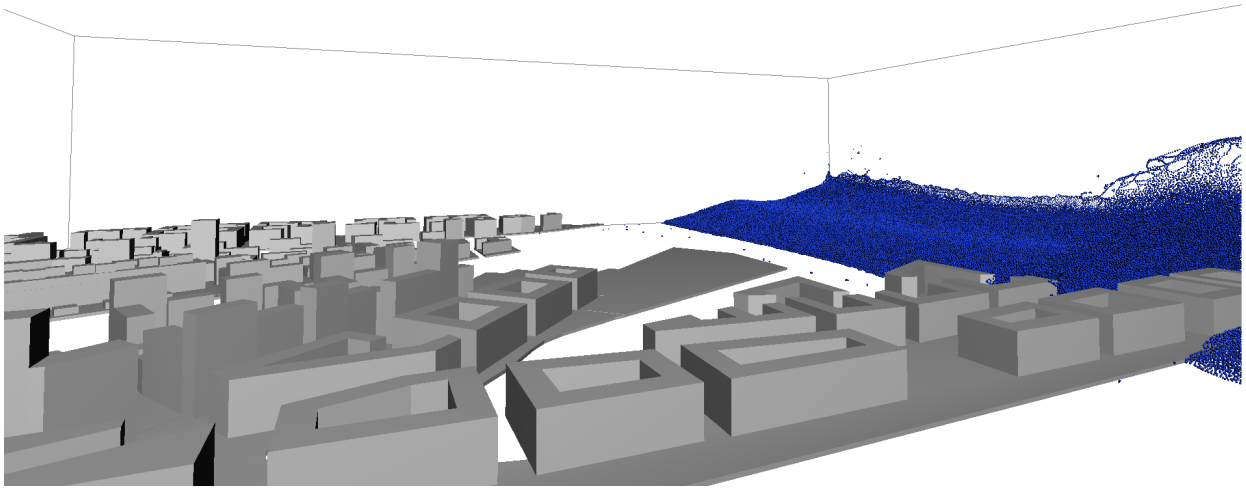

Parallel GPU-Based Fluid Animation

Master's thesis in Interaction Design and Technologies

JAKOB SVENSSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:122

Parallel GPU-Based Fluid Animation

JAKOB SVENSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Applied Information Technology
Chalmers University of Technology
Gothenburg, Sweden 2016

Parallel GPU-Based Fluid Animation
JAKOB SVENSSON

Supervisor: Marco Fratarcangeli
Examiner: Staffan Björk

Master's Thesis 2016:122
Department of Applied Information Technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Flooding a model of Frihamnen with the fluid animation using 1 million particles

Parallel GPU-Based Fluid Animation
JAKOB SVENSSON
Department of Information Technology
Chalmers University of Technology

Abstract

In fluid animation today a choice has to be made between accuracy and speed. Accuracy is chosen when animating fluids for special effects to achieve a believable animation. Speed on the other hand is chosen for interactive applications where a fast simulation is a must to keep the animation running in real-time. However, with increases in computational power in computer hardware and as more efficient simulation methods are developed a question arises. Is it possible to have both an accurate and fast fluid animation? In this thesis this will be investigated by looking at how the Jacobi method, which is used in real-time fluid simulation due to its ability to run in parallel, can be accelerated to achieve a greater accuracy in a real-time simulation. To try this an application for real-time fluid animation was developed. The computation in the simulation was run in parallel on the GPU. The results showed that the acceleration of the Jacobi method does increase the accuracy, but not enough to achieve both accurate and fast animations. A method for simulating viscoelastic fluids was also found when investigating techniques for accelerating the Jacobi method.

Acknowledgements

I want to thank my supervisor Marco Fratarcangeli for all his support during the project. Thank you for the great support and guidance during the implementation of the project as well as all the help and feedback while writing the report.

Jakob Svensson, Gothenburg, June 2016

Contents

List of Figures

List of Abbrevations

CPU	Central processing unit
GPU	Graphics processing unit
GPGPU	General-purpose computing on graphics processing units
PDB	Position Based Dynamics
SDF	Signed distance fields
SOR	Successive over-relaxation
SPH	Smoothed particle hydrodynamics
TBB	Thread Building Blocks
VFX	Visual Effects

1

Introduction

Computers have been used for simulating behaviours in the real world for almost as long as they have been around. Going back as far as to the Manhattan Project during World War 2 the process of nuclear detonation was simulated using a Monte Carlo algorithm [13]. As the computational power of computers increased over time more complex simulations are possible to run. This increase in complexity has allowed for many behaviours of the real world to be simulated. One of these is simulation of fluids.

Simulations to create 3D animations of fluids have been used for special effects in movies for almost two decades. The first movie to simulate a fluid, in this case water, using computer software was Dreamworks' Antz which was released 1998 [15]. An example of a commercial product for fluid simulation that has been used in movies for more than 10 years is Realflow [26]. One of the methods used in Realflow is called smoothed particle hydrodynamics (SPH), which is a well known method for simulating fluids through the use of particles.

While fluid simulation as special effects in movies gives realistic results, it is not interactive. An interactive simulation has the benefit of allowing the user to change parameters while the simulation is running and observe the results without waiting. In addition to changing parameters an interactive simulation allows for adding boundary conditions such as a wall to the scene and instantly see how the fluid's behaviour changes as well as how it interacts with the new object. This means that the user can try out many different scenarios very quickly in order to find the wanted scenario.

If the fluid animation is to be interactive the simulation needs to be run in real-time. This creates a problem where the simulation needs a lot of computation, but in a real-time application time is limited. There needs to be a trade-off between time and computation. In real-time games the animation is updated at least 30 times per second and often 60 times per seconds. In this later case there is only just under 17 ms available to do computations between each frame. This is not much time when tens- or hundreds of thousands particles are simulated.

However, if the particles can be kept independent the problem is possible to solve in parallel which can be done on the graphics processing unit (GPU). The parallel

task of drawing pixels on the screen has led the GPU to be designed to be good at parallel calculations [1]. There are examples of algorithms that simulates fluids with particles in parallel today, but the number of particles simulated has to be limited for the simulation to be done in real-time.

With this in mind there is still another problem to solve. The computation power of the GPU is mostly exploited for rendering 3D graphics. Usually only 2 or 3 ms for each frame are reserved for physics simulation on the GPU. This leads to the focus and goal of this thesis.

The purpose and overall objective of this thesis is to investigate novel iterative strategies for solving the constraints governing the motion of the fluid. More specifically the iterative Jacobi method will be investigated. The Jacobi method is used right now in real-time fluid animation since it is easy to parallelize making it suitable for simulation on the GPU. However, the Jacobi method is slow at converging. This thesis will therefore focus on accelerating the Jacobi method to achieve faster convergence while still solving the constraints in parallel. With this purpose in mind the research question that will be investigated in this thesis is:

Can we accelerate the Jacobi method in order to increase the number of particles simulated, without degrading the accuracy, in the short time available for physics simulation on the GPU?

With this question in mind, the goal of this project is to develop an interactive application that simulates fluids in real-time. The performance of the application and the accuracy of the simulation will be evaluated in order to answer the research question.

1.1 Problem Overview

In particle-based fluid animation the particles move in relation to their neighbors in a way that keeps the volume of the fluid the same over the course of the animation. Other properties of fluids such as surface tension also needs to be represented. In SPH these properties are represented by relationships between particles. Two problems then needs to be solved.

Find the neighbors for each particle. Solve an equation system based on the relationship between the neighbouring particles.

To make the fluid simulation interactive and in real-time these steps needs to be done at every frame update of the animation, which must update at 60 frames per second. It is therefore important that these steps are done fast enough to make sure that the simulation is not stuttering or unresponsive, while respecting the properties of the fluid.

1.1.1 Finding neighbours

A simple approach to solve the first problem is to choose one particle and compare the distance between it and all the other particles. If the distance is short enough the particles are neighbours. Then it is repeated for all particles. The problem with this approach is that as the number of particles is increased the computation time scales exponentially since all particles will have to be checked against all other particles.

A smarter approach is to divide the simulation space of the particles into a uniform grid. Then the search space for neighbours for each particle can be limited to a number of grid cells around the particle. In [14] this method is used to handle collisions between particles. In this case the 27 neighbouring grid cells (3x3x3 cells) of a particle is used to check for collisions.

1.1.2 Solve the equation system

The second problem is the main focus in the thesis. The problem consists of both setting up the constraints for all particles as well as solving the equation system of all the constraints. Solving of these equations makes sure that the particles move together as a single fluid.

In order to utilise the parallel performance available on a GPU this must be done in a way that is possible to do in parallel. A method achieving this which will be used in this thesis is the Jacobi method, which is an algorithm for solving a linear system of equations. However, solving the equations through the Jacobi method could take several iterations which is time consuming. In order to reduce the time consumption and thereby increase the performance of the simulation a method for accelerating the Jacobi method will be used. If successful the equation system will be solved faster achieving the same result with fewer Jacobi iterations. This would allow for either increasing the speed or the accuracy of the simulation.

2

Related Work

This chapter presents previous work in the area. Earlier attempts at solving the problem of particle-based fluid animation have mostly not had the purpose of running the simulation in real-time. The reason is that the computer hardware and the efficiency of the algorithms involved in the simulation did not allow for a lot of particles simulated in real-time. Therefore most papers have had the purpose of simulating fluids for offline rendering to use in for example visual effects in movies. There are still examples of real-time particle-based fluid simulations and increases in power of computer hardware and improvements of algorithms has started to allow for real-time simulations with a higher number of particles. Table 2.1 shows an overview of different methods for fluid animation from related works.

2.1 Real-time animations

In Particle-Based Fluid Simulation for Interactive Applications [25], Müller et al. introduce an interactive method to simulate fluids with free surfaces. Their method is an extension of an SPH-based method by Desbrun [11] that animates highly deformable bodies but they focus on simulating fluids instead. This is achieved by deriving the force density fields from the Navier-Stokes equation. They also added a term to model surface tension.

When rendering the fluid they show three versions. One where the particles are rendered as particles, one where point splatting is used to render a surface and in the third method an iso surface of the colour field is rendered. The first two versions was able to run in 20 frames per second when simulating water in a glass using 2200 particles. Using the third render version the frame rate dropped to 5 frames per second and can be seen in figure 2.1. Worth noting is that the paper was published in 2003 so the hardware used in the test is significantly less powerful than the typical computer hardware today.

In 2013 Macklin and Müller presented a position based fluid animation [22] based on the Position Based Dynamics framework described by Bender et al [8] and parts of the previous mentioned paper. An image of their method can be seen in figure 2.2

Method	Accuracy	Interactive	Real-time	Computation platform	Particle-based
Particle-Based Fluid Simulation for Interactive Applications	Focus on real-time over accuracy	Yes	Yes	CPU	Yes
Position Based Fluids	Focus on real-time over accuracy	Yes	Yes	GPU	Yes
Mass and momentum conservation for fluid simulation	Accurate representation of momentum conservation	No focus on direct interactivity	Yes	Unknown	No
Smoothed Particle Hydrodynamics on GPUs	Similar accuracy as other real-time methods	Yes	Yes	GPU	Yes
Realflow	High accuracy for realistic animations and renders	No	No	CPU and GPU	SPH version is, Hybrido is not
Efficient and Robust Position-Based Fluids for VFX	High accuracy with focus on robustness and surface tension effects	No	No	GPU	Yes
SPH Granular Flow with Friction and Cohesion	Accurate method for simulating granular materials	No	No	CPU	Yes
A parallel SPH implementation on multi-core CPUs	Focus was on performance rather than accuracy	No	No	CPU	Yes
Mixing Fluids and Granular Materials	Accurate interaction between sand and water	No	No	CPU	Yes
Versatile Surface Tension and Adhesion for SPH Fluids	Accurate method with more accurate surface tension than other methods	No	No	CPU	Yes
The method presented in this thesis	Focus on real-time with increased accuracy through Jacobi acceleration	Yes	Yes	GPU	Yes

Table 2.1: An overview of different methods for fluid animation

They formulate and solve a set of positional constraints that enforce constant density in the fluid which allows for similar incompressibility and convergence to modern smoothed particle hydrodynamic (SPH) solvers. The main difference from SPH however, is that it inherits the stability of the geometric position based dynamics method which allows for large time steps that are suitable for real-time applications.

Their method while running in real-time is too slow for a typical real-time scenario like a game. It needs a whole GPU for only the simulation and therefore an additional GPU needs to be used for rendering. This is acceptable in a research demonstration, but in order to use it in practice in a commercial game it needs to be able to run on a typical computer where usually only one GPU is available. Therefore the method needs to be faster so the simulation can run on the same GPU as the rendering.

The method presented in this thesis is based on the Position Based Fluids method so it is very similar. The main differences are the neighbour search which is implemented differently but still uses a uniform grid structure like in Position Based

Figure 2.1: Water poured into a glass rendered at 5 frames per second [25]

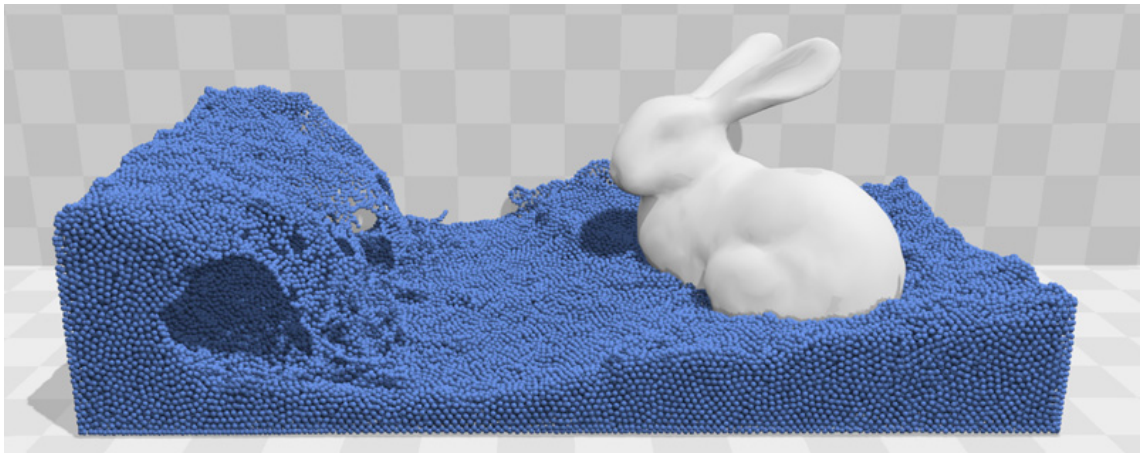

Figure 2.2: The underlying particles in the Position based fluid simulation by Macklin and Müller using 128k particles [22]

Fluids. Another difference is that a method to increase the convergence rate of the Jacobi method is tried in order to simulate more particles and to run both the simulation and the rendering on the same GPU.

Lentine et al. [21] propose a modification to the semi-Lagrangian method often used in fluid animation. The purpose of it is to make the fluid simulation conserve momentum, something that is not done otherwise in the semi-Lagrangian method. Their method is demonstrated in several demos simulating smoke in which it is compared to the standard semi-Lagrangian method. Their results show both that mass lost in the semi-Lagrangian method is still present in their method as well as artifacts that are present in the standard method does not appear in their method. In another demo the effect of mass conservation is also shown in an simulation of a splash in water where the splash is significantly larger in their method than the standard one, this is shown in figure 2.3. The method presented in this thesis uses a similar way of conserving momentum.

Harada et al. [16] presents an implementation of the SPH algorithm on GPUs. They

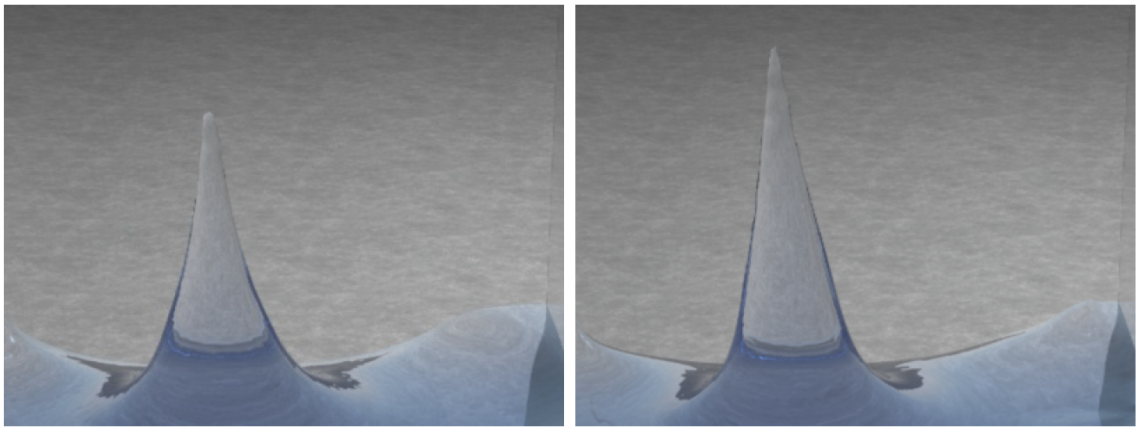

Figure 2.3: A comparison between water simulation using the traditional semi-Lagrarian method (Left) and the method by Lentine et al. (Right) [21]

achieved this by developing a method that searches for neighbouring particles on GPUs by dividing the simulation space into a three-dimensional grid. This reduces the computational cost since there is no need to search through the entire simulation space for each particle. Instead it is enough to search the grid boxes surrounding each particle since there is where the neighbouring particles can be found. To compute the rest of the SPH algorithm on GPUs they store physical values of the particles as textures on the video memory.

In their results they show real-time simulation consisting of approximately 60 000 particles running at 17 frames per second. Their method could also accelerate offline simulation. It is worth noting that the paper was published in 2007 and even though they used what was top hardware at the time, hardware today is significantly more powerful. In this thesis a similar grid is used to find the neighbours.

2.2 Offline simulations

Realflow is a product for fluid simulation that has been used in many movies. It features two main parts used for simulations of different scale [26].

The first one is a particle based fluid simulation which is based on SPH. It is used for smaller and mid range simulations and shows a very high level of detail. Since its main purpose is to be used in special effects it is not suitable to run in real-time.

The other one, called Hybrido, is used for larger scale simulations such as an ocean. It is a grid fluid solver in which the behaviour and the motion of the fluid is calculated from the interaction between neighbouring cells. In order to simulate wave splashes and foam, the grid solver is combined with particles, an example of a simulation of a large amount of water created with Hybrido is shown in figure 2.4. Similar scenarios are possible to simulate with the method presented in this thesis, even though it is

a particle-based method. However, similar to Hybrido a scene of this scale would not be able to run in real-time.

Figure 2.4: An example of a typical simulation created with Hybrido [26]

Alduán et al. [4] present a fluid simulation framework based on the previously mentioned Position based fluids work by Macklin et al. They focus on increasing the robustness of the simulation by implementing a different version of viscosity. It is designed to address VFX production demands and therefore not to be used in real-time. Instead they prioritised robustness and fluid properties that make the behaviour more detailed and realistic. The viscosity of the fluid and surface tension was improved at the cost of performance. With timesteps ranging from 600 ms to 4000 ms in the scenes tested it is far from running in real-time, but fast enough for artists to continuously review the resulting animation regularly while working.

In the paper they show how their method allow them to control the level of surface tension which can be seen in figure 2.5. The method in this thesis also includes the possibility to control the surface tension to a certain degree. However it is not as detailed and more approximated to keep the method fast. This means that if it is increased too much the simulation becomes unstable. Since performance was more important this was a needed trade off. It might be possible to implement the more detailed version of viscosity and surface tension without giving up much performance, but it is not investigated in this project.

In an earlier work by Alduán et al. [3] they presented an extension to the SPH algorithm that models the behaviour of granular materials such as sand. The purpose of their project was to simulate features such as flow which is closely related to fluids as well as more rigid body related features such as piles. Their results were successful in both areas. One demo shows an avalanche of sand flooding a city and another demo shows granular piles. There is also a demo that shows a sand castle

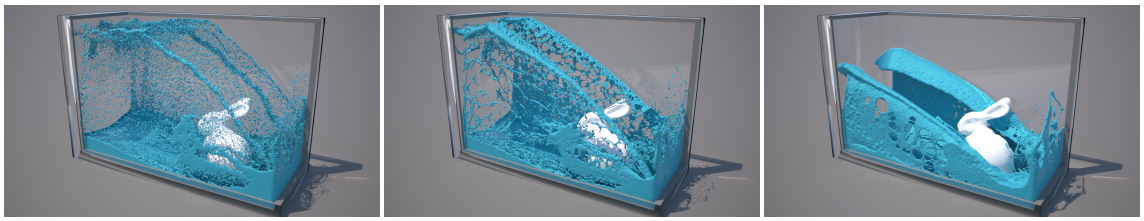

Figure 2.5: Different levels of surface tension showcased in a dam break scenario with growing surface tension from left to right [4]

stay completely intact until a couple of balls tear it down which can be seen in figure 2.6.

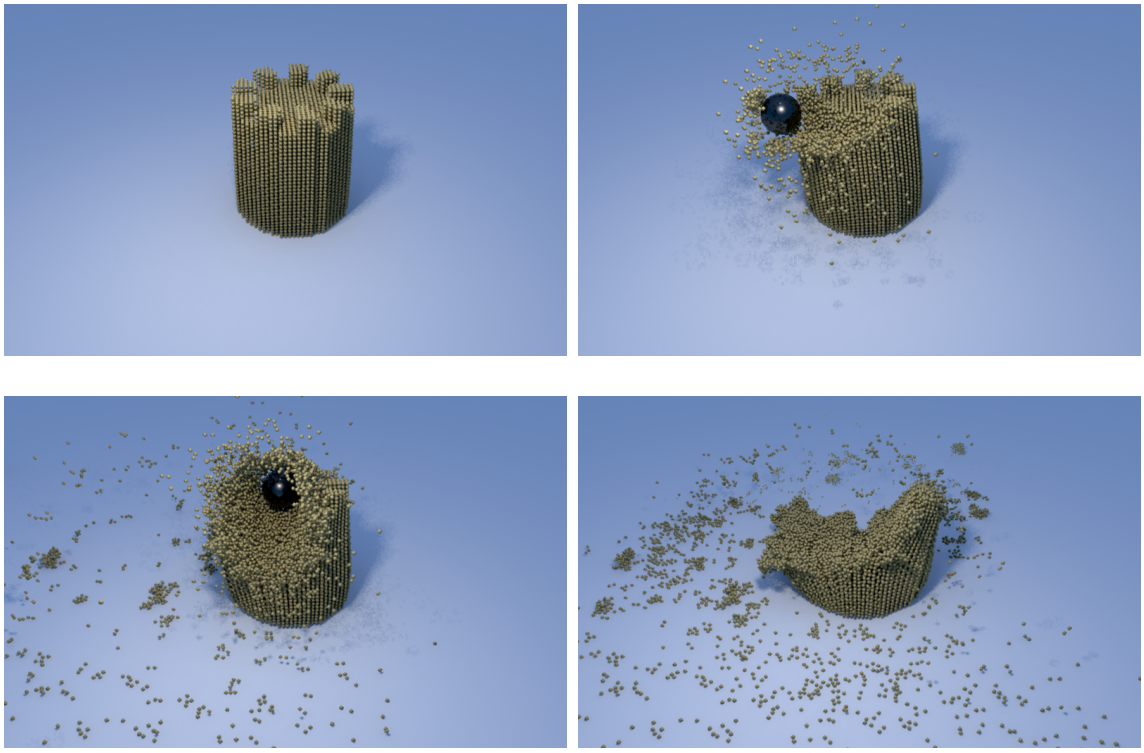

Figure 2.6: A showcase of the method by Alduán et al. keeping a sand castle intact until a ball tears it down. [3]

Ihmsen et al. [17] presents a parallel framework for simulating fluids on multi-core CPUs using the SPH method. They present and compare different ways of searching for neighbouring particles to find a way to lower the computational costs of the simulation. The methods they present and compare are, basic uniform grid, index sort, Z-index sort, spatial hashing and compact hashing. Their results shows that compact hashing and Z-index sort are the most efficient methods and that they were almost equally efficient. It was also concluded that one method is preferred over the other in different scenarios since the memory consumption of the methods scaled differently. In Z-index sort the memory consumption scaled with the domain while in compact hashing it scaled with the number of particles.

A similar method of searching for neighbours with the use of a uniform grid is used in this thesis as well, but instead the simulation runs on the GPU. The other methods presented in the paper could be interesting to try to see if they would help the neighbour search perform better. However, since the focus of the project is on accelerating the Jacobi method it is out the scope of the project.

In fluid simulation there is often interaction with other objects, both static and dynamic, but not with another fluid-like material. Lenaerts and Dutré [20] present a simulation of fine granular materials interacting with fluids. This is implemented as a unified framework based on the SPH method including simulation of both fluids and granular materials. In their demos they mix water and sand and the water interacts with the sand grains and changes their behaviour. The results demonstrates different scenarios, one being rain falling onto sand and transforming it into mud pools. In another scenario water is percolating into a rigid sand structure eroding pieces of it until it collapses completely which can be seen in figure 2.7.

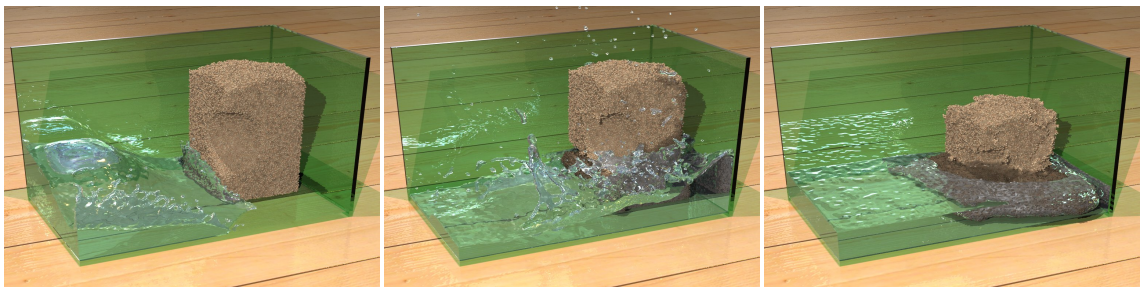

Figure 2.7: Water erodes a column of sand [20]

Akinci et al. [2] present a surface tension force and an adhesion force for SPH fluid simulation. With the help of these forces the simulation can handle large surface tensions in a realistic way. Water crown formation is an example of a behavior that previously had not been simulated realistically that is possible with these forces. This can be seen in figure 2.8. Simulation of droplets is also made possible. In their results they compare the simulation against other simulation methods in a water crown experiment. Their solution delivers a more convincing result without unnatural artifacts. The simulation for the demonstrated scenes took between 0.1 to 15 seconds per frame so it is not a real-time simulation.

If this could be implemented in the method of this thesis without a significant performance hit, which might be possible, it would increase the realism of the simulation significantly. However, it falls outside the scope of the project, but might be something for future work.

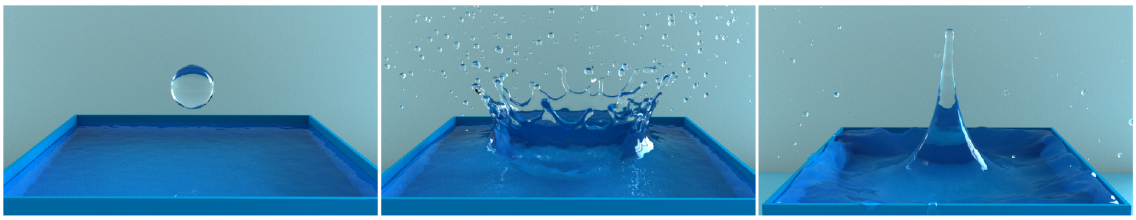

Figure 2.8: Water dropped into a filled container resulting in a realistic water crown due to the surface tension produced by the method by Akinci et al.[2]

3

Theory

In this chapter different methods and algorithms are presented that either are important for understanding the project or are going to be directly used in the implementation.

3.1 Smoothed particle hydrodynamics

Smoothed particle hydrodynamics (SPH) [24] is a well known computational method for simulating fluids. As the name of the method reveals it is based on particles that are used to represent the fluid. Each particle represents a mathematical interpolation where the fluid properties are known. More precisely the equation for any quantity A at any point r is given by the equation:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{r} - \mathbf{r}_j|, h) \tag{3.1}$$

where j represents a particle, m_j is the mass of the particle, $\mathbf{r}$ is the position, A_j is the value of any quantity A at $\mathbf{r}_j$, ρ_j is the density of the particle and W is the kernel function. The kernel function is used to weight the contributions of each particle according to the distance from the other particle. The advantage of this is that computational effort is saved by excluding the relatively small contributions of distant particles. The Gaussian function and the cubic spline are commonly used as kernel functions.

To get the density of a particle used in equation 3.1, the following equation is used:

$$\rho_i = \sum_j m_j W(|\mathbf{r} - \mathbf{r}_j|, h) \tag{3.2}$$

3.2 Parallel computing

Parallel computing is a type of computation in which many calculations are carried out simultaneously. Here two versions of parallel computing will be presented, Multi-core computing and General-purpose computing on graphics processing units (GPGPU).

A multicore CPU can issue multiple instructions each clock cycle which allows for an algorithm that is possible to run in parallel to perform up to as many times faster as there are cores on the CPU. In order to simplify the process of parallel development there are frameworks that can be used, one being TBB (Thread Building Blocks) by Intel [18]. TBB is a framework for C++ that simplifies the process of parallel development on the CPU. This is achieved by an abstraction level that removes the tedious and error prone task of working with native threads in C++.

In GPGPU on the other hand the large number of cores available on a GPU is used to do other calculations than graphics rendering. A single core of a GPU is not as powerful as a CPU core so in many cases a problem is solved faster when using the CPU. The exception is if the problem is solvable in parallel and thereby can be divided into a large number of parallel tasks. Then the GPU has a large advantage due to its architecture.

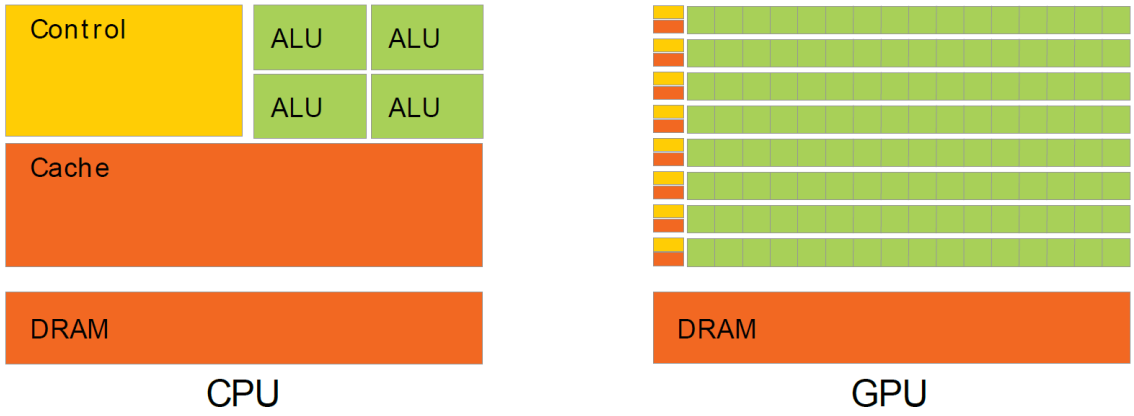

Figure 3.1: A simple overview of the architecture of the CPU and the GPU [27].

The GPU is specialised for problems which require a high arithmetic intensity, the ratio of floating point operations to data movement, that are highly parallel. This is what graphics rendering is about which the GPU has been designed for. The result of this design choice is seen in the design of the GPU compared to the design of the CPU. As figure 3.1 shows the CPU has more control units and memory for the arithmetic logic units (ALU) than the GPU [27]. This architecture allows the CPU to perform advanced computations fast due to the high resources available to each ALU. The GPU on the other hand is not as good at advanced computations but has a lot more ALUs. This gives the GPU an advantage in performance over the CPU in highly parallel problems.

3.3 CUDA and kernels

The CUDA platform by Nvidia is a tool for using CUDA-enabled GPUs for general purpose processing. As a software layer it gives direct access to the GPU's virtual instruction set and parallel computation elements which is done through execution of compute kernels.

A compute kernel or just kernel in this is a routine compiled for the GPU which is separated from the main program. In CUDA the kernels are defined as functions written in Cuda C which is an extension of C. The difference between a CUDA kernel and a C function is that a CUDA kernel is executed N times in parallel by N different CUDA threads. In order to run in parallel the threads are divided on the physical CUDA cores on the GPU.

3.4 Array of structures vs. structure of arrays

Usually when structuring data representing objects they are represented as a collection of structures. Taking particle simulation as an example, every particle will be represented as a structure of values such as velocity, position and mass describing the state of the particle. The particles are then held in a collection to allow for looping through the particles and updating their states when needed. The problem with this way of structuring data known as array of structures is that it does not align data well for vectorisation or caching which is important in parallel problems such as particle simulation.

The alternative approach is to instead use one collection for each of the values describing the objects. In the example of particle simulation this means one collection for the velocity of all particles, one for the position and one for the mass. This approach known as structure of arrays makes vectorisation easier [23].

3.5 Stationary linear iterative solvers

Stationary iterative methods are a class of solvers for linear systems like the following

$$\mathbf{Ax} = \mathbf{b} \tag{3.3}$$

where $\mathbf{A}$ is a given matrix and $\mathbf{b}$ is a vector [5]. Examples of stationary iterative methods that can be used to solve this equation include the Jacobi method and the Gauss-Seidel method as well as the SOR method.

3.5.1 Jacobi method

The Jacobi method is an algorithm in numerical algebra for solving a linear system of equations in the form of a matrix equation on a matrix that has no zeros along its main diagonal. By examining each of the equations in the linear system $\mathbf{Ax} = \mathbf{b}$ in isolation the ith equation can be expressed as the following [5]

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \tag{3.4}$$

Where a, b and x are elements of the matrix and vectors $\mathbf{A}, \mathbf{b}$ and $\mathbf{x}$, n is the length of $\mathbf{x}$ and i and j represents the current row and column in matrix $\mathbf{A}$. Then by solving for the value of x_i the following is obtained.

$$x_i = \frac{b_i + \sum_{i \neq j} a_{ij}x_j}{a_{ii}} \tag{3.5}$$

This leads to the iterative Jacobi method.

$$x_i^{(k)} = \frac{b_i + \sum_{i \neq j} a_{ij}x_j^{(k-1)}}{a_{ii}} \tag{3.6}$$

Where k is the current iteration and $k - 1$ the previous iteration.

3.5.2 Gauss-Seidel method

The Gauss-Seidel method is similar to the Jacobi method and can be used to solve the same type of equation. The equation for the Gauss-Seidel method is the following [5].

$$x_i^{(k)} = \frac{b_i + \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k-1)}}{a_{ii}} \tag{3.7}$$

The biggest difference with the Jacobi method is that in Gauss-Seidel each iteration is computed serially, because each component of the new iterate depends on all components previously computed. With each computed component used as soon as it is computed the Gauss-Seidel method is faster at converging than the Jacobi method. The downside is that the computations in an iteration cannot be computed in parallel.

3.5.3 Successive over-relaxation (SOR)

SOR is a variant of the Gauss-Seidel method with a faster convergence. With some modification the SOR method can also be used to accelerate other iterative methods such as the Jacobi method. The equation of the SOR method is very similar to the Gauss-Seidel equation and is the following:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)}\frac{\omega}{a_{ii}}b_i + \left(\sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right) \tag{3.8}$$

Where ω is called the relaxation factor and is a constant usually larger than 1. If ω is equalt to 1 the method simplifies to the Gauss-Seidel method and for $0 < \omega < 1$ it becomes a matter of under-relaxation resulting in slower convergence. Choosing the relaxation factor is not easy and it depends on the properties of the coefficient matrix. However, if the matrix is symmetric and positive definite the method is guaranteed to reach convergence for any value of ω between 0 and 2. Therefore, in order to achieve a faster convergence rate a value of ω between 1 and 2 is a good start [5].

3.6 Matrix splitting

Matrix Splitting is a method used for expressing a given matrix as a sum or a difference of matrices. An example can be seen in the following equation [30]

$$\mathbf{A} = \mathbf{M} - \mathbf{N} \tag{3.9}$$

Where $\mathbf{A}$ is a given nxn matrix and is non-singular ie., has an inverse. It is used to set up a matrix in a matrix equation to be easier and more efficiently solved through the use of iterative methods as the Jacobi method and Gauss-seidel method which both can be represented in matrix form.

The Jacobi method can be written in matrix form by splitting $\mathbf{A}$ in $\mathbf{Ax} = \mathbf{b}$ into $\mathbf{D} - \mathbf{U} - \mathbf{L}$ where $\mathbf{D}$ is the diagonal part of $\mathbf{A}$, $-\mathbf{L}$ is the strictly lower triangular part of $\mathbf{A}$ and $\mathbf{U}$ the strictly upper triangular part of $\mathbf{A}$. If $\mathbf{D}$ has an inverse the Jacobi method can be written as in equation 3.10 [9]

$$\mathbf{x}^{(k)} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k-1)} + \mathbf{D}^{-1}\mathbf{b} \tag{3.10}$$

With the same definitions of $\mathbf{D}$, $\mathbf{L}$ and $\mathbf{U}$ the Gauss-Seidel method can be expressed in matrix form as shown in equation 3.11 [9]

$$\mathbf{x}^{(k)} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{x}^{(k-1)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b} \tag{3.11}$$

3.7 Numerical integration of Newton's equations

When simulating physics on computers the physics equations needs to be described in discrete steps in order to model continuous physics equations on the computer hardware which work in discrete time steps. When working with moving objects such as particles Newton's equations of motion are often used as to model the movement and need to be integrated numerically.

A simple approach to solve this is the Euler method. In the case with a moving object affected by an acceleration the new position of the object is calculated at each time step through the following formulas:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v} \cdot \Delta t \tag{3.12}$$

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{a} \cdot \Delta t \tag{3.13}$$

Where $\mathbf{x}_{t+1}$ is the updated position, $\mathbf{x}_t$ is the current position, $\mathbf{v}$ is the velocity, $\mathbf{a}$ is the acceleration and Δt is the timestep. However this method is quite unstable especially at larger time steps [19].

Another method of integrating Newton's equations of motion is Verlet integration. In this method instead of storing the current position and the velocity, the current and the previous position is stored. The next position is then calculated through the following formula:

$$\mathbf{x}_{t+1} = 2\mathbf{x}_t - \mathbf{x}_{t-1} + \mathbf{a} \cdot \Delta t^2 \tag{3.14}$$

This means that the velocity is calculated each time step based on the previous ($\mathbf{x}_{t-1}$) and the current ($\mathbf{x}_t$) position. This method is more stable than the Euler method. It is also still a fast method and there is no extra memory needed since instead of storing the velocity and the current position the previous and the current position is stored.

3.8 Position Based Dynamics

Position Based Dynamics (PBD) is an animation method described by Bender et al [8]. It is used to simulate elastic bodies such as hair, cloth, deformable objects,

rigid body systems and fluids. Objects are represented by a set of particles and a set of positional constraints. Each constraint is assigned a stiffness parameter which defines the strength of the constraint in a range from zero to one. When the particles are moved by outer forces such as gravity or collisions their positions are set as a prediction. The predicted positions are then adjusted to make sure that all the constraints are satisfied which is achieved by solving an equation system of the constraints using the Gauss-Seidel method or the Jacobi method. The following function is an example of a positional constraint

$$C_i(\mathbf{x}_{i_1}, \mathbf{q}_{i_1}, ..., \mathbf{x}_{i_{n_j}}, \mathbf{q}_{i_{n_j}}) = 0 \tag{3.15}$$

where $\mathbf{x}$ is the position, $\mathbf{q}$ is the orientation represented as a quaternion, $i_1, ...i_{n_j}$ is a set of body indices and n_j the cardinality of the constraint.

3.9 Position Based Fluids

Position Based Fluid is an algorithm introduced by Macklin and Müller [22] and is based on the previously mentioned Position Based Dynamics method and uses aspects of smoothed particle hydrodynamics. Since it is based on the PBD method the fluid is represented by particles and constraints. Each particle has one constraint that is a function of the particle's position and the position of its neighbours. In the same way as in PBD when the particles are moved by external factors their positions are predicted. In the next step the neighbours of all the particles are found using the grid-based method explained earlier. Then the constraints can be set for all the particles using the neighbours and the constraints are solved using the Jacobi method to correct the predictions. The basic formula, without extra corrections for properties such as viscosity, used for correcting the prediction is the following

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0}\sum_j(\lambda_i + \lambda_j)\Delta W(|\mathbf{p}_i - \mathbf{p}_j|, h) \tag{3.16}$$

where λ_i is the particle's density constraint and λ_j is the density constraint from neighboring particles. λ_i is calculated using the following formula

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2} \tag{3.17}$$

3.10 Collision handling with signed distance fields

A signed distance field is defined by a signed distance function of a set in metric space, often defined as ω. The function determines the distance of a given point x from the boundary of ω. The sign of the distance determines if the point is in ω or not with a positive value for points inside ω, decreasing values towards the boundary and negative outside. The signed distance function can be defined as the following:

$$f(x) = \left\{ \begin{array}{cc} d(x, \partial\omega) & \text{if } x \in \omega \\ -d(x, \partial\omega) & \text{if } x \in \omega^c \end{array} \right\} \tag{3.18}$$

Where $\partial\omega$ denotes the boundary of ω and d the distance function [28].

An example of a simple signed distance field used for drawing text with pixels is shown in figure 3.2. In this example everything considered inside the field is painted black and everything outside white.

$\sqrt{8}$	$\sqrt{5}$	2	2	2	$\sqrt{5}$	$\sqrt{8}$
$\sqrt{5}$	$\sqrt{2}$	1	1	1	$\sqrt{2}$	$\sqrt{5}$
2	1	0	0	0	1	2
2	1	0	-1	0	1	2
2	1	0	0	0	1	2
$\sqrt{5}$	$\sqrt{2}$	1	1	1	$\sqrt{2}$	$\sqrt{5}$
$\sqrt{8}$	$\sqrt{5}$	2	2	2	$\sqrt{5}$	$\sqrt{8}$

Figure 3.2: A representation of a signed distance field for rendering text on a screen [10]

4

Method

This chapter describes how the simulation was implemented. The implementation process was carried out iteratively and consisted of seven main steps, verlet integrator, particle collision with uniform grid, neighbour search, SPH, SOR, SDF and offline rendering. These steps were chosen to fit the iterative working process since they easily build upon each other.

4.1 Algorithm overview

The algorithm of the simulation is outlined in Algorithm 1 and is an implementation of the method described in [22]. The for loops in the algorithm are in the actual implementation implemented as CUDA kernels and are therefore run in parallel. This means that each particle is updated through its own thread in a kernel.

4.2 Numerical integration

The updating of the particles position each frame was chosen to be implemented through verlet integration as opposed to Euler due to its better stability. It was implemented as its own kernel in Cuda. The kernel calculates the next position of a particle using a velocity, which is based on the previous positions of the particles, the gravity and the timestep.

In this step the rendering of the particles was also implemented. The particles positions are saved in a vertex buffer through a kernel. The buffer is then used to render points in opengl and through the use of a shader these points are made to look like spheres. This way the positions of the particles never have to be downloaded to the system memory which is slow, instead it is kept in the graphics memory.

Algorithm 1 The Simulation Algorithm

1: **for all** particles i **do**
    apply gravity $v_i \Leftarrow v_i + a\Delta t$;
    predict position $p_i^* \Leftarrow p_i + v_i\Delta t$;
2: **end for**
3: **for all** particles i **do**
    Find neighbouring particles $N_i(p_i^*)$
4: **end for**
5: **while** iter < maxIterations **do**
6: **for all** particles i **do**
     Calculate λ_i
7: **end for**
8: **for all** particles i **do**
     Calculate Δp_i
9: **end for**
10: **for all** particles i **do**
     perform collision detection and response
11: **end for**
12: **for all** particles i **do**
     $p_i^{\cdot} \Leftarrow p_i^*$
     update position $p_i^* \Leftarrow p_i^* + \Delta p_i$;
     apply SOR $p_i^* \Leftarrow \omega p_i^* + (1-\omega)p_i^{\cdot}$
13: **end for**
14: **end while**
15: **for all** particles i **do**
    Update velocity $v_i \Leftarrow \frac{p_i^* - p_i}{\Delta t}$
    apply vorticity and viscosity
    update position $p_i \Leftarrow p_i^*$
16: **end for**

4.3 Collision handling

The next step in the process consisted of implementing collision detection and response between particles through the use of a uniform grid. Even though the collision between particles is not a part needed for the fluid simulation it was an easy way to test the grid which is needed for the neighbour search. The code for the collision handling belongs to a code base written by Marco Fratarcangeli, the supervisor of this project, so this part of the implementation was mostly about integrating the code into the project. A picture of how the application looked like at this stage of development can be seen in figure 4.1.

The grid is generated to cover the world of the simulation. The size of the cells is set in such a way as only one particle fits in one cell. Then each particle is set to belong to a cell based on the particles position in the world. This is saved in an array with the length of the total number of particles in the simulation which

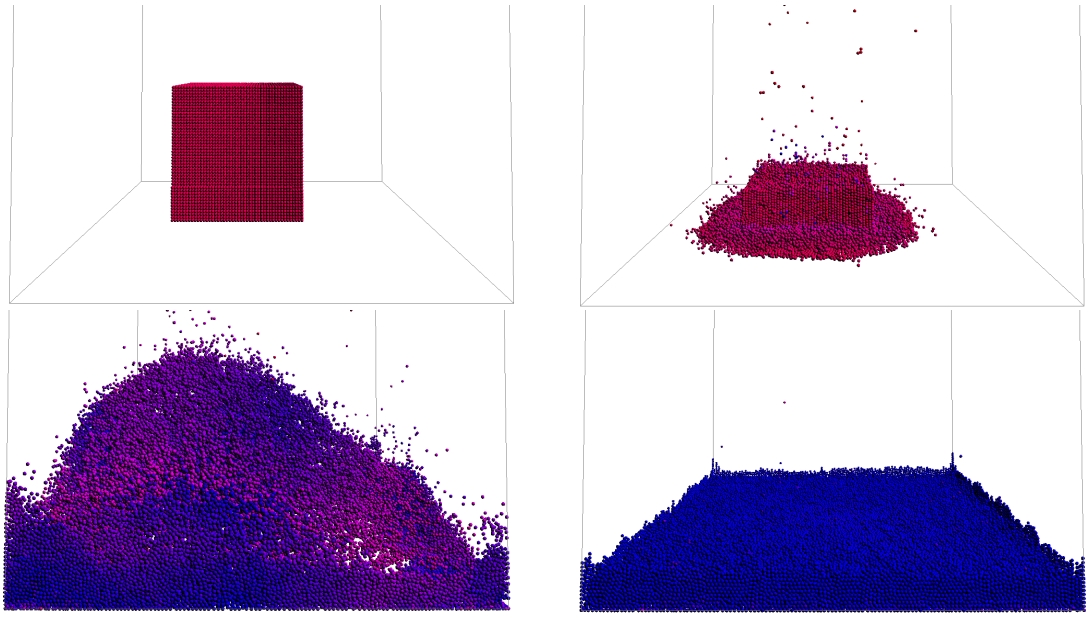

Figure 4.1: A cube of particles falling down to the ground with particle collision

holds the coordinate of the cell for each particle. The cell coordinate is saved in the format of an integer which is computed through the following formula:

$$i + j \cdot gridSize.x + k \cdot gridSize.x \cdot gridSize.y \tag{4.1}$$

Where $gridSize$ stores the number of cells in the grid in each dimension and i, j and k represents the particle's position in the grid. This makes it possible to save a three-dimensional position in a one-dimensional format. The coordinate is saved into the array in the way illustrated in figure 4.2.

Particle	Cell
0	111
1	112
2	222
3	551
4	321
5	752

Figure 4.2: An example of the first 6 indexes of the cell array, with the particle index on the left and the computed cell index on the right.

In order to detect collisions between particles the 3x3x3 area of cells with the particle's cell in the middle are checked for other particles. If any other particle is found the distance between the particles are calculated and if it is smaller than the diameter of a particle a collision is found and collision handling is applied to the particles.

4.4 Smoothed particle hydrodynamics

The implementation of the SPH was carried out in parts. First the base foundation of the SPH simulation was implemented and then an artificial pressure, vorticity and viscosity was added which increased both stability and accuracy of the simulation. A picture of the SPH simulation with vorticity and viscosity active can be seen in figure 4.3.

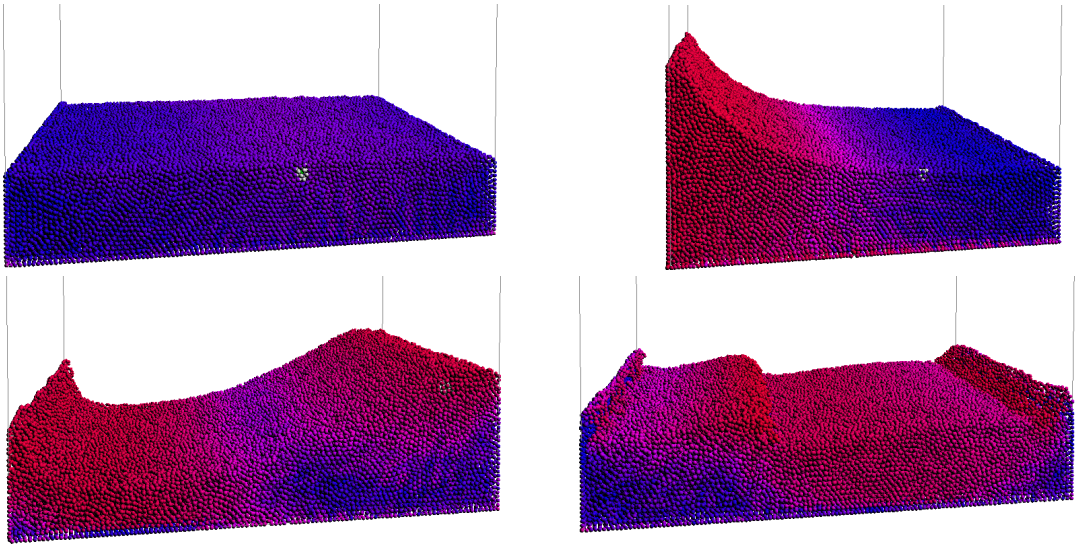

Figure 4.3: The application shown during a wave with SPH implemented with vorticity and viscosity active

The simulation consists of setting up and solving a system of positional constraint. These constraints enforces a constant density for the fluid which makes sure the particles moves together as a single fluid. In practice this is done by first calculating the density ρ_i for all particles in the following way:

$$\rho_i = \sum_j m_j W(|\mathbf{p}_i - \mathbf{p}_j|, h) \tag{4.2}$$

Where m is the mass of the particle, $\mathbf{p}_i$ is the position of particle i, $\mathbf{p}_j$ is the position of the j_{th} neighbour to particle i, and W is the poly6 kernel defined by the following formula:

$$W_{poly6} = \frac{315}{64\pi h^9} \left\{ \begin{array}{cc} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{array} \right\} \tag{4.3}$$

where r is the length of the vector $\mathbf{p}_i - \mathbf{p}_j$ and h is the smoothing length. The smoothing length is the maximum distance particles can be from eachother and still be neighbours.

The density value is then used together with the rest density to set up a constraint for each particle that should be fulfilled. The constraint is a function of the particle's position in relation to the position of its neighbours, denoted as $(\mathbf{p}_1, ..., \mathbf{p}_n)$, where n is the number of neighbours. The constraint is defined by the following equation:

$$C_i(\mathbf{p}_1, ..., \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1 \tag{4.4}$$

where ρ_0 is the rest density which is a constant set at the start of the simulation.

To solve the constraints a lambda value is calculated for each particle which is based on the constraint gradient and the constraint like the following:

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum\limits_k |\nabla_{\mathbf{p}_k} C_i|^2 + \varepsilon}, \tag{4.5}$$

where ε is a relaxation parameter that is constant over the simulation.

The gradient of the constraint function with respect to particle k has two cases depending on if particle k is a neighbouring particle (j) or the current particle (i). It is defined by the following formula:

$$\nabla_{\mathbf{p}k} C_i = \frac{1}{\rho_0} \left\{ \begin{array}{ll} \sum\limits_j \nabla W(|\mathbf{p}_i - \mathbf{p}_j|, h) & \text{if } k = i \\ \nabla W(|\mathbf{p}_i - \mathbf{p}_j|, h) & \text{if } k = j \end{array} \right\} \tag{4.6}$$

where ∇W is the gradient of the Spiky kernel which is defined by the following formula:

$$W_{spiky} = \frac{-45}{\pi h^6} \left\{ \begin{array}{cc} \frac{(h-r)^2}{r} & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{array} \right\} \tag{4.7}$$

Next the correction for the positions of the particles are computed through the following formula:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + S_{corr}) \Delta W(|\mathbf{p}_i - \mathbf{p}_j|, h) \tag{4.8}$$

The S_{corr} value is the previously mentioned artificial pressure.

In order to make the simulation both more stable and more realistic viscosity and vorticity was added. Both were added as an addition to the velocity calculated from the old position and the corrected position of the particles.

The vorticity was added to replace lost energy due to the position based simulation. It is calculated using the estimator given in [24]. The calculated vorticity is then used to calculate a correcting force which is multiplied with the timestep so it can be applied as a velocity.

The viscosity is important for the fluid to have a coherent motion. It is implemented as the XSPH viscosity introduced by Schechter and Bridson in [29]:

$$v_i^{new} = v_i + c \sum_j v_{ij} W(|\mathbf{p}_i - \mathbf{p}_j|, h) \tag{4.9}$$

where v_i is the velocity of particle i, v_{ij} is the difference of the velocities of particle i and j, $\mathbf{p}_i$ and $\mathbf{p}_j$ are the positions of particle i and j, W is the poly6 kernel showed in equation 4.7 and c is a user specified parameter.

4.5 Neighbour search

Finding the neighbours for each particle is an important step for the fluid simulation. It is similar to the collision detection and therefore the same grid implementation could be used with some modifications. However, since there is no collision between particles in the fluid simulation several particles can sometimes end up at the same position. Therefore the grid had to be modified to allow for several particles in the same cell. This meant that the implementation of how the cells each particle belongs to had to be modified. Instead of the previous array which held the cell coordinate for each particle, the array holds the particles that is inside each cell. An illustration of the new array can be seen in figure 4.4.

In order to keep track on how many particles there are in one cell the array is constructed so each cell uses several indexes to store the needed data. The first index stores the number of particles that currently are inside the cell. The next indexes stores the indexes of the particles that are inside the cell. The number of indexes reserved for each cell is determined by the cell capacity, which is a constant set to 8 in my simulation, plus one extra index to store the current number of particles in the cell. In order to store this information in a one-dimensional array the cell index is calculated like the following:

$$cellIndex \cdot (cellCapacity + 1) + count, \tag{4.10}$$

where $cellIndex$ is computed the same way as in equation 4.1, $cellCapacity$ is the previously mentioned cell capacity and $count$ is the current number of particles in the cell.

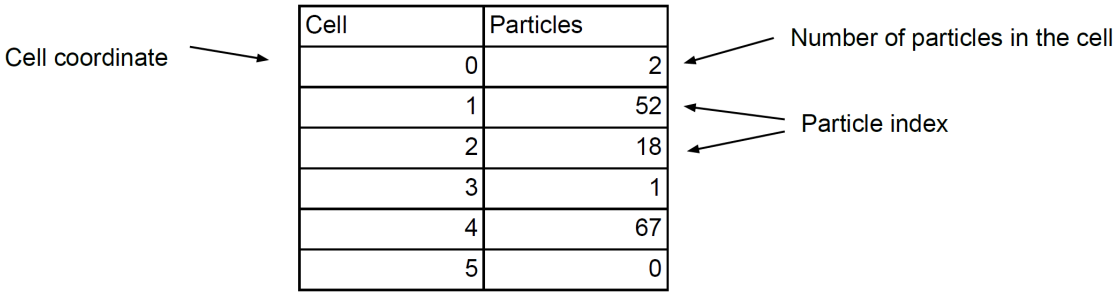

Figure 4.4: An example of the first 6 indexes of the cell array for the neighbour search, with the cell index on the left and the particle indexes on the right. The illustration is a simplified version of the array with a cell size of 2 which results in 3 indexes being reserved for each cell.

The count value is increased whenever a particle is added to the cell. However, since each kernel thread operates around one particle this could cause data to be overwritten if two particles is added to the same cell at the same time. In order to solve this a CUDA function called atomicAdd is used when the number of particles in a cell is updated. This function makes sure that only one thread edits the value at the same time. This breaks the parallelism at this part but since there usually are not that many particles in the same cell it does not have any noticeable effect on the performance.

In the neighbour search there was also a need to search a larger area for other particles than in the collision detection. Therefore the cell size was doubled which was now possible since they can hold several particles. The alternative would be to increase the number of cells to search through, but as the search distance is increased the number of cells needed to search through increases exponentially. Increasing the cell size was therefore the better option.

In order to find the neighbours for a particle the cells are searched through in the same way as in the collision detection, but for each cell there is a list of particles that are looked at. For each particle found the distance between the current particle and the particles found are calculated and if it is smaller than a threshold the particle is saved as a neighbour. The neighbours are saved by storing their particle index in a one-dimensional array with a number of indexes reserved for each particle, similar to how the cells are stored. The max number of neighbours stored for each particle is set to 60 and the index of a neighbour in the array is computed the following way:

$$particleIndex \cdot maxNeighbours + neighboursCounter, \tag{4.11}$$

where $particleIndex$ is the index of the particle, $maxNeighbours$ is the maximum number of neighbours allowed and $neighboursCounter$ is the index of the neighbour in the local list, meaning the list of neighbours for the current particle. A picture showing the result of a neighbour search for one particle is shown in figure 4.5.

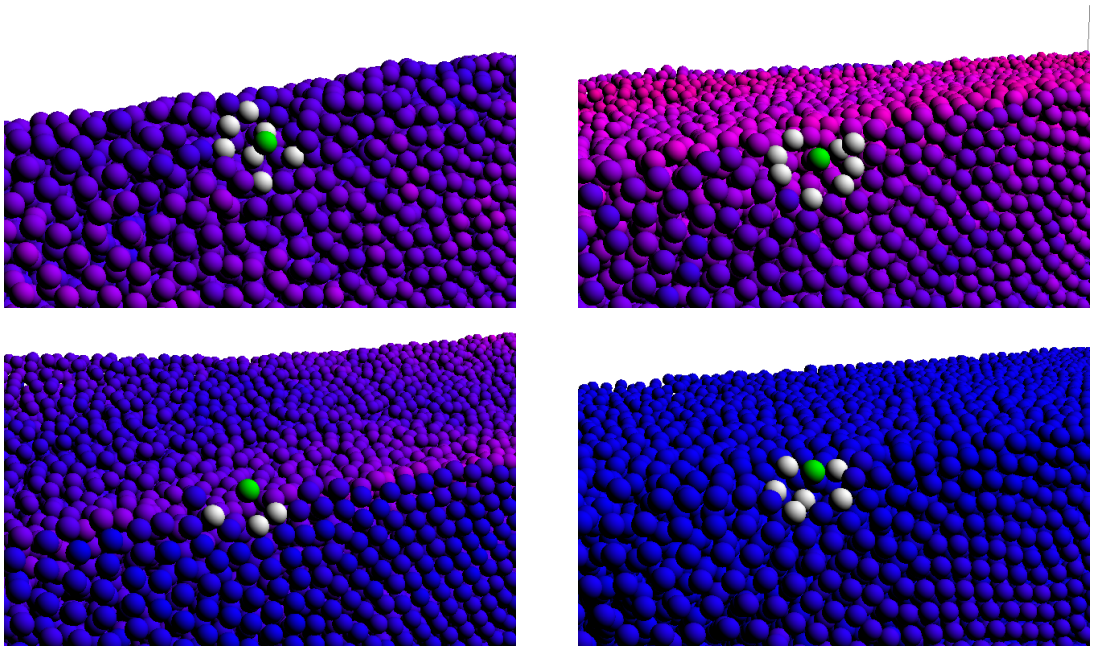

Figure 4.5: The neighbours found from the neighbour search shown during a wave, the white particles are neighbours to the green particle

4.6 Constraint solving

The constraints set up by the SPH part are solved independently with the Jacobi method. The process is outlined in algorithm 1 at row 5 to 14. The constraint is embedded in the λ value and is used to compute the Δp which is added to the position of the particle. The process is repeated a fixed number of iterations for the solution to move towards convergence. This means it will most likely not reach convergence, but hopefully get close enough. The reason behind using a fixed number of Jacobi iterations instead of running the Jacobi method until the solution converges is to have a constant performance suitable for a real-time application. Otherwise each update step of the simulation could take a different amount of time which would lead to the frame rate going up and down. Each update step would also take much more time since the number of iterations needed to reach convergence is large. The number of iterations can however be modified during run time with higher number of iterations resulting in a more accurate but slower simulation.

The final addition to the simulation was a variant of SOR for the Jacobi method. This was added with the hope of increasing the convergence of solution. SOR was

implemented using an acceleration factor denoted as ω and applied like the following formula:

$$p_i^* \Leftarrow p_i^* + \Delta p_i \tag{4.12}$$

$$p_i^* \Leftarrow \omega p_i^* + (1 - \omega) p_i^{\cdot} \tag{4.13}$$

where $p_i^{\cdot}$ is the position of particle i before Δp was added.

The acceleration factor is implemented so it is possible to change during run time in the same sense as the number of Jacobi iterations. This allows for quick testing of different combinations of values on the acceleration factor and number of iterations.

An implementation of the method described in [31] was tried using equation 4.12 followed by the following formula:

$$p_i^* \Leftarrow (1 - \omega) p_i^* + \frac{\omega}{N} \sum_j^N p_j^* \tag{4.14}$$

where N is the number of neighbours for particle i and p_j^* the position of neighbour j. Worth noting is that equation 4.12 is run for all particles before equation 4.14 so the positions of the neighbours are updated before they are used in the formula.

This version of SOR did however not have the expected result. For values of ω larger than 1 the simulation became completely unstable, but for omega values smaller than 1 something interesting happened. Then the fluid started behave like a viscoelastic fluid which can be seen in figure 5.9.

4.7 Signed distance field

To be able to create more interesting scenarios showing the true potential of the simulation collision handling against objects was implemented. This is handled with the use of signed distance fields. The code used for this in the project belongs to the code-base made by my supervisor Marco so this part of the implementation consisted mostly of integrating his code with my project.

In order to use a 3D model with the SDF to detect collision the SDF has to generated for the model. This is done by using a program called SDFGen written by Christopher Batty [6]. The SDF of the model is then loaded into the simulation and saved as a texture in the texture memory. The texture memory is cached which

means that a fetch from texture memory is faster than from the device memory if the data is present in the cache, and otherwise it is just as fast as the device memory.

When checking for collision the SDF value is fetched from the texture by inserting the position of a particle. If the returned value is less than zero, a collision is found. Then the appropriate collision response is applied by changing the position of the particle.

With the addition of collision handling against objects it was possible to create a scenario that pushed the simulation to its limit, flooding a 3D model of Frihamnen. This scenario was done with 1 million particles flowing towards and colliding against the 3D model. A picture of the scenario can be seen in fig 5.10.

4.8 Offline rendering

Due to the large number of particles in the Frihamnen scenario the simulation did not run fast enough for a real-time application any more. In order to get a smooth animation of the scenario a method of offline rendering was implemented. The implementation was done with the help of a library called FreeImage [12] which takes raw pixels from the graphics renderer and saves them as a bitmap which is saved to the hard drive as a png image. With the help of the tool FFmpeg [7] these images can then be used to create a video which shows the simulation as a smooth animation.

5

Results

To see how well the simulation performed and how accurate it was different tests were carried out. This chapter will present the results from those tests which consists of trying different values of Jacobi accelerator and number of iterations as well as a test consisting of flooding a model of Frihamnen. A link to videos of the tests can be found in Appendix A.

Using the Visual Profiler from Nvidia the time spent on the different kernels were measured. This was done by running only the particle simulation with 100k particles and with 5 Jacobi iterations in the Visual Profiler. The results can be seen in figure 5.1. The figure show the percentage of time spent on each kernel each update of the simulation. Two of the most time consuming kernels, calcLambda and solveConstraints are used for solving the constraints. The main reason for them taking so much time is that they are run 5 times each update since they are run once for each Jacobi iteration. If the number of Jacobi iterations could be lowered, by increasing the convergence without affecting the simulation, the performance of the simulation could be increased.

Figure 5.1: An overview of the percentage of time spent on each kernel in the simulation

In order to get a better understanding on how much the performance could be increased by lowering the number of iterations another test was done. The test consisted of measuring how many particles could be simulated at 60 frames per second with different number of Jacobi iterations. In order to be able to ensure a stable number of frames per second the test was carried out with the particle in a resting state after stabilising. In addition to measure the maximum number of particles the density was also measured for different values of the Jacobi accelerator. This makes it possible to see how much better the simulation becomes with an increased number of Jacobi iterations together with the cost it has on the performance. The goal for the solver is to reach the rest density which is set at 2000. The test starts with 2 Jacobi iterations due to the simulation not being stable enough to reach a resting state with only 1 Jacobi iteration. The results can be seen in figure 5.2 and 5.3.

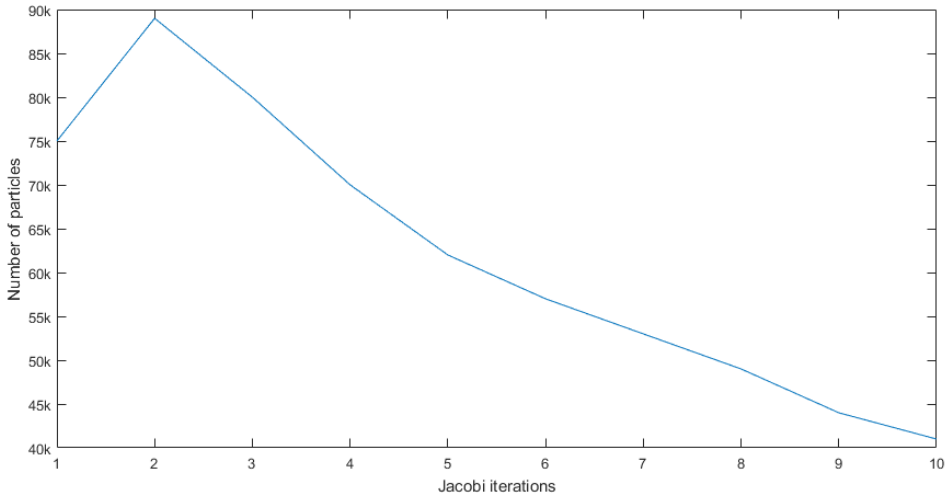

Figure 5.2: Number of particles [in thousands] animated at 60 fps varying the amount of solver´s iterations.

As can be seen in the graphs the performance drops the most from 2-5 iterations and then not as much. It is quite expected since the fewer Jacobi iterations that are used the larger performance impact it will have when the number of iterations is increased. The graph in figure 5.3 showing the density looks quite similar, as the number of iterations is increased the density gets closer to the rest density, but as in the other graph the difference between each added iteration get smaller. The density graph also show the effect of different values of the Jacobi accelerator. The same pattern is seen for all values of the accelerator, but the actual density measured gets lower as the value of the accelerator is increased. The effect of this is further investigated in the next tests.

Next the Jacobi accelerator was was investigated further to find out if it is possible to reduce the number of iterations while keeping the accuracy of the simulation. To test this SOR was used to see if different values of the Jacobi accelerator and number of iterations affected the convergence rate of the simulation. The test was

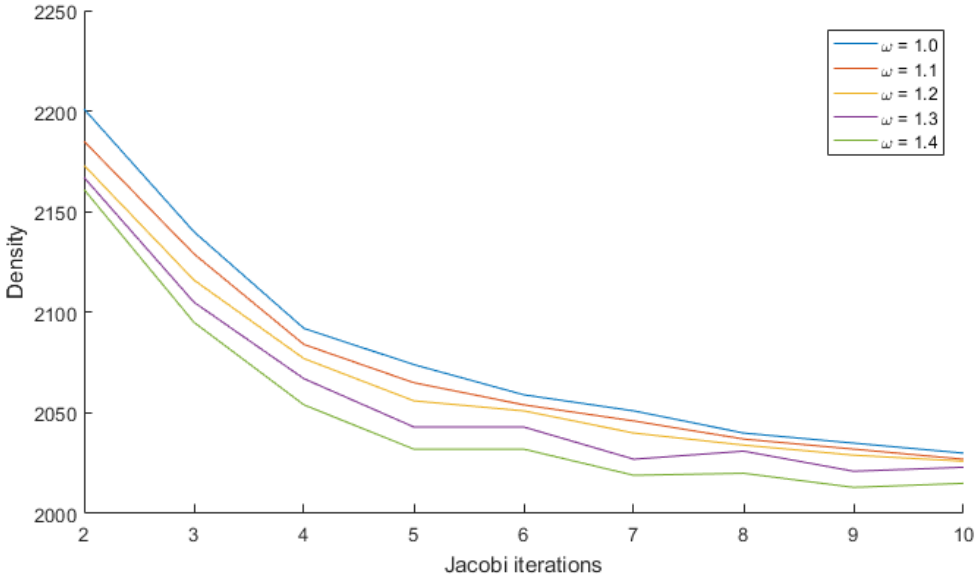

Figure 5.3: A graph of the density for different number of Jacobi iterations using the same number of particles as in figure 5.2

carried out in two ways. First the average density of all particles was measured at each 10th of a second over 10 seconds of simulation time. This was done with 3 Jacobi iterations and with the Jacobi accelerator set to values from 1 to 1.5. The results can be seen in fig 5.4. As can be seen in the graph, the density gets lower when the acceleration factor increases, but it does not converge faster.

Therefore, the density was monitored in a second way in order to see if the convergence would increase with a higher number of iterations combined with a higher acceleration factor. This time the average density over 10 seconds of simulation of the average density of all the particles was measured. The test was carried out with acceleration factors from 1 to 1.5 and with 1 to 20 Jacobi iterations. The results can be seen in figure 5.5. As can be seen in the figure the results are similar to the previous test, the density gets closer to the rest density with a higher acceleration factor and a higher number of iterations. However, it does not converge faster as the number of iterations is increased together with the acceleration factor. The same data can also be seen as a 3D surface graph in figure 5.6 for a better overview. In figure 5.7 and 5.8 the difference between an acceleration factor of 1.0 and 1.5 with 10 Jacobi iterations in the test scenario is shown. The difference is small as suggested by the graph, but it is noticeable that the particles sticks together better with the higher acceleration factor.

Next the convergence rate when using the Jacobi acceleration method outlined in equation 4.14 were to be tested. However as mentioned before this version did not result in a stable simulation, except when $\omega < 1$. Then the particles stuck together much more resulting in the fluid behaving more like a viscoelastic fluid. Therefore it was not possible to compare the convergence rate of this method with the other acceleration method. Images showing how the fluid behaved with an ω less than 1

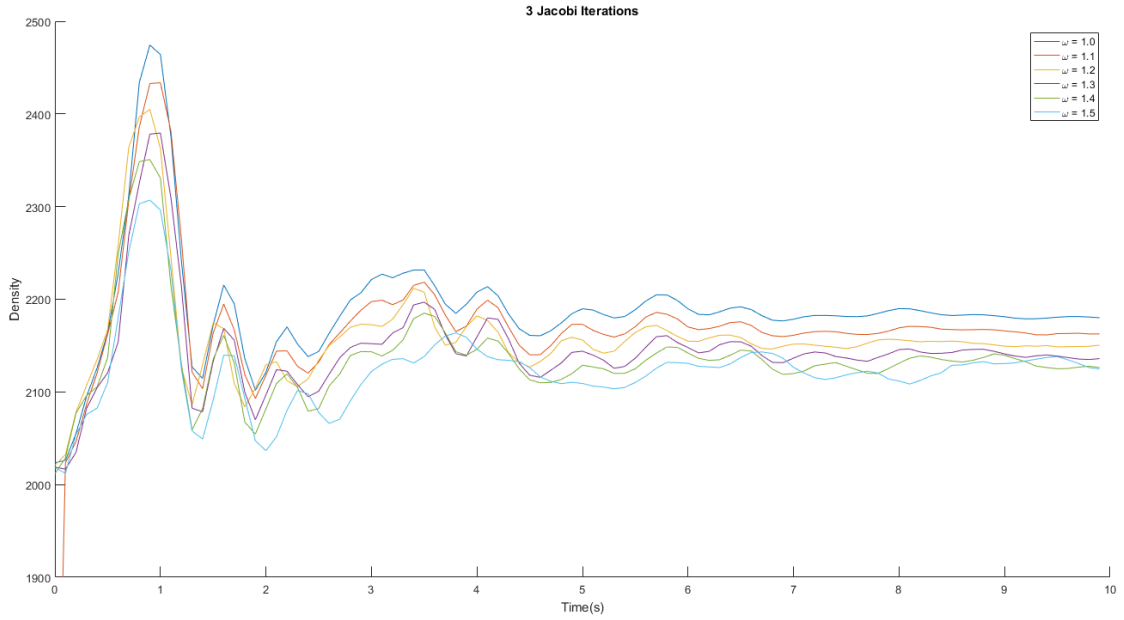

Figure 5.4: A graph of the density over time with 3 Jacobi iterations and an acceleration factor of 1.0 to 1.5

can be seen in figure 5.9.

The SDF collision handling was tested through the scenario of flooding Frihamnen. In order to make the scenario more believable 1000000 particles were simulated to create a large amount of water with small particles to achieve a high resolution. This caused the simulation to run much slower than a real-time application should, at around 2 fps which is understandable with the huge amount of particles simulated. In figure 5.10 the flooding scenario can be seen over four frames.

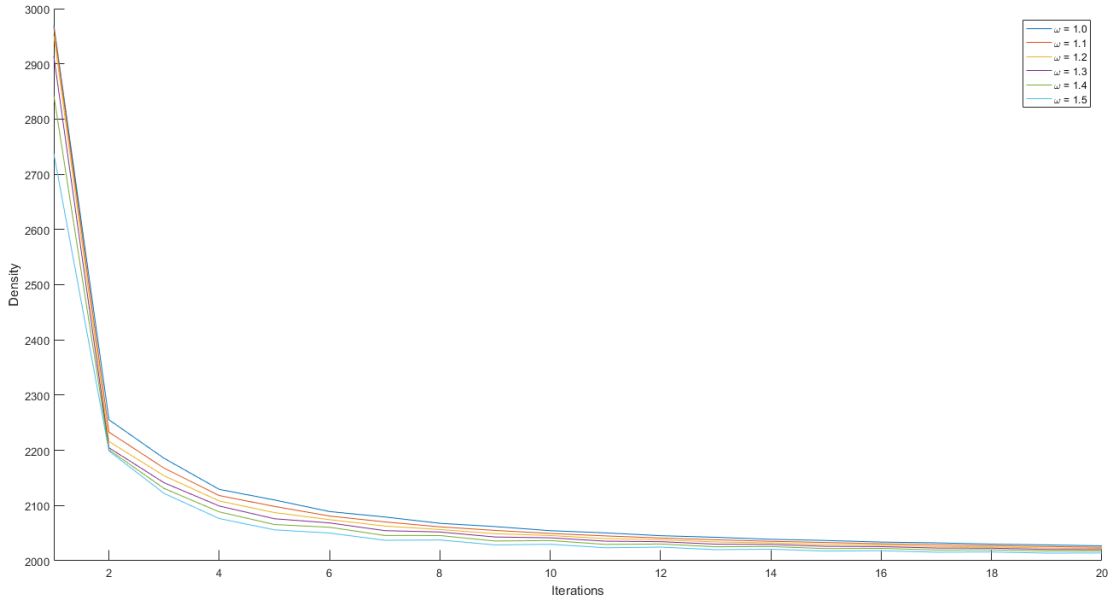

Figure 5.5: A graph of the density over time with 1 to 20 Jacobi iterations and an acceleration factor of 1.0 to 1.5

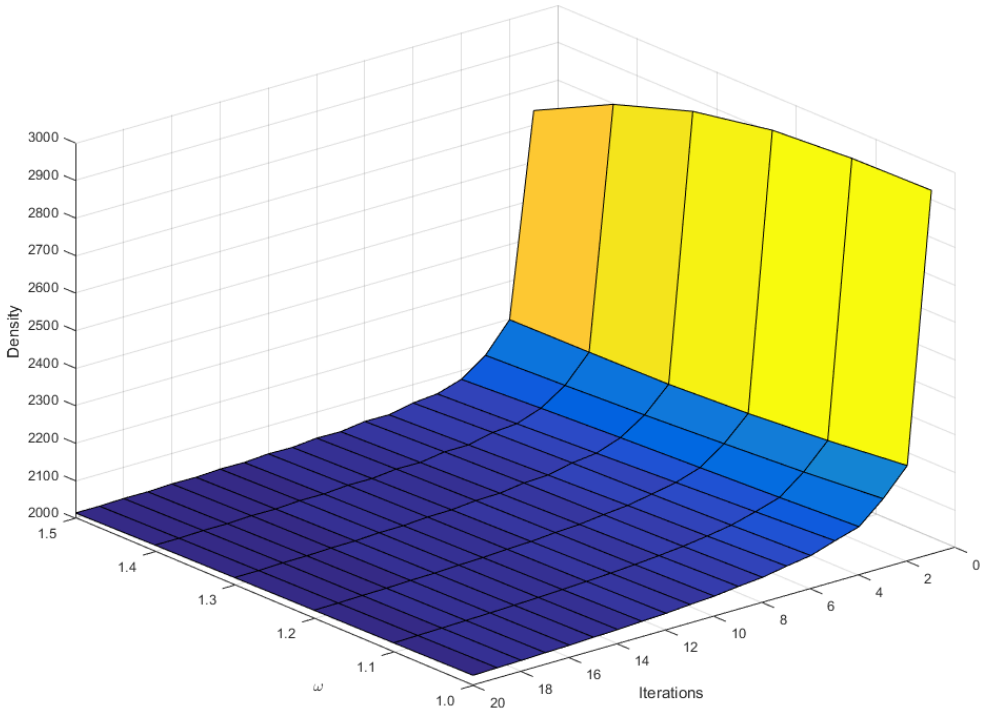

Figure 5.6: A 3D graph of the density over time with 1 to 20 Jacobi iterations and an acceleration factor of 1.0 to 1.5

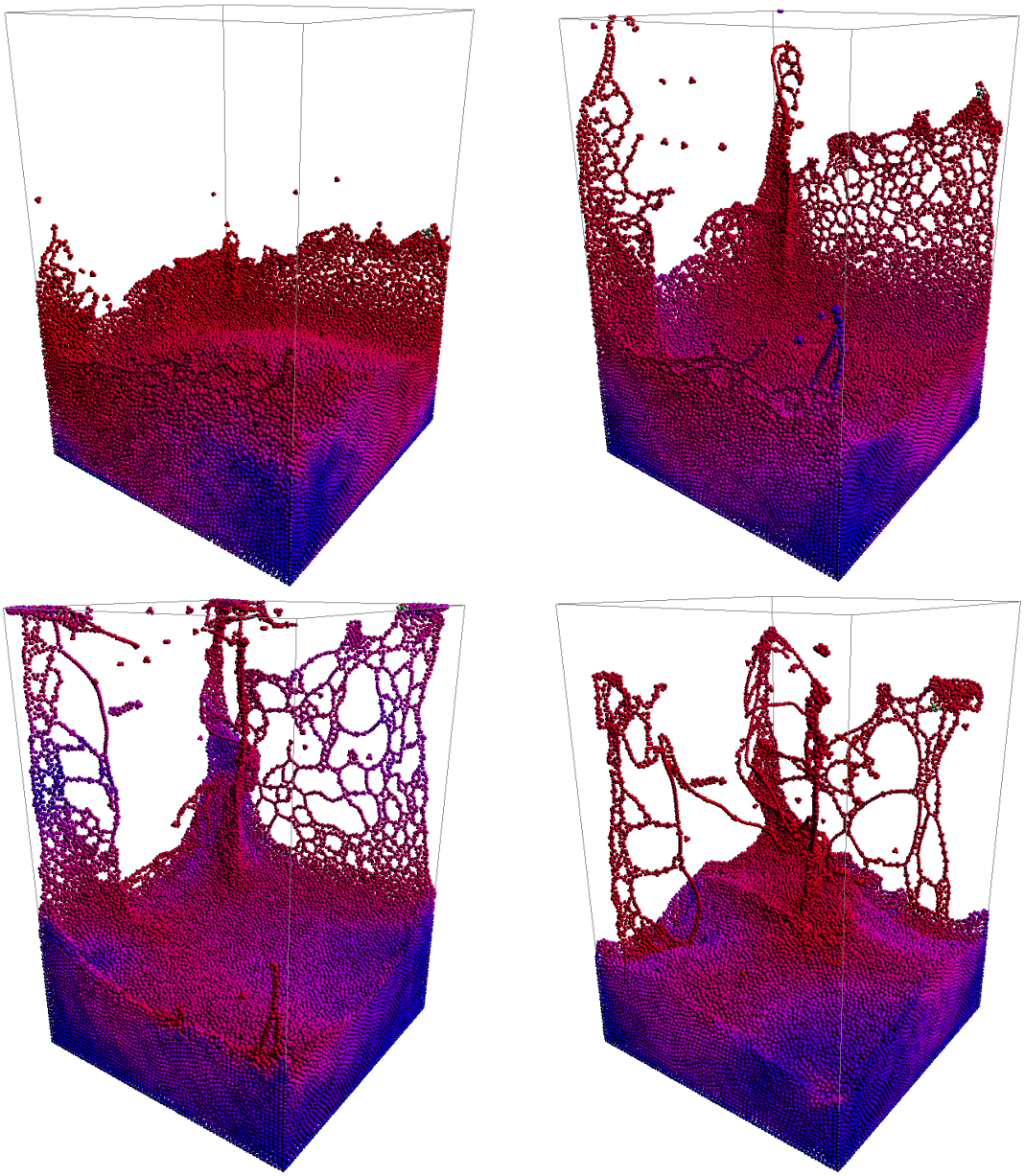

Figure 5.7: The splash in the test scenario with 10 Jacobi iterations and an acceleration factor set to 1.0

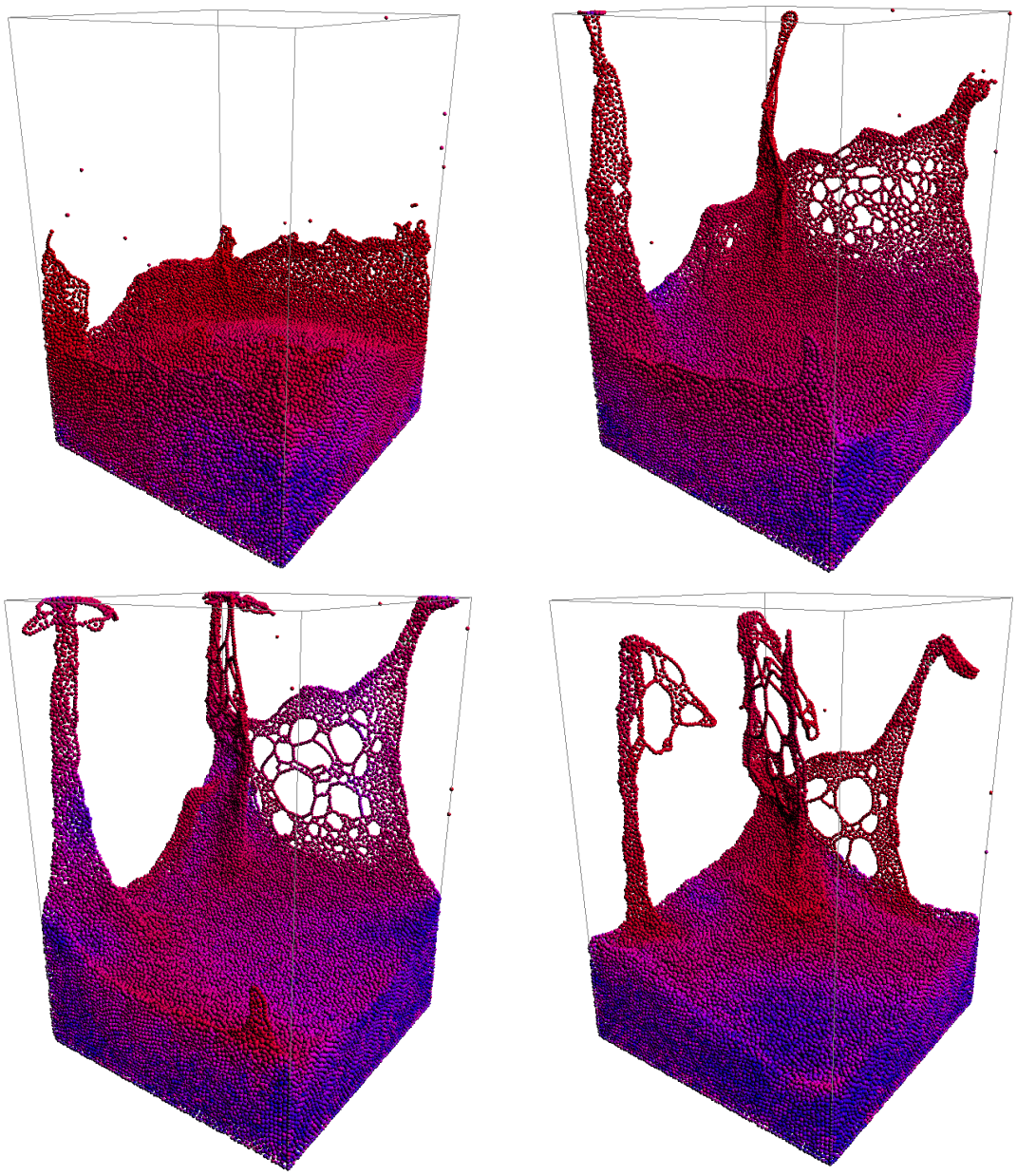

Figure 5.8: The splash in the test scenario with 10 Jacobi iterations and an acceleration factor set to 1.5

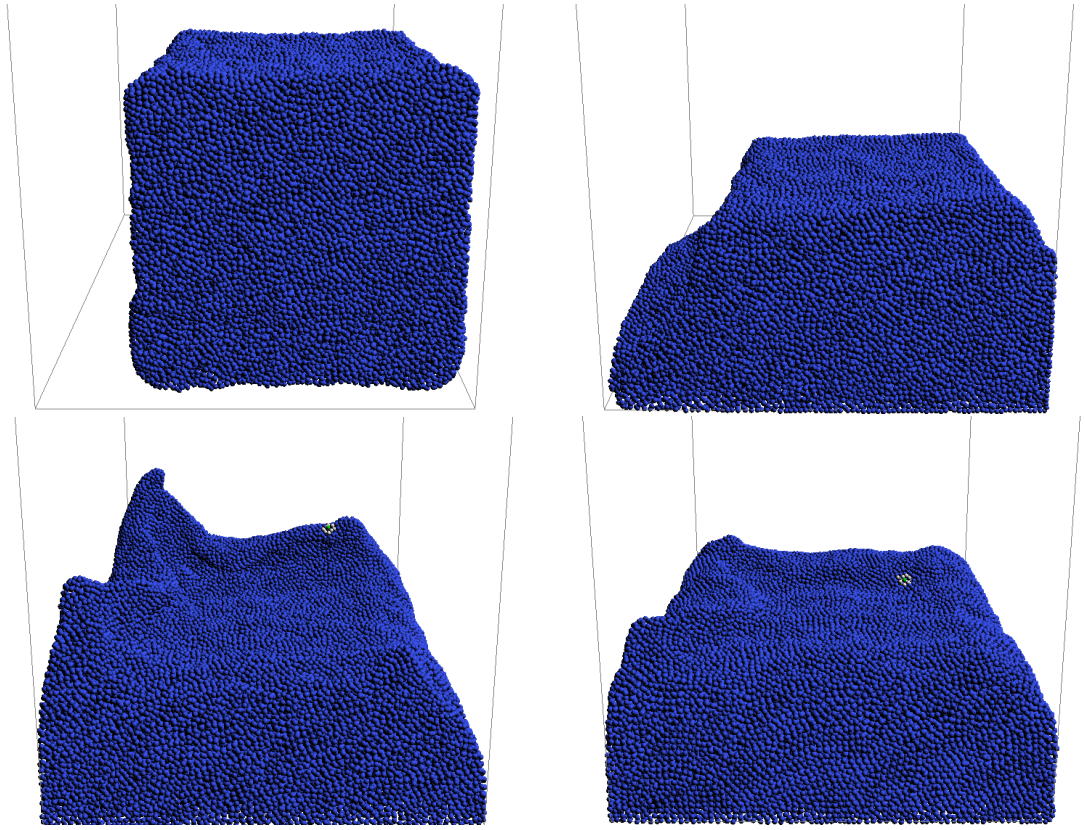

Figure 5.9: Four frames from the simulation with 5 Jacobi iterations, and an acceleration factor of 0.4 with the acceleration outlined in equation 4.14

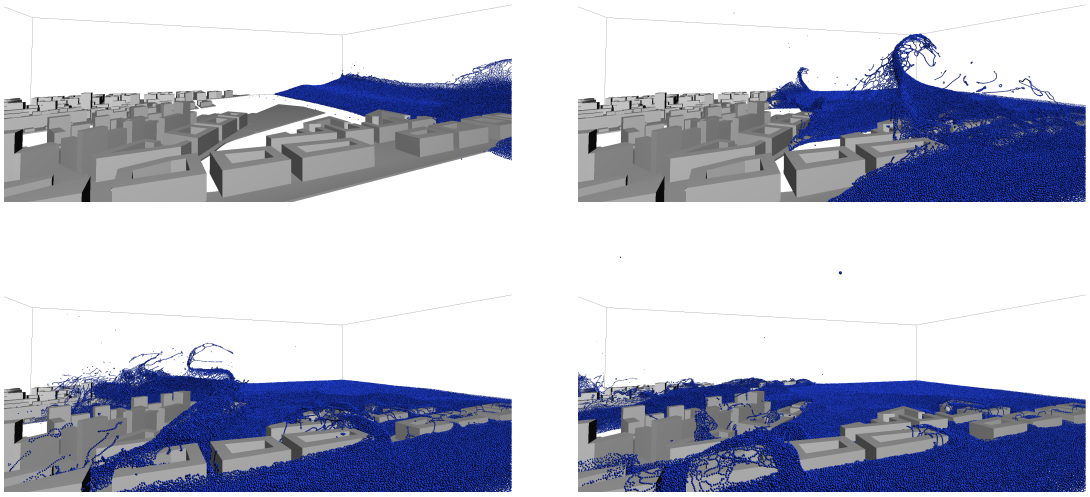

Figure 5.10: Four frames from the simulation flooding a model of Frihamnen with sdf collision handling, using 5 Jacobi iterations, and an acceleration factor of 1.2

6

Discussion

This chapter will take a look at the results and how well they fulfilled the goal of the project. Future work will also be brought up with possible improvements to the simulation or other areas of interest to investigate.

6.1 Result discussion

The results from the Jacobi acceleration did not reach the expected levels of improvement. The goal was to greatly increase the convergence rate with a higher acceleration value in order to either increase performance or accuracy or both. Instead the results show only a minor improvement in density of the particles which only had a small effect on the simulation by slightly improving the particles ability to stick together.

It is hard to say why it did not give the increase in convergence rate as was expected. One reason could be the complexity of the simulation due to the huge number of particles that keeps moving around all the time. Which could mean that moving a particle a bit more towards the suggested position to fulfill a constraint might have an almost as high negative effect on another constraint as it had a positive effect on the first constraint.

The other version of the Jacobi acceleration is easier to figure out why it did not work. Taking a look at equation 4.14, if ω is set to 1 the only thing left is the average positions of the neighbours since the term with the particles own position disappears. This is probably the reason why the simulation was completely unstable for $\omega = 1$ since the position of the particle is not taken into account at all. For $\omega > 1$ the position of the particle is starting to be taken into account, but multiplied with a negative factor. Perhaps this version of Jacobi acceleration is not suitable for this type of simulation or it was not translated correctly.

However, the result when using $\omega < 1$ is interesting and could be useful. The behaviour similar to a viscoelastic fluid is understandable when looking at the equation. With $\omega < 1$ the position of the particle is starting to be taken into account again while the neighbours effect starts to decrease, and when $\omega = 0.5$ the position

of the particle and the average position of its neighbours has equal effect on the next position. This balance between the two parts in the equation makes the particle go towards the suggested position to fulfill the constraint while at the same time move towards its neighbours. This is probably what causes the particles to stay together much more than in the normal fluid simulation resulting in the new behavior.

6.1.1 Real world scenarios

In one of the demos the water floods a model of Frihamnen which raises the question if the simulation model is realistic enough to simulate a real world event in an accurate way. The first thing that comes to mind while trying to answer this question is what scale the simulation represents. The simulation being based on particles is only an approximation of how a fluid behaves. In the real world it is much more complicated as well as much more detailed. If the particles are considered to represent small water droplets then perhaps it would be possible to determine a scale for the simulation. In that case, the scenario of flooding Frihamnen would require a huge increase in the number of particles which would take a huge amount of computing power with no guarantee that it would be close enough to reality to be useful for real world simulation.

Perhaps this type of simulation is more suitable for visual effects and interactive applications than for real world simulation. In these areas it is not very important that the simulation behaves exactly as a fluid does in the real world. What matters is that is looks and feels believable enough to be recognized as a fluid without breaking the immersion.

6.2 Future work

The viscoelastic fluid behaviour was an interesting discovery in the project. It was found as a coincidence when trying to achieve a faster convergence of the Jacobi method. Therefore, not much time have been put into testing and figuring out what it could be used for. With some more research on the topic and experimenting with the equation it could perhaps result in a good way of simulating viscoelastic fluids.

There are also other types of fluids that could be represented by the application with some more work. The fact that the simulation is about fluids and not just liquids opens up for possibilities of generalization of the simulation. With some changes in parameters and some other additions it would be possible to simulate other fluids like gases. It is also possible to simulate granular materials as for example sand using SPH which was brought up in the related works chapter. This could be done as future work, but it has mostly been done already so it may not be that interesting.

The methods tried in this project did not achieve the increased convergence rate that

was hoped for. However there are other options that could be tried in future work. One way would be to use the Gauss-Seidel method instead of the Jacobi method which would increase the convergence rate since the Gauss-Seidel method is much faster at converging than the Jacobi method. However, in order to do this and still use the GPU as the computation platform, the Gauss-Seidel method would have to be parallelised. This would be possible to do since each particle only depends on a small number of neighbours and therefore all particles that do not depend on each other could be updated in parallel. The problem would be to figure out which particles that are independent from each other.

There are other ways to increase the performance of the simulation. One way could be to look at how well the algorithm utilizes the GPU. This has been done during the project and the factor that holds back the simulation the most is memory dependency. The problem is that each particle needs access to neighbours and cells from the grid. These are not necessarily aligned well in memory which results in time spent waiting for memory transfers. A way to solve this could be to rethink some parts of the implementation in order to align the memory better between threads and thereby be able to increase the performance by sharing memory between threads.

7

Conclusion

This project resulted in a position-based fluid simulation with a solver based on the Jacobi method. It was used to answer the following question:

Can we accelerate the Jacobi method in order to increase the number of particles simulated, without degrading the accuracy, in the short time available for physics simulation on the GPU?

Accelerating the Jacobi method was tried through two different ways of using SOR with the Jacobi method. The result for the first method was a small increase in accuracy, but not large enough to achieve a accurate and fast simulation. The other method did instead not work at all when using over-relaxation resulting in a completely unstable simulation. However, when under-relaxation was used it ended up being a way to simulate viscoelastic fluids.

The simulation was also used to simulate flooding of Frihamnen with 1000000 particles. This showed that the simulation can be used for large scale simulations if it does not have to be real-time. It is however hard to say if it could actually represent a real world scenario of that scale and perhaps the simulation is more suited for simulation with entertainment purposes.

Bibliography

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, 2008. ISBN: 9781439865293.

[2] N. Akinci, G. Akinci, and M. Teschner. "Versatile surface tension and adhesion for SPH fluids". In: *ACM Transactions on Graphics (TOG)* 32.6 (2013), p. 182.

[3] I. Alduán and M. A. Otaduy. "SPH granular flow with friction and cohesion". In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics symposium on computer animation*. ACM. 2011, pp. 25–32.

[4] I. Alduán, A. Tena, and M. A. Otaduy. "Efficient and Robust Position-Based Fluids for VFX". In: *CEIG - Spanish Computer Graphics Conference* (2015), pp. 1–9.

[5] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[6] C. Batty. *SDFGen*. `https://github.com/christopherbatty/SDFGen`. 2015.

[7] F. Bellard. *FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video*. `https://ffmpeg.org/`. 2016.

[8] J. Bender, M. Müller, and M. Macklin. "Position-based simulation methods in computer graphics". In: *EUROGRAPHICS Tutorial Notes* (2015).

[9] R. L. Burden and J. D. Faires. *Numerical Analysis*. Brooks Cole, 2010. ISBN: 0538733519.

[10] I. Crawford. *Text Rendering using a Signed Distance Field in OpenGL*. `http://chikin.net/pages/blog/signed-distance-field-text-rendering.html`. 2015.

[11] M. Desbrun and M.-P. Gascuel. *Smoothed particles: A new paradigm for animating highly deformable bodies*. Springer, 1996.

[12] H. Drolon. *The Freeimage Project*. `http://freeimage.sourceforge.net/intro.html`. 2015.

[13] A. H. Foundation. *Computing and the Manhattan Project*. `http://www.atomicheritage.org/history/computing-and-manhattan-project`. Accessed: 2016-02-21.

[14] S. Green. "Cuda particles". In: *nVidia Whitepaper* 2.3.2 (2008), p. 1.

[15] Guinness World Records. *First film with digital water*. `http://www.guinnessworldrecords.com/world-records/first-film-with-digital-water`. Accessed: 2016-02-25.

[16] T. Harada, S. Koshizuka, and Y. Kawaguchi. “Smoothed particle hydrodynamics on GPUs”. In: *Computer Graphics International*. SBC Petropolis. 2007, pp. 63–70.

[17] M. Ihmsen et al. “A Parallel SPH Implementation on Multi-Core CPUs”. In: *Computer Graphics Forum*. Vol. 30. 1. Wiley Online Library. 2011, pp. 99–112.

[18] Intel Corporation. *General Questions about TBB*. `https://www.threadingbuildingblocks.org/faq`. Accessed: 2016-02-21.

[19] T. Jakobsen. “Advanced character physics”. In: *Game Developers Conference*. 2001, pp. 383–401.

[20] T. Lenaerts and P. Dutré. “Mixing fluids and granular materials”. In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 213–218.

[21] M. Lentine, M. Aanjaneya, and R. Fedkiw. “Mass and momentum conservation for fluid simulation”. In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM. 2011, pp. 91–100.

[22] M. Macklin and M. Müller. “Position Based Fluids”. In: *ACM Trans. Graph.* 32.4 (July 2013), 104:1–104:12. ISSN: 0730-0301.

[23] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439, 9780124159938.

[24] J. J. Monaghan. “Smoothed particle hydrodynamics”. In: *Annual review of astronomy and astrophysics* 30 (1992), pp. 543–574.

[25] M. Müller, D. Charypar, and M. Gross. “Particle-based fluid simulation for interactive applications”. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2003, pp. 154–159.

[26] Next Limit Technologies. *RealFlow*. `http://www.realflow.com`. Accessed: 2016-02-21.

[27] NVIDIA Corporation. *CUDA C Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`. Accessed: 2016-03-03.

[28] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Applied Mathematical Sciences. Springer New York, 2006. ISBN: 9780387227467.

[29] H. Schechter and R. Bridson. “Ghost SPH for Animating Water”. In: *ACM Trans. Graph.* 31.4 (July 2012), 61:1–61:8. ISSN: 0730-0301.

[30] Z. I. Woźnicki. “Matrix splitting principles”. In: *International Journal of Mathematics and Mathematical Sciences* 28.5 (2001), pp. 251–284.

[31] X. I. A. Yang and R. Mittal. “Acceleration of the Jacobi Iterative Method by Factors Exceeding 100 Using Scheduled Relaxation”. In: *J. Comput. Phys.* 274 (Oct. 2014), pp. 695–708. ISSN: 0021-9991.

A

Appendix 1

A playlist showing videos of the fluid animation can be found on the following link:
`https://www.youtube.com/playlist?list=PL128kqvaz4PeAKsBqt9JBvUvHvmVcqB8f`