Utilization of GPU-Based Smoothed Particle Hydrodynamics for Immersive Audiovisual Experiences

Nora Loimayr

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im November 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, November 26, 2019

Nora Loimayr

Contents

Abstract

Smoothed Particle Hydrodynamics is a particle-based technique used to simulate fluids. Since it is a computationally intensive method, today the required calculations are usually performed on the graphics processor. This accelerates the computations and, depending on the implementation, allows to achieve an interactive frame rate.

The interactive installation *liquidus* makes use of the described method to simulate various liquids. These liquids are projected onto the floor and can be controlled by the position of visitors. As a result, a collaborative, playful experience is created, which makes it possible to learn interesting things about the behavior of various liquids and gases.

SPH features a wide range of methods and strategies that can be used for implementation. In addition, results are strongly influenced by different input variables. These circumstances make it possible to create a functional setup, but there is no guarantee that it can be considered ideal.

This thesis presents several methods used for the implementation of *liquidus*. In the following evaluation relationships between different methods and their parameters are analyzed, where runtime behavior, stability and optical appearance are decisive criteria.

Although the relationships between the parameters are mathematically justifiable, their interdependence can often be difficult for developers of interactive applications to comprehend. This thesis should provide an overview of the interaction of different parameters and thus facilitate the configuration and optimization of different setups, with particular regard to the use case *liquidus*.

Kurzfassung

Smoothed Particle Hydrodynamics ist eine partikelbasierte Technik zur Simulation von Flüssigkeiten. Da es sich um eine rechenintensive Methode handelt, werden die benötigten Berechnungen heute üblicherweise auf dem Grafikprozessor durchgeführt. Dies ermöglicht, die Berechnungen zu beschleunigen und damit, je nach Implementierung, eine interaktive Bildrate.

Die interaktive Installation *liquidus* macht von der beschriebenen Methode gebraucht, um verschiedenste Flüssigkeiten zu simulieren. Diese Flüssigkeiten werden auf den Boden projiziert und können von Besuchern durch deren Position beeinflusst werden. Als Folge entsteht eine gemeinschaftliche, spielerische Erfahrung, die ermöglicht, Interessantes über das Verhalten von verschiedensten Flüssigkeiten und Gasen zu lernen.

SPH weist eine große Bandbreite an Verfahren und Strategien auf, die für eine Umsetzung genutzt werden können. Zusätzlich werden Resultate stark durch verschiedene Eingangsgrößen beeinflusst. Diese Umstände ermöglichen zwar die Erstellung eines funktionstüchtigen Setups, jedoch kann nicht garantiert werden, dass dieses Setup als ideal anzusehen ist.

In dieser Arbeit wird eine Anzahl von Verfahren vorgestellt, die für die Implementierung von *liquidus* eingesetzt wurden. In der anschließenden Evaluierung werden Zusammenhänge zwischen verschiedenen Verfahren und den zugehörigen Parametern analysiert, wobei Laufzeitverhalten, Stabilität und optisches Erscheinungsbild maßgebliche Kriterien sind.

Die Zusammenhänge zwischen den Parametern sind zwar mathematisch begründbar, deren gegenseitige Beeinflussung ist für Entwickler von interaktiven Applikationen jedoch oftmals schwer überschaubar. Diese Thesis soll somit einen Überblick über das Zusammenspiel verschiedenster Parameter bieten und somit das Konfigurieren und Optimieren verschiedener Setups, besonders im Hinblick auf den Anwendungsfall *liquidus*, erleichtern.

Chapter 1

Introduction

The Deep Space 8K is a show room in the Ars Electronica Center in Linz. It is mainly used to present high resolution videos, media art, as well as interactive, co-located installations. The room offers two projection surfaces, one on a wall and one on the floor, resulting in 8K image resolution. Additionally, a laser tracking system can identify twenty and more people moving, allowing to experience various co-located adventures.

liquidus is an interactive, physics-based fluid simulation that was developed specifically for Deep Space. It turns the show room into a playful laboratory and allows

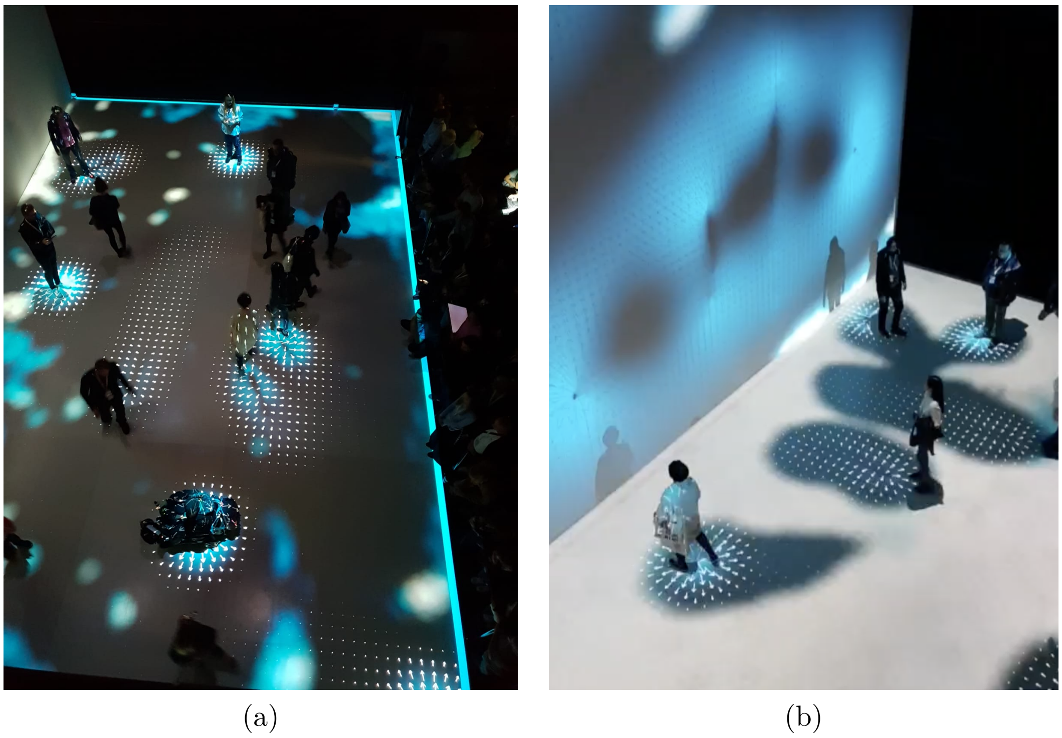


Figure 1.1: *liquidus* presented at Deep Space 8K at the Ars Electronica Festival 2019 in Linz.

exploring various properties of liquids and gases. A particle-based fluid simulation is projected onto the ground and fills up the entire area. Visitors can walk on the floor and interact with the fluid. They can model shapes, try to collectively displace the liquid or simply watch it. Depending on the parameter setup, the fluid can resemble different types of liquids such as water or honey. Additionally, by extracting various parameters from the simulation and feeding them into a sound synthesis module, a soundscape that supports the overall visual experience is generated.

liquidus can be visited freely. It serves as a playful application that gives people understanding of the behavior of fluids. By playing around with the provided fluids, people can get a deeper understanding of their behavior as well as physical theory behind them. Figure 1.1 provides two images that were taken at the premiere of *liquidus*[1] at the Ars Electronica Festival 2019[2].

1.1 Problem Description

This thesis focuses on the implementation of the fluid simulation used for *liquidus*. As method to simulate fluid dynamics at interactive frame rates, Smoothed Particle Hydrodynamics (SPH), a particle based approach to simulate fluids, was chosen. This method, however, offers a large range of implementation strategies and variations.

Furthermore, it is a simulation method whose quality, accuracy as well as behavior are strongly dependant on the selection of values for numerous parameters. Additionally, the interdependence of SPH-parameters is very high. Changing one parameter's value affects the influence of another parameter's value. As a result, the variety of possible setups is large.

The circumstances described allow to find a suitable, running setup using a number of selected procedures. However, it is by far not guaranteed that the resulting setup can be seen as ideal. This thesis therefore focuses on the determination of an ideal combination of methods, so that for the described use case best results are attained. Furthermore, it will be evaluated how said methods influence performance.

1.2 Requirements and Goals

One major requirement of the application is its real-time capability, meaning a stable frame rate from at least 60 frames per second is targeted. Since the projections in Deep Space generally run with a refresh rate of 120 Hz, a frame rate of 120 frames per second is preferable. Doubling the frame rate leads to a reduction of the time frame given for the simulation as well as for the rendering process by half.

Another requirement is to ensure the stability of the simulation. When visitors interact with the fluid the simulation must remain stable, no matter how fast visitors move or in which direction. The movement of visitors can generate undesirable forces that might have crucial impacts on the simulation. In order to provide a stable simulation, integration methods and the used delta time have to be chosen wisely, since they are

[1]https://ars.electronica.art/outofthebox/liquidus/
[2]https://ars.electronica.art/outofthebox/

responsible for the accuracy of the simulation. Choosing them wrong could lead to the need of compensating their effects elsewhere.

Furthermore, the simulation needs to be configurable at run-time. This allows experimenting with SPH-parameters during test scenarios, as well as exploring the influence they have on the simulation interactively.

In the end, a compromise between accuracy and interactivity must be found. The more accurate the simulation is, the longer will be the time a simulation step takes to be computed. As a consequence, simulations that are very accurate might run with a low frame rate, while simulations that profit from a stable, interactive frame rate might induce stability problems or provide erroneous results.

Throughout the thesis, a selection of procedures used to implement a fluid simulation based on the SPH concept is defined. The main goal of the thesis is to evaluate to which extent those procedures influence the quality of the simulation, both performance- and stability-wise. Furthermore, it is attempted to find an ideal combination of those procedures and define and justify the reasons of such. In order to provide a basis the evaluation can be built on, performance data is gathered. The collected data will further be used to draw conclusions about an ideal setup of the simulation.

1.3 Outline

In Chapter 2 a basic but precise overview about the topic computational fluid dynamics is offered, with special regard to the particle-based approach Smoothed Particle Hydrodynamics. Furthermore, installations that resemble *liquidus* are presented and discussed. This chapter serves as foundation to the following chapters and mainly focuses on the theoretical and physical background of fluid simulations. Chapter 3 presents how the simulation was eventually implemented. Thus, several implementation details are provided and justified. A strong focus as well lies on the GPGPU implementation of the simulation. In Chapter 4 a number of optimization techniques are introduced. Particularly neighbor search and time integration are relevant topics. In Chapter 5 the applied optimization measures are evaluated and discussed. Finally, Chapter 6 provides the conclusion of the thesis.

Chapter 2

Fundamentals and State of the Art

This chapter sums up and explains fundamental knowledge of the topic computational fluid dynamics. Special regard will be given to Smoothed Particle Hydrodynamics, a particle-based concept to simulate fluids, since it was chosen as an implementation method. Furthermore, a number of installations comparable to *liquidus* are presented. They closely resemble *liquidus* due to their focus on topics as fluid dynamics, particle systems or interactivity.

2.1 Computational Fluid Dynamics

Fluids, in particular liquids, gases and plasma, have always surrounded human beings. Rivers floating down steep hills resulting in massive waterfalls, oceans set in motion by the power of nature, puddles that evaporate and turn to gas – destined to eventually come back down to the ground when turning into rain. It is by far no coincidence that mankind has strived to gain knowledge about fluids and their physical origins. The field of computational fluid dynamics (CFD), is a sub-branch of the field of fluid dynamics, and uses the numerical analysis of *fluid flow* in order to simulate the behavior of liquids and gases that are in motion.

2.1.1 Application Areas

The field of CFD has been studied extensively in the last decades. Today, the range of applications covers artistic, but also scientific areas. CFD provide assistance when designing objects such as aircrafts, cars and ships. Simulations help to ensure safety and efficiency of the designed objects, since the objects are exposed to fluid flow throughout their lifetime. Furthermore, a lot of numerical techniques contribute a major part to weather forecasts. They help to understand and predict real-life weather scenarios such as heat transfer. Additional fields of applications are e.g., astrophysics, meteorology and oceanography [2, p. 2].

Another major application field is computer animation, where the goal is to generate realistic animations using fluid simulations. Examples are smoke, water, fire, and various other liquids or gases. These simulations find place in computer generated movies, architectural visualizations, or applications with interactive frame rates such as computer games and other interactive media.

As opposed to the use of CFD in mechanical engineering, in terms of animation visual credibility and fast simulation cycles generally are of higher relevance than accuracy [17]. This is due to frame rates usually running somewhere between 24 and 60 frames per second. Particularly real-time applications require a stable and minimized update rate. In order to increase performance, acceleration methods such as the use of graphics cards, are an active area of research.

2.1.2 Domain Discretization

The mathematical equations that describe fluid flow are usually so complex that by using analytical methods only, no solution will be obtained. Therefore, in the field of CFD, numerical analysis is performed, with the aim to receive approximate but accurate results. First, the domain is discretized: The fluid continuum is represented by discrete elements. Subsequently, a numerical algorithm can be developed and implemented in order to solve the problem [11].

There are two fundamental domain discretization methods when tracking a fluid's motion: The *Eulerian*, a grid-based method, as well as the *Lagrangian*, a particle-based method. Within the Eulerian viewpoint the fluid is represented by a number of grid cells with fixed locations, whereas within the Lagrangian viewpoint the fluid is partitioned, resulting in a combination of particles with variable locations. The difference between those two methods is demonstrated in Figure 2.1.

Both Lagrangian and Eulerian concepts are widely used in the field of CFD. Particle-based solutions are typically better suited for real time applications, since they are based on approximations rather than on solving linear systems of equations and therefore require less computational effort. Once the concept has been understood, they are comparatively straightforward to implement. Furthermore, they do not present any spatial restrictions. One major drawback of particle-based methods is the lesser accuracy as opposed to grid-based methods. This is why a large application area of Lagrangian methods is the game industry, where interactivity (and therefore performance) has priority over accuracy.

2.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics is one of the most known particle-based approaches to simulate fluids, and therefore belongs to the Lagrangian viewpoint. The concept was first brought up 1977 by Gingold and Monaghan [6], originally for solving astrophysical problems. Later, it was also extended to be used for Solid Mechanics by Libersky and Petschek [10] and actively discussed in the area of CFD. Today, the SPH method is also used in the field of computer animation, i.e., in commercial products as X-Particles[1].

The following sections provide a compact but precise overview of the SPH technique and its theoretical background, as well as information about important properties as density, pressure or viscosity.

[1]A used within the 3D modeling application Cinema 4D.

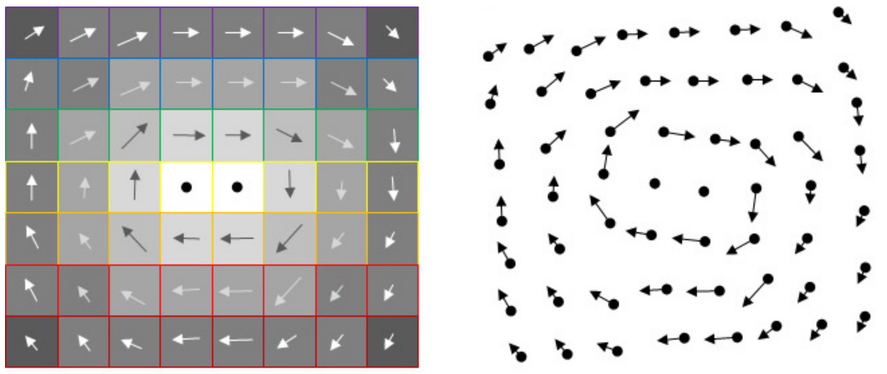


Figure 2.1: Eulerian (a) and Lagrangian (b) viewpoint. The Eulerian approach tracks fluid information at fixed locations, represented by grid cells, whereas the Lagrangian approach treats the continuous fluid as a discrete particle system with variable locations. Source: [27].

2.2.1 General Concept

The main concept of SPH is the discretization of a fluid resulting in a number of elements called particles. Each element has a number of properties, among them also phyical quantities. The expression *smoothed* originates from the idea to use a smoothed distribution of said quantities over a specific area in order to define the quantities of a certain point of interest. This results in an approximation of the continuous field by using finite data points [9]. Consequently, field quantities with any location in space can be calculated. In order to create a smooth distribution a smoothing kernel is used. Fundamentally, the properties are weighted by this kernel, depending on their spatial distance from the point of interest. The size of the area of interest is defined by the smoothing radius.

Figure 2.2 illustrates how the smoothing radius h defines the distribution area and hence also the neighboring particles. Each particle which is located within the smoothing radius lies inside the influence domain and therefore is smoothed by the kernel. Particles outside the influence domain are ignored.

SPH Interpolation Model

The idea of SPH interpolation is to approximate any physical quantity A at any location $\mathbf{r}$ by looking at neighboring particles. This is done by creating a weighted sum of the contributions from all neighbor particles. The SPH model therefore is defined as

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h), \tag{2.1}$$

where j iterates over all particles, m_j is the mass of particle j, $\mathbf{r}_j$ its location, A_j and ρ_j the field quantity and the density field at $\mathbf{r}_j$, respectively. W denotes the smoothing kernel and h the smoothing radius [14].

While the interpolation location $\mathbf{r}$ generally could be set to be at any location in space, for building fluid simulations field quantities are solely calculated at particle

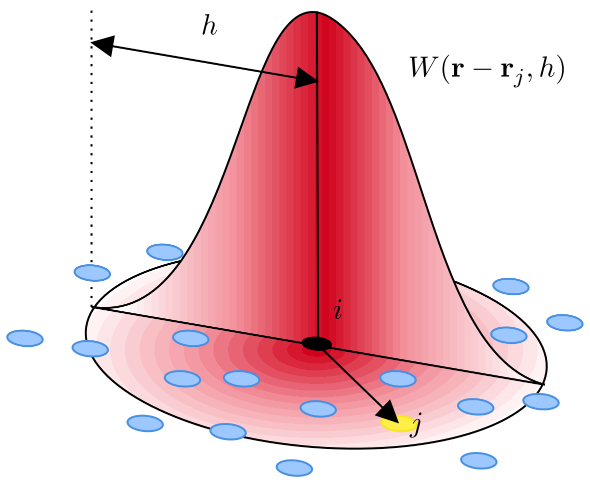


Figure 2.2: The principle of SPH. The contributions of each particle are weighted using a smoothing kernel h. Source: [25].

locations and assigned to the associated particles. This turns particles to discrete entities representing a continuous system and eventually allows updating particle locations over time.

Smoothing Kernel

The smoothing kernel is a weighting function that allows to estimate fluid quantities for any location. Figure 2.2 shows that contributions of particles close to the interpolation location are larger than contributions of particles further away from the interpolation location. Particles outside the influence domain do not contribute to the estimation. To put this mathematically, the function value has its peak at the center point and decreases constantly while moving further away from it, until finally fading to zero. Theoretically, any function could be used as kernel function. In [12] certain constraints are pointed out that each kernel needs to fulfill. A selection of the most relevant ones is listed and explained in the following:

- The kernel function must be even. A kernel function is even if

  $$W(\mathbf{r}) = W(-\mathbf{r}), \tag{2.2}$$

  meaning it is symmetric with respect to the y-axis. A symmetric kernel ensures equal weighting of properties in all directions.
- The kernel function must be normalized. A kernel is normalized, if

  $$\int W(\mathbf{r})d\mathbf{r} = 1, \tag{2.3}$$

  meaning the integral over its full domain always yields 1. At this point, it must be stated that a 2D-kernel cannot be used for 3D simulations, since the integral would

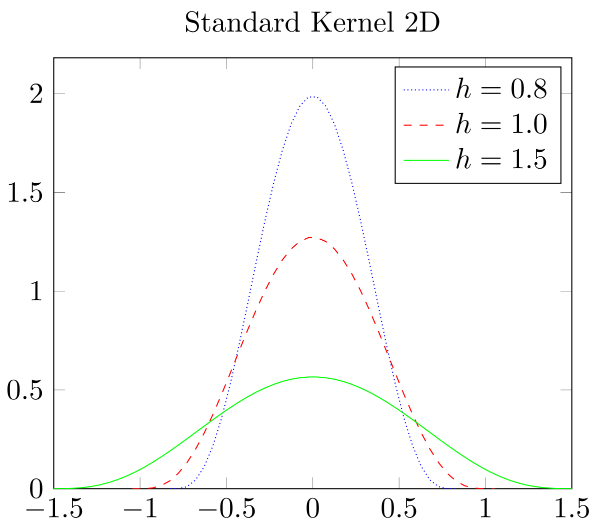


Figure 2.3: A Gaussian smoothing kernel with different smoothing radii. The smoothing radius defines the severity of the particles contributions. While a large smoothing radius lessens the contribution of very close particles, a small smoothing radius leads to stronger weightings.

not yield 1 in a different dimension. By multiplying the kernel with a different normalization constant it can be modified so it satisfies this condition.

- The kernel function must have finite support. A kernel function has finite support if

$$\begin{cases} W(\mathbf{r}-\mathbf{r}_j,h) \neq 0 & \text{for} \quad \left|\mathbf{r}-\mathbf{r}_j\right| < h, \\ W(\mathbf{r}-\mathbf{r}_j,h) = 0 & \text{for} \quad \left|\mathbf{r}-\mathbf{r}_j\right| \geq h, \end{cases} \tag{2.4}$$

  meaning it is zero outside a compact set.

In literature, various different smoothing kernels have been proposed. There is no consensus about a perfect kernel function, since they bring a trade-off between accuracy and computational costs with them. For specific kernel types and a discussion of their effects on the simulation, see Section 3.2.1.

Smoothing Radius

The smoothing radius defines the area of interest and therefore the subset of particles that are treated as neighbors. A particle j is a neighbor of particle i when $\left|\mathbf{r_i}-\mathbf{r_j}\right| \leq h$, meaning their spatial distance is less or equal the smoothing radius.

Additionally, the smoothing radius specifies the fall-off of the kernel. Figure 2.3 shows that while increasing the smoothing radius, the amplitude of the kernel is reduced significantly. The larger the radius, the lesser is the impact of close particles, in comparison to kernels with a smaller radius. This makes the choice of the kernel size resolution-dependent, since a high number of particles would result in very high property summations.

2.2.2 Moving the Particles

In order to move the particles, the acceleration of each particle has to be calculated first. The Navier-Stokes equations for incompressible flow describe the motion of fluids:

$$\rho \frac{D\mathbf{v}}{Dt} = \underbrace{-\nabla p}_{\text{pressure}} + \underbrace{\mu \nabla^2 \mathbf{v}}_{\text{viscosity}} + \underbrace{\rho \mathbf{g}}_{\text{external}}, \tag{2.5}$$

where μ describes the viscosity factor and $\mathbf{g}$ external body accelerations. In literature, a variety of versions of the Navier-Stokes equations exist. Equation 2.5 is taken from [14]. The term on right-hand side of the equation can be seen as a composition of three force fields: the pressure field ($-\nabla p$), the viscosity field ($\mu \nabla^2 \mathbf{v}$) and an external force field ($\rho \mathbf{g}$). Summing up those force fields yields in the change of momentum $\rho \frac{D\mathbf{v}}{Dt}$ which is situated on the left-hand side of the equation. With that said, the acceleration of a fluid particle can be represented by a summation of pressure, viscosity and external forces, since

$$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{f}_i}{\rho_i}. \tag{2.6}$$

Summing up the partial forces, namely pressure force, the viscosity force and potential other external forces, eventually yields the net force that acts on each particle:

$$\mathbf{f}_i = \mathbf{f}_i^{\text{pressure}} + \mathbf{f}_i^{\text{viscosity}} + \mathbf{f}_i^{\text{external}}. \tag{2.7}$$

The calculation of those quantities generally follows the SPH interpolation model explained in Section 2.2.1: Each quantity is approximated by a summation of weighted contributions of neighboring particles. However, in certain cases calculations are slightly modulated in order to guarantee what is called *force symmetry.* Newtons third law of motion states that there is always an opposed and equal reaction to each action, meaning that forces always come in equal pairs. This also holds true for interacting particles. When the number of particles interacting is limited to two, SPH force calculations could yield different results [9]. An example scenario is demonstrated in Figure 2.4. Consider there are two particles i and j in a system and the quantity A is to be interpolated at both particle positions. For calculating the force acting on particle i, we use

$$\mathbf{f}_i = m_j \frac{A_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{2.8}$$

and for calculating the force acting on particle j

$$\mathbf{f}_j = m_i \frac{A_i}{\rho_i} W(\mathbf{r}_j - \mathbf{r}_i, h), \tag{2.9}$$

respectively. In general, the weighted contributions A_i and A_j are not said to be equal. As a consequence, with A_i and A_j not being equal, both calculations yield different forces and result in an asymmetry of forces:

$$A_i \neq A_j \Rightarrow \mathbf{f}_i \neq \mathbf{f}_j. \tag{2.10}$$

In order to avoid such asymmetries, forces are usually symmetrized. Various methods have been proposed in literature, this thesis refers to symmetrization methods described in [14]. Since the methods are different depending on the quantity that is calculated, they are explained individually for each quantity in the relevant sections.

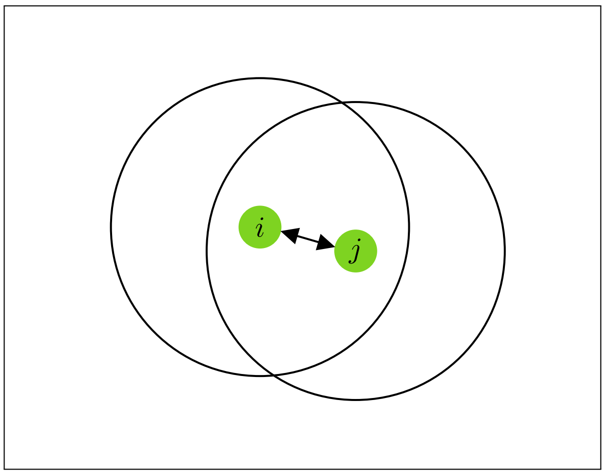


Figure 2.4: The problem of two forces not being symmetric. Having only two particles i and j in the system, the force calculated with respect to each other can result in different vectors, depending on which particle is the particle of interest.

2.2.3 Field Calculations

A precise explanation of the calculation of the density field, as well as the force density fields pressure and viscosity is given in the following.

Density Field

The density ρ is a physical quantity that describes weight per unit volume. In SPH, each particle i can be seen as a representation of a fluid volume V_i. Therefore, the density at the particle's location can be defined as

$$\rho_i = m_i/V_i. \tag{2.11}$$

In order to approximate the density at particle locations, the SPH interpolation model is used. Since the density itself is a prerequisite for SPH interpolation, it needs to be evaluated each time-step, before calculating other quantities. The density $\rho(\mathbf{r})$ at particle location $\mathbf{r}$ is obtained by substituting the density quantity into Equation 2.1, resulting in

$$\rho(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{2.12}$$

Due to cancelling out the density part ρ_j/ρ_j of the equation, the prerequisite of knowing the density when calculating the density can be circumvented. As a result, the density calculation merely remains a summation of weighted particle mass contributions – an approximation of the density calculation in Equation 2.11.

Pressure-Gradient Force Field

Pressure is a physical quantity that defines the force that acts on a surface per unit area $p = F/A$, where p denotes pressure, F the force and A the unit area. When there is an imbalance in pressure, meaning a difference in pressure from one location to another, a force that redresses this imbalance is implied: The pressure-gradient force ∇p, which is always pointing from high-pressure regions to low-pressure regions. This force is what is responsible for e.g., wind, since it causes air to move across different pressure regions. In terms of particles, the pressure-gradient force occurs when having higher pressure on the one side of a particle than on the other. The resulting force directly influences the acceleration of the particle and tries to balance the pressure.

To calculate such force, a vector that defines the rate and direction of change in a scalar field is used. It is called gradient operator in two dimensions and is defined as

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x}(\mathbf{x}), \frac{\partial f}{\partial y}(\mathbf{x}) \right), \tag{2.13}$$

which is a vector composed of the partial derivatives in all directions. The derivatives return the tangents of a given field at a given location. Therefore, the pressure gradient ∇p is pointing to the direction of the steepest ascent. However, as mentioned above, the result is supposed to be a field that is pointing away from high-pressure regions and towards low-pressure regions. Therefore, the negative gradient of the pressure field is used, denoting the pressure-gradient force $-\nabla p$, where p is the pressure at particle location. Substituting $-\nabla p$ into the SPH interpolation rule described in Equation 2.1 leads to the following approximation of the pressure-gradient force field:

$$\mathbf{f}_i^{\text{pressure}} = -\nabla p(\mathbf{r}_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{2.14}$$

As previously described in this section, the resulting forces are not symmetric. In order to symmetrise them, the plain use of the arithmetic mean of both pressure quantities of interest is proposed by [14] and yields:

$$\mathbf{f}_i^{\text{pressure}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{2.15}$$

It is a commonly used alternative to other, more costly methods.

In order to eventually interpolate the pressure-gradient force, the pressure quantity p has to be evaluated first. Since a deeper description of pressure calculations would go beyond the scope of this thesis, they are outlined only briefly. For a more precise explanation the references can be consulted.

In the SPH literature, various methods to calculate p have been used. Most methods make use of an *equation of state* – equations that present the relation between different state variables, as e.g., pressure and density. Pressure depends highly on density, since high density leads to high pressure. One of the most common equations used was proposed by [4]. It is a modified version of the ideal gas state equation and is defined as

$$p = k(\rho - \rho_0), \tag{2.16}$$

where k is the gas constant, which determines the temperature increment per mole, and ρ_0 is the rest density. While this method has the benefit of being computationally quite inexpensive, it results in high compressibility.

A method that enforces low density variations is the use of the Tait equation. It was proposed by [1] and involves the speed of sound of the fluid in pressure calculations. A completely different approach was proposed by [3], where a pressure Poisson equation was solved to compute the pressure term in order to lessen compressibility, which also is a common technique in the field of Eulerian fluid solvers.

Viscosity Field

Viscous fluids are fluids that resist deforming. The intensity of this resistance is defined by the viscosity: While honey has a high viscosity, water is a less viscous fluid. In terms of particles, we speak of the viscosity force (compare to Equation 2.5), which is denoted as $\mu\nabla^2\mathbf{v}$. The viscosity force acts as a damping force, since it aims to minimize the differences in velocity between the particles, by measuring the difference in velocity to the average around it. To calculate the viscosity force, the Laplacian operator in two dimensions

$$\nabla^2 f(\mathbf{x}) = \nabla \cdot \nabla f(\mathbf{x}) = \frac{\partial^2 f(\mathbf{x})}{\partial x^2} + \frac{\partial^2 f(\mathbf{x})}{\partial y^2}. \tag{2.17}$$

is used. It measures how far a quantity is from the average around it. In order to calculate the viscosity force field, again the SPH rule is applied and the viscosity term $\mu\nabla^2\mathbf{v}$ is substituted into Equation 2.2, which yields in

$$\mathbf{f}_i^{\text{viscosity}} = \mu\nabla^2\mathbf{v}(\mathbf{r}_i) = \mu\sum_j m_j \frac{\mathbf{v}_j}{\rho_j}\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{2.18}$$

where μ is the viscosity factor of the fluid. The higher this factor is, the more viscous is the fluid. When $\mu = 0$, the fluid is not viscous at all. It is then referred to as superfluid.

Again, this calculation yields asymmetric forces. Since the viscosity forces are only dependent on absolute velocities, one way to symmetrize them is by using velocity differences, resulting in

$$\mathbf{f}_i^{\text{viscosity}} = \mu\sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j}\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{2.19}$$

External Force Fields

In addition to the force fields already mentioned, external force fields can be included in force calculations as well. An external force field is defined as

$$\mathbf{f}_i^{\text{external}} = \rho_i \mathbf{g}, \tag{2.20}$$

where $\mathbf{g}$ is an external body acceleration at particle position. An example for such force field is the earth's gravity. Since the acceleration due to gravity by definition yields around 9.8 at all field positions, the gravity force field at particle position can be calculated as $\mathbf{f}_i^{\text{gravity}} = \rho_i \cdot 9.8$ and therefore does not bring up the need of performing SPH interpolation on the field.

2.2.4 Simulation

Time Integration

After summing up the previously computed forces, the acceleration of each particle needs to be evaluated. For each particle, the resulting force is divided by the particle's density as

$$\mathbf{a} = \frac{\mathbf{f}}{\rho}, \tag{2.21}$$

where $\mathbf{a}$ is the new acceleration of the particle and $\mathbf{f}$ is the result of the force addition. After calculating the new acceleration of each particle time integration is performed, in order to compute the new particle positions. A quick and straight-forward to implement method is the Semi-Implicit Euler Integration. However, in the SPH literature, the common integration scheme is Leap Frog integration [14].

Time integration also involves choosing a suitable delta time for the use case. Generally speaking, the smaller the delta time is, the more exact is the simulation. When the delta time is chosen too small, severe stability problems can occur. However, small time steps result in multiple simulation iterations per frame. In concrete terms, this means that SPH calculations are performed multiple times per frame. Since they can not be seen as efficient calculations, interactivity might suffer from it. Again, this results in a decision problem between accuracy and interactivity.

Simulation Loop

The theoretical background that is needed to create a basic SPH simulation was described in the previous sections. This section section sums up the SPH algorithm and gives a list of the steps that are required in order to perform one simulation step.

1. Calculate the needed properties. For each particle i do:
   (a) compute the density ρ_i,
   (b) compute the pressure p_i.
2. Compute the forces and update the particle's position. For each particle i do:
   (a) compute the pressure force $\mathbf{f}_i^{\text{pressure}}$,
   (b) compute viscosity force $\mathbf{f}_i^{\text{viscosity}}$,
   (c) compute additional external forces $\mathbf{f}_i^{\text{external}}$,
   (d) compute the total force $\mathbf{f}_i = \mathbf{f}_i^{\text{pressure}} + \mathbf{f}_i^{\text{viscosity}} + \mathbf{f}_i^{\text{external}}$,
3. Perform time integration and handle collisions. For each particle i do:
   (a) compute the particle acceleration $\mathbf{a}_i = \mathbf{f}_i/\rho_i$,
   (b) perform time integration to get the new particle position $\mathbf{x}_i$,
   (c) handle possible border collisions,
4. Render the particles.

For reasons of simplicity, the specification of a neighbor search method and a time integration scheme was left aside.

Fixed-Radius Near Neighbor Search

When performing SPH interpolation, each particle's neighbors that are located within a fixed radius, specifically the smoothing radius, have to be looked for. This variant of the nearest neighbor search problem is referred to as *fixed-radius near neighbor search.* Since particle locations can change each simulation step, neighbors need to be re-evaluated each simulation step as well. The resulting costs are always dependent on the number of particles n, which generally is aimed to be maximized, for reasons of accuracy or visual appearance. As a consequence, neighbor search is the main bottleneck in the field of SPH. Therefore, in order to avoid time-consuming pre-simulation cycles, as well as performance problems within real-time applications, an efficient neighbor search algorithm is required.

2.3 Similar Installations

This section covers several installations that show similarities to *liquidus* of technical as well as conceptual nature, with particular focus on connecting relevant topics such as fluid dynamics, particle systems, real-time applications as well as interactivity.

2.3.1 Au-delà des Limites

The Japanese collective *teamLab* focuses on what they call *ultrasubjective spaces* – spaces representing images that can be entered by viewers. Viewers can move around freely and interact with the artwork and therefore can become part of it. As opposed to traditional Japanese paintings, which are generally described as flat, the collective creates immersive art that diminishes the boundaries between viewer and the artwork [28].

Au-delà des limites is an interactive installation by teamLab that follows this exact concept. As the french title suggests, viewers can go *beyond their limits* and become part of the installation. The installation itself is based on a real-time, interactive particle system representing waterfalls. When visitors move around on the ground, they displace the water and flowers start blooming instead. The resulting artworks are non-reproducible. Figure 2.5 shows a picture of the installation.

2.3.2 Fluid Structure 360

Fluid Structure 360 is an interactive installation created by Vincent Houzé. It takes place in a room of which all four walls are acting as a canvas, displaying the motion of a surreal fluid. The main intent is to visualize the abstract shape's reaction to various impulses. Visitors can manipulate the fluid by getting closer to it. It reacts to their body movements. The fluid behaviour is, as the artist states, based on a state of the art fluid solver [23]. Unfortunately, no further implementation details were accessible. Figure 2.6 shows a picture of its premiere in Paris.

2.3.3 Particle-Based Granular Synthesis

Particle-based Granular Synthesis is a sound synthesis prototype by Tadej Droljc. The idea was to use particle positions of a three-dimensional particle system as input pa-

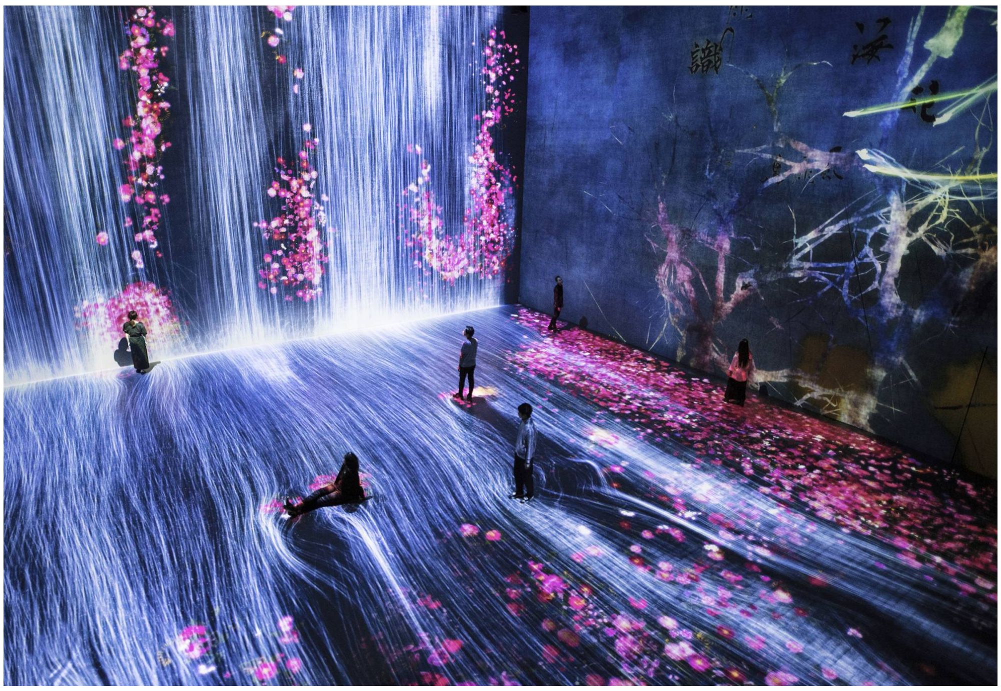


Figure 2.5: *Au-delà des limites* by *teamLab*. An interactive installation where visitors can interact with waterfalls consisting of particles. Source: [29].

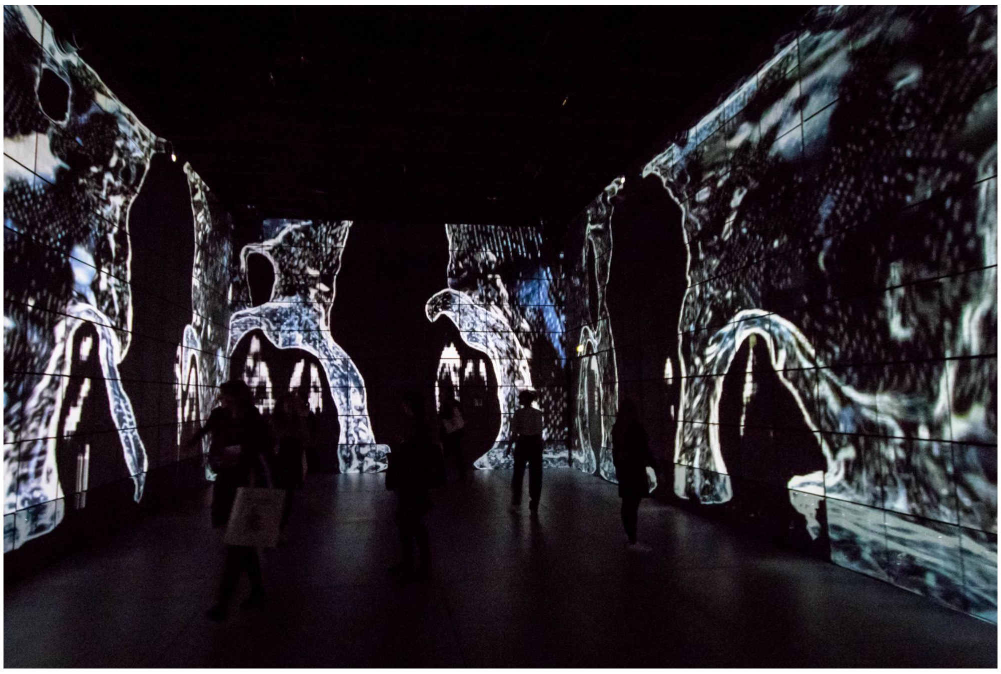


Figure 2.6: *Fluid Structure 360* by Vincent Houzé. An immersive installation where viewers can interact with fluids. Source: [23].

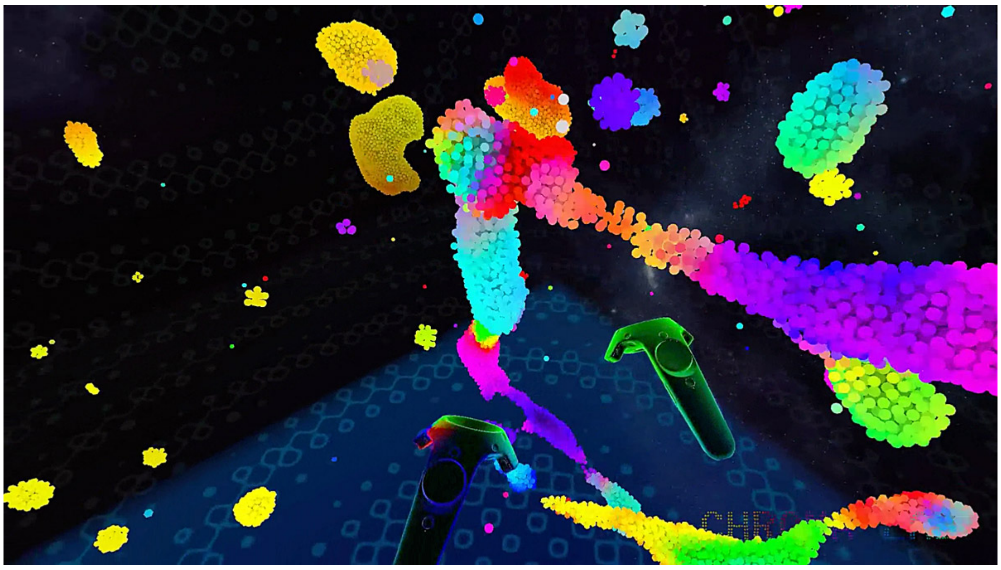


Figure 2.7: Screenshot of the VR-game *Chroma Lab*. Source: [19].

rameters for a granular sampler. The implemented particle system holds 16 particles. An audio sample presenting a female voice is split in 16 grains, of which each grain is assigned one specific particle. The x-coordinate of the particle controls the stereo panning of the grain, the y-coordinate pitch and the z-coordinate the sample reading position, respectively [24].

2.3.4 Singing Sand

Singing Sand is an installation conceptualized and implemented as well by Tadej Droljc. It can be seen as the final outcome of the above described prototype. Particles are seen as sand grains that create sound by providing parameters to a granular synthesizer. Particle velocities determine the particles' color and therefore the simluation's visual appearance. The sequel of the installation *Singing Sand 2.0* was intentionally created for the Deep Space in the AEC and rendered offline in stereopairs [26].

2.3.5 Chroma Lab

Chroma Lab is a virtual reality application providing a physics-based particle sandbox. The player can accumulate a large amount of particles with fluid behavior around them and play with it. The player is offered various possibilities of interacting with the particles, such as picking them up, hitting or shooting them, or simply give them a color. The application is based on a custom physics engine that runs on the GPU in order to ensure to achieve the sophisticated frame rates of today's virtual reality displays [20]. Figure 2.7 shows a screenshot from the application.

Chapter 3

Implementation Details

This chapter provides an explanation of the basic application structure, broken down into three parts: the particle system, with the main focus on its GPGPU implementation, the fluid system, and eventually the application's interactivity concept. It covers details that were crucial for the implementation process. Additionally, it offers justifications for the implemented schemes.

3.1 GPU-Based Particle System

The application was implemented using POWIDL[1]. Since POWIDL itself is a high-level library, mainly the underlying POWIDL-Win, the API between the application and the operating system Windows, had to be modified. It was expanded by the functionality of GPU-based particle systems, providing the ability of various adjustments regarding emitters, behaviors of particles and others.

In order to implement the particle simulation exclusively on the GPU, an API dedicated for *general-purpose computing on graphics processing units* (GPGPU) needed to be chosen. The decision was made in favor of Microsoft DirectCompute, because it is part of the Microsoft DirectX API collection and therefore lies very close to the rendering pipeline of the application.

3.1.1 GPGPU Execution Model

The particle simulation is implemented using GPGPU acceleration. Particle properties are therefore updated in parallel. Since data transfer between CPU and GPU would slow down the process significantly, particle data is stored in video memory only. The simulation as well as all calculations related to it run exclusively on the GPU. A computational overhead due to data transfer can be avoided this way.

Throughout each update cycle, properties such as position, color, and age are evaluated per particle by dispatching a number of compute shaders: The shaders are able to fetch particle data from buffers storing required information as, for example, particle states. When dispatching a compute shader, the number of threads that should be invoked has to be defined. Within the basic particle system as many threads as particles

[1]A development library for mainly interactive applications (http://quantumreboot.com/powidl).

Name	Type	Description
`pos`	`float3`	Position and angle
`ttl`	`float`	Time-to-live in seconds
`age`	`float`	Current age, counting down from ttl to zero
`acc`	`float3`	Acceleration and angular acceleration
`dns`	`float`	Density field at particle location
`prs`	`float`	Pressure field at particle location

Table 3.1: Properties and storage of `PointParticle2D`

are started. Each thread then is assigned to a single particle and is responsible to update its properties. This way, large numbers of particles can be simulated while maintaining an appropriate frame rate.

After processing each particle, the particle information is passed to the rendering stage of the application, which is responsible for converting particle locations and additional information into pixels and drawing the particles eventually.

3.1.2 Data Flow

Particle Data

In this implementation, each particle holds six properties to define its current state: both position and acceleration are key simulation properties. Position defines the final rendering location. Acceleration serves as basis for time integration, providing essential information for updating the position. The density- as well as the pressure-field are properties that need to be cached per particle during SPH fluid simulation. Additional state properties are the time-to-live as well as the particle age. They can be used for age-dependant applications, such as visualization. Also, they are needed for particle emission. A representation of the particle structure `PointParticle2D` is shown in Table 3.1.

Particle States

Particle data is stored in video memory, specifically in the buffer `ParticleStates`. The buffer contains a structure `PointParticle2D` for each particle. Compute shaders can obtain access to the particle states buffer. They can fetch a specific particle's state by accessing the buffer with the particle ID as index.

The particle states buffer is created having the size of the maximum amount of particles possible and cannot be changed in size dynamically. Therefore, dead as well as alive particles are stored in the buffer at the same time. The reason for such implementation is performance: Since individual memory allocation for each particle being spawned would yield in high perfomance costs, memory is allocated beforehand. Consequently, particle states only have to be written and read.

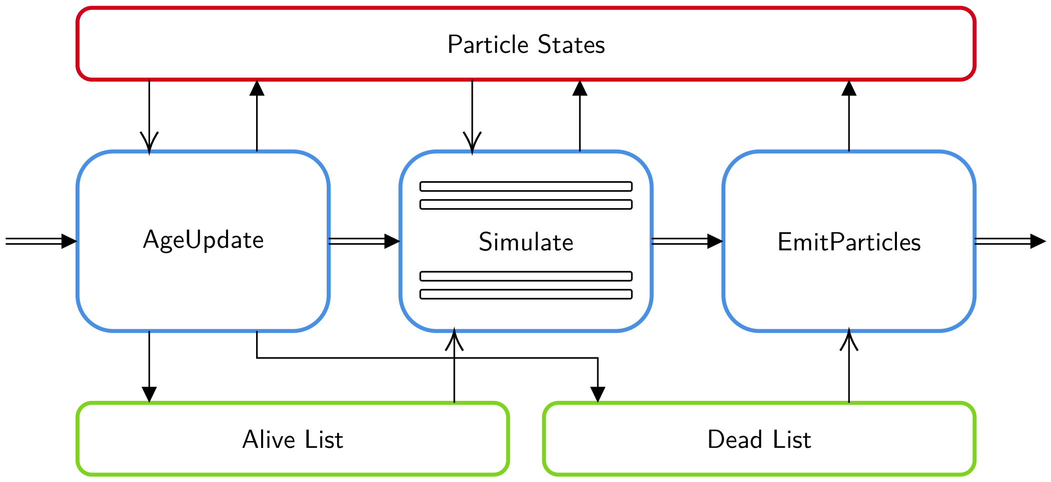


Figure 3.1: Visualization of the emission and destruction process: The shader `AgeUpdate` reads and updates particle states (PS). Additionally, it handles alive- and dead-list creation. Next, the main simulation part is performed by multiple shaders. Eventually, the shader `EmitParticles` sets dead particles back to being alive and therefore triggers their emission.

Particle Emission

The main structure of the simulation is defined by the emission and destruction process. Each simulation cycle, a number of compute shaders acts on alive particles and updates their states, while dead particles are disregarded. Therefore, a means to differentiate between alive and dead particles is required. A simple solution would be to have each compute shader determine if the particle of interest is dead or alive. However, performing the same determination process within each shader would be repetitive and result in inefficiency. Another solution would be to have one compute shader determine which particles are alive and copy their states in a separate buffer, exclusively containing states from alive particles. This as well would be inefficient, since copying data is a time-consuming task. Also, it would be a waste of memory, since the same particle states would be stored twice. To avoid this, the buffers `AliveList` and `DeadList` were introduced. The alive-list list stores particle IDs of active particles, the dead-list of inactive particles, respectively. Shaders then primarily access particles by using those lists. The resulting emission and destruction process can be broken down into three major parts.

1. First, dead- and alive-list are filled up with particle IDs. This is done by the shader `AgeUpdate`. Relevant particle information, as particle age and time-to-live, is read from the particle states buffer. Using this information, each particle's age is updated and written back to the buffer. When a particle is considered alive, its ID is appended to the alive-list. When it is considered dead, it is appended to the dead-list, respectively.
2. Next, the main part of particle simulation is performed. Particle information is read from buffers and updated. This is the part of the application, where particle accelerations are updated due to forces resulting from, for example, fluid behavior

or user input. This also involves the calculation of said forces. Subsequently, the particle accelerations are integrated and positions can be updated. In practice, This process is split in many small parts and each part is executed by a separate shader.

3. At last, the actual emission process is performed. The shader `EmitParticles` is started with as many threads as particles should be emitted. Each thread pulls index values from the list and thereby consumes one particle ID stored in the dead-list. The particle ID allows access to the particle information, using the particle states buffer. The dead particles are set back to alive. Later, during the rendering stage, particles declared as dead will not be rendered.

Figure 3.1 provides a visualization of the process outlined above: Both AgeUpdate and EmitParticles represent one single shader unit each. Simulate represents a collection of shaders that are executed one after another, e.g. shaders responsible for fluid dynamics, etc.

3.1.3 Simulation

Integration Scheme

As first approach to perform time integration, the Semi-Implicit Euler method was chosen as integration scheme. It is a first order integration method that profits from its ease of implementation, as well as its computational efficiency. According to the scheme, the updated values for $\mathbf{v}_{n+1}$ and $\mathbf{x}_{n+1}$ are calculated as

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_n + \mathbf{a}_{n+1} \cdot \Delta t,\\ \mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_{n+1} \cdot \Delta t,\end{aligned} \tag{3.1}$$

where $\mathbf{v}$ denotes the velocity, $\mathbf{a}$ the acceleration and $\mathbf{x}$ the position of a particle at step index n, and Δt is the time step of the integration. When all forces are calculated and summed up, a particle's acceleration can be calculated. The integration calculations then are performed at the end of each simulation step.

Variable Time Step

When using a plain fixed time step, meaning performing time integration with, for example, $\Delta t = 1/60$, it can bring up problems when the refresh rate of the display does not equal the delta time of the simulation, in this case 60 Hz. The simulation will not match to the frame rate. Such scenario might occur when the simulation is performed on a machine that is not powerful enough to update and render the simulation in a time frame of up to 60 frames per second, or simply when vertical synchronization (VSYNC) is turned off, since the display refresh rate then might exceed the 60 Hz [22].

In order to attain frame time independent simulation steps, a variable time step was implemented. The duration of the previous frame is measured and used as delta time for simulating the upcoming frame. As a result, the simulation always matches the refresh rate of the display.

Boundary Treatment

In order to prevent particles from leaving the screen, a boundary collision detection method had to be implemented. As boundary treatment method a classical border collision system was chosen. Four solid borders – one on the top, bottom, left and right each – can be defined. Having performed time integration, each particle is checked for its x- or y-position being outside those borders. When a particle does cross the borders on one axis, the velocity on this axis is inverted and scaled by a coefficient of restitution.The updated velocity therefore is calculated as

$$\mathbf{v} = -\mathbf{v} \cdot s_{\text{rest}}, \tag{3.2}$$

where s_{rest} denotes the coefficient of restitution. It describes the relation between the final and the initial relative velocity. In practice, it mainly scales down the original force in order to generate the repulsive force that arises when the particle hits the boundary. This type of collision detection results in a number of if-clauses and therefore might not be the optimal solution. For a different concept aiming to exploit the GPGPU technique, see Section 6.2.3.

3.2 Fluid Dynamics

This section focuses on the implementation of SPH behavior extending the existing particle system. It serves as overview providing relevant information about the parallel implementation on the graphics processing unit by using compute shaders.

3.2.1 Kernels

For the simulation it was chosen to use three different smoothing kernels. The decision of which kernels to choose was made in favor of the kernels proposed by [14], since they are specifically aimed for real-time applications and follow the concept of interactivity over accuracy. The proposed kernels can exclusively be used for 3D simulations, since their integrals do not yield unity. In order to fit for 2D-simulations as well, they were adapted by replacing the normalization constant. The following sections list the chosen kernels and justify their application.

Standard Kernel 2D

The first kernel is referred to as standard kernel in the SPH community, since it is probably the most frequently used kernel in this area. The kernel is denoted as

$$W_{\text{std2D}}(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{4}{h^2\pi} \begin{cases} \left(1 - \frac{r^2}{h^2}\right)^3 & \text{for } 0 \leq r \leq h, \\ 0 & \text{otherwise,} \end{cases} \tag{3.3}$$

and is used for density calculations. It is a Gaussian-shaped kernel, which is truncated at the finite distance of the smoothing radius. Thus, the relatively small contributions of particles outside the smoothing radius are ignored, offering a benefit in terms of computational complexity. Figure 3.2 plots the kernel function itself, as well as its first and second derivative.

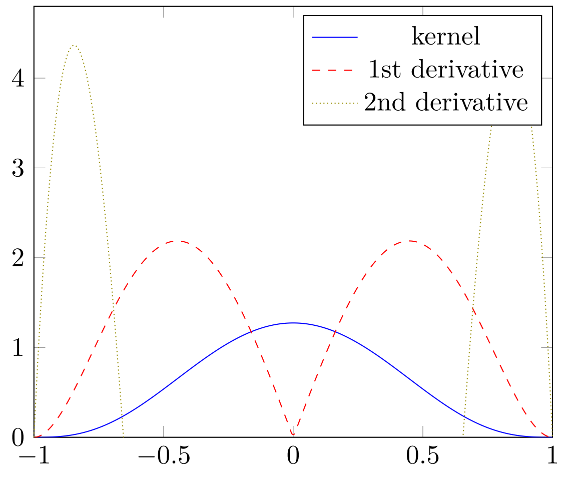


Figure 3.2: The standard kernel for 2D simulations as well as its gradient and Laplacian.

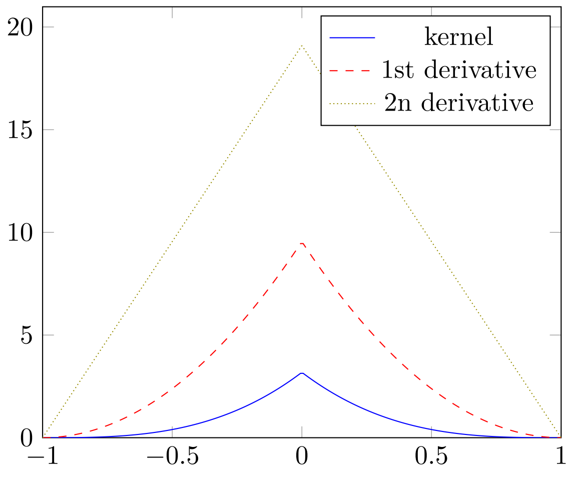


Figure 3.3: The spikey kernel for 2D simulations as well as its first and second derivative.

Spikey Kernel

Pressure calculations demand the use of the gradient of the kernel. As can be seen in Figure 3.2, the gradient of the standard kernel approaches zero at the center. Thus, when using the standard kernel function for pressure calculations, the resulting forces will vanish, when particles get too close. Since the calculated forces are supposed to represent repulsive forces, particles will start to build clusters in high pressure regions. Therefore, the so-called spikey kernel

$$W_{\text{spikey2D}}(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{10}{h^2\pi} \begin{cases} \left(1 - \frac{r}{h}\right)^3 & \text{for } 0 \leq r \leq h, \\ 0 & \text{otherwise,} \end{cases} \tag{3.4}$$

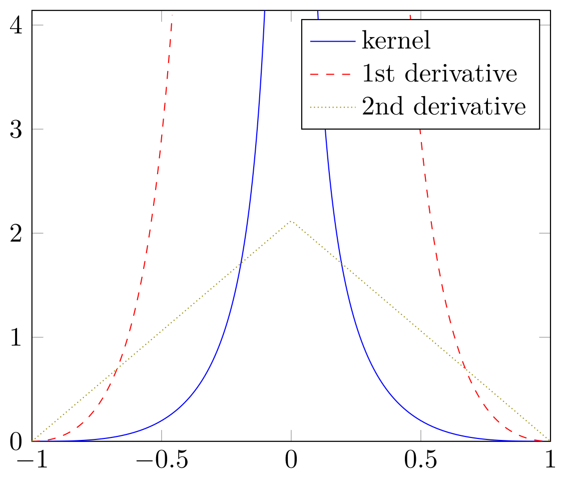


Figure 3.4: The viscosity kernel for 2D simulations as well as its first and second derivative.

was used for this implementation. It was first proposed by [4] and later readdressed by [14]. As opposed to the standard kernel, the spikey kernel's gradient does not vanish to zero in the center, which can be seen in Figure 3.3. Instead, both kernel and its gradient have a spikey shape, resulting in the repulsion forces that pressure is supposed to produce.

Viscosity Kernel

The third kernel used is the viscosity kernel which is defined as

$$W_{\text{visc2D}}(\mathbf{r}_i - \mathbf{r}_j, h) = \frac{10}{3h^2\pi} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & \text{for } 0 \leq r \leq h, \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

As its name implies, it is the kernel used for viscosity calculations. Figure 3.4 shows the viscosity kernel and its derivatives. The second derivative of the kernel results in a linear function, similar to the second derivative of the spikey kernel. In [14] no concrete reasons for its use are proposed. It is therefore doubted that the viscosity kernel provides any advantages to the use of the spikey kernel.

3.2.2 Naive GPU Implementation

In order to avoid errors coming from neighbor search and guarantee the validity of the simulation, the first SPH implementation was done using a brute-force neighbor search strategy. For a detailed explanation of the final neighbor search method implemented, see Section 4.1.

Algorithm 3.1 provides a naive SPH implementation using two nested for-loops traversing all particle states and is based on the principles explained in Section 2.2. In the first for-loop, density and pressure quantities per particle are evaluated. In the second for-loop, the previously evaluated density and pressure values serve as inputs for

Algorithm 3.1: Naive implementation of Smoothed Particle Hydrodynamics using a brute-force neighbor search method and the ideal gas equation for pressure calculations. The algorithm consists of two for-loops. The first one is responsible for pressure and density calculations, the second one for pressure and viscosity forces.

```
1:  procedure SmoothedParticleHydrodynamics($P$, $h$, $k$, $\rho$)
        Input: $P$, particle list; $h$, smoothing radius; $k$, gas constant; $\rho$, rest density.
2:      for each particle $i$ in $P$ do
3:          $\rho_i \leftarrow 0$
4:          for each particle $j$ in $P$ do
5:              $d \leftarrow |\mathbf{r}_i - \mathbf{r}_j|$
6:              if $d < h$ then
7:                  $\rho_i \leftarrow \rho_i + m \cdot W_{\text{std2D}}(\mathbf{r}_i - \mathbf{r}_j)$
8:              end if
9:              $p_i \leftarrow k(\rho - \rho 0)$
10:         end for
11:     end for
12:     for each particle $i$ in $P$ do
13:         $\mathbf{f}_i^{\text{pressure}} \leftarrow 0$
14:         $\mathbf{f}_i^{\text{visc}} \leftarrow 0$
15:         for each particle $j$ in $P$ do
16:             if $d < h$ then
17:                 $\mathbf{f}_i^{\text{pressure}} \leftarrow \mathbf{f}_i^{\text{pressure}} + \nabla W_{\text{spikey2D}}(\mathbf{r}_i - \mathbf{r}_j) \cdot m \cdot (p_i + p_j)/2\rho_j$
18:                 $\mathbf{f}_i^{\text{visc}} \leftarrow \mathbf{f}_i^{\text{visc}} + \nabla^2 W_{\text{visc}}(\mathbf{r}_i - \mathbf{r}_j) \cdot m \cdot \mu \cdot (v_j - v_i)/\rho_j$
19:             end if
20:         end for
21:     end for
22: end procedure
```

pressure and viscosity force calculations. The task has to be split in two parts, because for updating pressure and viscosity forces according to the SPH principle, density and pressure at particle locations are prerequisites and have to be computed first.

When parallelizing this algorithm and porting it to the GPU, the task as well is split up in two tasks, where each SPH compute shader corresponds to one of the inner for-loops. Therefore, two shaders responsible for SPH calculations are needed in total, `UpdateDensityPressure` and `UpdateForces`. Within those two compute shaders, the SPH interpolation concept is applied. Splitting up the task in two shader using separate dispatch calls will assure all threads are synchronized when the second shader is dispatched.

3.2.3 SPH Shader Structure

The intent of the SPH shaders is to add SPH contributions to each particle's acceleration. Both of the implemented shaders fundamentally follow the same concept. Program 3.1 shows a simplified version of the HLSL-shader `UpdateDensityPressure`, providing an

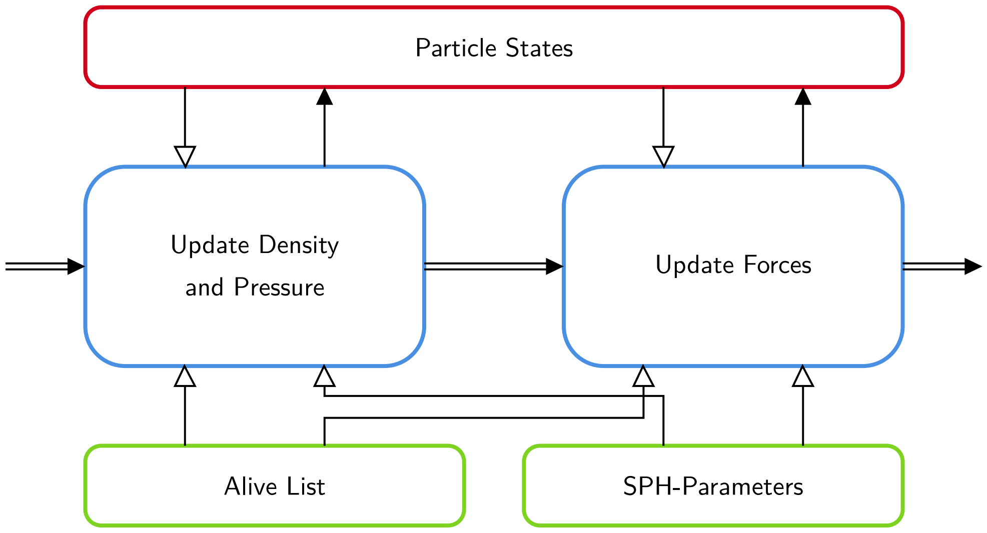


Figure 3.5: SPH shader pipeline. The first compute shader `UpdateDensityPressure` updates pressure and density values and updates the particle states buffer. The second compute shader `UpdateForces` fetches the updated data and calculates acceleration and viscosity forces based on it. Both shaders access the alive list and a buffer containing various SPH-specific parameters, however they do not write any data to them.

example of how SPH-specific shaders can be structured.

The shader invokes one thread for each particle. Input buffers for the shader are the buffer containing the particle states, the alive-list and a constant buffer storing its length, as well as one constant buffer containing SPH-specific constants. First, the particle of interest i is tested for being alive. This is determined by comparing the thread index with the alive-list length. If the index is less, the particle is considered alive. Subsequently, the alive-list can be accessed with the thread index, returning the actual particle index. The state of particle i is fetched from the particle states buffer and copied to a local variable.

Before starting density calculations, the particle's current density value has to be set back to zero. A for-loop iterates over each other particle j and fetches its state information. Particle i is not excluded from this iteration. Within the loop, the distance between both particles is determined. When the distance is less than the smoothing radius, particle j is considered a neighbor. In this implementation, the squared distance is used. Distance comparisons can be performed with squared distances as well, which avoids performing square root computations and thus unnecessary computational overhead.

When particle j is considered a neighbor, its density contribution is calculated by weighting its mass with the smoothing kernel. The mass and a part of the kernel are factored out and multiplied with the temporary density after the loop. Using the calculated density value, the pressure at particle location is evaluated according to the ideal gas equation. Eventually, the updated particle state is copied back to the particle states buffer.

The second shader, `UpdateForces`, fundamentally works following the same principle as the described one. Again, a for-loop iterates over the neighboring particles and

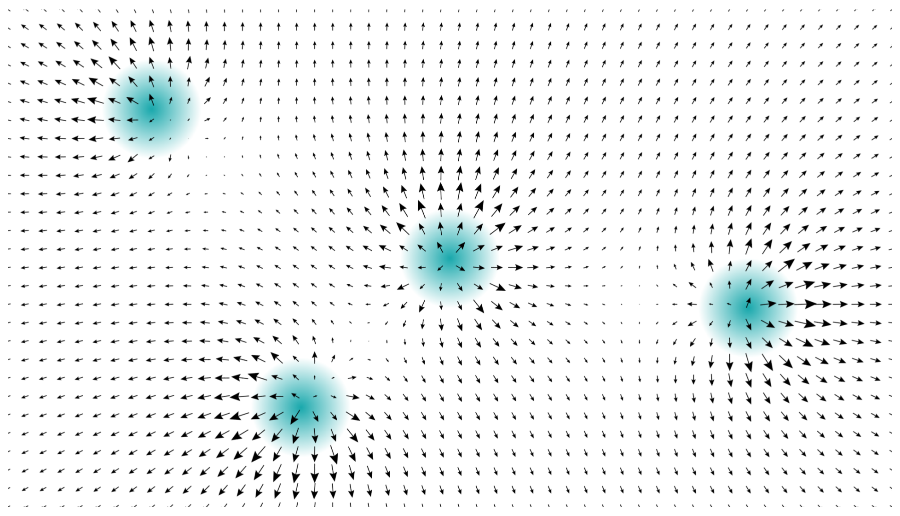


Figure 3.6: The underlying flow field of the simulation. It is used to process user input. The blue circles indicate players' positions. The arrows depict the magnitude and direction of the forces the players create.

calculates pressure and viscosity force, each weighted by a separate kernel. The resulting forces are converted to accelerations and added to particle accelerations.

3.3 Interactivity

As main means of interaction, the players' positions are tracked by a laser ranger. By walking around, they can generate both forces that act attracting on the particles, as well as forces, that act repulsive on them, respectively. The magnitude of those forces can be predefined, but manipulated during runtime as well. In case of a setup outside of Deep Space, the application can be executed on a device supporting multi-touch gestures.

3.3.1 User Defined Flow Field

The technique that lies behind this kind of interactivity is a flow field. It acts as force field on particles. Figure 3.6 shows how such flow field could look like. In order to adapt the creation of the flow field to the existing GPGPU pipeline, the flow field was completely implemented on the GPU. User positions are passed from CPU to GPU, as input parameters for the compute shader responsible for the flow field. The flow field itself consists of a number of discrete data points stored in a texture. While the r- and g-channels of the texture are used for storing the flow field's two-dimensional force vectors, the b and a- channel remain empty. The two remaining channels could be reserved for an additional flow field. If unused, they should be removed in order to avoid unnecessary memory consumption. Since the data is passed as texture, user positions can be quantized to the flow field's data points by mapping them to the texture as UV-coordinates. An additional decay compute shader can be activated. It diminishes the magnitude of the forces over time. This implementation method was mainly chosen

```
// The buffer containing the particle states.
RWStructuredBuffer<PointParticle2D> particleStates : register(u0);

// The list of alive particles.
RWStructuredBuffer<uint> aliveList : register(u1);

// Contains size of alive- and dead-list.
cbuffer ListSizeConstantBuffer : register(b0)
{
    uint numAlive;           // The number of alive particles.
    uint numDead;            // The number of dead particles.
}

// Contains SPH-specific constants
cbuffer FluidDynamics2DConstantBuffer : register(b1)
{
    float gasConstant;       // The gas constant of fluid particles.
    float restDensity;       // The rest density of the fluid particles.
    float smoothingRad;      // The smoothing radius of fluid particles.
    float smoothingRad2;     // The smoothing radius of fluid particles squared.
    float mass;              // The mass of fluid particles.
};

static const float PI = 3.14159265358979;
static const float KERNEL_POLY6 = 4.0 / (PI * smoothingRad2);

[numthreads(PARTICLE_UPDATE_GROUP_SIZE, 1, 1)]
void main(uint3 id : SV_DispatchThreadID)
{
  if (id.x < numAlive) {  // Check if particle is alive.

    uint id1 = aliveList[id.x];
    PointParticle2D ptcl1 = particleStates[id1];  // Fetch current particle state.
    ptcl1.dns = 0;  // Reset particle density.

    for (uint i = 0; i < numAlive; i++) {  // Iterate over all alive particles.

      uint id2 = aliveList[i];
      PointParticle2D ptcl2 = particleStates[id2];  // Fetch other particle state.

      float2 dv = ptcl2.pos.xy - ptcl1.pos.xy;
      float dst2 = dot(dv, dv);
      if (dst2 < smoothingRad2) {
        float x = 1.0 - dst2 / smoothingRad2;
        ptcl1.dns += x * x * x;
      }
    }
    ptcl1.dns *= mass * KERNEL_POLY6;  // Apply factored out term.
    ptcl1.prs = gasConstant * (ptcl1.dns - restDensity);  // Update pressure.
    particleStates[id1] = ptcl1;  // Write updated particle state back to particle buffer.
  }
}
```

Program 3.1: The HLSL shader `UpdatePressureDensity`. It performs density and pressure calculations while using a brute-force neighbor search method.

because its parallel structure matches the parallel architecture of the GPU.

3.4 Results

An image sequence of the simulation is shown in Figure 3.7.

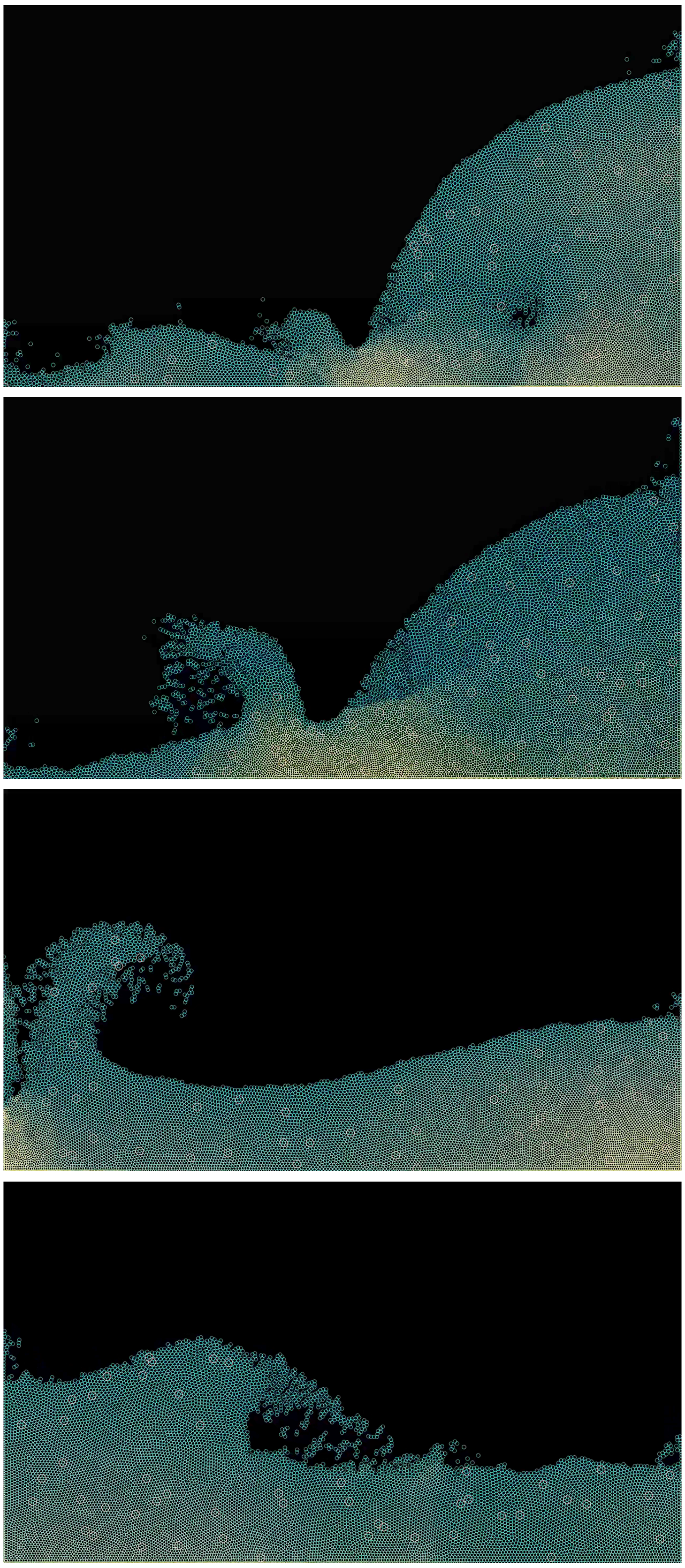


Figure 3.7: Image sequence of of the final simulation rendered with a debug shader.

Chapter 4

Optimization Measures

This chapter covers several techniques that were implemented in order to enhance performance. It offers justification for each optimization measure as well as a precise explanation on how they were implemented. Furthermore, various adjustments concerning the simulation's numerical stability are presented.

4.1 Neighbor Search

The most simple and naive way of fixed-radius near neighbor search (fixed-radius NNS) is a brute-force approach: Comparing each particle location $\mathbf{r}_i$ with each particle location $\mathbf{r}_j$ and performing a distance check. This approach results in a computational complexity of $O(n^2)$, with n denoting the number of particles used. Certainly, such complexity is not acceptable. Even small particle systems, such as systems with $n = 1000$ will already require one million distance checks. Particularly when targeting an interactive simulation such method will raise problems, since such applications require a certain frame rate one the one hand, and a certain amount of simulations per frame on the other hand.

4.1.1 Spatial Hashing

As acceleration method *Spatial Hashing* was chosen. It is a technique that uses a uniform spatial grid as conceptual data structure [5]. Particles are assigned to grid cells with a specified cell size by using a hash function. When looking for a particle's neighbors, only a number of grid cells have to be checked, in particular the cell the particle is located in as well as its adjacent cells. Figure 4.1 illustrates the basic concept of the neighbor search method.

This reduces the computational complexity of neighbor search itself to $O(kn)$, with k being the average number of particles per grid cell multiplied with the number of grid cells that have been looked through, since only the neighboring grid-cells and the contained particles have to be checked per particle. The complexity reduction as well as the parallelization ability and therefore GPGPU compatibility of this algorithm were determining factors leading to implement it.

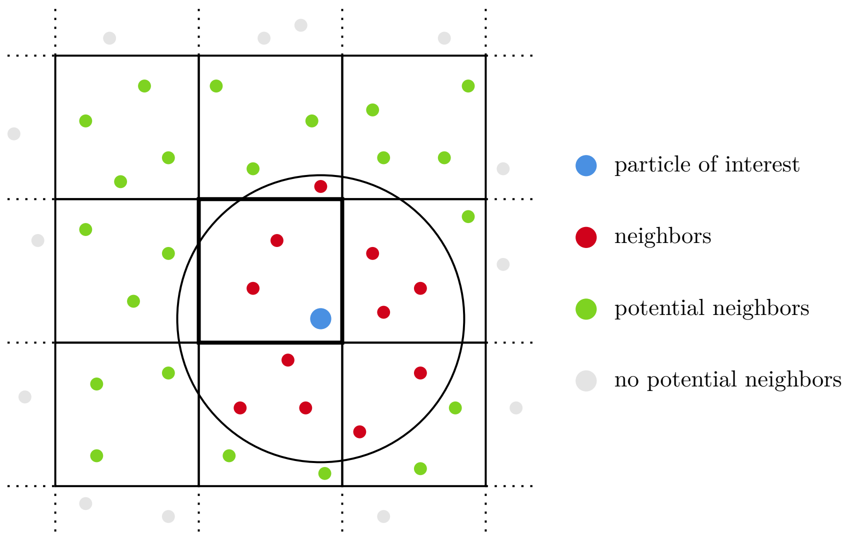


Figure 4.1: Spatial Hashing used for neighbor search. Each particle is assigned to a grid cell. When searching a particle's neighbors, only the grid cell the particle is located in as well as the adjacent cells contain potential neighbors. Distance checks then only have to be made with those potential neighbors, instead of with all particles.

Parallel algorithm

The implemented method was proposed in [7] and is designed to fit for GPU-based particle simulations. Originally, it was implemented with CUDA and therefore makes use of GPGPU acceleration. The basic idea is to cache particle IDs and their assigned grid-cells in a data structure in order to access them when needed.

The algorithm itself is rather easy to parallelize, since grid indexes can be calculated in parallel. What limits parallelization is mainly the sorting part of the algorithm.

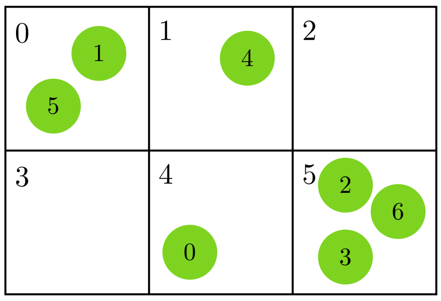
Hash Generation: Each grid-cell is assigned a hash-value. It is based on the grid-cell indexes c_x and c_y, representing the row and column index of the cell. The grid cell index c_x of a particle is calculated as:

$$c_x = \left\lfloor \frac{p_x}{c_{size}} \right\rfloor , \tag{4.1}$$

where p_x is the x-value of the particle position and c_{size} denotes the grid cell size in both x- and y-direction. The same scheme has to be applied for the grid cell index c_y in the y-direction, respectively. When having both indexes calculated, the final hash-value h can be created as

$$h = \operatorname{mod}\left(res_x \cdot c_y + c_x, \quad res_x \cdot res_y\right), \tag{4.2}$$

where res_x is the resolution of the grid cells in the x-direction. While grid-indexes are unique, hash values are not. This is due to the modulo operator that is used within the hash calculation. Therefore, it might be the case that two different grid-cells share the same hash value. As a consequence, particles that are located in a non-adjacent



(a)

pID	cellID
0	4
1	0
2	5
3	5
4	1
5	0
6	5

(b)

pID	cellID
1	0
5	0
4	1
0	4
2	5
3	5
6	5

(c)

startIdx	endIdx
0	1
2	2
-1	-2
-1	-2
3	3
4	6

(d)

Figure 4.2: The buffers needed for neighbor search. (b) stores the particle-IDs and their associated cell-ID. (c) is the same list, but sorted by cell-ID. (d) stores start- and end-index for looking up particles in (c).

cell are checked as well. However, this mainly happens when particles shoot out of their restricted area by crossing the borders and is seen as safety measure for handling rebellious particles.

Creating the Data Structures: Setting up neighbor search involves three data structures. An example is shown Figure 4.2. The creation of those data structures can be broken down into three main parts.

1. The first part involves the hash value creation for each particle: The grid cell indexes in both x- and y-direction are evaluated. Using those indexes, the hash value is created after the model mentioned above. The particle ID and the associated hash value are stored in a buffer `particleCellIDs`.
2. The buffer containing the particle ID and the associated hash value is sorted by hash value and stored in the buffer `particleCellIDsSorted`.
3. The start end indexes of each cell-ID in previously sorted list are stored in the buffer `startEndIdx`.

Retrieving the Data: After the creation of the required buffers all particle IDs of a specific cell can be retrieved by accessing the `startEndIdx` buffer using the specified cellID as index. The buffer returns both the start- and end index of the range containing relevant particle IDs. This circumvents the need of having each thread traverse through the entire sorted list, in order to retrieve all relevant particle IDs.

When searching for a particle's neighbors, not only the cell the particle is located in is of relevance, but also adjacent cells are. Which cells have to be checked and how many, depends on the grid-cell size. Depending on the setup, it might be necessary to define the quarter the particle is located in its cell.

GPGPU Implementation

The actual neighbor search implementation involves nine shaders in the end, where six are exclusively responsible for the sorting part of neighbor search. The three remaining shaders are responsible for the rest of neighbor search, meaning mainly buffer creation. The first shader invokes one thread per particle. The cell ID for each particle is calculated and stored together with the associated particle ID in a separate buffer. Next, six shaders are responsible for creating a buffer sorted those particle IDs by their cell IDs in ascending order. A detailed explanation on sorting is provided in Section 4.1.2.

After that, the buffer containing start- and end-indexes will be created. A compute shader is dispatched, each thread executed deals with resetting the buffer at its index. Start-indexes are set to -1 and end indexes to -2. Thus, when a cell does not contain any particles, a for-loop with said start- and end indexes will not perform any iterations. Subsequently, the last compute shader is invoked. It is responsible for writing the start- and end-indexes to the associated buffer. The shader uses one thread per particle. Each thread reads the grid cell at current and previous position from the sorted particle-cell buffer. Both cell indexes are compared. If they are not equal, a cell change is indicated. Thus, the current index is saved as start-index and the previous index as end-index. An additional if-clause handles the first and the last element of the buffer. Since this implementation will not introduce any write conflicts, no atomic operations are required.

4.1.2 Counting Sort

Sorting is an essential part of the spatial hashing neighbor search model, since particles have to be sorted by their grid cell ID. The procedure of sorting has to be performed each simulation step. Depending on the time step, this could be multiple times per frame. Furthermore, the number of elements to be sorted is equal to the amount of particles simulated. Since the amount of particles generally is to be maximized, an efficient way of sorting the grid-cell data had to be chosen. The solution of using *Counting Sort* was proposed in [21] and has been shown to outperform the method proposed in [7], which is using a parallel-radix sort implementation. Counting sort is not based on the comparison of numbers, but rather works address-based, which makes an implementation on the GPU feasible.

Algorithm

The algorithm consists of three parts. First, the number of particles per grid cells is counted and stored per cell ID. The resulting histogram provides the distribution of particles in the cells. As second step, the histogram's prefix-sum is computed. The

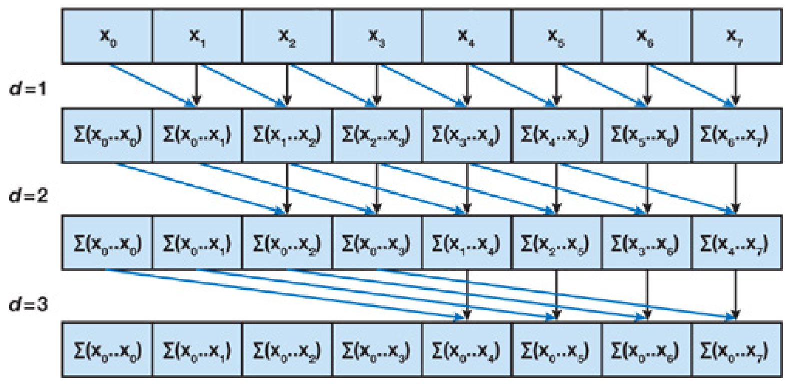

d=1
d=2
d=3
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

Figure 4.3: Principle of parallel prefix-sum creation. Source: [8]

prefix-sum of a sequence of numbers $s_{\text{init}} = (x_1,\ x_2,\ x_3)$ is another sequence of numbers $s_{\text{pref}} = (x_1,\ x_1 + x_2,\ x_1 + x_2 + x_3)$, built by adding up the prefix numbers of the initial sequence. Given the sequence $s_{\text{init}} = (1, 2, 3, 4, 5)$, its prefix-sum therefore is the sequence $s_{\text{pref}} = (1, 3, 6, 10, 15)$. In the case of counting sort, the prefix-sum serves as an address indicator. Each number of the sequence represents the address where a new grid cell starts. Once having calculated the prefix-sum, the data that is to be sorted only has to be copied from the histogram to the given addresses, which represents the third and last step of the algorithm. By decrementing each histogram value, each address that was already written to is removed from the histogram. The decremented value serves as address for the next particle that is located in the same cell.

Parallel Implementation

Since counting sort originally is a sequential algorithm, it had to be adapted in order to work on parallel systems as well. The following paragraphs describe the final GPGPU implementation of the algorithm. The implementation involved the creation of 6 shaders and several buffers.

Histogram: The histogram itself is a buffer created with size of the amount of grid cells. Each index of the buffer represents a grid cell. The histogram creation happens by dispatching two compute shaders. The first shader is responsible for resetting previous histogram data. One thread for each histogram bin is executed. Each thread sets the assigned bin back to zero. The second shader is responsible for the actual histogram creation. For each particle, a thread is executed. At position of the particle's cell ID, the histogram value is incremented. In order to avoid race conditions when two threads want to write into the same histogram bin, for the increment part the intrinsic function `InterlockedAdd` is used. It performs an atomic add of a given value, in this case 1.

Prefix Sum: As base for prefix-sum calculations the CUDA-specific parallel scan proposed in [8] was used. It was adapted to work with DirectCompute. The concept behind it is building partial sums of the individual elements, which are then added up again. Each stage is executed by a separate shader. The entire process of prefix sum creation

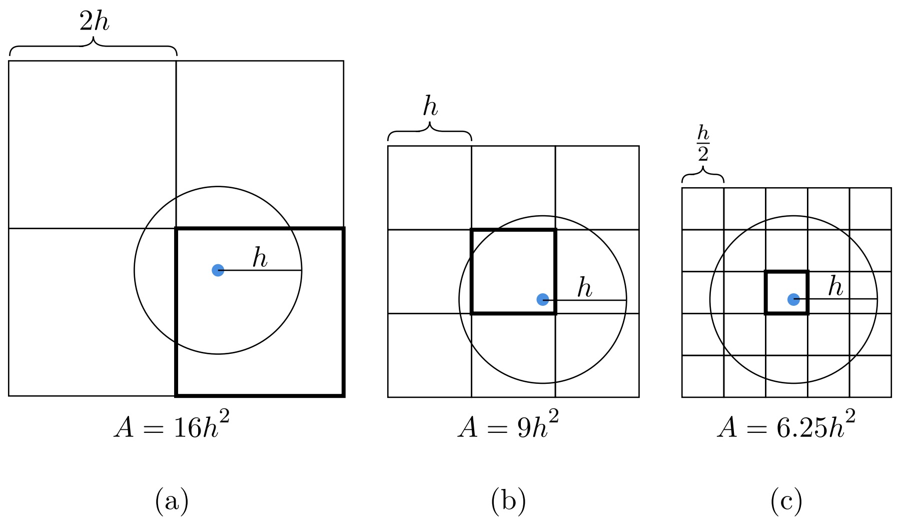


Figure 4.4: Demonstration of the grid-cell size c_{size} set to the values $2h$ (a), h (b) and $\frac{h}{2}$ (c). The smaller a grid-cell size is chosen, the smaller and therefore more exact becomes the area that has to be searched. However, the number of grid-cells that have to be searched is increased.

is performed by three shaders exclusively responsible for it. A visualization of the concept is provided in Figure 4.3. Further implementation details can be looked up in the reference.

Sorting: The sorting part of the algorithm mainly is based on copying values and is performed by one single shader, starting as many threads as active particles. An output buffer having the same size of the initial buffer is created. Each thread then reads the cell ID of the assigned particle and uses it as an index for accessing the histogram. The histogram returns the index of the output buffer where the particle ID and the associated cell ID is to be copied. After copying the data, each thread has to decrement the histogram. Again, this is done by performing an atomic add, however passing the negative value -1 to the function. Each thread therefore copies data to the output array and decrements the histogram for the subsequent threads. The result is a buffer containing the particle IDs sorted by their cell ID.

4.1.3 Grid-Cell Size

The bare implementation of the Spatial Hashing algorithm will not be a guarantee for exploiting performance capacities, when using the wrong parameters. An essential parameter within neighbor search is the size of the uniform grid cells. The grid-cell size determines not only the resolution of the grid, but it also defines the amount of grid-cells that have to be checked when looking for a particle's neighbors. Figure 4.4 illustrates the influence that the chosen grid-cell size has on grid construction and also data retrieval. Various aspects have to be considered when choosing the grid-cell size:

- The smaller the grid-cell size is chosen, the smaller becomes the area of interest.

On average, an area of interest that is smaller will hold lesser particles which are no neighbors. Reducing the cell size therefore should have a positive impact on performance.

- A higher grid resolution leads to a higher memory footprint, since the buffer storing start- and end-indexes will increase in its size.
- The range of the histogram created when performing counting sort is widened. The histogram itself will require more memory. Also, it could have an influence on the performance of prefix-sum creation.
- The number of particles has an impact on grid size choice. Dealing with huge amounts of particles will lead to the need of a smaller smoothing radius, which in turn will lead to a higher grid resolution.

A brief examination of a generally reasonable grid-cell size configuration is provided in [21]. However, it cannot be guaranteed that the given cell size serves as best option for this particular implemenetation. Therefore, during the implementation process the use of multiple cell sizes was tested. The influence the different cell sizes have on performance was evaluated and is discussed in Section 5.2.1. In any case, the cell size is dynamically adapted to the smoothing radius. This allows changing the smoothing radius during runtime, which is of high importance when it comes to parameter tuning.

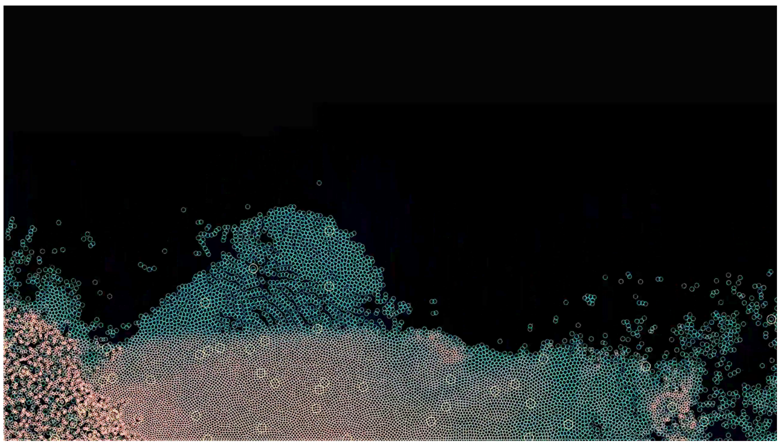
4.2 Time Integration

Initially, as integration method, the first order Euler integration method was chosen. Furthermore, a variable time step was chosen, meaning the delta time passed to the simulation was based on frame time.

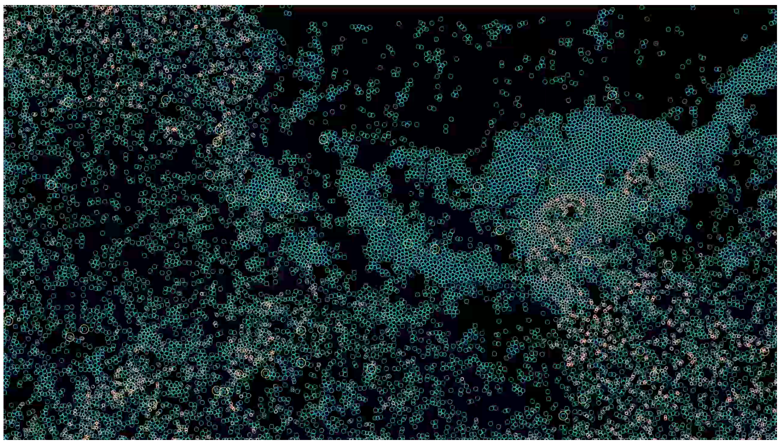
Performing some tests, it could be verified that the simulation brought up severe stability problems under certain circumstances. When particle accelerations exceeded to a certain level, the error produced by time integration was too large to be compensated. In certain cases, this seemed to be induced by only a couple of particles. When they accelerate too fast, their forces are passed to surrounding particles, eventually resulting in a chain reaction. Particles then start dashing around randomly. At times, stability could be regained. Other times the simulation had to be restarted. Figure 4.5 provides examples for instabilities of the simulation. As numerical stability is of high importance, several adjustments were made in order to improve the simulation's stability.

4.2.1 Time Step Adjustments

The problem of the variable time step the accuracy of the physics simulation depends on the time how long the last frame was calculated. Less powerful, or simply old machines, will therefore take longer to produce frames and in turn the time step will be larger than on newer, more potent machines. As a consequence, older machines will produce less exact simulations, since generally speaking the larger the time step, the larger the simulation error. What is more, a variable time step can introduce major stability problems, since the scaling of the delta time will not be controllable at all. When a frame takes too long to be calculated, the simulation could become instable because the resulting error might be to large to overcome it.



(a)



(b)

Figure 4.5: The simulation showing instabilities. They are caused by inaccuracies starting to develop in the lower left corner (a). If these inaccuracies cannot be compensated, the simulation will not stabilize (b).

It is therefore conceivable that the occurrence of stability problems can be reduced using a different way to determine the delta time that is passed to the simulation. An additional option to evaluate the time step was chosen to be implemented. The procedure is explained in [22] and is referred to as *semi-fixed time step*. The concept behind it is to define a fixed time step, but perform the simulation step multiple times per frame update, more specifically as often as frame time allows it. This provides the ability of controlling the size of the simulation step and being frame time independent at the same time.

4.2.2 Leap-Frog

Since SPH calculations are very expensive regarding the computational effort, decreasing the delta time another time and therefore performing even more simulation steps per frame was not an option. Instead, the *Leap Frog* integration scheme was implemented and tested as second integration option. It is an integration method of second order, and thus more accurate than the previously implemented Semi-Implicit Euler integration, which is a scheme of first order. Since Leap-Frog is computationally not vastly more expensive, it was used to achieve higher accuracy and stability while maintaining the time step.

The principle of Leap Frog is simular to the Euler method. Both position and velocity values are updated each time step. However, the updated position is calculated before the updated velocity, based on velocity and acceleration values. Then, the updated velocity is calculated, based on previous acceleration that was cached,as well as current acceleration. As a result, updated position and velocity are calculated as

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n \cdot \Delta t + \mathbf{a}_n \cdot \tfrac{\Delta t^2}{2}, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + (\mathbf{a}_n + \mathbf{a}_{n+1}) \cdot \tfrac{\Delta t}{2}, \end{aligned} \tag{4.3}$$

where $\mathbf{v}$ denotes the velocity, $\mathbf{a}$ the acceleration and $\mathbf{x}$ the position of a particle at step index n, and Δt is the time step of the integration [13].

The implementation of the Leap-Frog scheme entailed two changes in the original implementation:

- The process of time integration had to be split into two parts and therefore two different shaders, each performing one step: Each update, the first shader is dispatched before calculating particle forces, the second shader is dispatched after calculating particle forces.
- The struct PointParticle2D holding particle data had to be extended by the property `float3 prevAcc` in order to cache the acceleration of the previous time step.

Using a fixed timestep with $\Delta t = 1/120$, it provided more stable results than the previous scheme. An evaluation of both integration schemes is provided in Chapter 5.

4.3 Final Shader Pipeline

The described adjustments entailed several changes to the compute shader pipeline as well as the structure of the particle system. The particle system itself should be configurable, meaning it should be possible to choose which procedures are used to perform the simulation.

In order to provide the ability of configurating the setup, the application fetches Windows Registry entries that determine which combination of procedures is executed. This involves settings such as the neighbor search method, the integration method, which emitter should be used and where it should be positioned. Further settings are different rendering methods, forces that are added to the particles etc. The configurability of the entire system also offers the possibility of automatically testing different scenarios when evaluating them, without having to change the overall structure each time. This requires an application structure that can be variably adapted to the selection of specific processes. The simulation process has therefore been divided into four steps.

1. Preparation: Various types of preparations, which are prerequisites for the following procedures, are made during this period.

2. Forces: This part deals with the calculation of different forces applied to the particles. Possible forces are, for example, SPH-specific forces, gravitation, forces caused by user interaction.
3. Constraints: Here, any types of constraints can be added to the system. Currently, this features only the border constraints.
4. Emission: This section handles the particle emission as described in Section 3.1.2. The emission process is separated in two parts, one executed at the beginning of the update cycle, one at the end.

Figure 4.6 visualizes an example of how the compute shader pipeline can look like. Within the presented setup leapfrog integration as well as spatial hashing as neighbor search method are performed.

What might seem irritating is that the emission process is performed as last step, when it usually is performed as first. The reason for this is that the responsible compute shader `EmitParticles` performs changes to the alive list and makes it unusable. Since most other shaders are dependent on the alive list as input buffer, the emission process is performed last. This does not raise any problems, since the entire update loop is repeated anyways.

Depending on the type of setup, the shader pipeline changes its appearance. For instance, when using Semi-Implicit Euler integration, only one shader is needed in order to update particle positions. `LeapFrog1` therefore will not be dispatched. Similarly, when a brute-force neighbor search method is used, the shaders in orange colors do not have to be dispatched. However, the SPH-specific shaders, which can be used exclusively for NNS, are then replaced with shaders intended for the brute-force approach.

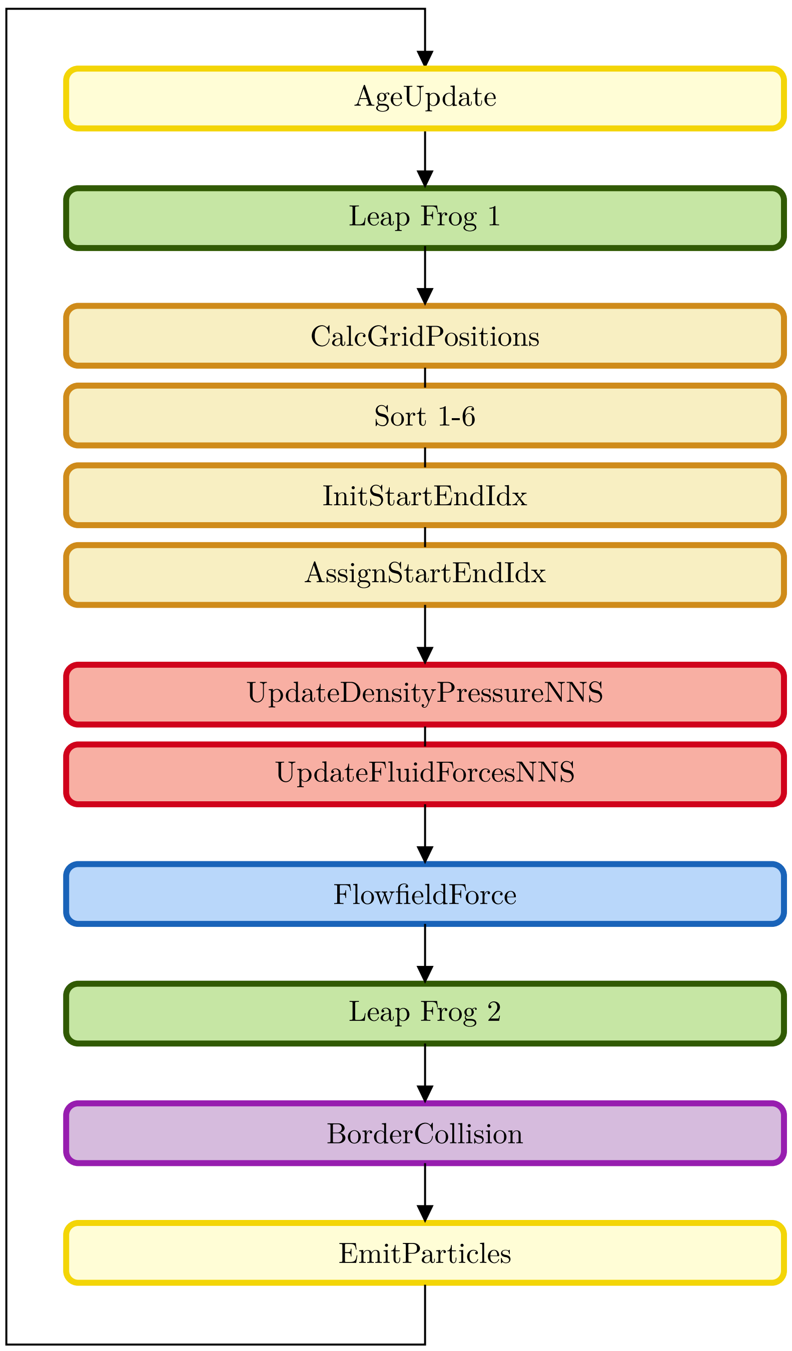


Figure 4.6: Compute shader pipeline of the final application. The different colors classify the shaders into sections: yellow, the emission process; green, the integration process; orange, preparation for Spatial Hashing as neighbor search method; red, fluid calculations using NNS; blue, flow field calculations; purple, boundary handling.

Chapter 5

Evaluation

This chapter deals with the performance evaluation of several processes as well as parameters which are part of fluid simulation. The main emphasis lies on defining optimal values for parameters such as grid-cell size and others as well as evaluating the influence of different integration methods. The gathered performance data will be used to draw conclusions about an ideal composition of said methods and parameters, in order to provide the best setup possible for the use case with the existing methods. Subsequently, in Section 5.3 it is discussed if the expectations are met or disproved as well as if the evaluation method even provides a definite conclusion.

5.1 Evaluation Environment

In order to evaluate the project, a test and measurement environment had to be created first. The live states of the actual application *liquidus* could not be used for this purpose, because they involve additional graphics and sound output that could affect performance measurements. Instead, the evaluation is carried out via a separate application state that was specifically implemented for evaluation and performance measurement. In addition, the evaluation state provides automated measurement methods and allows a reproducible scenario without random elements and user input.

A reasonable default setup for the evaluation state was created. The goal was to provide a stable setup, that can be used as basis for all measurement runs. Generally, each test setup involves the emission of 25000 particles over multiple seconds, representing a liquid that is poured into a 15 meter wide tank over time. Particle emission is conducted over the first 10 seconds. Performance then is measured for a total amount of 60-240 seconds, depending on the purpose of the measurement. This is done to assure the simulation can overcome any fluctuations resulting due to the emission process. Table 5.1 provides a summary of the used parameter values for this setup. Any exceptions due to specific measurements will be stated explicitly when presenting the results.

Data measurement itself is performed in constant intervals. Various experiments have shown that an interval of 500 milliseconds is sufficient and that a reduction to 250 milliseconds does not produce more detailed results. Since the evaluation effort is largely independent of the interval size, no further optimization of the interval size was seen as required.

Simulation Parameter	Value	Unit
Particle Count	25000	particles
Particle Mass	0.54	kg
Smoothing Radius	0.2	m
Rest Density	20.01	-
Gas Constant	37.44	-
Viscosity Coefficient	2.65	-
Integrator	Leapfrog	-
Timestep	120	Hz
Damping	0.1	-
Gravity	-10	m/s^2

Table 5.1: Default simulation parameters used for the evaluation setup. If not explicitly stated differently, this exact setup was used for measurements.

Each test was performed in *non real-time*. This means, a fixed delta time and a fixed number of simulation steps per frame were used. The whole process was performed as fast as possible, without considering the frame rate. On the one hand, this has the advantage that the simulation always runs under the same conditions, and on the other hand, the time required to wait for the measurement results is shortened.

Performance measurements were carried out on a single platform, using a NVIDIA GeForce GTX 1080 with 8GB video memory. As screen resolution, Full HD (1920×1080) in full screen mode was chosen.

5.2 Results

5.2.1 Grid-Cell Size

When performing Spatial Hashing as neighbor search method, the grid-cell size determines not only the resolution of the grid, but it also defines the amount of grid-cells that have to be checked when looking for a particle's neighbors and therefore also the area that is relevant for search. The question arises, which grid-cell size is suitable for this use case and also if it is setup-independent.

Taking a look at the SPH literature, the recommended cell size is typically $2h$ or h. Unfortunately, it is never justified why a particular size is used. A first approach to determine an optimal cell size was performed in [21]. As a result, the optimal cell size was defined to be 2/3. However, the way the results are presented does not indicate which values exactly were evaluated. Furthermore, no information about the setup is given, neither about the hardware used, nor about the defined parameters. It is therefore questionable if the results coming from the reference can be seen as valuable for this use case in particular, but as well in general.

In order to prove to find the best cell size for the use case, the test setup was run

Cell Size	$2h$	h	$2/3h$	$1/3h$	$2/5h$	$1/3h$	$2/7h$	$1/4h$
Resolution	2×2	3×3	4×4	5×5	6×6	7×7	8×8	9×9
Cells	4	9	16	25	36	49	64	81

Table 5.2: The number of grid cells that have to be checked depending on the cell size.

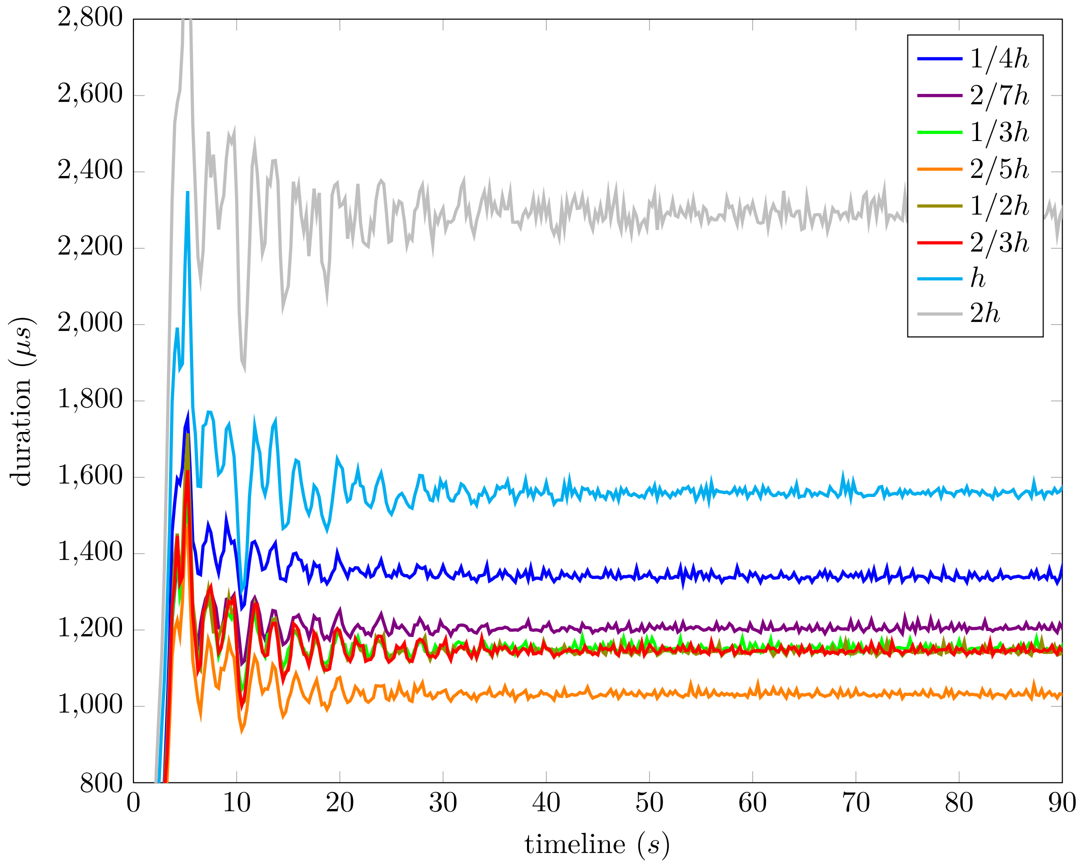


Figure 5.1: Duration of fluid force calculations using Spatial Hashing as neighbor search method. Each simulation period was performed with a different grid-cell size, each proportional to the smoothing radius. With gravity.

with eight different grid-cell sizes. Each cell size is proportional to the smoothing radius h, since it defines the size of the neighborhood that has to be searched. The factors proportional to h that were investigated were 2, 1, 2/3, 1/3, 2/5, 1/3, 2/7 and 1/4. Each of those values determines the number of cells that have to be searched. Table 5.2 provides the tested cell sizes and the associated number of cells that have to be searched when performing neighbor search. As shown in the table, the smaller the grid cells become, the more cells have to be searched.

Regarding the simulation setup, the test scenario was performed twice, once with the use of a gravity force and once without. This should prevent gravity from affecting

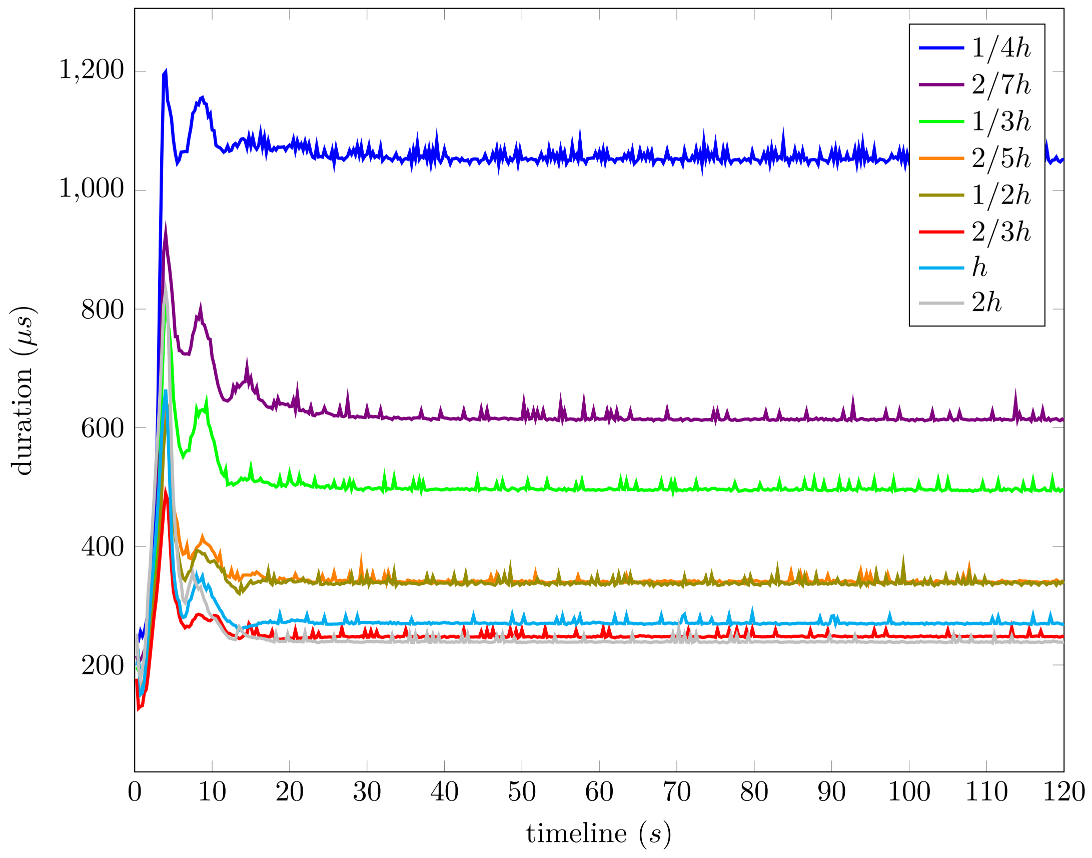


Figure 5.2: Duration of fluid force calculations using Spatial Hashing as neighbor search method. Each simulation period was performed with a different grid-cell size, each proportional to the smoothing radius. No gravity force was applied.

the result. Without gravity, particles are equally spread over the domain, not leading to any density fluctuations.

Apart from that, the test simulation was run each time under the same circumstances. The results are visualized in Figures 5.1 and 5.2. They show the duration of fluid force calculations with respect to time. Since the creation of the lists used for neighbor search do not show significant differences in this respect, they are not presented here.

5.2.2 Integration Method

During the development process two integration schemes were chosen to be implemented and tested. First, semi-implicit Euler integration. It is straight forward to implement and computationally quite inexpensive, which were decisive factors leading to utilise it as integration scheme initially. Literature, however, showed that the most common integration scheme used in the SPH community tended to be leapfrog. It therefore was assumed that using leapfrog would lead to improved results to some extent. In order to evaluate to what extent the two methods differ in terms of stability and performance, multiple test scenarios were created and measured. The goal was to find reasons to determine which of the both is more suitable for the use case.

Stability

The integration schemes to be evaluated differ considerably in terms of accuracy. While semi-implicit Euler is an integration scheme of first order, leapfrog is a scheme of second order. The integration error produced by leapfrog therefore is smaller when using the same delta time. As a consequence, it was hypothesized that leapfrog will have a positive impact on stability in relation to the results that semi-implicit Euler produces.

In order to evaluate if the hypothesis can be verified, both integration schemes had to be compared regarding their influence on stability. A method to observe the current stability of the simulation, not based on visual perception but rather on measurements, had to be determined. Although numerical instability generally can be detected mathematically, it was chosen not to in this case, since it would go beyond the bounds of this thesis. Instead, a simplified procedure to evaluate the simulation's current stability was developed.

When the simulation becomes unstable, particles start jumping around in an uncontrolled manner. They generate velocities that are outside the usual scale. For this reason, the idea was to detect particles whose speed values exceed a usual value. Since it is not feasible to build an average over speed values on the GPU, it was chosen to create a histogram of speed values instead. The histogram consists of three bins, each one associated with a range of speed. Each particle is assigned to a histogram bin, determined by defined speed thresholds. As a result, the histogram provides the distribution of *slow*, *medium* and *fast* particles. A particle is declared as

- slow, when $|\mathbf{v}| < 0.1$,
- medium, when $0.1 \leq |\mathbf{v}| < 0.5$,
- and fast, when $0.5 \leq |\mathbf{v}|$,

while the related unit is meters per second. Particles declared as fast exceed speed values that are considered as normal and are therefore classified as particles that indicate instability. As opposed to this, medium particles are considered as particles in motion and slow particles as still particles. Consequently, the velocity histogram provides valuable information about the current stability situation.

The evaluation setup was run using both integration schemes. Figure 5.3 provides the result of the measurements. It shows the distribution of fast, medium and slow particles over the time.

Critical Frequency

The method developed to observe the simulation's stability is also used to determine what was classified as *critical frequency*. The critical frequency describes the minimum frequency the simulation has to be run with in order to avoid stability problems. If a value below this limit is used, the simulation runs the risk of becoming unstable. This is due to the circumstance that the integrator then produces errors that cannot be compensated anymore. As a result, the fluid constantly oscillates, and in extreme

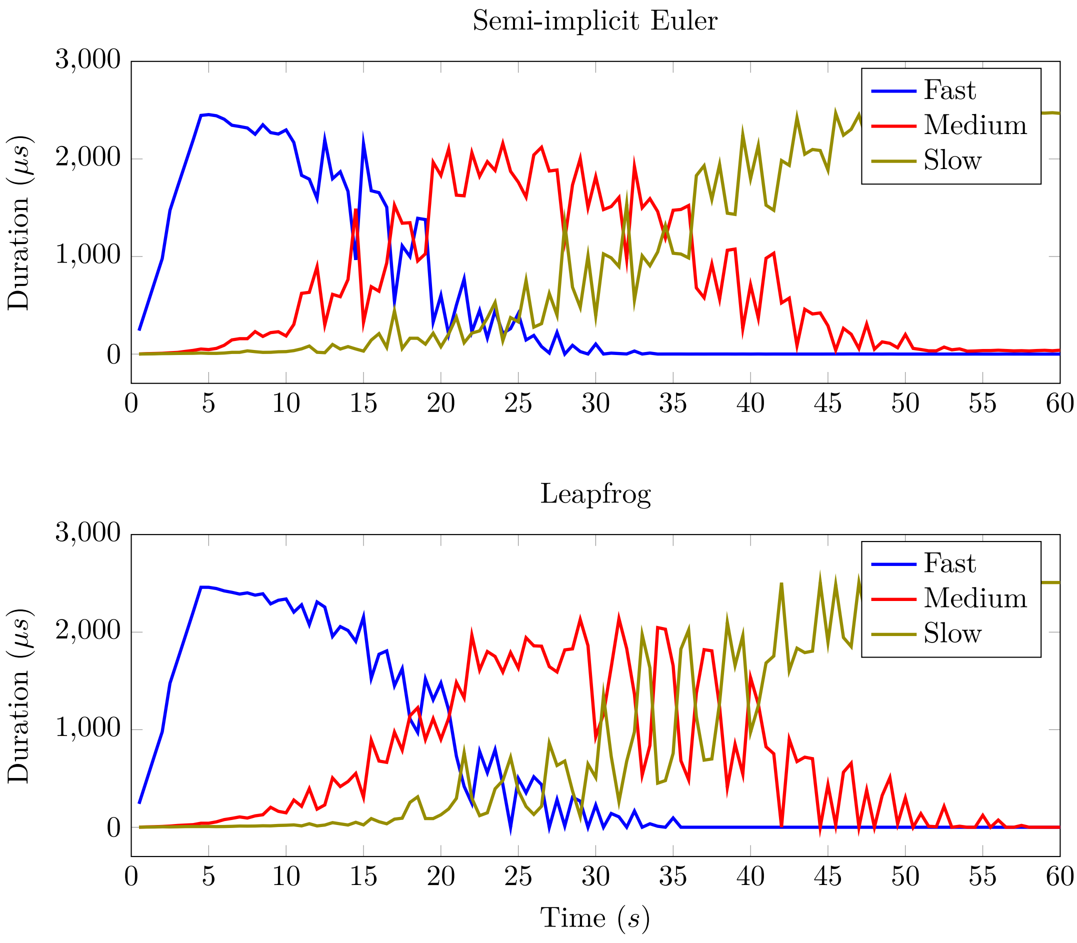


Figure 5.3: Stability observation of the simulation using semi-implicit Euler integration as well as leapfrog integration at 60 Hz.

cases, all particles race around chaotically. Once unstable, the fluid will not come to rest, which is indicated by the number of fast particles over time. The amount of fast particles should converge towards zero, if it does not, the simulation can be classified as unstable.

In order to limit the range of relevant frequencies, first, approximate critical frequencies were determined by visual evaluation. Thus, a frequency range was determined, which was regarded as relevant. Measurements were carried out afterwards with the contained frequencies. As soon as the number of fast particles decreased at one frequency during the simulation time, it was defined as the valid critical frequency.

5.2.3 Overall Performance

For the use case *liquidus* it was particularly important to create visually appealing and realistic fluids. Therefore, the goal was to optimize the simulation as much as possible. When providing a high resolution fluid, more exact details can be displayed than with

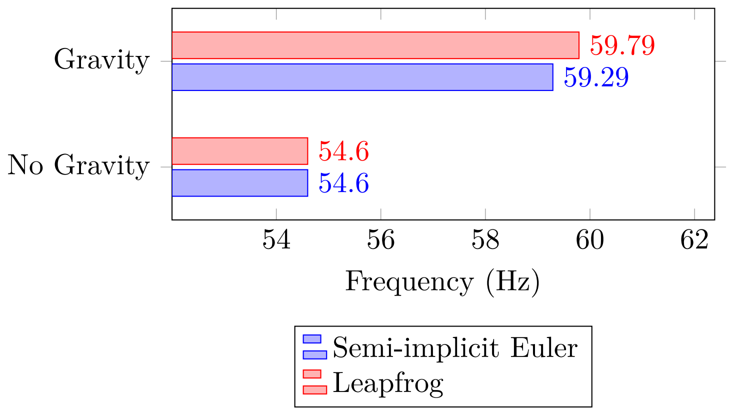


Figure 5.4: The critical frequency results of a setup with and without gravity.

a low resolution fluid.

In order to know the maximum amount of particles that can be simulated, the simulation was tested on five different system configurations using a simulation frequency of 120 Hz. Figure 5.6 provides the results of the simulation.

5.3 Discussion

5.3.1 Fluid Calculations Discussion

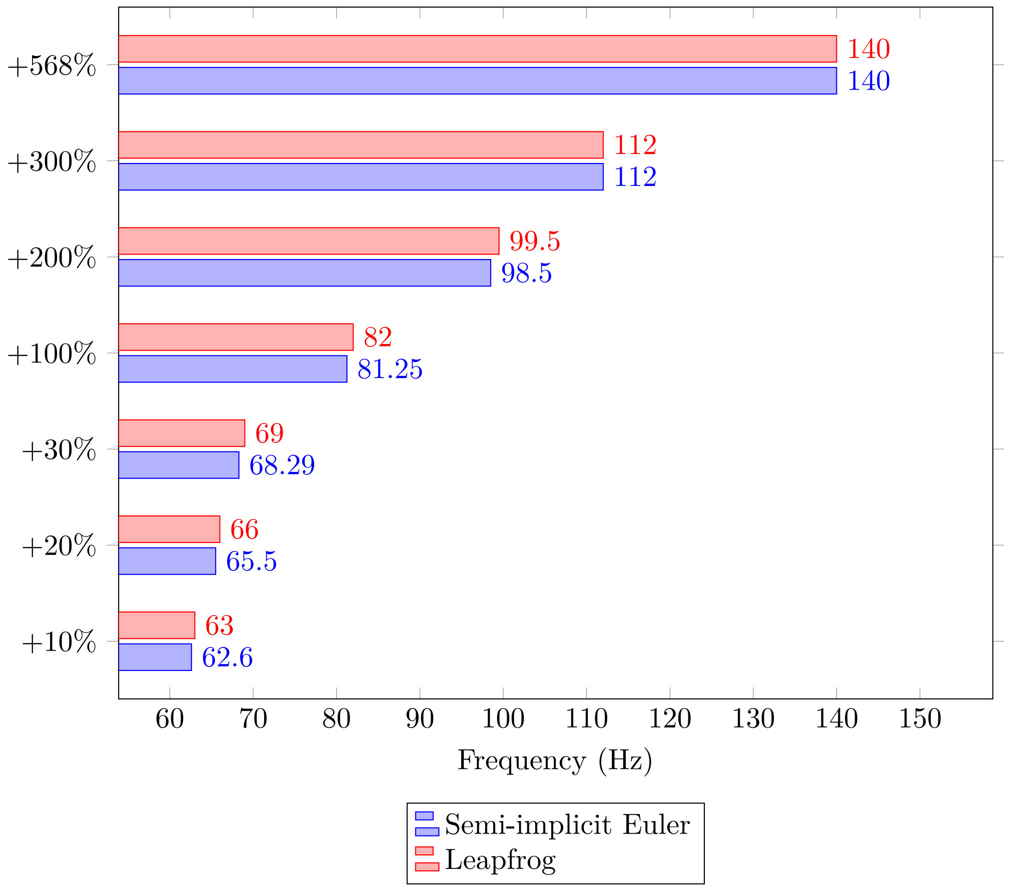
Grid-Cell Size

Several different cell sizes have been tested to evaluate if an optimum cell size can be defined. The results provided in 5.1 and 5.2 show the duration of fluid calculations over time. The high amplitudes that occur during the first seconds of the simulation are due to the emission process.

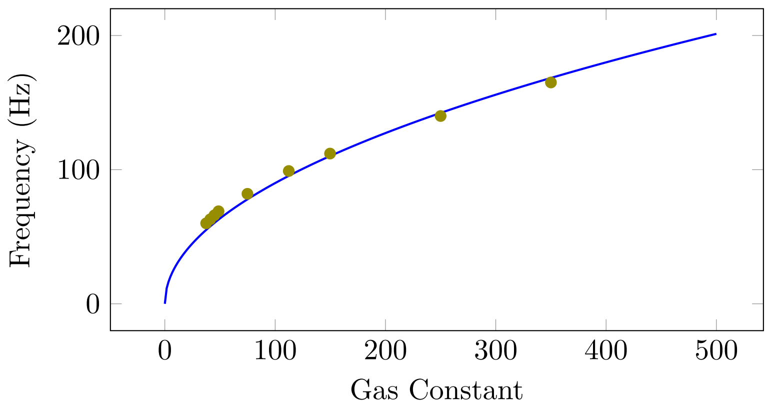
Taking a look at performance, the differences using scenarios are higher than expected. In certain cases, they even exceed one millisecond, which can be seen as significant amount considering that each frame offers a maximum of 16 milliseconds and that in most cases multiple simulation updates are performed per frame. Interestingly, both measurements result in completely different ideal cell sizes. While the setup using gravity clearly shows $2/5h$ as optimum, the setup without gravity shows $2h$ as optimum. At the same time, however, $2h$ serves as worst cell size for the setup with gravity.

A higher grid resolution generally leads to a search area that better suits the actual area of interest. Therefore, when increasing the resolution of the grid, less particles that are not in the neighborhood have to be checked. However, the increase of the resolution also leads to a rise in grid cells that do not contain any particles. Nevertheless, those grid cells are checked for particles and several read accesses of the neighbor lists are performed for each empty grid cell that is searched. It is therefore assumed that an optimum value for the cell size must provide a well-balanced compromise between a search area that is too large and too many unnecessary memory accesses.

When applying gravity to the simulation, the density of the fluid will increase as a



(a)



(b)

Figure 5.5: Results for the evaluation of critical frequencies. The bar plot (a) shows results for two integration schemes using different values as gas constant. The tested values are based on the initial gas constant used in the default setup. The scatter plot (b) shows the same results, however absolute gas constant values were used. The data points resemble a square root curve.

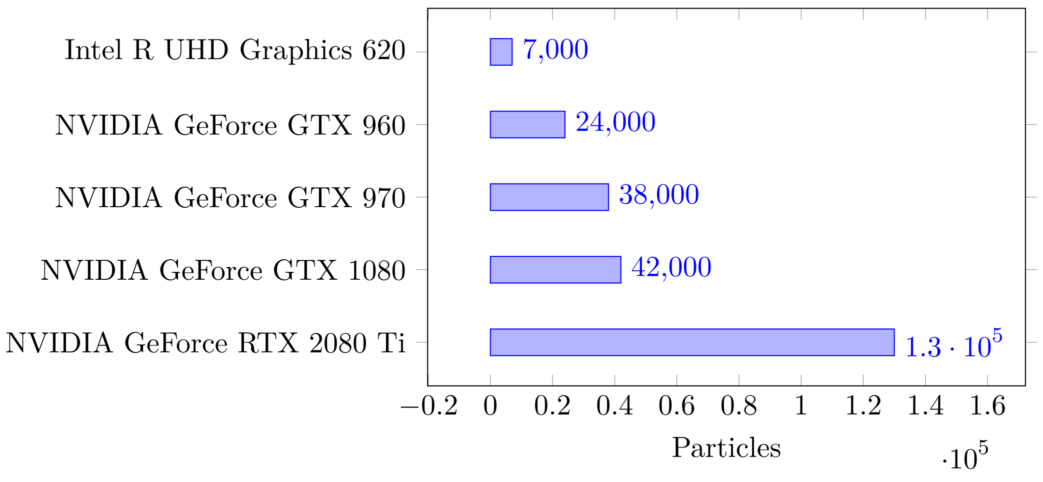


Figure 5.6: Maximum number of particles possible using different hardware.

side effect. The average number of particles per grid cell will increase, since particles are pushed towards the lower border of the tank. As a result, said compromise might be given under the different conditions, since the distribution of particles vastly changed. An ideal grid cell size therefore is highly dependant on the setup of the simulation and can not be defined in general.

Density

Density has a major influence on performance. Figure 5.1 shows the duration of fluid force calculations over time with gravity. Figure 5.1 shows the same measurements, but the simulation was not influenced by gravity. Both measurement runs show severe performance differences. Looking at cell size of $2/5h$, the calculations for the setup influenced by gravity take about twice as long as the same calculations without gravity. The least performant cell size $2h$ takes even more, on average. This is due to the smoothing radius enclosing more particles in its area when the density of the fluid is increased. The more particles are included in the area of interest of particles, the higher the computational effort will be.

Sloshing

Taking a look at Figure 5.1, all fluid force calculations show noticeable sinusoidal shapes. Due to its parameter setup, the default simulation represents a liquid. When the liquid is poured into the tank defined by the borders, the sloshing of the liquid leads to a short-term compression of the liquid. Since the density determines the average number of particles in a grid cell and also in the smoothing radius, sloshing has an effect on the performance of fluid calculations. Therefore, it is assumed that the sinusoidal oscillation in fluid force calculations could be due to the sloshing behavior of the fluid.

Comparing the results with the results from Figure 5.2, it can be seen clearly that gravity has an influence on the oscillation. While all simulations using gravity show those deflections, simulations performed without gravity, do not show any of them. This is assumed to be due to the fluid not sloshing during the simulation.

5.3.2 Integration Discussion

Stability

Figure 5.3 shows the results of the stability observation of both integrators. Both simulation scenarios initially show a high number of fast particles. This can be traced back to the emission process. Particles are thrown into the tank from a certain height and therefore have high initial velocities. Additionally, they are often thrown apart when their spawn positions lie too close to each other. Both can lead to high velocities.

Once the emission process is complete, the simulation begins to calm down and stability returns. This is visualized by the decrease in the number of fast particles and the increase in the number of slow particles. After one minute, the simulation has calmed down completely and all particles have entered the state of rest. Against expectations, both procedures behave largely similarly when looking at the results. Due to the large amount of fluctuations, no tendency of which scheme being more stable can be determined. Both schemes seem not to differentiate greatly.

Figure 5.4 and Figure 5.5 also provide information about the stability of the simulation. Comparing the bar plots, they show that both integration schemes have frequencies with inconsiderable differences. This confirms the assumption that they do not differ in terms of stability. Leapfrog originally was thought to provide better results than semi-implicit Euler due to its higher order, resulting in smaller integration errors. As reason for not determining any significant differences, it is assumed that Euler causes increased stability as it underestimates the overall energy of the system. The original hypothesis is based on the false assumption that a higher error simultaneously leads to higher instability.

Critical Frequency

The critical determined cannot be considered suitable for practice. the reason for this is that there is no headroom available for situations where stability is at risk. However, they are values that can be determined and therefore compared as well. They provide basic information about the conditions that have to be fulfilled in order to guarantee the stability of the simulation.

Interestingly, the results concerning the gas constant, which are presented in Figure 5.5, show that the simulation's minimum frequency changes in a square root relation to the gas constant. Higher values as gas constant generally improve the fluid's visual quality, since it lessens compressibility. Particles close to the borders therefore will be less compressed than when using low values as gas constant. As a consequence, the gas constant can be tuned up to higher values without suffering a loss in stability.

A similar relation could also apply to different values for gravity. However, since for the use case of *liquidus* only the two states *with gravity* and *without gravity* are relevant, the measurement of further values was left out of consideration. The results in Figure 5.4 show that there is a difference when comparing both setups. While for the test setup without gravity a frequency of 60 Hz would provide enough headroom to ensure stability, the same setup using gravity would not provide any headroom for critical situations. To solve this, a higher frequency could be used to provide the headroom needed.

5.3.3 Overall Performance Discussion

Again, the measurements visualized in Figure 5.6 are strongly dependant on the simulation setup, which complicates any comparisons. The numbers determined will not fit for all setups, since they are dependant on e.g. the water depth, gravity, the fluids density and even more. However, these numbers offer the possibility to examine the performance variations on different hardware. Looking at the results, the NVIDIA GeForce RTX 2080 Ti produces the best results without surprise. It is even managed to simulate more than twice the amount of particles as the NVIDIA GeForce GTX 1080. These large differences in performance may be due to the different handling of memory management by the different graphics cards. Low-end graphics cards definitely cannot compete with those results.

In the case of SPH generally, as well as concerning this specific implementation, performance is mainly dependant on fluid calculations. To further accelerate the simulation, the neighbor search methods should be further optimized, since it poses the main bottleneck throughout the simulaton.

Chapter 6

Conclusion

To realize *liquidus*, the concept of SPH was implemented exclusively on the GPU. The parallel execution of particle force calculations allows acclerating the computationally expensive force calculations and therefore provides the ability of simulating a large number of particles while maintaining interactive frame rates.

In addition, a number of optimization measures were implemented, which are specifically suited for the parallel architecture of the GPU. A neighbor search method built on the concept of spatial hashing replaced the naive brute-force neighbor search approach and contributed to the reduction of expensive fluid force calculations. Counting sort was implemented to be executed exclusively on the GPU, efficiently sorting hundreds of thousands of particles by their grid cells.

Regarding the optimization of neighbor search, an ideal cell size could not be determined. Since in the referenced literature mainly cell sizes of $2h$ or h were specified, it was assumed initially that one of them would lead to the optimum. However, performance measurements show that the ideal cell size is highly dependant on the simulation setup, in particular on the fluid's density. The cell size that is proposed in [21] might be ideal for the setup, but cannot be seen as ideal in general. Nevertheless, since *liquidus* provides a number of specific setups, an ideal cell size for those setups can be evaluated in order to maximize the number of particles for the use case while maintaining a stable frame rate.

Contrary to expectations, it was found that leapfrog has no specific advantages over semi-implicit Euler in the context of interactive SPH. It does neither contribute more to the simulation's stability, nor lower the minimal amount of simulation steps per frame required. For the further development and application of *liquidus*, it was therefore decided to use semi-implicit Euler as integration scheme, as it is slightly more performant. A simulation with such computational effort needs to be optimized as much as possible in order to ensure that sufficient computing time is available for other tasks such as rendering.

Furthermore, some valuable insights regarding the influence of parameters have been gained. It could be determined that he simulation's minimum frequency changes in a square root relation to the gas constant. The gas constant can thus be scaled up without strongly influencing performance and stability. It can be interpreted from the results that the density has the greatest influence on the simulation. Both performance and stability depend on it.

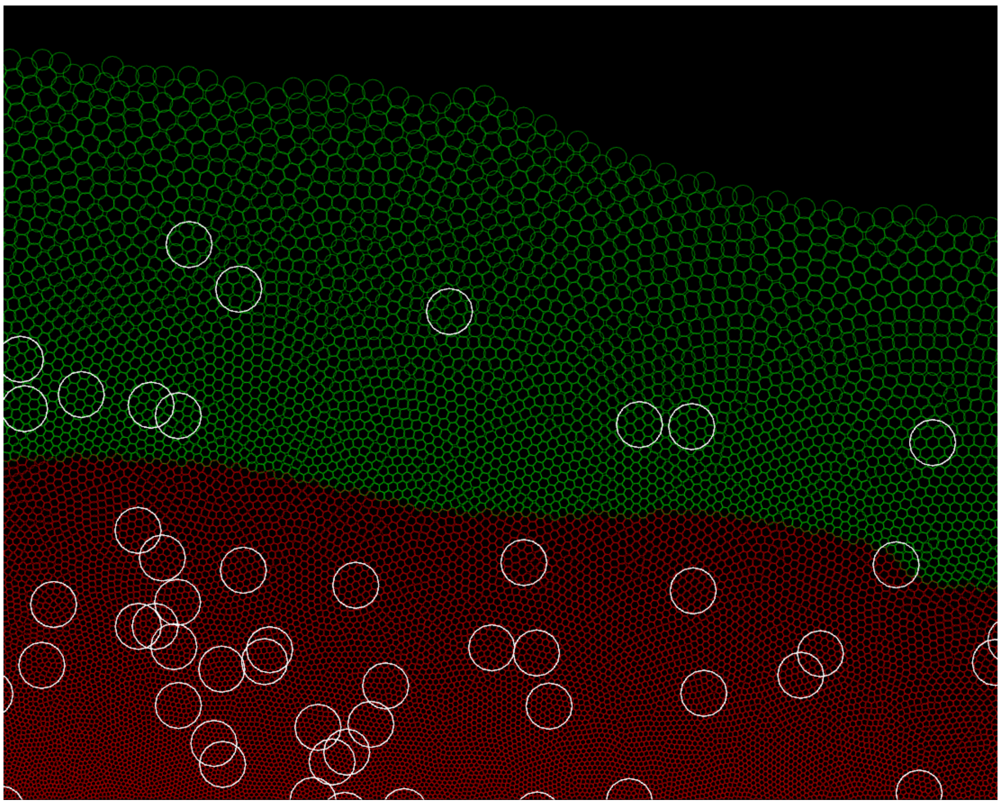


Figure 6.1: Part of a screenshot of the simulation. Particles are rendered using a debug shader. The particles' color visualizes the density at particle location. While a red color indicates a very dense region, green depicts less dense regions. The white circles visualize the current smoothing radius. Due to differences in density within the fluid, different numbers of particles are enclosed by the smoothing radius.

6.1 Limitations

Most of the measurements performed are only comparable to a limited extent, since they are strongly setup-related. During the evaluation it was therefore tried to use the same setup continuously. Measurements under different conditions can lead to completely different results. The massive number of possible combinations of the different methods thus makes the measurement procedure very difficult, which is why it was not possible to evaluate the entire range of considered parameters.

6.2 Future Work

6.2.1 Adaptive Smoothing Radius

Density differences pose a problem in the current implementation of SPH calculations, since they largely affect performance. This is due to the number of neighbors of each particle which varies with the diverging density of the fluid. Figure 6.1 visualizes the problem. While the smoothing radius encloses 30 and more particles in very dense regions, around five or less particles are enclosed in less dense regions. Regions close to the borders result in higher densities and therefore particles have more neighbors that have to be included to SPH force calculations. As a result, the computational time needed to perform SPH-specific calculations increases within those regions.

The varying density can be related to the fluid's compressibility. Unfortunately, methods aiming to achieve incompressibility, or at least approximating it, are generally computationally vastly more expensive and therefore do not offer an adequate solution for the problem.

What could be done, however, is adapting the smoothing radius of each particle to the density at its location. Particles in denser regions then would be assigned a smaller smoothing radius and particles in less dense regions a larger one. As a consequence, the number of neighbors would roughly be equal over the domain. Furthermore, an approximately equal number of neighbours could contribute positively to the calculations performed on the GPU, since they would be distributed to all threads approximately equally as well, which could have positive effects on GPU occupancy and therefore increase performance.

Nevertheless, the implementation of an adaptive smoothing radius would introduce challenges. The uniform grid used for neighbor search would not serve well then. Instead, the grid's resolution would as well have to adapt to the smoothing radius. In dense regions, the grid might need to have a higher resolution than in less dense regions.

6.2.2 Data Reorganization Using Space Filling Curves

As described in Section 4.1.1, partitioning the particles in a uniform grid in order to perform neighbor search brings large performance improvements regarding. With this approach, the amount of particles that have to be checked for being neighbors is reduced significantly. However, neighboring particles usually are spread across memory. Accessing the data in a coalesced way therefore is not possible. In [18] a method to optimize data access is proposed: Particle data is reorganized in global memory using a Hilbert space-filling curve. As a result, particles that lie close to each other in space also lie close in memory. Then, the idea is to copy particle data into shared memory. Since this is a time-consuming process, only particle data that is seen as relevant according to the Hilbert curve is copied.

Currently, throughout the execution process, all particle data is read from global memory only. This indeed can be seen as a naive way of implementation, since global memory provides relatively slow memory access [16, pp. 79]. Switching to a shared memory implementation definitely holds potential, however, the proposed method must be implemented and evaluated in order to find if it is suitable as an optimization technique in this case. Copying particle data to shared memory is an expensive task that might only provide advantages with very large amounts of particles. Also, the proposed method is based on the use of a *Verlet List*, a list where neighbors are cached per particle. This is a completely different approach than the one implemented.

6.2.3 Boundary Treatment

The current boundary treatment method requires branching, since each particle needs to be tested on being outside of the specified borders. The resulting thread divergence can have a substantial impact on execution time, since threads could remain idle while waiting for other threads to finish processing instructions [15, p. 89]. This results in unexhausted GPU occupancy. One option to resolve this issue is to implement a separate flow field exclusively used for border collision. Such flow field offers the advantage that its containing forces just have to be added to the other particle forces. This follows the same principle as the flow field responsible for user interactions, which means boundary treatment would therefore adapt to the base structure of the simulation.

Another possibility is to instantiate static particles that serve as boundaries. The particles are excluded from fluid calculations, however, they contribute to other particles' fluid calculations. They mainly store the same properties as each other particle. Such boundary treatment method could entail the benefit of reducing particle compressibility at the borders, since border particles then have a fixed location.

Appendix A

CD-ROM/DVD Contents

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Files

Path: /

_thesis_loimayr.pdf . .	Thesis Document

A.2 Project-Files

Path: /

_project_source.zip . .	Relevant Project Source Code

A.3 Miscellaneous

Path: /

_online_sources.zip . .	Cited Online Sources
_videos.zip	Video Examples of the Simulation

References

Literature

[1] Markus Becker and Matthias Teschner. "Weakly Compressible SPH for Free Surface Flows". In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* San Diego, California: Eurographics Association, 2007, pp. 209–217 (cit. on p. 12).

[2] Jiri Blazek. *Computational Fluid Dynamics. Principles and Applications.* 3rd ed. Oxford: Butterworth-Heinemann, 2015 (cit. on p. 4).

[3] Sharen J. Cummins and Murray Rudman. "An SPH Projection Method". *Journal of Computational Physics* 152.2 (1999), pp. 584–607 (cit. on p. 12).

[4] Mathieu Desbrun and Marie-Paule Cani. "Smoothed Particles: A new paradigm for animating highly deformable bodies". In: *Eurographics Workshop on Computer Animation and Simulation.* Published under the name Marie-Paule Gascuel. Poitiers, France: Springer-Verlag, Aug. 1996, pp. 61–76 (cit. on pp. 11, 23).

[5] Christer Ericson. *Real-Time Collision Detection.* San Francisco, California: Morgan Kaufmann Publishers, 2005 (cit. on p. 30).

[6] R. A. Gingold and J. J. Monaghan. "Smoothed particle hydrodynamics: theory and application to non-spherical stars". *Monthly Notices of the Royal Astronomical Society* 181.3 (Dec. 1977), pp. 375–389 (cit. on p. 5).

[7] Simon Green. *Particle Simulation using CUDA.* Tech. rep. NVIDIA, May 2010. URL: http://developer.download.nvidia.com/assets/cuda/files/particles.pdf (cit. on pp. 31, 33).

[8] Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA". In: *GPU Gems 3.* Ed. by Hubert Nguyen. Boston: Addison-Wesley Professional, 2007. Chap. 39, pp. 851–875 (cit. on p. 34).

[9] Doyub Kim. *Fluid Engine Development.* Boca Raton, Florida: CRC Press, 2017 (cit. on pp. 6, 9).

[10] Larry D. Libersky and A. G. Petschek. "Smooth particle hydrodynamics with strength of materials". In: *Advances in the Free-Lagrange Method Including Contributions on Adaptive Gridding and the Smooth Particle Hydrodynamics Method.* Ed. by Harold E. Trease, Martin F. Fritts, and W. Patrick Crowley. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991 (cit. on p. 5).

[11] G.R. Liu and M.B. Liu. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method.* Singapore: World Scientific Publishing Co. Pte. Ltd., 2003 (cit. on p. 5).

[12] M.B. Liu, G.R. Liu, and K.Y. Lam. "Constructing smoothing functions in smoothed particle hydrodynamics with applications". *Journal of Computational and Applied Mathematics* 155.2 (2003), pp. 263–284 (cit. on p. 7).

[13] James M. Van Verth. "Mathematical Background". In: *Game Physics Pearls.* Ed. by Gino van den Bergen and Dirk Gregorius. Natick, Massachusetts: A K Peters, Ltd., 2010. Chap. 1, pp. 3–28 (cit. on p. 38).

[14] Matthias Müller, David Charypar, and Markus Gross. "Particle-based Fluid Simulation for Interactive Applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* SCA '03. San Diego, California: Eurographics Association, 2003, pp. 154–159 (cit. on pp. 6, 9, 11, 13, 21, 23).

[15] Jason Sanders and Edward Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010 (cit. on p. 55).

[16] Duane Storti and Mete Yurtoglu. *CUDA for Engineers. An Introduction to High-Performance Parallel Computing.* Addison-Wesley Professional, 2015 (cit. on p. 54).

[17] Mauricio Vines, Won-Sook Lee, and Catherine Mavriplis. "Computer animation challenges for computational fluid dynamics". *International Journal of Computational Fluid Dynamics* 26 (July 2012), pp. 407–434 (cit. on p. 5).

[18] Daniel Winkler et al. "gpuSPHASE – A shared memory caching implementation for 2D SPH using CUDA". *Computer Physics Communications* 213 (2017), pp. 165–180 (cit. on p. 54).

Online sources

[19] *Chroma Lab.* URL: https://futurism.com/wp-content/uploads/2017/04/chroma-lab.jpg (visited on 10/29/2019) (cit. on p. 16).

[20] *Chroma Lab.* URL: https://store.steampowered.com/app/587470/Chroma_Lab/ (visited on 10/29/2019) (cit. on p. 16).

[21] *Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids.* URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf (visited on 10/17/2019) (cit. on pp. 33, 36, 42, 52).

[22] Glenn Fielder. *Fix Your Timestep! How to step your physics simulation forward.* June 2004. URL: https://gafferongames.com/post/fix_your_timestep/ (visited on 11/09/2019) (cit. on pp. 20, 37).

[23] *Fluid Structure 360.* URL: https://vincenthouze.com/portfolio/fluid-structure-360/ (visited on 10/29/2019) (cit. on pp. 14, 15).

[24] *Particle-based Granular Synthesis - Test 1.* URL: https://www.youtube.com/watch?v=b5xZ5Urc3kY (visited on 10/29/2019) (cit. on p. 16).

[25] *Schematic view of a SPH convolution.* URL: https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics#/media/File:SPHInterpolationColorsVerbose.svg (visited on 11/25/2019) (cit. on p. 7).

[26] *Singing Sand 2.0.* URL: https://ars.electronica.art/center/en/singing-sand/ (visited on 10/29/2019) (cit. on p. 16).

[27] *Sponsored Feature: Fluid Simulation for Video Games (Part 1).* URL: https://www.gamasutra.com/view/feature/132552/sponsored_feature_fluid_.php?page=2 (visited on 11/04/2019) (cit. on p. 6).

[28] *Ultrasubjective Space.* URL: https://www.teamlab.art/concept/ultrasubjective-space (visited on 10/29/2019) (cit. on p. 14).

[29] *Universe of Water Particles on Au-delà des limites.* URL: https://www.teamlab.art/w/large-waterparticles/ (visited on 10/26/2019) (cit. on p. 15).