TDT4590
Complex Computer Systems,
Specialization Project

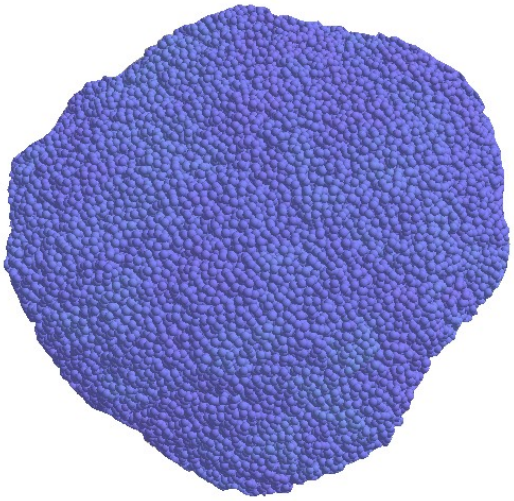
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Fredrik Fossum

Surface Tension in Smoothed Particle Hydrodynamics on the GPU

Supervisor: Dr. Anne C. Elster

Trondheim, Norway, December 21, 2010



NTNU
Norwegian University of
Science and Technology

Contents

List of Figures

Abstract

A fluid simulation using Smoothed Particle Hydrodynamics on CUDA GPUs was developed by a former Master student. However, this simulation exhibits less than realistic results when simulating fluids like water. It does not, for example, capture their pronounced splashing behaviour.

In this project this simulation is extended to improve upon this behaviour. A surface tension force is implemented into the simulation. This force was chosen for how it largely affects the behaviour of water. Efforts are made in order to improve both the performance and behaviour of this force.

We show how this extension greatly affects the behaviour of the fluid by causing drop formations, and enhancing splash effects. While suffering a performance hit, the simulation still retains reasonable performance, reaching around 70 % of the original simulation.

Surface detection used during the surface tension calculation could also be used if trying to extract a mesh surface from the particle simulation, a possible future work.

Acknowledgements

I would like to thank Dr. Anne C. Elster for her advice, and Øystein E. Krog for his assistance.

Trondheim, Dec 2010

Fredrik Fossum

Chapter 1

Introduction

Simulating realistic fluids is typically a computationally costly operation. Traditionally it has not been something that one could hope to simulate in real time on an average workstation. However, with the emergence of more and more powerful graphics accelerators, combined with new frameworks for programmers to program these devices, real time simulation of realistic fluids is quickly becoming reality. Technologies such as CUDA from NVIDIA has opened up many new possibilities for practical and affordable parallel computing.

A very highly performing fluid simulator using Smoothed Particle Hydrodynamics was developed for CUDA by former master student at the HPC-group at NTNU, Øystein E. Krog, capable of simulating both simple fluids and more complex snow avalanches. However, the simple fluid suffers from less than realistic behaviour in many instances, especially in water-like fluids, where it does not exhibit enough of a "splashing" effect.

The goal of this project is to improve the realism of the simple fluid simulator such that it can produce more realistic simulations of water-like fluids. At the same time it is important to maintain reasonable performance in the simulation. We have chosen to do this by implementing surface tension into the simulation, a property which greatly affects the behaviour of fluids such as water, and is very important for them to behave correctly.

Chapter 2

Surface Tension

2.1 Introduction

Surface tension is a property of the surface of a liquid. It allows the surface to resist an external force and the effects can be easily be seen every day in the behavior of water. Surface tension holds the fluid together, causing the formation of drops. It is also what allows certain insects to walk on the surface of water, and makes it possible for small objects such as a paper clip to float on the surface of water, even though the density of the object is greater than the water itself.

Surface tension is caused by cohesive forces between the liquid's molecules. Inside the liquid each molecule is pushed and pulled equally in every direction by nearby molecules. The net force acting on these molecules is zero. The molecules at the surface however, do not have molecules on all sides. This results in a net force pointing inwards into the liquid. An illustration of this is provided in figure 2.1.

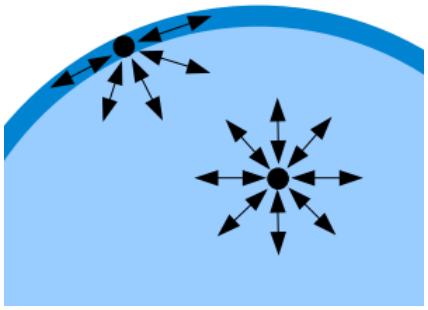


Figure 2.1: Illustration of the forces acting on two molecules in a liquid. One molecule is inside the liquid, while another is located on the surface.

This creates internal pressure and forces the liquid surface to contract to the smallest possible area. This is what makes liquid droplets take a spherical shape.

2.2 The Continuum Surface Force Method

The Continuum Surface Force (CSF) Method was developed by Brackbill et al. [2] in 1990. Processes that are localized to a fluid's interface are modeled in the CSF method by applying them to fluid elements in the transition region of the interface. Phenomena in the interface of a fluid, such as surface tension, is translated into a volume process with a net effect that emulates the physical behaviour of the phenomenon. [12]

In the CSF model, surface tension is represented by a force per volume F_s, which is given by the following equation:

$$\mathbf{F_s} = \mathbf{f_s}\delta_s \tag{2.1}$$

where δ_s is a normalized function, known as the surface delta function, which has its peak value at the interface of the fluid, and f_s is the force per area, which is given by

$$\mathbf{f_s} = \sigma\kappa\hat{\mathbf{n}} + \nabla_s\sigma \tag{2.2}$$

where σ is the surface tension coefficient, a constant which varies from fluid to fluid, $\hat{\mathbf{n}}$ is the unit normal to the interface, κ is the curvature of the interface and ∇_s is the surface gradient.

The first term in equation (2.2) is a force which acts along the interface normal, and is the surface tension force due to local curvature. This force smooths regions with high curvature, and works to reduce the total surface area. An illustration of how this force is affected by curvature can be seen in figure 2.2.

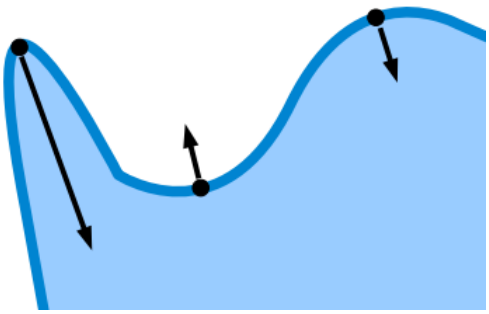


Figure 2.2: Surface tension force at points with varying curvature

The second term is a force which acts along a tangent of the interface. This force works to move fluid from regions of low surface tension to higher surface tension.

Chapter 3

Parallel computing and GPU programming

3.1 Parallel Computing

Parallel computing is a form of computation in which calculations are performed "in parallel". This is based on the principle that large problems can often be divided into smaller problems, which can be solved concurrently, opposed to sequentially like traditional non-parallel computing.

The main driving force between the recent development into parallel computing comes from the stagnation of the frequency of modern processors. The power density in modern processors are approaching the limit of what silicon can handle with current cooling techniques. This is known as the *power wall*.

Parallel computing can be embodied in many different forms, many of which do not exclude each other. The following are some of the layers of parallelism exposed by modern hardware [3]:

Multi-chip parallelism This means having several physical processor chips in a single computer. These chips share resources such as system memory through which the chips can relatively inexpensively communicate.

Multi-core parallelism This is similiar to multi-chip parallelism, with the distinction that the cores are all contained in a single chip. This lets the cores share resources like on-chip cache, allowing for less expensive communication.

Multi-context (thread) parallelism This is when a single core can switch between multiple execution contexts with little or no overhead.

Instruction parallelism This is when a single processor can execute more than one instruction simultaneously.

Traditionally, floating-point operations were considered expensive, while memory accesses were considered cheap. These roles have since been reversed, and

memory has become the limiting factor in most applications. Data has to be transferred through a limited number of pins at a limited frequency, causing what is known as the *von Neumann bottleneck*.

3.1.1 Flynn's Taxonomy

Michael J. Flynn proposed the following four classifications of parallelism in 1966:

SISD: Single Instruction, Single Data No parallelism.

SIMD: Single Instruction, Multiple Data Performing identical instructions on different data streams in parallel.

MISD: Multiple Instruction, Single Data Performing different instructions on a single data stream. Uncommon architecture.

MIMD: Multiple Instruction, Multiple Data Performing different instructions on different data streams.

In addition, MIMD is further divided into two different types.

SPMD: Single Program, Multiple Data When the different instructions actually stem from an identical program where branching has altered the control flow.

MPMD: Multiple Program, Multiple Data When the different instructions stem from different programs.

As we will see in section 3.3 modern GPUs expose a SPMD programming model on SIMD hardware.

3.1.2 Heterogeneous Computing

Recent developments are also showing rising interests in heterogeneous computing. Heterogeneous computing refers to systems that use a combination of different types of computational units.

The motivation for using heterogeneous systems is that although problems can be divided into smaller problems, all these smaller problems might not be of the same nature, and might benefit from different hardware architectures.

In the past, advances in technology and frequency scaling allowed most applications to increase in performance without structural changes. However, today, the effect of these advances are less dramatic since new obstacles such as the von Neumann bottleneck and the power wall have been introduced. Brodtkorb et al. [3] mentions the combination of a *Central Processing Unit* (CPU) and a *Graphics Processing Unit* (GPU) as one of the most interesting examples of heterogeneous systems for node level heterogeneous computing.

While neither the CPU or the GPU are heterogenous systems by themselves, they form a heterogeneous system when they are used together.

3.2 GPGPU

Using the GPU for computations traditionally handled by the CPU is known as General Purpose computing on Graphics Processing Units (GPGPU). The GPU is a specialized accelerator, designed for the acceleration of graphics computations. Their parallel nature allows them to efficiently perform large numbers of floating-point operations.

In terms of Floating Point Operations per Second (FLOPS), the GPU has has far surpassed the CPU, as can be seen in figure 3.1.

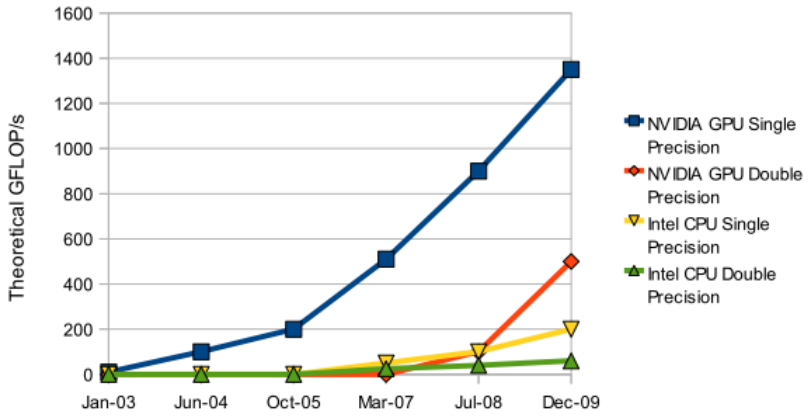


Figure 3.1: Development of Floating Point Operations per Second (FLOPS) in NVIDIA GPUs and Intel CPUs. [14]

The reason for this huge discrepancy in floating-point capability is that the GPU is specialized for intensive, highly parallel computation, and is designed such that more transistors are devoted to data processing rather than flow control and data caching.

An illustration of the distribution of transistors in a CPU and a GPU can be seen in figure 3.2.

The GPU also has much higher memory bandwidth than the CPU, which can be seen in figure 3.3.

These two factors make the GPU a very interesting platform for computationally intense applications. A GPU can almost be seen as a small supercomputer, allowing a single computer to perform simulations that previously belonged to the domain of larger clusters of computers.

This has created ample new possibilities for the scientific community. Problems such as fluid simulation, molecular dynamics, medical imaging and many more are experiencing drastic performance improvements. This can often mean the difference between seeing the results of an operation almost immediately, as opposed to waiting for hours or even days.

However, a downside to using GPUs is that they typically use a PCI express bus, which can become a very serious bottleneck in cases where the entire problem

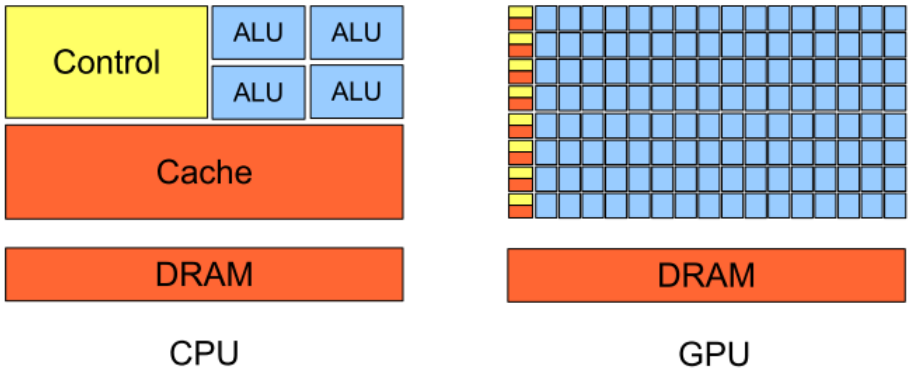


Figure 3.2: Distribution of transistors in a CPU and a GPU [14].

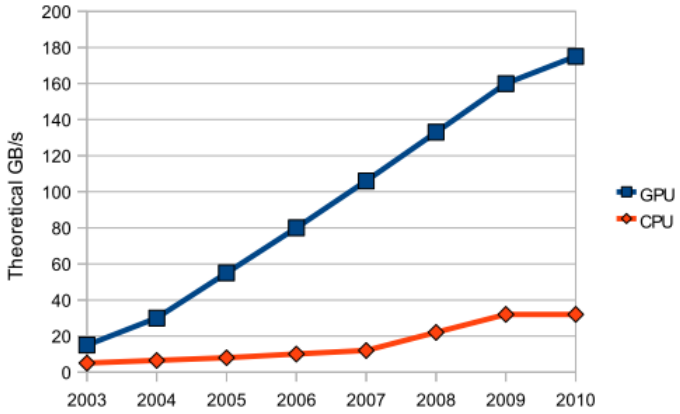


Figure 3.3: Development of Memory Bandwidth for NVIDIA GPUs and Intel CPUs. [14]

does not fit in the GPUs memory. A second generation PCI express x16 bus allows a theoretical maximum of 8 GB/s of data transfer between CPU and GPU memory.

Initially, GPUs were not designed for GPGPU. GPUs were programmed using shaders, a set of software instructions performed on the GPU. These shaders are tightly knit to graphical concepts such as vertices and pixels. In order to perform operations that were not related to graphics, the problem had to be transformed so that the GPU could solve it as if it were graphics-related. This was often not only difficult, but also very limited, since a problem could not always be solved efficiently with the limitations of the shaders.

Since then, more sophisticated frameworks for programming GPUs have been developed, the most mature of which is the *Compute Unified Device Architecture* (CUDA).

3.3 CUDA

CUDA is a parallel computing architechture developed by NVIDIA, and can be found in all newer NVIDIA GPUs. CUDA includes a software environment that allows developers to use a high-level programming language based on C called CUDA C. Other languages such as CUDA FORTRAN, OpenCL and DirectCompute are also supported. This makes programming for a CUDA device much easier than by using the old paradigm of shaders, and also gives the programmer more possibilities.

To describe the CUDA architecture we will first go through the older GT200 architecture and then look at some of the improvements that have been introduced with the new GT100 architecture, which was codenamed "Fermi". [3]

3.3.1 CUDA Hardware Model

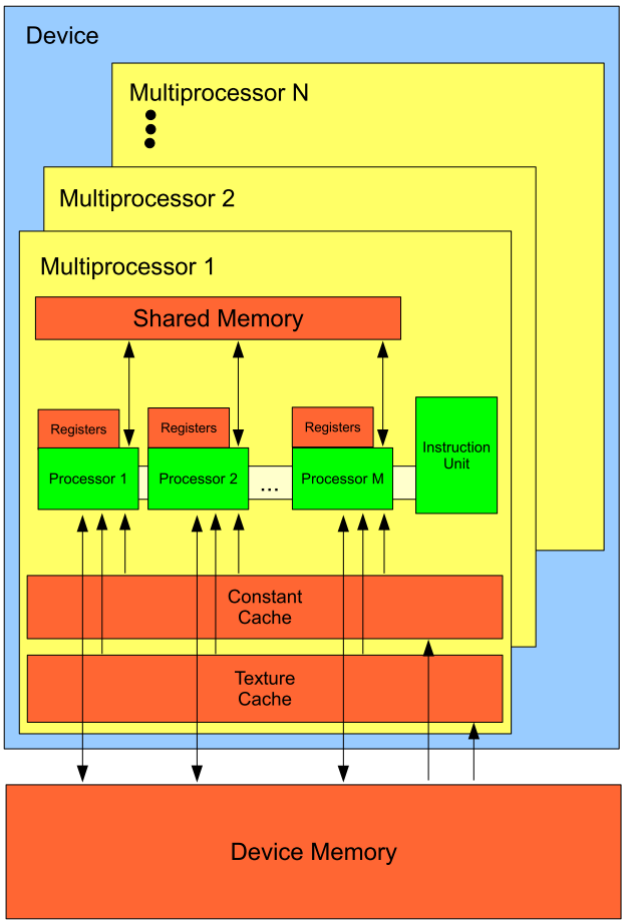


Figure 3.4: CUDA Hardware Model [14].

In the CUDA hardware model the device contains a number of multiprocessors, which in turn contain a number of processors. An illustration can be seen in figure 3.4.

Each processor is a fully pipelined ALU capable of integer or single precision floating point operations. Each multiprocessor has eight of these processors. In addition, each multiprocessor also has a double precision unit, and two special function units which are used to accelerate certain mathematical functions.

A large number of *threads* are organized into *blocks*, which in turn make up a *grid*. All blocks run the same program, which is known as the *kernel*, and threads within the same block can communicate and synchronize using a kind of local store memory called *shared memory*. However, there is no communication between blocks, except for the ability to perform atomic operations on global memory. The connection between threads, blocks and grid can be seen in figure 3.5.

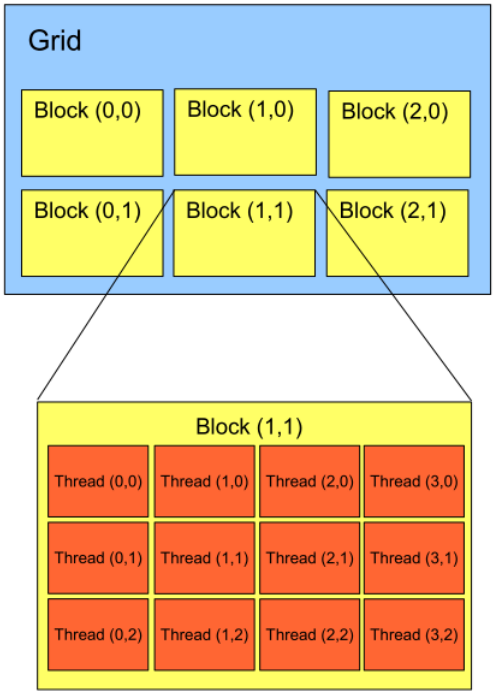


Figure 3.5: CUDA hierarchy of grid, blocks and threads [14].

The blocks are automatically divided into *warps*, which are groups of 32 threads. These warps are then scheduled to the Multiprocessors at runtime. In the GT200 architecture each Multiprocessor has eight Processors, and each warp is executed in a SIMD fashion by issuing the same instruction through four runs on these eight processors.

If a branch diverges within a warp, all non-diverging threads are masked out- until the threads reconverge. A high amount of divergence will therefore make the threads execute in a serial fashion. Divergence inside warps thus degrades

performance. Divergence between warps, however, does not have any impact on performance.

The multiprocessors can switch between warps without any overhead. By keeping many active warps ready, they can be switched between, in order to hide latencies. A block is always designated to a single multiprocessor, however, a multiprocessor can have more than one block designated to it. The number of blocks each multiprocessor can run is limited by the amount of registers and shared memory used by the warps.

3.3.2 CUDA Programming Model

A core concept in CUDA is the kernel, mentioned earlier, which is a function that is executed on the GPU by many threads concurrently. The threads are grouped in thread blocks, and when launching a CUDA kernel one has to specity both the number of thread blocks (*grid size*) and the number of threads per thread block (*block size*).

Threads within a block can communicate through the shared memory, as mentioned in section 3.3.1. This memory is much faster than the device memory, and managing this memory well is key to achieving good performance in applications that use a lot of memory. Also important is to make the memory requests *coalesced*, in other words that all simultaneous memory requests within a warp access the same segment in memory.

The ratio of active warps to the maximum number of supported warps is known as the *occupancy*, and can be used as an indication of how well latencies are hidden by the multiprocessors. As long as an application is bound by memory bandwidth, improving the occupancy will generally lead to improved performance.

3.3.3 Fermi

Fermi is based on the same concepts as the older architecture, however it implements several major improvements. Firstly, the number of processors has more than doubled, from 240 to 512. Secondly, double precision performance has been improved greatly, from being 1/8 of the floating point performance in the old architecture, to now being 1/2. A new cache hierarchy allows redesigned algorithms to achieve even greater performance. All vital parts of memory are now protected by ECC, and the memory space has also been unified, allowing C++ code to be run directly on the GPU.

The new architecture also allows up to 16 kernels to execute simultaneously, as opposed to the old architecture which only allowed a single kernel at a time.

Chapter 4

Smoothed Particle Hydrodynamics

4.1 Fluid representations

There are two conceptually different types of methods that can be used to represent fluid dynamics. They are Eulerian methods and Lagrangian methods [16]. In the Eulerian methods the movement of the fluid is observed by the velocity of the fluid in fixed points which do not move. In Lagrangian methods, however, the movement is instead observed through the position of points that move with the fluid.

Besides being either Eulerian or Lagrangian, methods are also distinguished as being either mesh-based or mesh-less. The difference between the two is that mesh-based methods contain explicit information about connections between points. While Eulerian methods are normally mesh-based, both mesh-based and mesh-less Lagrangian methods are common. However, since the points in Lagrangian methods are moving, unless special measures are taken, the mesh can become deformed causing bad conditioning of the simulated problem [15].

4.2 Smoothed Particle Hydrodynamics

The Smoothed Particle Hydrodamics method (SPH) was first developed in 1977 by Bob Gingold and Joe Monaghan [5], and independently by Leon Lucy [8]. It was originally intended to simulate the flow of gas clouds in astrophysics, but has since been extended to simulate various other physical phenomena, including but not limited to lava flow, snow avalanches and computational fluid dynamics.

SPH is a mesh-less Lagrangian method for approximating numerical solutions to the equations of fluid dynamics. To do this, the fluid is divided into a set of particles. Each of these can be viewed as a material particle, representing a small

part of the fluid itself. However, these particles also function as interpolation points, which can be used to calculate various properties of the fluid [11].

Since it is a Lagrangian method, SPH has many advantages over Eulerian methods which makes it well suited for real time simulations [6].

- Computation is only performed where it is necessary.
- Less storage and bandwidth is required because the fluid properties is only stored at the particle positions and not at every point in space.
- The fluid is not necessarily constrained to a finite box.
- Ensuring conservation of mass is trivial, since each particle represents a fixed amount of mass.

A main disadvantage of Lagrangian methods is that they can require a large number of particles to produce realistic results. However, Lagrangian methods can more than make up for this by being very highly parallelizable.

SPH is based on the following integral interpolant [11]:

$$A_I(\mathbf{r}) = \int A(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h)d\mathbf{r}' \tag{4.1}$$

where A is an arbitrary quantity, $\mathbf{r}$ is any position in space, W is a smoothing kernel and h is known as the smoothing length. The smoothing length manipulates the shape of the smoothing kernel W, which guarantees that for any $h < \mathbf{r} - \mathbf{r}'$ the smoothing kernel will be 0.

Equation (4.1) above can be approximated by the following sum:

$$A_S(\mathbf{r}) = \sum_j V_j A_j W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.2}$$

The summing of contributions to the attribute of a particle is illustrated in figure 4.1.

In SPH we use the particles as summation points. However, particles do not have any explicit volume. The volume is therefore replaced with the mass and density to find the volume which is implicitly represented by the particle. The resulting equation is the basis formulation of SPH.

$$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.3}$$

This equation can be used to approximate a new value for a property A at any position $\mathbf{r}$. However, it requires that the values for mass and density are known in all surrounding positions. The usual approach is to let all particles have an identical constant mass. This also has the advantage of making conservation of mass in the system trivial, as mentioned earlier. The density ρ_j can then be calculated using equation (4.3) by inserting density ρ for A:

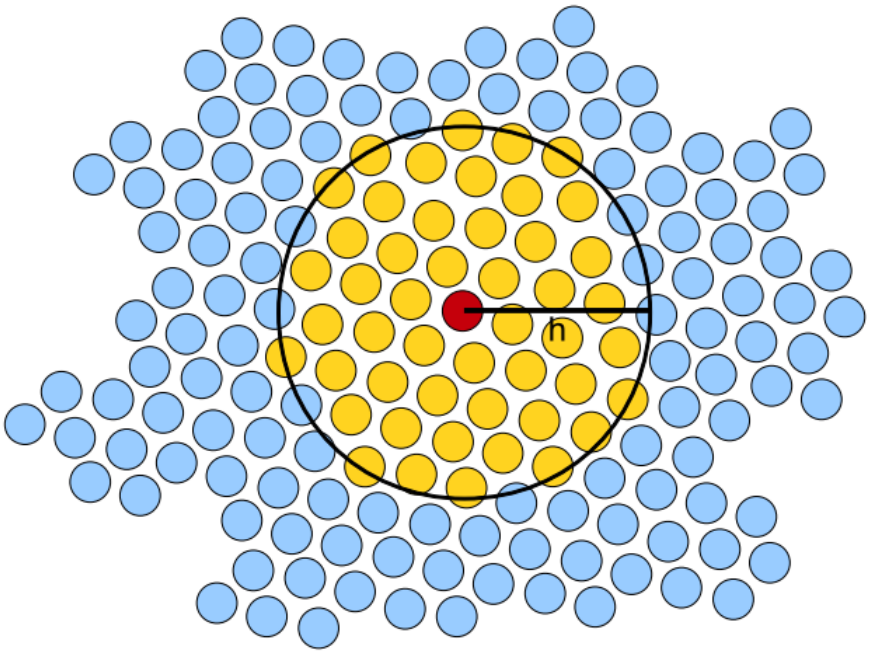


Figure 4.1: An attribute in the center particle is found by summing the contributions from the particles within the smoothing length h. The particles that contribute are colored yellow.

$$\begin{aligned} \rho_S(\mathbf{r}) &= \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r_j}, h) \\ &= \sum_j m_j W(\mathbf{r} - \mathbf{r_j}, h) \end{aligned} \tag{4.4}$$

However, if we are simulating an incompressible fluid, we can not use SPH to calculate the pressure, since SPH is inherently compressible [17]. In this case there are essentially two options. One is to find the pressure by solving the Poisson equation directly:

$$\nabla^2 P = \rho \frac{\nabla \mathbf{v}}{\Delta t} \tag{4.5}$$

Unfortunately this is computationally expensive. Another approach is to use an equation of state. An equation of state provides a mathematical relation between two or more state variables.

A commonly used equation of state is the ideal gas state equation:

$$P = k\rho \tag{4.6}$$

However, it is often modified by adding a rest density ρ_0 and a rest pressure P_0 to the system.

$$P = P_0 + k(\rho - \rho_0) \tag{4.7}$$

4.3 Smoothing Kernel

The function $W(\mathbf{r} - \mathbf{r_j}, h)$ is known as the smoothing kernel. The smoothing kernel is a scalar weight function, and its behaviour is modified by the smoothing length h. The smoothing length works as a cutoff value and directly affects how many particles are considered in the sum in equation (4.3). The choice of smoothing kernels is significantly important, and largely effects the behaviour and accuracy of a simulation. Different kernels may also be better suited for smoothing the various different quantities that are in play. The choice of smoothing kernel also largely affects the performance of the simulation. The smoothing kernel and smoothing length should be chosen with this in mind, since an overly accurate smoothing kernel will needlessly degrade performance.

For a smoothing kernel there are several properties that must hold [7]:

1. The normalization condition:

$$\int_{\mathbf{r}} W(\mathbf{r}, h) d\mathbf{r} = 1 \tag{4.8}$$

   This condition ensures that the maximum and minimum values of a quanitity are not enhanced after being smoothed by the kernel.

2. The kernel must be positive:

$$W(\mathbf{r}, h) > 0 \tag{4.9}$$

3. The delta function property:

$$\lim_{h \to 0} W(\mathbf{r} - \mathbf{r_j}, h) = \delta(\mathbf{r} - \mathbf{r_j}) \tag{4.10}$$

   where δ is Dirac's delta function:

$$\delta(\mathbf{r}) = \begin{cases} \infty & |\mathbf{r}| = 0, \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

4. The compact condition:

$$W(\mathbf{r} - \mathbf{r_j}, h) = 0, |\mathbf{r} - \mathbf{r_j}| > h \tag{4.12}$$

   which guarantees that the contribution from any particle outside of the smoothing length is 0.

If the normalization condition holds, and the kernel is even ($W(\mathbf{r}, h) = W(-\mathbf{r}, h)$) then the kernel is of second order accuracy [10]. This means that when approximating (4.1) by (4.3) the error is $O(h^2)$ or better.

4.3.1 Derivatives

One of the main features of SPH is that when finding the derivative of an attribute A, only the derived smoothing kernel is needed, since all the other variables can be considered constant [15]. The gradient is as follows:

$$\begin{aligned}\nabla A_S(\mathbf{r}) &= \nabla \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r_j}, h) \\ &= \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r_j}, h)\end{aligned} \tag{4.13}$$

and similiarly, the Laplacian is:

$$\begin{aligned}\nabla^2 A_S(\mathbf{r}) &= \nabla^2 \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r_j}, h) \\ &= \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r_j}, h)\end{aligned} \tag{4.14}$$

However, using the gradient and laplacian in this form may in some cases produce undesirable results. An example are cases where the calculated attribute should be symmetric, such as forces. Forces should adhere to Newton's third law, stating that for any force F there exists an equal, opposite and collinear force $-F$.

A suggested symmetrized version of the gradient was developed by Monaghan [9], and is known as the Symmetric Gradient Approximation Formula (SGAF).

$$\nabla A_s(\mathbf{r}) = \rho \sum_j m_j (\frac{A}{\rho^2} + \frac{A_j}{\rho_j^2}) \nabla W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.15}$$

This gradient conserves linear and angular momentum, and is commonly used for the pressure gradient.

An alternative, and simple, pressure gradient was suggested by Müller et al. [13], which focuses on speed and stability.

$$\nabla A_s(\mathbf{r}) = \sum_j m_j \frac{A - A_j}{2\rho_j} \nabla W(\mathbf{r} - \mathbf{r_j}, h) \tag{4.16}$$

4.4 Existing SPH implementation

In this work we extend an existing SPH implementation developed by Øystein E. Krog [17]. The simulation is implemented in CUDA, and includes two different SPH models.

The first is a "simple" SPH model, based on the work of Müller et al. [13]. This model simulates an incompressible Newtonian fluid. It is designed for interactivity and trades accuracy for performance.

The second model is a more complex model which uses more accurate smoothing kernels and supports a range of rheological models which enables the simulatino of Non-Newtonian fluids. In Krog's work, this model is used to simulate snow avalanches.

Since this project is about simulating water-like liquids in real time, rather than snow avalanches, focus has been given to the simple SPH model. However, the modifications made could easily be applied to the complex model as well.

Both models is built around the same framework. Only the calculation of the forces on the SPH particles is different. The simulation uses a data structure which stores particles in a grid, based on their position in the domain. Each cell in the grid is set to be equal to the smoothing length h. This allows the search for nearby particles to be accelerated greatly. Only the cell of the particle, and the 26 surrounding cells have to be searched, since particles in any other cells would be outside of the smoothing length and their contribution zero.

An 2D example of this is shown in figure 4.2. In 2D there are only 8 surrounding cells. In 3D, an additional 9 surrounding cells would be added on each side of the 2D cell "plane".

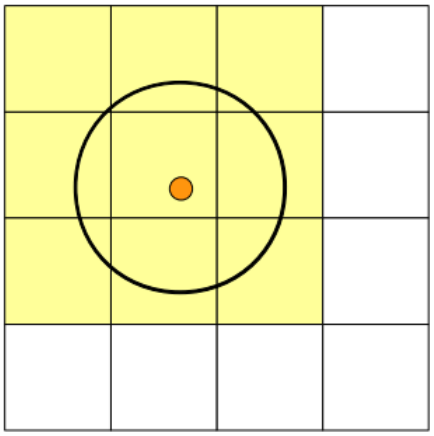


Figure 4.2: Particle grid. When checking for particles near the orange particle, only the surrounding yellow grid cells could possibly contain particles that lie within the smoothing length, illustrated with a black circle.

The implementation of this is based on NVIDIA's implementation in their CUDA Particles Demo [6].

This grid has to be updated at the beginning of every iteration.

Then the forces of each particle is calculated according to the SPH equations of the chosen model. After that, external forces are added. The external forces, come from gravity and the boundaries, or walls, of the simulation. The gravity is easily added by adding accelerating each particle by the gravitational constant g.

The boundary forces exist in order to keep the particles within a cube shaped domain. Recall from chapter 4.2 that particles in a lagrangian simulation did not necessarily have to be constrained in a finite box. However, it is still done in this simulation for several reasons. The boundaries function as a container which the fluid can fill up. Constraining the fluid to a cube shaped domain is also vital for the grid data structure described earlier to function properly. Particles outside the grid would not be able to take advantage of the faster neighbor search.

Finally, the new acceleration, velocity and position of each particle is calculated, and optionally, each particle is given a color based on an attribute. In most images in this report, the particles are colored based on their velocity, where blue represents the slowest velocity, and red the fastest.

4.4.1 Simple SPH model

The simple model simulates an incompressible fluid by calculating the forces on the particles by approximating the Navier-Stokes equations with SPH.

The Lagrangian formulations of the Navier-Stokes equations for simulating an imcompressible fluid are as follows [17]:

1. Conservation of mass:
$$\frac{d\rho}{dt} = -\rho \nabla \cdot \mathbf{v} \tag{4.17}$$

2. Conservation of momentum:
$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla P + \frac{\mu}{\rho}\nabla^2 \mathbf{v} + \mathbf{f} \tag{4.18}$$
where P is pressure, μ is the *dynamic viscosity* which is a constant in a Newtonian fluid, and $\mathbf{f}$ is external force.

The mass in the system is considered constant which, as mentioned earlier, is trivial in SPH by keeping particle mass constant and the number of particles constant. This allows the first Navier-Stokes equation, the conservation of mass, to the ignored. What remains to be calculated is then only the equation of conservation of momentum.

The simple SPH model first calculates the density as follows:

$$\rho_i = \sum_j m_j W_{poly6}(\mathbf{r} - \mathbf{r_j}, h) \tag{4.19}$$

where W_{poly6} is a smoothing kernel developed by Müller et al. [13], which provides a good compromise between performance and accuracy.

$$W_{poly6}(\mathbf{r}-\mathbf{r_j},h)=\frac{315}{64\pi h^9}\begin{cases}(h^2-|\mathbf{r}-\mathbf{r_j}|^2)^3 & 0\leq|\mathbf{r}-\mathbf{r_j}|\leq h\\ 0 & otherwise\end{cases} \tag{4.20}$$

Then, the pressure is calculated using the ideal gas equation of state, equation (4.7).

Once the pressure is calculated, the terms in equation (4.18) can be approximated with SPH. The first term, or the *pressure force*, is calculated using SGAF from equation (4.15), in combination with a special kernel W_{spiky} developed by Desbrun and Gaschuel [4].

$$-\frac{1}{\rho}\nabla P=-\sum_j m_j(\frac{P}{\rho^2}+\frac{P_j}{\rho_j^2})\nabla W_{spiky}(\mathbf{r}-\mathbf{r_j},h) \tag{4.21}$$

$$W_{spiky}(\mathbf{r}-\mathbf{r_j},h)=\frac{15}{\pi h^6}\begin{cases}(h-|\mathbf{r}-\mathbf{r_j}|)^3 & 0\leq|\mathbf{r}-\mathbf{r_j}|\leq h\\ 0 & otherwise\end{cases} \tag{4.22}$$

The second term in equation (4.18), the *viscosity force* is calculated using a special smoothed version of the SPH laplacian and a smoothing kernel $W_{viscosity}$ both developed by Müller et al. [13].

$$\frac{\mu}{\rho}\nabla^2\mathbf{v}=\frac{\mu}{\rho}\sum_j\frac{\mathbf{v_j}-\mathbf{v}}{\rho_j}\nabla^2 W_{viscosity}(\mathbf{r}-\mathbf{r_j},h) \tag{4.23}$$

$$W_{viscosity}(\mathbf{r}-\mathbf{r_j},h)=\frac{15}{2\pi h^3}\begin{cases}-\frac{|\mathbf{r}-\mathbf{r_j}|^3}{2h^3}+\frac{|\mathbf{r}-\mathbf{r_j}|^2}{h^2}+\frac{h}{2|\mathbf{r}-\mathbf{r_j}|}-1 & 0\leq|\mathbf{r}-\mathbf{r_j}|\leq h\\ 0 & otherwise\end{cases} \tag{4.24}$$

The calculations described in this subsection are are implemented in CUDA kernels that spawn one thread for each particle. For each iteration the density has to be calculated first since it is a dependency for the next operations. Density calculation is therefore performed in its own kernel. The pressure force and viscosity force is then calculated simultaneously by a single kernel, since there are no dependencies between them. A graph showing the calculation of forces in the simple SPH model can be seen in figure 4.3.

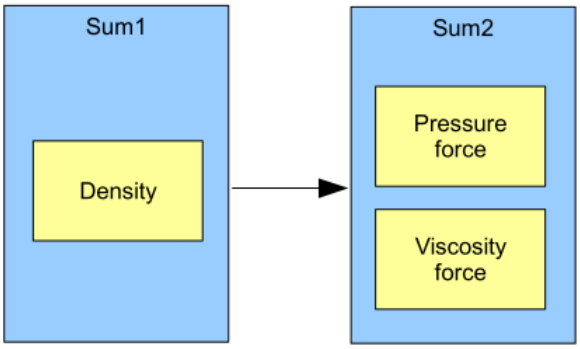


Figure 4.3: Graph showing the calculation of forces in the simple SPH model

Chapter 5

Implementation

5.1 Surface Tension in SPH

To add surface tension to the simulation we use SPH to implement the Continuum Surface Force method (CSF) described in chapter 2.2. We use methods based on the work of J. P. Morris [12], and Müller et al. [13].

Recall from chapter 2.2 that the surface tension was given by a force per volume, equation (2.1),

$$\mathbf{F_s} = \mathbf{f_s}\delta_s$$

where $\mathbf{f_s}$ is a force per unit area defined by equation (2.2),

$$\mathbf{f_s} = \sigma\kappa\hat{\mathbf{n}} + \nabla_s\sigma$$

We assume that the surface tension is constant throughout the fluid, therefore the second term, the force which worked to move the fluid from areas with low surface tension to areas with higher surface tension, can be ignored.

The force density to be calculated then reduces to:

$$\mathbf{F_s} = \sigma\kappa\hat{\mathbf{n}}\delta_s \tag{5.1}$$

The first thing to consider is how to find the unit normal to the surface of the fluid, $\hat{\mathbf{n}}$.

To find this normal, we use a new field quantity which is 1 at particle locations and 0 anywhere else. This field is called a *color field*.

We use SPH to get the smoothed color field by inserting $A_j = 1$ into the SPH equation (4.3),

$$c_s(\mathbf{r}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{r} - \mathbf{r_j}, h) \tag{5.2}$$

For a smoothing kernel we chose the W_{poly6} smoothing kernel mentioned in chapter 4.4.1, because it provides good performance, while still offering acceptable stability and accuracy.

The gradient of the smoothed color field yields a surface normal pointing into the surface.

$$\mathbf{n} = \nabla c_s \tag{5.3}$$

This normal is not a unit normal, however, the unit normal is easily found by dividing by the length of the normal.

$$\hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|} \tag{5.4}$$

Next, we consider the kurvature κ. The curvature can be calculated using [12, 13]

$$\kappa = -\nabla \cdot \hat{\mathbf{n}} = \frac{-\nabla^2 c_s}{|\mathbf{n}|} \tag{5.5}$$

The only part missing from equation (5.1) is now the surface delta function δ_s. Recall that the surface delta function was supposed to be a normalized function that peaked at the interface, i.e. the surface, of the fluid.

It turns out we already have such a function available: $|\mathbf{n}|$. This function clearly peaks at the surface, and is normalized given that the smoothing kernel is normalized as well, which was one of the properties that had to hold for smoothing kernels, from chapter 4.3.

Putting it all together we end up with the following expression for the surface tension:

$$\mathbf{F_s} = \sigma \frac{-\nabla^2 c_s}{|\mathbf{n}|} \frac{\mathbf{n}}{|\mathbf{n}|} |\mathbf{n}| = -\sigma \nabla^2 c_s \frac{\mathbf{n}}{|\mathbf{n}|} \tag{5.6}$$

However, evaluating $\frac{\mathbf{n}}{|\mathbf{n}|}$ at positions where $|\mathbf{n}|$ is a small number would cause numerical problems. We therefore only calculate the surface tension if $|\mathbf{n}|$ is larger than a certain value. Changing this value would change the thickness of the "surface layer" of particles that were affected by surface tension. We experimented with different values and a value of 20 seemed to create a reasonably thick layer. By changing the colors of the particles based on whether their value of $|\mathbf{n}|$ were larger than 20 or not, we could get a visual representation of which particles were considered surface particles. A screenshot of this can be seen in image 5.1

5.2 Initial approach

We approximated the gradient and laplacian of the color field by using the straight forward SPH gradient and laplacian.

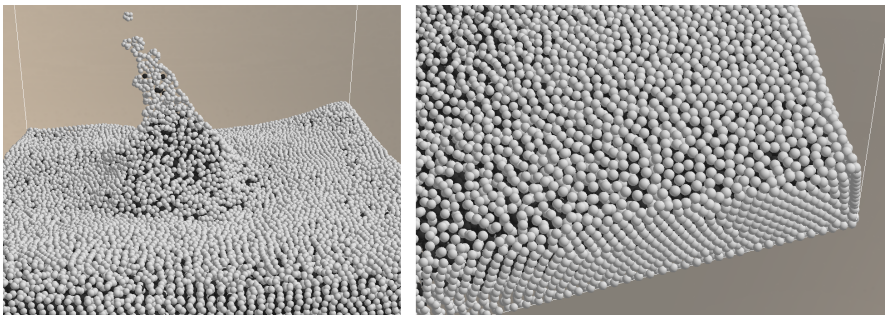


Figure 5.1: Particles colored based on whether they are considered surface particles or not. Surface particles are white, rest are black.

$$\nabla c_s = \sum_j m_j \frac{1}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r_j}, h) \tag{5.7}$$

$$\nabla^2 c_s = \sum_j m_j \frac{1}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r_j}, h) \tag{5.8}$$

We created a kernel that would compute these SPH sums and calculate the surface tension by spawning a thread for each particle, the same way the other SPH sums were calculated in the simulation. We made these calculations happen after the calculation of the pressure force and the viscosity force. The updated graph of the force calculations can be seen in figure 5.2.

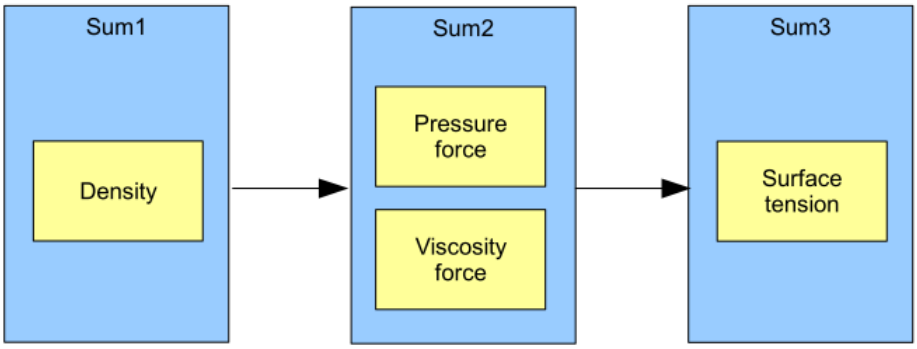


Figure 5.2: Calculation of forces in initial approach

5.3 Improvement efforts

Although the initial approach successfully applied surface tension to the simulation, some efforts were made to improve upon it.

We tried using more advanced and symmetrized versions of the SPH gradient and laplacian for calculating the color field gradient and laplacian, such as the SGAF. However, this yielded no perceivable change in behaviour for the simulation, so it was scrapped in favor of the simple straight forward approach which was initially used, because of its smaller computational cost.

We then made some efforts to improve performance. There were no computational dependencies between second and the third summing kernels, that is, the values for pressure force and viscosity force are not needed in order to compute surface tension. The calculation of surface tension could therefore be included in the second kernel, without having to execute its own. The resulting graph of force calculation can be seen in figure 5.3.

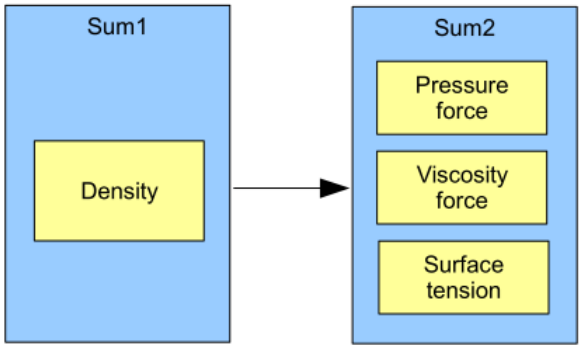


Figure 5.3: Surface tension included into second kernel

This resulted in a decent performance boost, the details of which can be seen in chapter 6.2.

For testing the surface tension in our simulation, we deactivated gravity and studied the behaviour of a cube shaped body of fluid. This is a classic testing case for surface tension that has been frequently used [1]. Ideally, the cube should morph into a sphere. However, a problem was discovered during this test, which was that close to the boundary of where surface tension was applied, the particles had an erratic movement. In order to remedy this, a smoothing procedure inspired by [1] was applied after the calculation of the surface tension.

$$\mathbf{F}_{smooth} = \frac{\sum_j \mathbf{F}_j^s W(\mathbf{r} - \mathbf{r_j}, h)}{\sum_j W(\mathbf{r} - \mathbf{r_j}, h)} \tag{5.9}$$

The calculation of this new surface tension was dependent on the calculation of the original surface tension, and therefore had to be executed by a new kernel after the second kernel. The final graph of calculation of forces can be seen in figure 5.4.

A problem which arose from this was that the original surface tension force had to be stored in memory. Before, this force was simply added to a total force in combination with the pressure and viscosity forces, with no way of knowing afterwards how much of this force was provided by the surface tension.

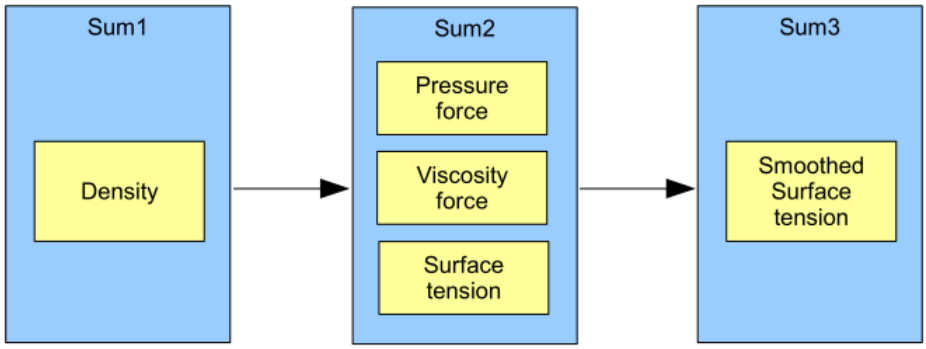


Figure 5.4: Smoothing of surface tension in new kernel

In order to avoid a needless increase in memory use, we chose to store the surface tension force in the particles' color values. This color value was always recalculated at the end of every iteration anyway, and was not dependent on its previous value.

Chapter 6

Results and Discussion

6.1 Behaviour

Results from the surface tension test described in chapter 5.3, where a cube of fluid is released in zero gravity, can be seen in figures 6.1, 6.2 and 6.3.

For reference purposes, the results of the test when there is no surface tension present at all can be seen in figure 6.1. Since there are no forces holding the fluid together, the cube simply explodes.

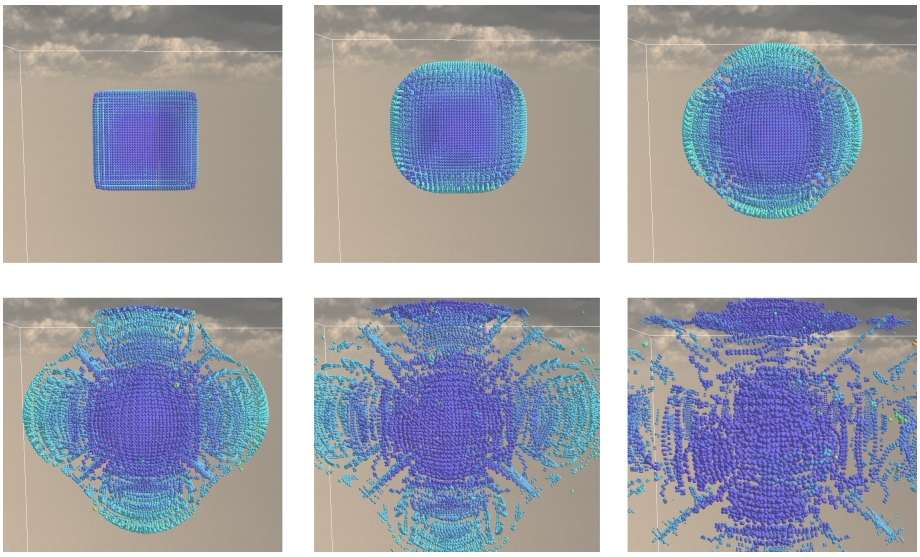


Figure 6.1: Cube of fluid in zero gravity with no surface tension

The results of the test when surface tension is implemented, but not smoothed, is shown in figure 6.2. The problem of erratic movement of particles at the boundary of the surface particles can be seen in the colors. The particle colors are based on particle velocity, and ideally all particles should be calm, and thus

be colored blue. However, the particles just inside the surface are turquoise, and some even yellow or orange, indicating movement.

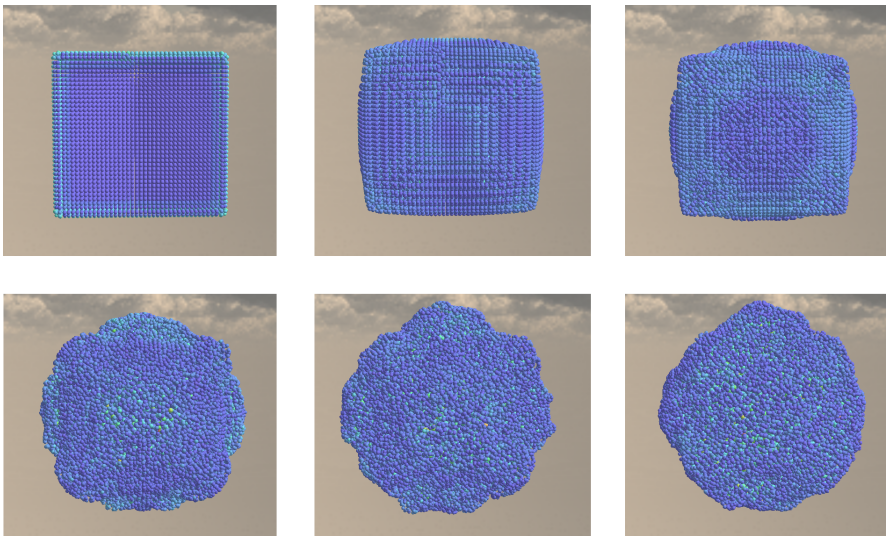


Figure 6.2: Cube of fluid in zero gravity with non-smoothed surface tension

Finally, the results of the zero gravity cube test when surface tension is implemented and smoothed is seen in figure 6.3. The problem with the erratic particle movement is gone.

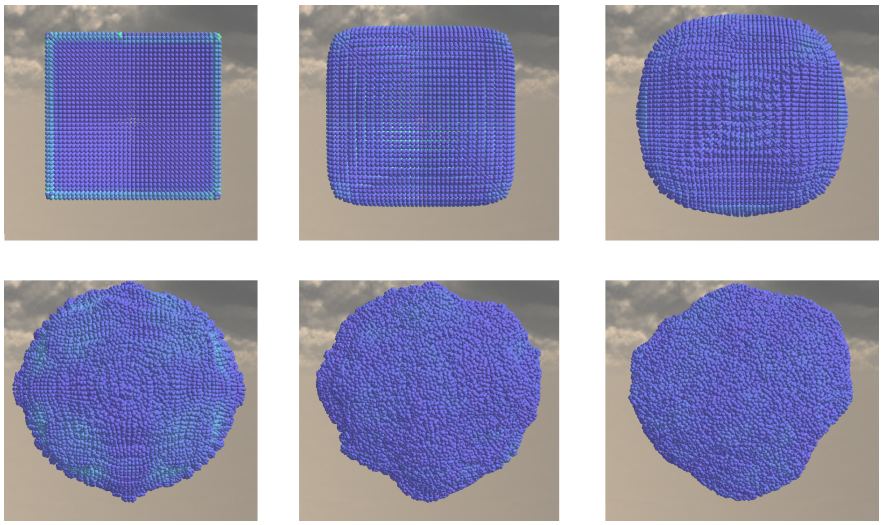


Figure 6.3: Cube of fluid in zero gravity with smoothed surface tension

With the new surface tension implemented the behaviour of the simulation is very visibly altered. The surface tension not only causes the fluid to form droplets, it also increases the effect of splashing in the fluid. This greatly increases the realism of what happens when a drop falls into a body of fluid, as

can be seen in figure 6.4.

The surface tension also alleviates a problem with SPH. SPH only works properly when there are many particles together. When a single or couple of particles become isolated, their behavior becomes unpredictable, and erroneous. In the old simulation isolated particles could often be seen floating around in mid-air. Since there were no forces keeping the particles together, other than gravity and walls, particles could easily become isolated. While surface tension does not fix the behavior of isolated particles, it does however provide a force that holds particles together, and thereby reduces the likelyhood of particles becoming isolated in the first place.

As can be seen from the screenshots the fluid does not obtain a perfect spherical shape. The reason behind this is not clear. It might be a problem with the surface tension force itself, however it might also be due to the other forces working within the fluid. In either case, it is quite likely that the current state should be good enough for simulations where very high accuracy is not required, which is the case for most interactive real time simulations.

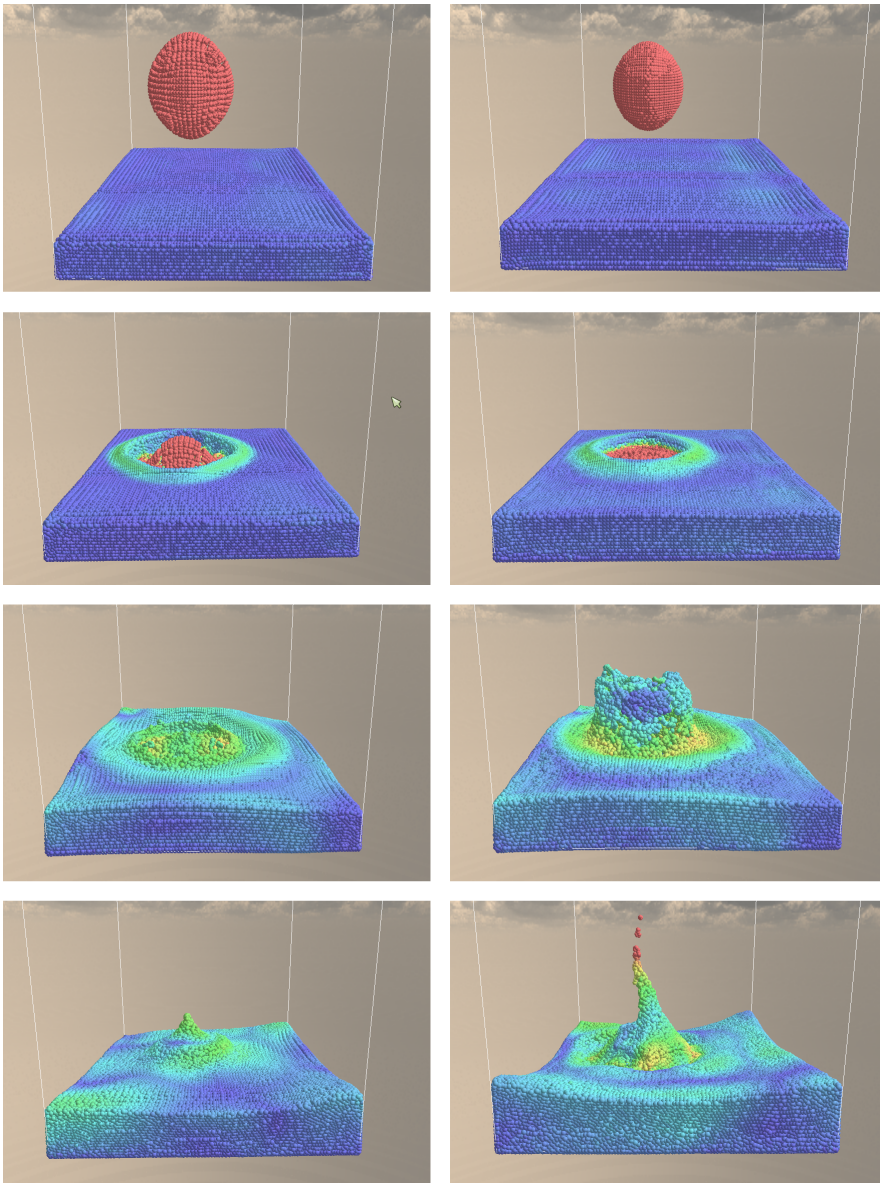


Figure 6.4: A drop of fluid falling into a pool. Simulation on the left has no surface tension, while simulation on the right does.

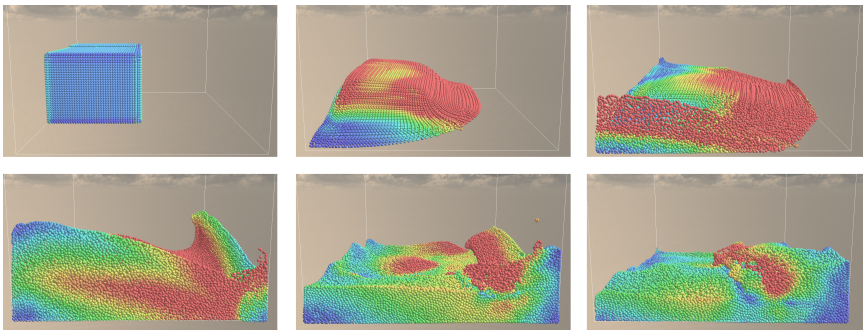


Figure 6.5: Dam break test scenario

Parameter	Value
Timestep	0.002
Grid World Size	1024
Simulation Scale	0.0005
Rest Density	1000
Rest Pressure	100
Ideal Gas Constant	1.5
Viscosity	1.2
Boundary Stiffness	30000
Boundary Dampening	256
Static Friction Limit	0
Kinetic Friction	0
Surface Tension Coefficient	0.07

Table 6.1: Performance test simulation parameters

6.2 Performance

To test the performance of the simulation, a dam break scenario was used. The fluid was positioned in a box shape in one corner of the domain, after which it was allowed to flow freely. A visual example of the dam break scenario can be seen in figure 6.5.

The parameters used in the test can be seen in table 6.1. These parameters create a very stable and realistic fluid.

The test was performed on two different systems. One had a NVIDIA GTX260 graphics card, an Intel Core 2 Quad Q9550 CPU running at 2.83 GHz and 4 GB RAM. This graphics card uses the old NVIDIA GT200 architecture.

The second machine had a NVIDIA GTX460 graphics card, an Intel i7 CPU running at 3.07 GHz and 6 GB RAM. This graphics card uses the newer GF100 "Fermi" architecture.

The interesting part is the performance difference between the two graphics cards, and it therefore would have been best to test the two cards on an otherwise identical system. Unfortunately we did not have the opportunity to do so. However, the result would most likely have been very similiar, since the

system specifications beyond the graphics card should have very little impact on performance.

The average frames per second (FPS) when running the test cases with the SPH model and various variations of surface tension can be seen in figures 6.6 and 6.7.

Figure 6.6 shows the performance on the system with the older GTX260 graphics card, while figure 6.7. The interesting part of these results are how the SPH model performs after surface tension has been implemented, compared to how it performed before. Krog [17] has already explored in detail how his implementation of the simple SPH model compares to other known implementations of SPH.

The performance of the initial approach of implementing surface tension can be seen in figure 6.6. It reaches around 60 % of the performance of the original simple model. The faster surface tension, where calculation was combined in the second kernel, but not smoothed, reaches almost 80 %, while the smoothed surface tension reaches around 70 %.

While this is a considerable slowdown from the original model, the performance is still quite high considering that the original SPH model was very fast.

It is interesting to note that the smoothed surface tension performs better than the initial approach, even though they both require three kernel executions. This is most likely caused by a memory bottleneck in the initial approach. The calculation of surface tension requires many values that are also used by pressure and viscosity force calculation, such as mass and pressure, resulting in better memory efficiency when these calculations are combined.

The smoothing of the surface tension by comparison does not use as much memory. It only requires the stored surface tension force. This makes the penalty of executing a separate kernel for this calculation less than that for the calculation of the surface tension force.

Figure 6.7 shows the performance when running the test on a GTX460 Fermi card. The initial approach of surface tension was omitted in this test. The other versions perform better in all cases, making the initial approach rather uninteresting, since it would never be beneficial to use it.

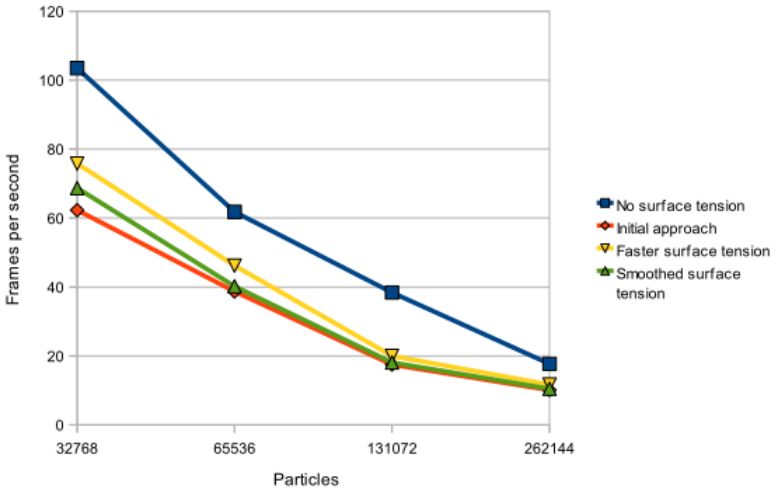


Figure 6.6: Performance of dam break simulation on a GTX260

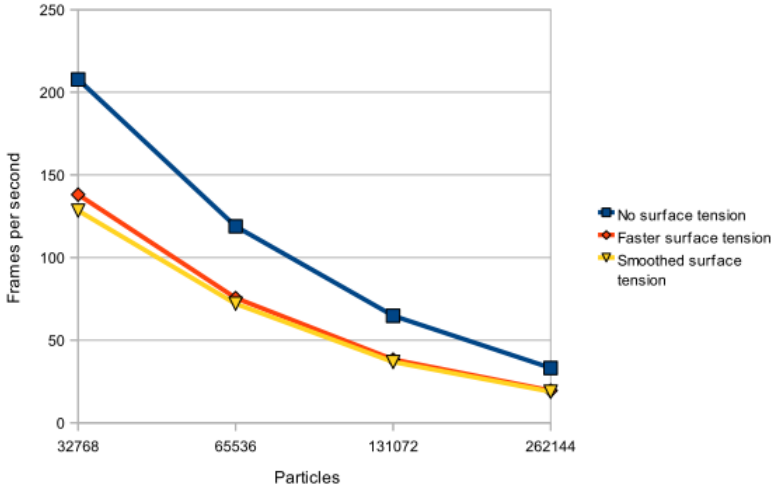


Figure 6.7: Performance of dam break simulation on a GTX460

Chapter 7

Conclusions and Future Work

In this project a fluid simulator using Smoothed Particle Hydrodynamics (SPH) was extended to improve the realism of water-like fluid simulation.

A description of SPH as well as the original fluid simulation was presented, along with CUDA, a GPGPU architecture for which the simulation was created.

Surface tension was implemented into the simulation, and it was shown how this greatly enhanced the behaviour of the fluid, and made it look more water-like. Efforts were made to improve both performance and quality of the surface tension.

The new fluid simulation, while suffering a performance hit, still retained reasonable performance with the final version having around 70 % of the performance of the original.

7.1 Future Work

A potentially interesting future work could be to extract a mesh surface out of the fluid particles, which would improve the visual quality of the fluid drastically. The surface particle detection mentioned in chapter 5.1 and illustrated in figure5.1could be used to greatly lower the computational cost of this, since it would allow for focusing only the surface particles which will be visible, while ignoring the hidden inner particles.

A downside of using the CUDA to implement the simulation is that it can only be run on machines with an NVIDIA graphics card. Reimplementing the simulation in OpenCL would enable support for graphics cards by other vendors. It could also be interesting to implement a more lightweight graphics application to render the simulation. Currently, the application uses the open source OGRE 3D engine, which is a bit overkill for the simple visualization needed for this simulation.

Beyond this, there are still other improvements that can be made to the original simulation, and the suggestions for future work that Krog mentioned in his thesis [17] still apply. These include using more advanced SPH models to get more physically accurate results, although this would kill any real-time interactivity of the simulation, and also plenty of work related to the snow avalanche part of the simulation, which has been largely untouched by this project.

Bibliography

[1] Kai Bao, Hui Zhang, Lili Zheng, and Enhua Wu. Pressure corrected sph for fluid animation. *Computer Animation and Virtual Worlds*, 20:311–320, 2009.

[2] J. U. Brackbill, D. B. Kothe, and C. Zemach. A continuum method or modeling surface tension. *Journal of Computational Physics*, 100:335 – 354, 1992.

[3] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18:1 – 33, 2010.

[4] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. *In Computer Animation and Simulation (Proceedings of EG Workshop on Animation and Simulation)*, pages 61 – 76, 1996.

[5] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, 1977.

[6] Simon Green. *CUDA Particles*. NVIDIA, November 2007.

[7] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Master's thesis, University of Copenhagen, 2006.

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

[9] J. J. Monaghan. An introduction to sph. *Computer Physics Communications*, 48(1):89 – 96, 1988.

[10] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543–574, 1992.

[11] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

[12] Joseph P. Morris. Simulating surface tension with smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 33:333 – 353, 2000.

[13] Mathias Muller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154 – 159, 2003.

[14] NVIDIA. *NVIDIA CUDA C Programming Guide*, 3.2 edition, September 2010.

[15] Marcus Vesterlund. Simulation and rendering of a viscous fluid using smoothed particle hydrodynamics. Master's thesis, Umea University, 2004.

[16] F. M. White. *Fluid Mechanics*. McGraw-Hill, 1994.

[17] Ø ystein E. Krog. Gpu-based real-time snow avalanche simulations. Master's thesis, Norwegian University of Science and Technology (NTNU), 2010.

Appendix A

Poster displayed during SC10

At the SC10 supercomputing convention in New Orleans in November 2010, the following poster was displayed at the NTNU HPC-group booth. The poster was created in cooperation with fellow master student Yngve S. Lindal.

Smoothed Particle Hydrodynamics (SPH) on GPUs

Master students: Fredrik Fossum (1), Yngve Sneen Lindal (1)
Supervisors: Drs. Anne C. Elster (1) and Håvard Holm (2)
(1) Dept. of Computer & Info. Science (2) Dept. of Marine Technology
Norwegian University of Science and Technology (NTNU) Trondheim, Norway

Smoothed Particle Hydrodynamics

The SPH method is a Lagrangian interpolation method for approximating a solution to the Navier-Stokes equations. It works by dividing the fluid into a set of particles. A particle's properties are "smoothed" over a distance *h* (known as the smoothing length) by a *kernel function*. This means that any physical quantity, such as density, pressure, temperature etc., of a particle can be obtained by summing the relevant quantity from particles within the smoothing length. The contributions from each particle is weighted by their mass, density and distance.

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

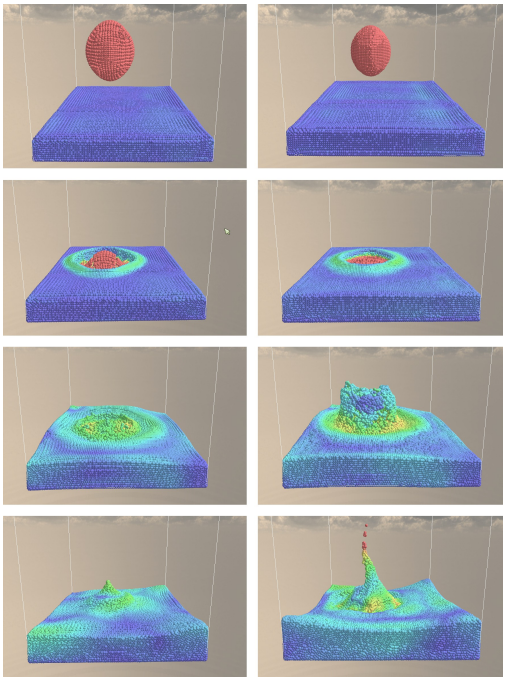
Where *A* = *arbitrary property*

Assigning the particles to a grid, and setting the grid cell size equal to the smoothing length in all dimensions allows us to save computation time. Since we only have to check the neighboring cells for particles within the smoothing distance, the cost of searching for neighboring particles is significantly lowered. This is well worth the small cost of updating the grid at the beginning of each iteration.

A high performing implementation of SPH on GPU using CUDA was implemented by former HPC-Lab student Øystein E. Krog with the aim to simulate snow avalanches [1]. This implementation also included a simpler SPH model for simulating fluids [2].

Surface Tension

With the goal of increasing realism, we have added surface tension to the previous simple SPH implementation. Surface tension is a property of liquid surfaces caused by cohesion of the liquid's molecules. Surface tension is what makes liquid form into spherical drops, and is also what allows certain insects to run on top of water. Surface tension is very important for realistic free surface flow simulations. It increases realism of splash effects and drop formation in the fluid.



Surface tension in the simulation on the right creates a clearly visible splash effect.

State-of-the-art SPH solver

We are also parallelizing a state-of-the-art SPH solver developed at the department of marine technology at NTNU and Sintef Marintek. The solver is implemented in Fortran and our work will include porting the code to the CUDA platform, verifying correctness and optimizing it.

From the definition of the SPH formula, a derivation of the smoothed density for each particle would be

$$\varrho(\mathbf{r}) = \sum_{b=1}^{N} \varrho_b W(\mathbf{r} - \mathbf{r}', h) dV_b = \sum_{b=1}^{N} \varrho_b W(\mathbf{r} - \mathbf{r}_b, h) = \sum_{b=1}^{N} m_b W(\mathbf{r} - \mathbf{r}_b, h) \quad (1)$$

However, this gives a faulty picture of the density for boundary particles, see figure 1.

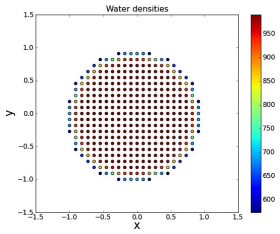


Figure 1. The boundary particles suffers from having too few neighbors, which results in a density far below ~1000, which is the ideal value in this case.

The SPH solver uses a modified version of the continuity equation, making the density calculation dependent of the velocity field,

$$\frac{d}{dt}\varrho_a = -\varrho_a \sum_{b=1}^{N} m_b(\mathbf{u}_a - \mathbf{u}_b) \cdot \nabla_a W_{ab} \quad (2)$$

which achieves a correct density distribution for the entire field.

This is just one example of the physical modifications implemented in the solver. Other calculations include artifical viscousity, repulsive forces to prevent unphysical particle «clumping» and reinitialization of the particle density each 20th time step to minimize numerical errors. Also, the fluid is treated as slightly compressible, to allow for a so-called «equation of state», i.e. An equation that describes the relation between density and pressure for each particle.

To achieve high accuracy and allow for large time steps, the Runge-Kutta 4 integrator is used, demanding more memory and computation power than typical interactive SPH formulations.

The solver is developed by Dr. Csaba Pakozdi, the author of [3], and has been shown to produce results very near analytical solutions of well known fluid modeling problems (drop case, dam break etc.). This is particularly true for large n, and therefore a fast, parallelized implementation is needed. We're still early in our implementation phase, so no work has been done on profiling and optimization yet.

References:
[1] Øystein E. Krog, *"GPU-based Real-Time Snow Avalanche Simulations"*, Master's Thesis, NTNU, 2010
[2] Øystein E. Krog and Anne C. Elster, *"Fast GPU-based Fluid Simulations Using SPH"*, Extended abstract presented at EuroGPU and PARA 2010
[3] Csaba Pakozdi. "*A Smoothed Particle Hydrodynamics Study of Two-dimensional Nonlinear Sloshing in Rectangular Tanks"*, *PhD thesis, NTNU, 2008.*

Acknowledgements
We would like to thank NVIDIA for their hardware contributions to Dr. Elster's research lab.

NTNU
Norwegian University of Science and Technology

HPC-Lab
Computer & Info. Science
Norwegian University of Science and Technology
http://research.idi.ntnu.no/hpc-lab