid flow, we present some visualizations of exemplary implemented scenarios. Figure 4.1 shows water particles falling into a glass bowl visualized at particular steps of the simulation time.

Figure 4.1**a**) shows the initial arrangement of the 2,197 water particles. Accelerated by gravity the particles start falling into the bowl (see 4.1**b**)). The impact of the surface tension force provides for the minimization of the fluid's surface causing a more or less spherical formation of the particle crowd. In **c**) the amount of water spreads over the bottom ground of the bowl, while in 4.1**d**) the steepness of the bowl's wall enforces the water to flow back causing the particles to mount up in the middle. During **e**) and **f**) the effect of spreading and flowing back together is repeated and attenuates over time until the water comes to rest. Just to get a feeling of the real duration of such a small-scale simulation, the elapsed real-time from **a**) to **f**) is approximately 1.5 seconds. Another typical water scenario is the *Breaking Dam* simulation depicted in Figure 4.2 using 3,000 particles. The general idea of this scenario is to place a certain body of a fluid in a fractional area of the whole simulation domain. After the fluid has come to rest, the panel that separates the domain volume is removed. Consequently, the body of the fluid propagates in the entire simulation domain. Regarding our visualization, in **a**) the dam is broken (the panel is removed) and causes the body of water to flow freely in the open direction visualized in **b**). As the particles reach the solid right

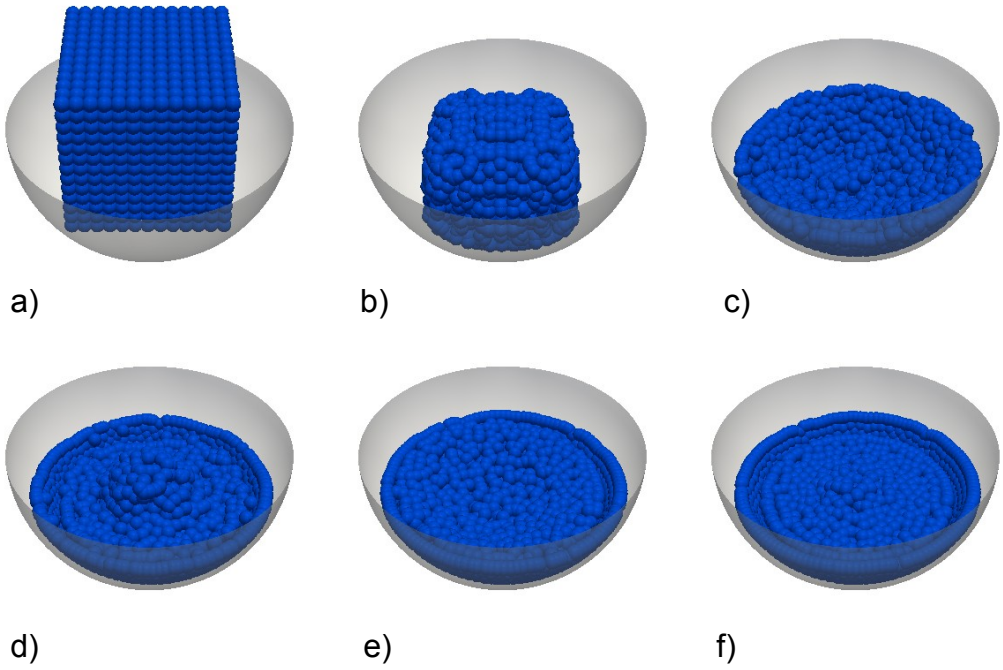

Figure 4.1: Simulation of 2,197 water particles falling into a solid bowl.

wall in **c**), they are forming a wave spilling back to the opposite direction in **d**). In **e**) a second minor wave is formed, and again the water spills over to the opposite wall in **f**). Obviously, these visualization scenes demonstrate an unusual behaviour of the fluid particles at the walls of the OBB. There, the particles tend to adhere to the surface of the wall. We believe that this results from the applied no-slip boundary condition (2.51) as well as the suboptimal rectangular geometry. Further issues concerning the use of the OBB container and the simulation as a whole, are discussed in Section 4.2.

The examination of fluid flows disturbed by obstacle collisions has been a field of interest throughout this work as well. Therefore, Figure 4.3 depicts an examplary scenario where a large body of water, comprising 3,000 particles, approaches a spherical obstacle object with a size of $9.2\,dm^3$. When the particles collide with the object in **b**), they need to evade the obstacle causing them to become denser around the obstacle object. However, the kinetic energy of the colliding particles in combination with the following body of water pushes the particles laterally past the obstacle visualized in **c**) and **d**).

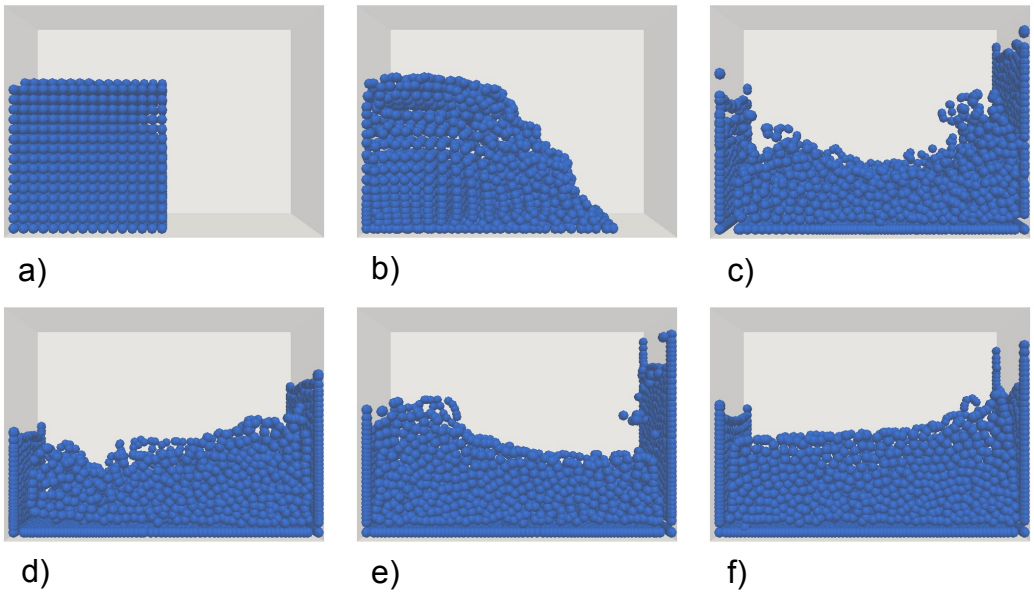

Figure 4.2: Simulation of the *Breaking Dam* scenario. Visualization of 3,000 particles modeling approximately 50 liter of water.

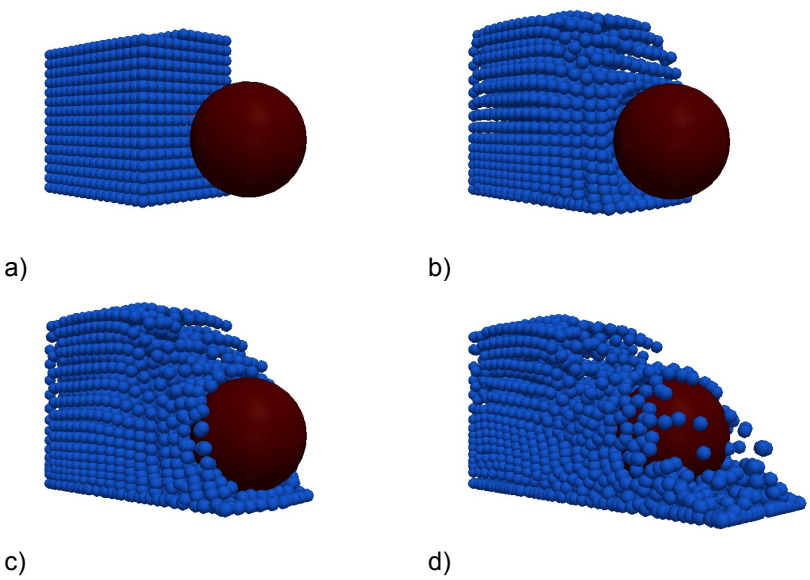

Figure 4.3: Visualization of 3,000 water particles (about 50 liter) flowing towards a spherical obstacle object.

4.1 Performance

The performance measurements are supposed to highlight the run-time of several different developed SPH implementations. For the run-time examination we employed an *Intel Quad-Core i5 520M* 2.4 GHz mobile CPU with 4GB main memory.

So far, we justified the appliance of the Linked Cell method only on the complexity notation for the particle-particle interactions but we have not yet presented the actual performance increase we are able to obtain by that. Figure 4.4 depicts the run-time performance of several different SPH solver implementations using the same preconditions. The number of applied optimization techniques generally increases from the left bar to the right bar. Furthermore, we intend to present the time scaling of each implementation by modifying the number of simulated water particles. The first bar in each group represents the run-time of the implementation without the Linked Cell method, while the other three versions employ this technique. The difference of these three implementations are additional successive optimizations of the Linked Cell support region as well as certain data access improvements as presented throughout Subsection 3.1.2 and 3.1.3.

The main statement of the performance measurement proves that with the application of the Linked Cell method alone we are actually reducing the run-time complexity of the SPH solver by approximately 85% simulating 500 particles and for $10,000$ fluid particles even by approximately $94,5\%$ compared to the unoptimized implementation. Moreover, we examine that the exploit of the simultaneous mutual particle contributions, introduced in Subsection 3.1.2, in contrast to the single particle contributions provides a further reduction of the run-time complexity of up to 76%. Besides these major improvements, the impact of the calculation optimization by reusing already evaluated terms seemingly plays only a minor role for the run-time measurements. Nevertheless, we obtain a small performance increase using this additional optimization. Concerning the time measurement for the simulation of $10,000$ particles, we examine an over all run-time improvement of about 98.8% between the unoptimized and the fully optimized implementation of the SPH solver.

Regarding the computational complexity, we examine that the unoptimized version is truly an $\mathcal{O}(N^2)$ scheme because the measured time is affected quadratically by the number of particles. For example, if we double the number of particles, we will quadruple the run-time of the unoptimized solver and if we increase the amount of particles by a factor of 5, the resulting time will be multiplied by a factor of 25. This effect can be directly obtained by the measurements depicted in Figure 4.4. Considering our final implementation, the run-time increase factor caused by the doubling of the particle amount actually cannot be determined as a constant value. Contrarily, it strongly de-

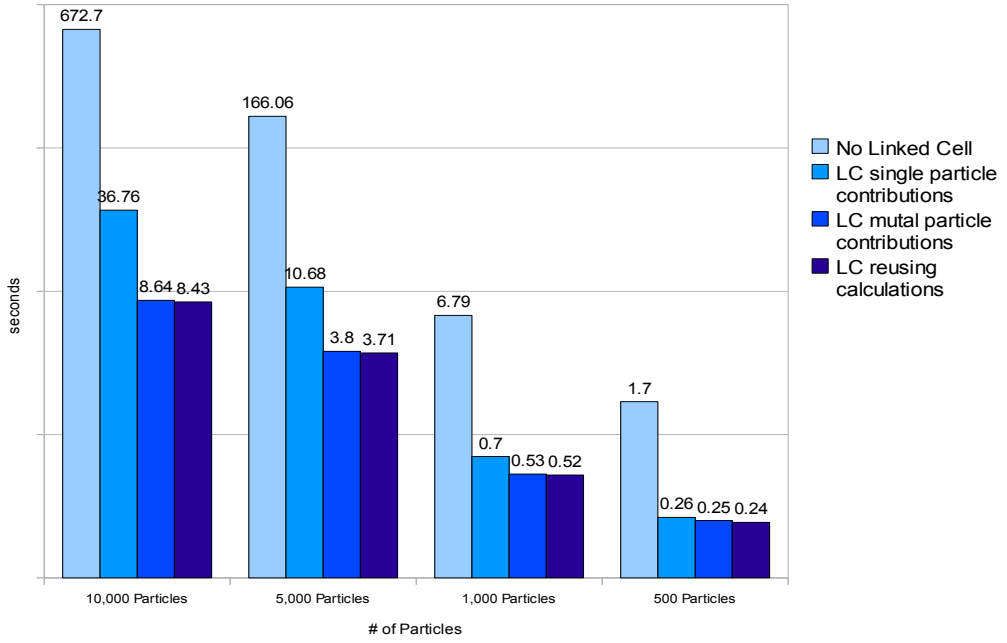

Figure 4.4: Run-time of different sequential SPH implementations for 100 iteration steps. The x-axis depicts the number of simulated particles and the y-axis shows the run-time of the solver in seconds scaled logarithmically. Each presented time measurement shows the fastest run-time out of three successive experiments.

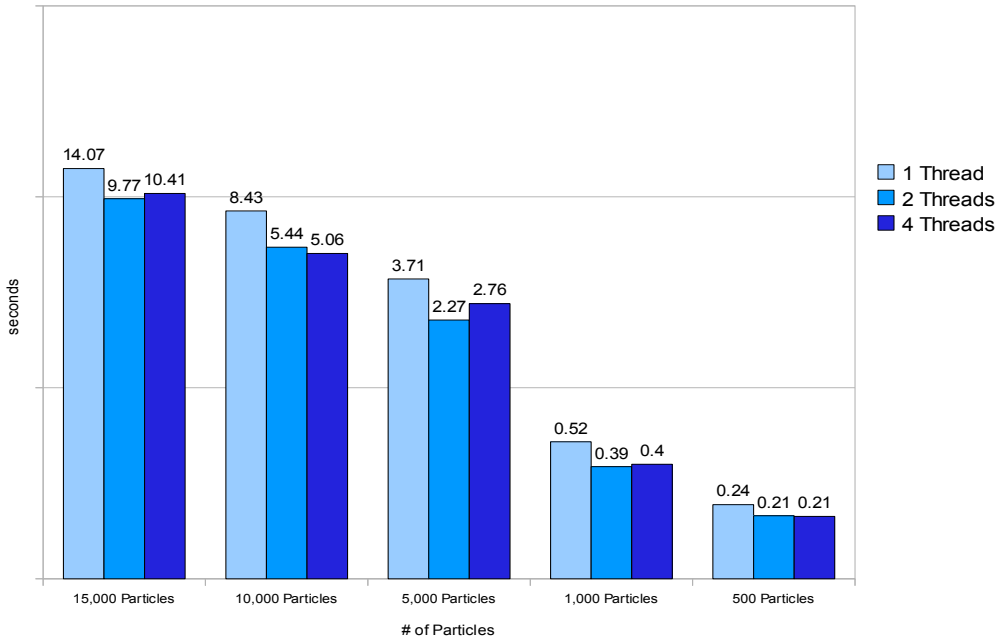

Figure 4.5: Run-time of the chosen SPH implementation with regard to the impact of the OpenMP parallelization using 100 iteration steps. Each presented time measurement shows the fastest run-time out of three successive experiments. As before, the y-axis is scaled logarithmically.

pends on the mean amount of particles involved in the Linked Cell support region, already introduced as the quantity M in Subsection 3.1.1 that affects the complexity $\mathcal{O}(M \cdot N)$ using the Linked Cell method.

Another important topic of this section is the observation of the performance improvement by the parallelization of the particle-based simulation. Figure 4.5 shows the impact of parallel programming on the run-time of our final implementation. We measured several different particle amounts under the same conditions as in the general run-time measurement before and composed the results of the sequential implementation to a two-threaded and a four-threaded parallelized version. Generally, the experiments show that there is definitely a performance gain caused by the parallel execution of the solver, however, we seemingly are not able to linearly decrease the run-time by increasing the number of parallel OpenMP tasks. The reason is that we are not able to parallelize each of the implementation parts introduced in Section 3.3 and that the parallelized workload is possibly not always distributed ideally among the threads. Furthermore, each thread's data accesses cause an arbitrary memory accessing behaviour of the program as a whole due to the fact that we are working with a shared memory system.

We believe that the unstructured memory accesses become even a bigger

problem for a furtherly increased number of threads. Consequently, this seems to be the main reason why the performance of the four-threaded implementation is generally inferior to the two-threaded version. The calculation of the SPH terms implemented with the presented Linked Cell-loops are obviously not optimally suitable for a satisfying parallelization as we employed. Nevertheless, the two-threaded implementation shows a significantly superior performance compared to the sequential program and is therefore chosen as the reference implementation for the examination of the simulation's real-time capability. Using the time step as listed in Table 3.1, $\Delta t = 0.01s$, our simulator needs to perform 100 SPH iterations per second to fulfill real-time requirements optimally. The maximum amount of particles we are able to simulate with our final two-threaded implementation in real-time actually depends on the applied force field equations and on the chosen boundary container. Generally, the results show a significant speed advantage for the spherical container in favor of the OBB. In fact, we are able to calculate the fluid flow of $4,100$ water particles in real-time using a spherical container and Equation (2.21) for the evaluation of the pressure force field. Under the same conditions, the employment of Equation (2.22) instead of (2.21) leads to slower results. The possible reasons for these major performance differences are discussed in the following Section 4.2.

4.2 Simulation Issues

During the development of the SPH simulation we have faced several problems regarding the stability of the method. While the water simulation using a spherical boundary object works very reliably with the parameters given in Section 3.2, the OBB container causes some major stability issues due to its edged form and the unsatisfactory particle projection already discussed in Subsection 2.5.3. Using the OBB in combination with the introduced collision handling procedure, it is not principally guaranteed to obtain a realistic and stable result for every water simulation. The reason for this lies in the complex behaviour of the particles at the edges of the OBB. There, the particles tend to congregate and possibly build up very strong repulsive pressure force fields causing the particles to be accelerated very quickly in the opposite direction. The visualization result of the fluid flow for such a scenario does not model a realistic behaviour at all. However, we are able to avoid the coalescence and the resulting high velocities of adjacent particles at the OBB's edges by decreasing the time step Δt by a factor of 10. Unfortunately, this way we heavily impair the interactivity of our solver.

Another problem we encountered during the development of the SPH implementation is the reliability of the evaluation of the mass density ρ. In the case of a single particle i flowing through the simulation domain apart from

any other fluid particle, the resulting value of particle i's mass density is obviously zero due to the fact that there are no adjacent particles inside i's support region. Consequently, the acceleration (2.16) of particle i would be undefined and the balance of the whole simulation is falsified. In order to avoid this unstable behaviour, we rather choose to employ the rest density ρ_0 instead of the mass density for the time integration scheme (2.16) in cases where $\rho = 0$.

Generally, this brings us to a discussion about incompressibility of the fluid. For our small-scale purposes, water is actually an incompressible liquid and as such should employ a constant mass density. However, as we already mentioned in Chapter 2, the SPH method has been developed for compressible fluid flows and it is therefore not built to model incompressibility explicitly. We are not allowed to fix the mass density to a constant value, e.g. the rest density, because this causes unreliable results for the evaluation of the pressure field (2.19). In fact, we would not be able to model repulsion and attraction anymore. In contrast to that, the method's lack of incompressibility does not guarantee the volume conservation of the simulated fluid convincingly. Figure 4.6 demonstrates that equal fluid volumes possibly occupy different volumes of the boundary container strongly dependent on the position of the primitive. Regarding the upright box container in **a**), the volume occupied by the fluid particles is about $\frac{1}{3}$, while the horizontally positioned container in **b**) is half-full. The cause is the dense congregation of adjacent particles in regions where a high pressure field is acting on the particles. The pressure acting on the bottom of the upright OBB is higher because there is less space to distribute the pressure than in the case of the horizontal OBB. Therefore, the particles in **a**) are enforced to compress in a stronger way than in **b**).

Obviously, the aforementioned issue concerning the reliability of the simulation results using a boxed container object are based on the lack of incompressibility as well. Hence, there is actually a close connection between compressibility and the stability of the simulation as well as real-time capabilities. In [Kel06], it is stated that *near-incompressibility* can be obtained by decreasing Δt and simultaneously increasing the gas stiffness constant k. The lack of incompressibility is also the reason for the major performance differences measured at the end of Section 4.1. We already mentioned that due to compressibility the particles can become very dense in locations with high pressure fields. The number of particles occupying the support region of an arbitrary particle at locations with strong compressibility can become significantly larger than usual. Hence, the computational complexity of the Linked Cell iterations rises as well and causes worse run-time performances. For the same reason, the choice of the pressure force field equation can become a decisive element concerning compressibility issues and therewith real-time performance.

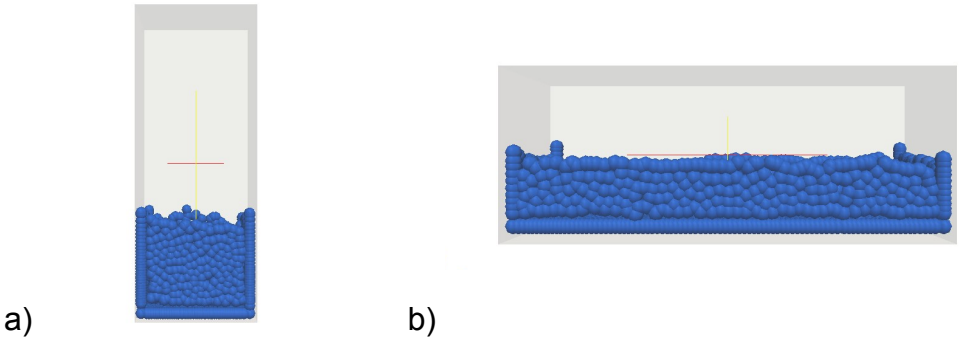

Figure 4.6: Simulation of 1,815 particles in an OBB of equal size but differently positioned.

Chapter 5

Conclusion

Throughout this thesis we have collected, described and presented the necessary elements for a realistic and efficient SPH fluid flow implementation. Based on the standard SPH equation (2.4) and the simplification of the Navier-Stokes equations (B.1), we derived the density and force fields providing convincing dynamics of the fluid particles. In combination with algorithmical optimization techniques, we managed to develop a proper implementation of the SPH method simulating up to 4,100 particles in real-time.

The smoothed particle hydrodynamics method really qualifies for a serious competitor for the complex Eulerian methods as it reduces the computational effort for solving the fluid mechanical equations significantly. Unfortunately, it does not fulfill the incompressibility assumption and is therefore not exactly adhering to the derivation of the Navier-Stokes equations for incompressible fluid flows. Adjusting the SPH method to obtain a state of near-incompressibility, we thereby have to abandon satisfactory results concerning real-time capabilities.

In conclusion, it seems to be very difficult to permanently provide stable water simulations in every case and at the same time not to lose focus on the interactivity and speed of the implementation. As we already mentioned in the introductory chapter, there is always a trade-off between complexity and accuracy of a certain CFD method. While the performance and visualization results using the OBB implicit primitive turned out to be less satisfactory, the results using the spherical boundary object have met our previous expectations. Furthermore, the application of the Linked Cell method shows an impressive impact on the method's performance measurement presented in Section 4.1. The major drawbacks of the method are the issues concerning the lack of incompressibility as well as the incoherent cache accesses. If the memory accesses of the mutual particle computations were improved, the impact of parallel programming could possibly be exploited much better than we were able to observe.

Nevertheless, we think that there is lots of space for upcoming improvements and extensions regarding the theory and the implementation of the SPH method to furtherly exploit and increase its entire performance. We think that the acquired fluid flow simulation using the SPH method attained our previously declared goals and is qualified to form the basis for any further development on this topic.

Bibliography

[Ebe03] D. Eberly. "Game Physics". Morgan Kaufmann, 2003.

[ESHD05] K. Erleben, J. Sporring, K. Henriksen and H. Dohlmann. "Physics-Based Animation". Charles River Media, 2005.

[DeCa96] M. Desbrun and M.-P. Cani. "Smoothed Particles: A new paradigm for animating highly deformable bodies". In Computer Animation and Simulation, pp. 61-76, 1996.

[GCKn03] M. Griebel, A. Caglar and S. Knapek. "Numerische Simulation in der Moleküldynamik". Springer, 2003.

[GiMo77] R. A. Gingold and J. J. Monaghan. "Smoothed particle hydrodynamics". Monthly Notices of the Royal Astronomical Society 181, pp. 375-389, 1977.

[Jos01] Nicolai M. Josuttis. "The C++ Standard Library". Addison-Wesley, 7th Printing 2001.

[Kel06] Micky Kelager. "Lagrangian fluid dynamics using Smoothed Particle Hydrodynamics". University of Copenhagen, 2006.

[Kit10] Kitware Inc. Group. "ParaView" is an open-source, multi-platform data analysis and visualization application. "http://www.paraview.org/". Nov, 2010.

[Luc77] L. B. Lucy. "A numerical approach to the testing of the fission hypothesis". Astrophysical Journal 82, pp. 1013-1024, 1977.

[MCGr03] M. Müller, D. Charypar and M. Gross. "Particle-Based Fluid Simulation for Interactive Applications". Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation, pp. 154-159, 2003.

[Mon88] J.J. Monaghan. "An introduction to SPH". Computer Physics Communications 48, pp. 89-96, 1988.

[Mon92] J.J. Monaghan. "Smoothed Particle Hydrodynamics". Annual Review of Astronomy and Astrophysics 30, pp. 543-574, 1992.

[Mon94] J.J. Monaghan. "Simulating free surface flows with SPH". Journal of Computational Physics 110, pp. 399-406, 1994.

[Omp10] OpenMP Architecture Review Board. "OpenMP Application Programming Interface 3.0", May 2008. http://www.openmp.org/mp-documents/spec30.pdf. Last access: Nov, 29, 2010.

[Pov10] Persistence of Vision Raytracer Pty. Ltd.. "Persistence of Vision Raytracer" (POV-Ray). http://www.povray.org. Last access: Nov, 28, 2010.

[StFi95] J. Stam and E. Fiume. "Depicting Fire and other Gaseous Phenomena using Diffusion Processes". Computer Graphics, 29th Annual Conference Series, pp. 129-136, 1995.

[Vest04] Marcus Vesterlund. "Simulation and rendering of a viscous fluid using Smoothed Particle Hydrodynamics". Master's thesis, Umea University, Sweden, 2004.

Appendix A

Math Reference

- Dirac delta function:

$$\delta(x) = \begin{cases} \infty & x = 0 \\ 0 & x \neq 0 \end{cases} \tag{A.1}$$

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1 \tag{A.2}$$

- Gradient vector field of a scalar function:

$$\nabla f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = f \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{pmatrix} \tag{A.3}$$

- Laplacian (Laplace operator):

$$\triangle f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \nabla^2 f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \nabla \cdot \nabla f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = f \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} \\ \frac{\partial^2 f}{\partial y^2} \\ \frac{\partial^2 f}{\partial z^2} \end{pmatrix} \tag{A.4}$$

- Signum function that returns the sign of a real number

$$sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases} \tag{A.5}$$

Appendix B

Classical CFD Equations

- The Navier-Stokes equations of an incompressible fluid describing the conservation of momentum (vector form):

$$\rho\left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v}\right) = -\nabla p + \mu\nabla^2\boldsymbol{v} + \boldsymbol{f} \tag{B.1}$$

- Continuity equation for the description of mass conservation (incompressible fluid)

$$\nabla\cdot\boldsymbol{v} = 0 \tag{B.2}$$

- Euler equations (simplification of Navier-Stokes equations neglecting viscosity and heat conduction)

$$\rho\left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v}\right) + \nabla p = 0 \tag{B.3}$$

Appendix C

Kernel Functions

The following smoothing kernels are used throughout the SPH computations.

- 6th degree polynomial kernel as the default kernel:

$$W_{poly6}(\boldsymbol{r}, h) = \frac{315}{64\pi h^9} \begin{cases} \left(h^2 - \|\boldsymbol{r}\|^2\right)^3 & 0 \leq \|\boldsymbol{r}\| \leq h \\ 0 & \|\boldsymbol{r}\| > h \end{cases} \tag{C.1}$$

$$\nabla W_{poly6}(\boldsymbol{r}, h) = -\frac{945}{32\pi h^9} \boldsymbol{r} \left(h^2 - \|\boldsymbol{r}\|^2\right)^2 \tag{C.2}$$

$$\nabla^2 W_{poly6}(\boldsymbol{r}, h) = -\frac{945}{32\pi h^9} \left(h^2 - \|\boldsymbol{r}\|^2\right) \left(3h^2 - 7\|\boldsymbol{r}\|^2\right) \tag{C.3}$$

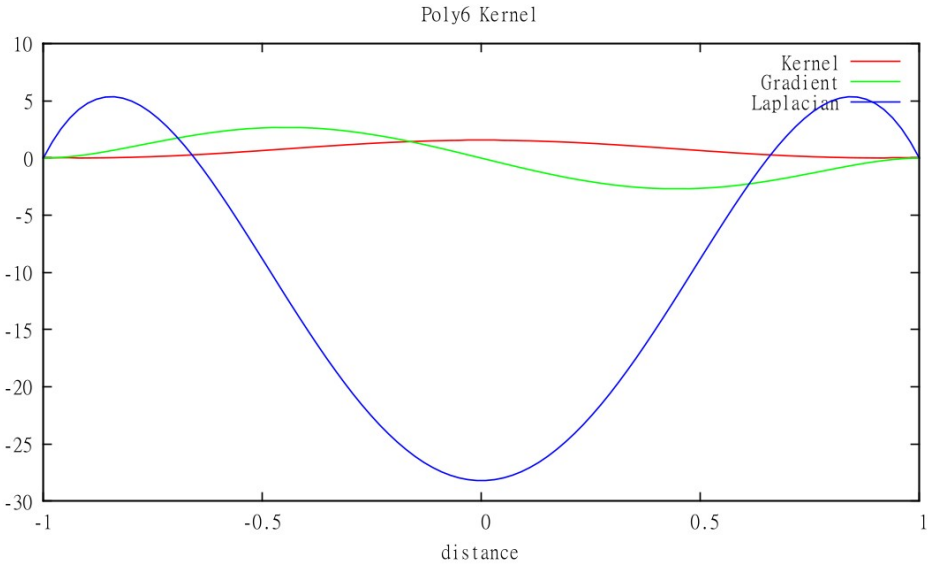

Figure C.1: The 6th degree polynomial kernel with its gradient and Laplacian in 1D, $h = 1$.

- Spiky kernel as the pressure kernel:

$$W_{spiky}(\boldsymbol{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\boldsymbol{r}\|)^3 & 0 \leq \|\boldsymbol{r}\| \leq h \\ 0 & \|\boldsymbol{r}\| > h \end{cases} \tag{C.4}$$

$$\nabla W_{spiky}(\boldsymbol{r}, h) = -\frac{45}{\pi h^6} \frac{\boldsymbol{r}}{\|\boldsymbol{r}\|} (h - \|\boldsymbol{r}\|)^2 \tag{C.5}$$

$$\nabla^2 W_{spiky}(\boldsymbol{r}, h) = -\frac{90}{\pi h^6} \frac{1}{\|\boldsymbol{r}\|} (h - \|\boldsymbol{r}\|)(h - 2\|\boldsymbol{r}\|) \tag{C.6}$$

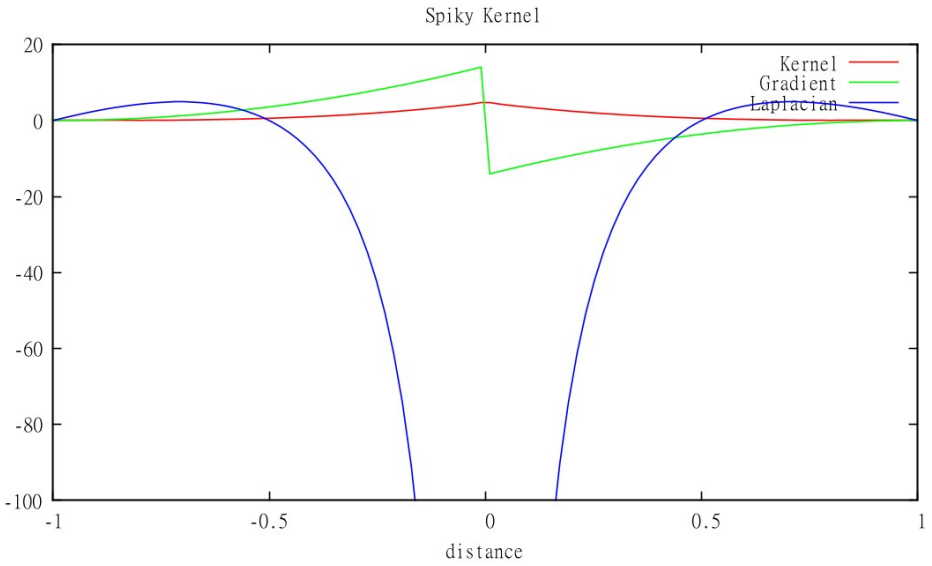

Figure C.2: The spiky kernel function with its gradient and Laplacian in 1D, $h = 1$.

- Viscosity kernel:

$$W_{viscosity}(\boldsymbol{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{\|\boldsymbol{r}\|^3}{2h^3} + \frac{\|\boldsymbol{r}\|^2}{h^2} + \frac{h}{2\|\boldsymbol{r}\|} - 1 & 0 \leq \|\boldsymbol{r}\| \leq h \\ 0 & \|\boldsymbol{r}\| > h \end{cases} \tag{C.7}$$

$$\nabla W_{viscosity}(\boldsymbol{r}, h) = \frac{15}{2\pi h^3} \boldsymbol{r} \left(-\frac{3\|\boldsymbol{r}\|}{2h^3} + \frac{2}{h^2} - \frac{h}{2\|\boldsymbol{r}\|^3} \right) \tag{C.8}$$

$$\nabla^2 W_{viscosity}(\boldsymbol{r}, h) = \frac{45}{\pi h^6} (h - \|\boldsymbol{r}\|) \tag{C.9}$$

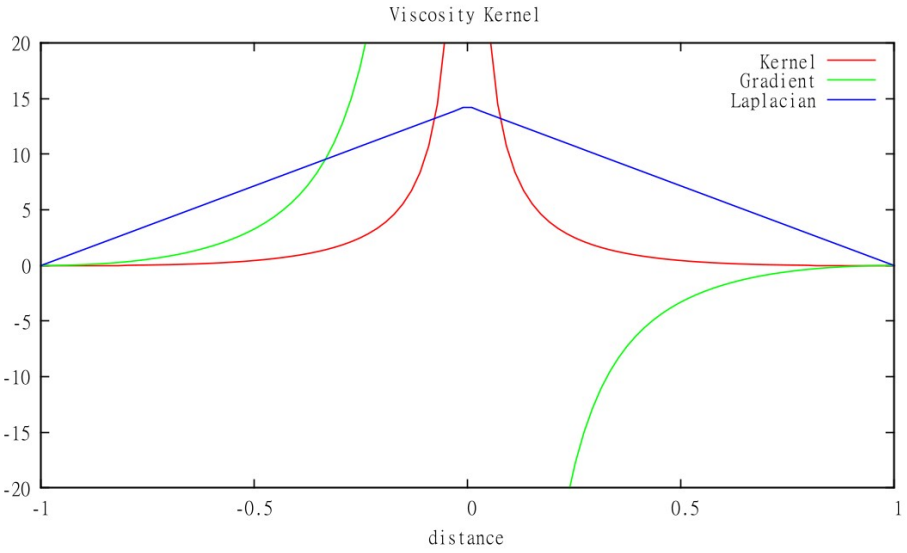

Figure C.3: The viscosity kernel function with its gradient and Laplacian in 1D, $h = 1$.