Bachelorprojekt

A Simple 2D SPH Fluid Solver

Jan Hinsch

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Computer Graphics

Date: September 11, 2024

Supervised by Prof. Dr.-Ing. Matthias Teschner

Contents

Chapter 1

Introduction

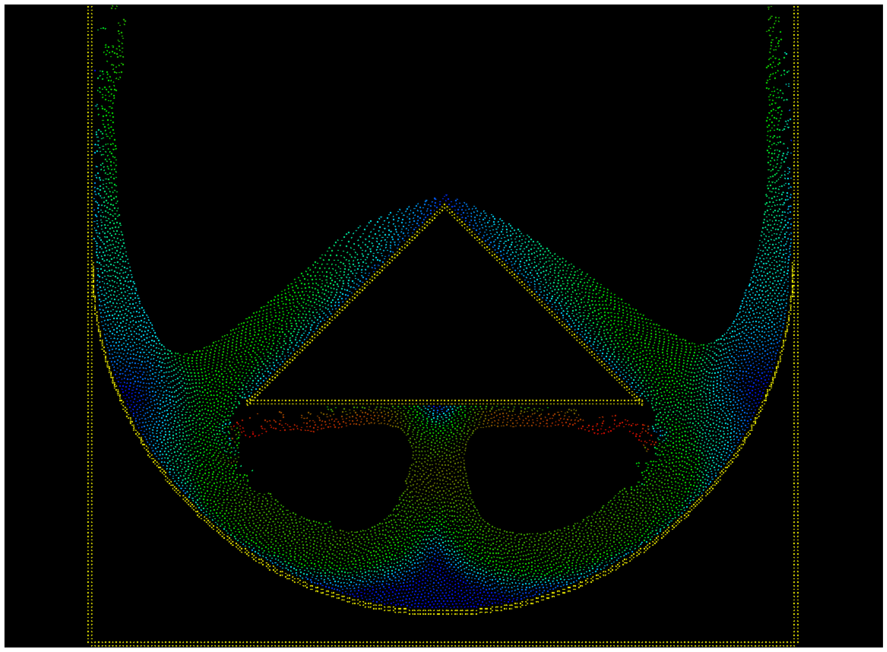


Figure 1.1: A fluid interacting with boundaries, simulated with my implemented fluid solver.

Fluids surround us everywhere - whether in the flow of a river, the smoke rising from a chimney, or the simple pouring of water into a glass. They all share a common mathematical basis. Understanding this mathematical framework becomes critical when our goal is to capture the dynamics and subtle details of fluid behavior for simulation. Whether our goal is to create visually stunning animations for movies or video games, or to apply them in scientific contexts, fluid simulation serves as a flexible and essential tool that bridges the realms of artistic creativity and scientific exploration.

In this project I implemented a simple Smoothed Particle Hydrodynamics (SPH) fluid solver from the ground-up. My primary goal was to deepen my understanding of fluid solvers, so I tried to limit myself to basic concepts. For the sake of simplicity I used an Equation of State (EOS) for the pressure calculation. This approach depends on fixed parameters, but since my goal is to deepen my understanding, it is an appropriate approach and is often used by people developing their first fluid solver.

Another goal for me was to enable real-time parameter adjustments (such as viscosity, gravity, stiffness constant, and surface tension) in the fluid simulation. In order for my fluid solver to achieve this with a reasonable number of particles, I quickly realized that a naive neighbour search of $\mathcal{O}(n^2)$ would not suffice. Consequently, I implemented Index Search to optimize the neighbour search process. Figure 1.1 shows an example scene of my fluid solver using the mentioned approaches.

In the following report I will show how my fluid solver works and how I implemented it. In Chapter 2 and 3 I will present a broad overview of the general concepts of a fluid solver as well as the underlying physics. Chapter 4 will focus on the SPH method. Next, I have dedicated Chapter 5 to highlight the importance for an efficient neighboursearch in SPH. Chapter 6 will cover the implementation details of my fluid solver, and in Chapter 7, I will analyze and evaluate the accuracy and performance of the fluid solver.

Chapter 2

Basic Concept of a Fluid Solver

In the real world, fluids are continuous entities, so in order to simulate fluid behavior in a computational setting, we must discretize the fluid into manageable elements. There are two main approaches for doing this. One where the simulation domain is defined by a fixed grid in space and at these fixed points changes in quantities such as velocity and pressure are tracked over time. This is based on Leonhard Euler's description of a fluid and is often referred to as the grid approach or Eulerian fluid. The other approach (and the subject of this project) is particle-based and is based on Joseph-Louis Lagrange's description of a fluid. Therefore it is often called Lagrangian fluid or particle fluid. Here we focus on tracking the motion of individual particles or elements within the fluid. In this approach, each particle represents a specific volume of the fluid (or a boundary) and is treated as a distinct entity with unique properties such as mass and velocity. Thus, the principle is that we compute forces for each particle in each iteration, and these forces act on the individual velocity of each particle, which in turn affects the position of each particle.

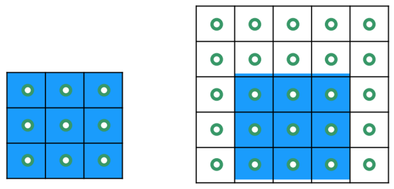


Figure 2.1: Concept of particle fluid on the left and grid fluid on the right. The blue box represents the fluid and the green dots are the sample points [7]

Both approaches divide the fluid into small volumes and compute velocity changes at the sample points, but the main difference is that a particle fluid solver advects its samples with the flow of the fluid. In the Eulerian approach, the samples do not move with the fluid, but remain fixed in space.

Because samples in particle fluids are advected with the flow, and because particles can also represent boundaries, the boundary/fluid interaction and the inclusion of free surfaces (the interaction of the fluid with air or, more simply, the surface of a fluid) are easier to implement.

There also exist hybrid methods that use both a fixed grid and particles, but I will not go into detail here.

Chapter 3

Governing Equations

The heart of all fluid solvers lies in the computation of accelerations. Here we have to solve partial differential equations that describe the motion of fluid substance such as liquids and gases. Before I explain the differential equations used in my solver, first, two assumptions about our fluid:

1. Assumption: Our Fluid is a Newtonian Fluid.

This means that the viscosity of our fluid remains constant regardless of the shear rate. This linear relationship simplifies the governing equations, making the calculations more straightforward.

There also exist Non-Newtonian fluids which do not have a constant viscosity and thus the computation of their behavior requires more complex models.

2. Assumption: Our Fluid is incompressible.

This assumption implies that the density of the fluid remains constant throughout the flow, meaning that the fluid's volume does not change under pressure. Mathematically, this is expressed as the divergence of the velocity field $\mathbf{u}$ being zero $\nabla \cdot \mathbf{u} = 0$, and is referred to as the incompressibility condition.

The first differential equation we use governs the position of particles:

$$\frac{dx_i}{dt} = u_i \tag{3.0}$$

It states that the time rate of change is equal to the velocity. Particle positions and their respective attributes are advected with the local fluid velocity.

The change in u_i of an advected particle over time is governed by the Lagrange form of the Navier-Stokes equation, which I will explain in the next section.

3.1 Navier-Stokes Equations

The differential equations that need to be solved are the incompressible Navier-Stokes equations, they are usually written as:

Momentum Equation

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} * \nabla \vec{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u} + \vec{g} \tag{3.1}$$

Incompressibility condition

$$\nabla * \vec{u} = 0 \tag{3.2}$$

In the following, I will explain the individual terms, operators and symbols used in the momentum equation (3.1) and the incompressibility condition (3.2).

3.1.1 Pressure

The first is the pressure term $-\frac{1}{\rho}\nabla p$. The general concept behind pressure is that high pressure regions push on lower pressure regions.

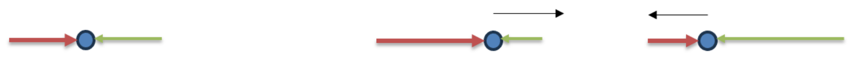


Figure 3.1: Pressure is applied to a particle (blue dot)

Figure 3.1 illustrates this principle. The two arrows represent pressures acting on a particle (the blue dot). In the left example, the two forces acting on the particle are of equal strength, so the particle remains stationary. In the two examples on the right, there is an imbalance of forces, with one force being stronger than the other. As a result, the particle is accelerated in the direction of the weaker force. To measure this behavior, we are interested in this imbalance of forces, so we use the negative gradient $-\nabla p$. Through this principle, pressure accelerations preserve the volume and density of the fluid.

The goal of pressure in a fluid solver is to tolerate only minimal deviations in density. This is a very important aspect that I will come back to.

3.1.2 Viscosity

The second fluid force is viscosity: $\nu \nabla^2 \vec{u}$. It is a frictional force between molecules and represents the resistance of a fluid to deformation or flow. High viscosity implies a thicker and more resistant fluid such as honey, while low viscosity implies a thinner and less resistant fluid such as alcohol. This force causes a particle to move at the average velocity of nearby particles, so the Laplacian operator ∇^2 is used here. ν is a hyperparameter, also often called kinematic viscosity, that governs the amount of viscosity our fluid will have.

3.1.3 External forces

The last term, written as $\vec{g}$, represents external forces. Almost always we use it for the gravitational force, but it can also be used for user input forces. So if we want to enable the user to interact with the fluid (e.g. with the mouse cursor), this could also be handled here.

3.1.4 Momentum Equation

We can derive the momentum equation from Newton‘s second law $\vec{F} = m\vec{a}$. If we imagine that every particle represents a blob of our fluid with each with its own properties such as mass m, volume V, and velocity $\vec{u}$, and we want to simulate the behavior of this single particle, basic physics tells us that $\vec{F} = m\vec{a}$ applies here. What we now need to compute are the forces $\vec{F}$ acting on each particle. For that we insert the three forces (pressure, viscosity, external forces) mentioned above as a sum multiplied by V. So we get:

$$-V\nabla p + V\mu\nabla^2\vec{u} + \rho * V\vec{g} = m * \vec{a}$$

The mass of an object can be computed with $m = \rho * V$, with ρ for density and V for volume. The acceleration term I will set to $\vec{a} \equiv \frac{D\vec{u}}{Dt}$. The big D notation is called the material derivative, which I will explain in a moment. For now this leads us to:

$$\rho * V\frac{D\vec{u}}{Dt} = -V\nabla p + V\mu\nabla^2\vec{u} + \rho * V\vec{g}$$

Dividing by $\rho * V$ and inserting our kinematic viscosity $\nu = \frac{\mu}{\rho}$ gives us

$$\frac{D\vec{u}_i}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\vec{u} + \vec{g} \tag{3.3}$$

This is the general form of the Navier-Stokes equation using the material derivative. The definition of the material derivative depends on whether we are working with advected Lagrangian sampling points (as in our particle fluid solver) or fixed Eulerian sampling points (in a grid fluid solver).

In our case we are working with advected sample points and thus the material derivative is simply defined as the total derivative $\frac{D\vec{u}_i}{Dt} = \frac{d\vec{u}_i}{dt}$. We can set the material derivative this way because our first differential equation $\frac{dx_i}{dt} = u_i$ holds, as stated in [2]. If we insert this definition into the general form of the Navier-Stokes equation, we get the Lagrangian form of the equation:

$$\frac{d\vec{u}_i}{dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\vec{u} + \vec{g} \tag{3.4}$$

Now the term on the left side translates to the Lagrangian question, as in [1], which is: How fast does the velocity change as a function of time for the particle whose position is given by $\vec{x}(t)$?

This is the form of the Navier-Stokes equation used in my project.

The definition of the material derivative changes when we work with fixed sample points: $\frac{D\vec{u}_i}{Dt} = \frac{\partial \vec{u}_i}{\partial t} + \vec{u}_i * \nabla \vec{u}_i$ this is the form we also see in (3.1). The additional term $\vec{u}_i * \nabla \vec{u}_i$ is called convective acceleration. Simply put, this term measures how much change is due solely to differences in the fluid flowing past a fixed location in space. However we do not need to compute this as we work with advected samples and can ignore this overhead term.

3.1.5 Incompressibility condition

As already mentioned we assume our fluid to be incompressible. This assumption implies that the fluid's density remains constant over time, regardless of the pressure changes within the fluid.

$$\nabla \cdot \vec{u} = 0$$

In practice, the incompressibility condition simplifies the Navier-Stokes equations because it eliminates the need to account for changes in fluid density over time.

As we will see later, this part can cause some problems in our approach to simulating a fluid. Due to the discrete nature of SPH, density can fluctuate significantly, leading to local compressibility even when the fluid is supposed to be incompressible. In my project, I used an equation of state to calculate pressure, which relates pressure to density. To enforce incompressibility, the pressure must be adjusted to counteract any density variations with a hyperparameter k, but I will go more into detail here later on.

Chapter 4

Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a versatile computational method that interpolates quantities at discrete sample positions and approximates spatial derivatives using sums over neighbouring particles. Although commonly associated with fluid simulation, SPH fundamentally serves as an interpolation and discretization scheme to numerically approximate the Navier-Stokes equations. The resulting forces, such as pressure and viscosity, are then used within a particle-based fluid solver.

Introduced in 1977 by Gingold and Monaghan, SPH was initially developed for astrophysical problems, such as the formation and evolution of proto-stars and galaxies [5]. Over time, it has become a powerful tool in various fields, including fluid dynamics.

In a particle fluid solver, SPH is primarily utilized to compute fluid forces, namely pressure and viscosity. Gravity, however, does not require SPH, it is modeled as a constant vector pointing downward.

The core of the SPH method is the kernel function, $W(x_i - x_j, h)$. This function weights the contributions of neighbouring particles based on their distance from the particle at x_i, enabling the approximation of an unknown quantity A_i at position x_i using known quantities A_j at positions x_j.

4.1 Kernel Function

The kernel function $W_{ij} = W(x_i - x_j, h)$ is as a weighting function that determines the influence of a sample j based on its distance x_j from x_i. If this distance is less than a fixed distance h, called the kernel support (or smoothing length), the weight is positive. If the distance exceeds the kernel support, the weight is zero:

$$W_{ij} = W\left(\frac{\|x_i - x_j\|}{h}\right) = W(q)$$

as stated in [7]. The parameter h is typically the particle size, and the kernel support is usually set between $2h$ and $5h$. In my project, I chose a kernel support of $2h$. This parameter is crucial because it controls how many neighbouring samples j are included in our SPH sums. A larger kernel support increases the number of neighbours, enhancing the accuracy of the simulation, but also increases the computational cost.

A kernel function is often at least twice differentiable and resembles a Gaussian function. Even though I am using finite differences in my project to approximate the second derivative of the kernel function, choosing a twice differentiable kernel remains advantageous.

A kernel function should satisfy the following properties, as outlined in [7]:

1. Normalization condition:
$$\int_\Omega W(x_j - x_i, h) dx_j = 1$$
This ensures that the kernel is properly normalized, so integrating the function over the entire space results in 1.

2. Compact support condition:
$$W(x_j - x_i, h) = 0 \quad \text{for} \quad \|x_j - x_i\| > h$$
The kernel function should be zero outside its support range.

3. Symmetry condition:
$$W(x_j - x_i, h) = W(x_i - x_j, h)$$
Symmetry ensures that the interaction between any two points is treated equally, regardless of direction.

4. Positivity condition:
$$W(x_j - x_i, h) \geq 0$$
As noted in [3], some kernel functions do not satisfy this condition, but negative kernel values may lead to incorrect mass density estimates. Thus, positivity is desirable.

5. Dirac-δ condition:
$$\lim_{h \to 0} W(x_j - x_i, h) = \delta(x_j - x_i)$$
Here, $\delta(x_j - x_i)$ is the Dirac-δ function, which is zero everywhere except when $x_j = x_i$, where it integrates to 1. This property ensures that as the kernel becomes narrower, it becomes more localized around the point x_i.

A popular kernel variant, which I also use in my project, is cubic spline:

$$W(x_j - x_i) = \alpha \begin{cases} (2-q)^3 - 4(1-q)^3 & 0 \leq q < 1, \\ (2-q)^3 & 1 \leq q < 2, \\ 0 & q \geq 2, \end{cases}$$

with $q = \frac{\|x_j - x_i\|}{h}$. The normalization factor α varies depending on the dimensionality of the simulation: for 1D, $\alpha = \frac{1}{6h}$; for 2D, $\alpha = \frac{5}{14\pi h^2}$; and for 3D, $\alpha = \frac{1}{4\pi h^3}$.

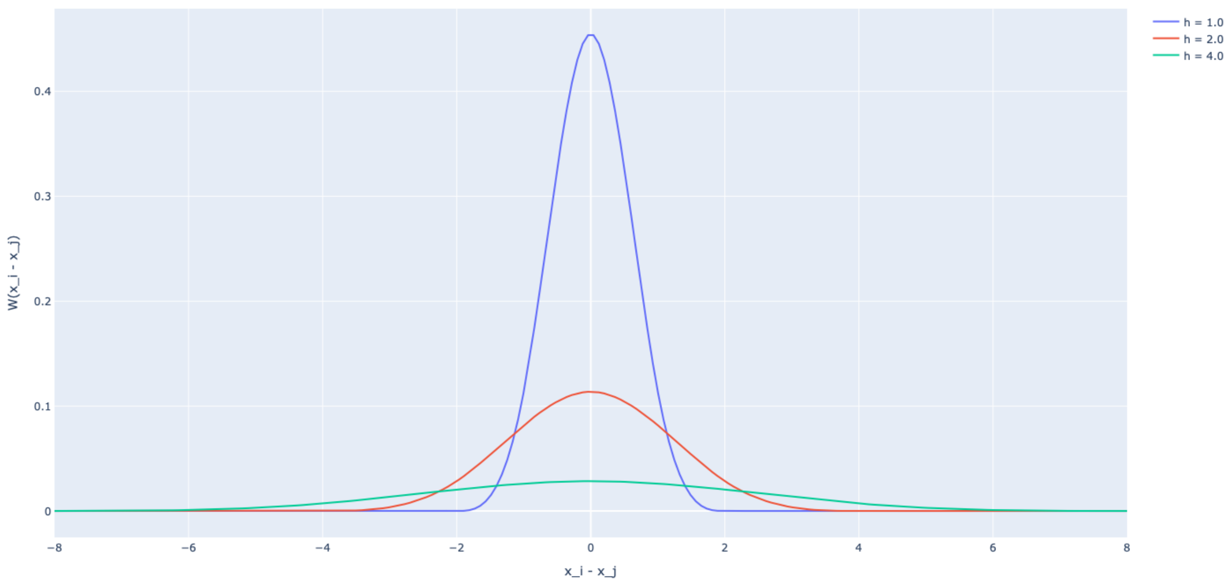


Figure 4.1: Plot of cubic spline kernel function with three different h values

In my project, the kernel function is primarily used to compute the density of individual particles.

4.2 Kernel Derivative

The kernel derivative is essential for computing the pressure and viscosity accelerations of individual particles. It is important to note that the gradient computation here results in a vector:

$$\nabla W(x_j - x_i) = \alpha \frac{x_j - x_i}{\|x_j - x_i\| h} \begin{cases} -3(2-q)^2 + 12(1-q)^2 & 0 \leq q < 1, \\ -3(2-q)^2 & 1 \leq q < 2, \\ 0 & q \geq 2. \end{cases}$$

In my project, the kernel derivative is also used to compute surface tension, which will be discussed in the corresponding chapter.

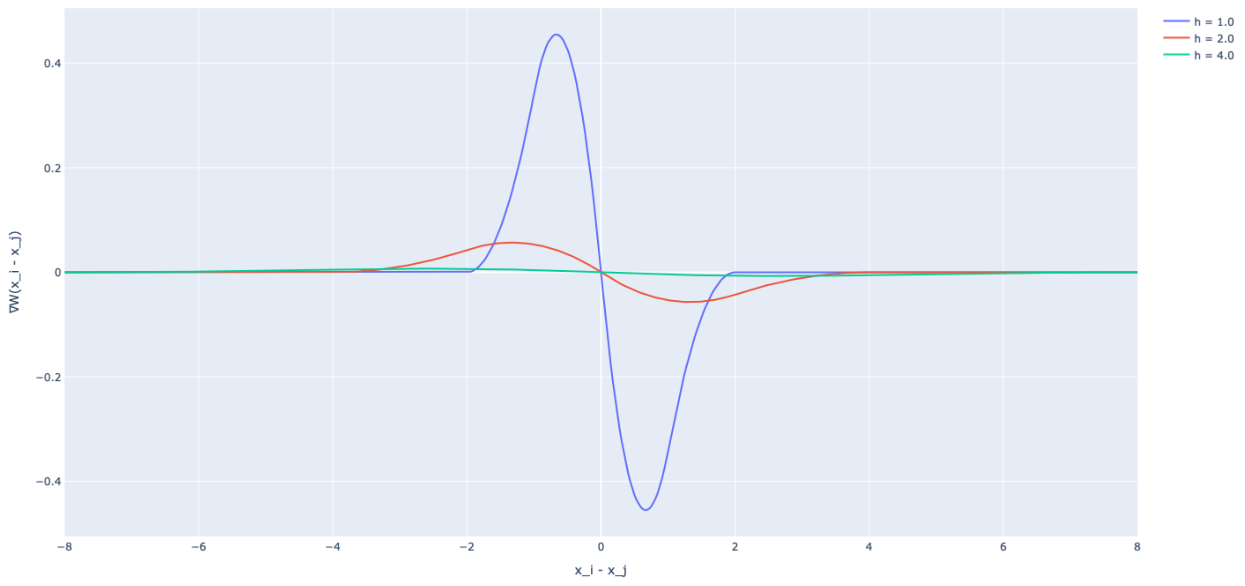


Figure 4.2: Plot of kernel derivative function with three different h values

4.3 SPH Discretizations

This section explains how each component, density, pressure acceleration and viscosity acceleration, is computed using SPH. Since the definitions for each component are continuous functions, we need to discretize them to compute their values numerically in our simulation. SPH achieves this discretization by approximating each component with sums over neighbouring particles j:

1. Density computation:
$$\rho_i = \sum_j m_j W_{ij} \tag{4.3}$$

2. Pressure acceleration:
$$-\frac{1}{\rho_i}\nabla p_i = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W_{ij} \tag{4.4}$$

3. Viscosity acceleration:
$$\nu\nabla^2\mathbf{u_i} = 2\nu\sum_j \frac{m_j}{\rho_j}\frac{(u_j - u_i)\cdot(x_j - x_i)}{(x_j - x_i)\cdot(x_j - x_i) + 0.01h^2}\nabla W_{ij} \tag{4.5}$$

In the computation of viscosity acceleration, the Laplacian ∇^2 is approximated using a first derivative. Direct computation of the second derivative is often error-prone due to particle disorder and the lack of sufficient samples to accurately approximate the second derivative of the kernel function. Thus, the Laplacian, inherently a second derivative, is approximated using a finite difference method based on the first derivative, a more robust approach.

The SPH forms shown here are only some of the possible variants. There are many valid SPH approximations for spatial derivatives, each with its own advantages and disadvantages.

4.4 Pressure Computation

To compute the pressure gradient, we first need to determine the pressure p_i from the density ρ_i. In my project, I use a state equation defined as follows [2]:

$$p_i = k \left[\left(\frac{\rho_i}{\rho_0} \right)^7 - 1 \right]$$

The term ρ_0 is a hyperparameter representing the desired rest density of the fluid. The density when the fluid is at rest, meaning no particles are moving, and no density deviation is present.

The constant k is a stiffness parameter. A higher k value increases the pressure response for a given deviation from the rest density, reducing the fluid's compressibility. Ideally, we aim for incompressibility, but due to approximation errors, perfect incompressibility is not achievable. Increasing k makes the fluid less compressible, but demands smaller time steps to maintain stability. In the Analysis chapter, I will explore this trade-off in more detail, discussing how k influences both the stability and accuracy of our fluid simulation.

Chapter 5

Neighbour Search

In this chapter, I will highlight the importance of an efficient neighbour search in SPH fluid solvers. In the previous chapter, I introduced the concept of approximating spatial derivatives of quantities using SPH sums. To effectively compute these sums, it is crucial to identify the neighbours of each particle within a certain distance, defined by the kernel support.

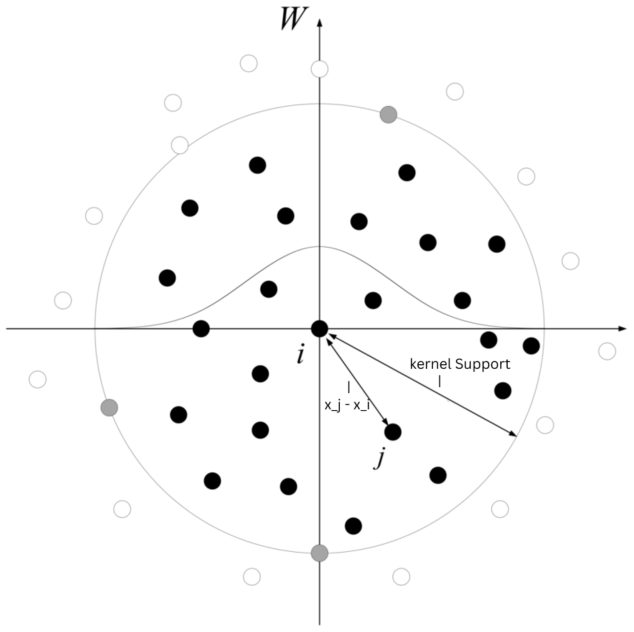


Figure 5.1: Visualization of the kernel support radius and kernel function. The particles within this radius (inside the gray circle) contribute to the SPH sums, weighted by the kernel function. [4]

Figure 5.1 illustrates this concept. All particles located within a distance $x_j - x_i$ smaller than the kernel support h must be considered in our SPH sums. The kernel function, depicted in the figure, determines the weight of each neighbouring particle's contribution based on its distance from the particle at x_i.

A naive approach to finding these neighbours would be to iterate over all particles in the simulation, resulting in a computational complexity of $O(n^2)$. This quadratic complexity quickly becomes impractical as the number of particles increases, making this method infeasible for simulations with a large number of particles. Efficient neighbour search algorithms are thus essential to reduce this complexity.

5.1 Uniform Grid methods

A commonly used concept to achieve efficient neighbourhood search is the uniform grid method. The general idea is to divide the simulation domain into a grid of uniformly sized cells, where each cell stores information about the particles inside it.

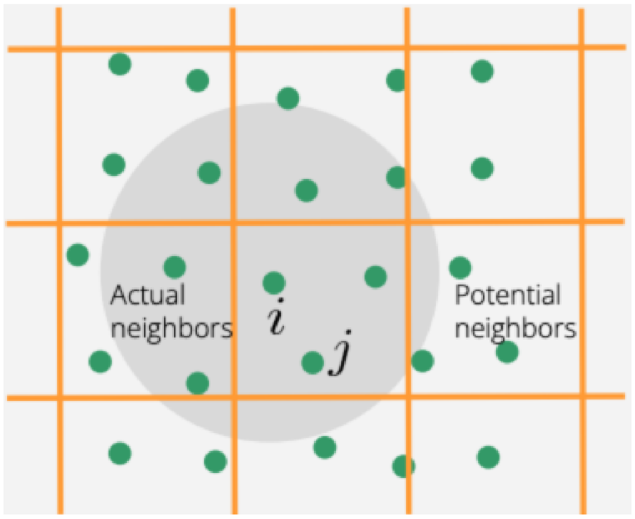


Figure 5.2: Uniform grid visualized. The gray circle around i highlights the kernel support. [7]

This concept can be seen in Figure 5.2 [7]. It is important that the edge length of a grid cell is equal to the kernel support. This ensures that we only need to check potential neighbours in the adjacent grid cells (9 in 2D and 27 in 3D, including the own grid cell) to find every actual neighbour.

In the next section I will introduce a way to actually implement the concept of uniform grids.

5.2 Index Sort

In this approach, the simulation domain is divided into a grid of cells, each of which can contain multiple particles. The general idea is to compute cell indices for all particles at each simulation step and sort the particles based on these cell indices. So from each particle's spatial coordinates, we determine the corresponding grid cell and compute its cell index, which is then stored with the particle.

Next, the particles are sorted according to their cell indices. This sorting ensures that particles within the same grid cell are adjacent to each other in the sorted list, and particles in neighbouring cells are also close to each other in this list.

To access particles within a specific cell, references are created from a list of grid cells to the particle list. For each grid cell, we store a reference to the first particle in the sorted list, as illustrated in Figure 5.3.

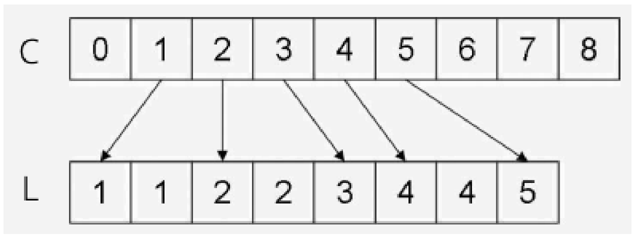


Figure 5.3: The two arrays used for Index Sort visualized. C is the array where each element represents a cell of the uniform grid. L is the particle list, sorted with respect to cell index. The arrows represent references pointing to the first particle in the sorted list. [7]

In the Implementation chapter, I will go into more detail on how I implemented this approach in my project.

Chapter 6

Implementation

I used C++ as the programming language and used the SFML Framework. In the following sections I will show how I tested the functionality of my solver and showcase my features.

6.1 Tests

It is essential for every fluid solver that we test the central components. In particular, the neighbour search, kernel functions, and fluid force computations must be tested. By doing so, we can identify and correct potential errors in the implementation that could otherwise be difficult to detect.

To perform these tests, I construct a $11 * 11$ grid of particles. Each particle is spaced at a distance h from its neighbours. This setup creates an ideal environment where each particle represents a volume of $\frac{1}{h^2}$. Through this ideal environment, certain properties and behaviors can be predicted and verified. If the tests pass, we can be confident that our implementation is free from significant bugs.

6.1.1 Kernel Functions

For the Kernel functions I constructed this ideal $11 * 11$ grid and performed the following tests on all particles inside the grid. As a reminder I implemented cubic spline as my kernel function, which is defined as follows:

$$W(q) = \frac{5}{14\pi h^2} \begin{cases} (2-q)^3 - 4(1-q)^3 & 0 \leq q < 1, \\ (2-q)^3 & 1 \leq q < 2, \\ 0 & q \geq 2, \end{cases}$$

with $q = \frac{\|x_j - x_i\|}{h}$.

To ensure the correctness of my implementation, I verify that the kernel function satisfies the following properties [7]:

- $W(3) = 0$: This represents the kernel function value applied to a particle outside of the kernel support.
- $W(0) = \frac{5}{14\pi h^2}\left((2-0)^3 - 4(1-0)^3\right) = \frac{20}{14\pi h^2}$: This represents the kernel function value when applied to the particle itself.
- $W(1) = \frac{5}{14\pi h^2}(2-1)^3 = \frac{5}{14\pi h^2}$: This represents the kernel function value for a neighbouring particle at a distance h.
- $W(\sqrt{2}) = \frac{5}{14\pi h^2}(2-\sqrt{2})^3 \approx \frac{1.005}{14\pi h^2}$: This represents the kernel function value for a neighbouring particle at a diagonal distance $\sqrt{2}h$.
- $\sum_j W_{ij} = \frac{1}{h^2}$: This sum represents the normalization condition, ensuring that the kernel function correctly sums to 1 over the particle's neighbours j.

The definition of my kernel derivative is:

$$\nabla W(x_j - x_i) = \alpha \frac{x_j - x_i}{\|x_j - x_i\| h} \begin{cases} -3(2-q)^2 + 12(1-q)^2 & 0 \le q < 1, \\ -3(2-q)^2 & 1 \le q < 2, \\ 0 & q \ge 2. \end{cases}$$

with $\alpha = \frac{5}{14\pi h^2}$.

I test the kernel derivative function to the following properties:

- $\nabla W((0,0) - (0,0)) = (0,0)$: Gradient of the same particle position should be zero.
- $\nabla W((0,0) - (h,0)) = -\nabla W((0,0) - (-h,0)) = \left(\frac{3\alpha}{h}, 0\right)$: Off-diagonal test.
- $\nabla W((0,0) - (0,h)) = -\nabla W((0,0) - (0,-h)) = \left(0, \frac{3\alpha}{h}\right)$: Off-diagonal test.
- $\nabla W((0,0) - (h,h)) = -\nabla W((0,0) - (-h,-h)) = \left(-\frac{1}{h\sqrt{2}}\alpha\beta, -\frac{1}{h\sqrt{2}}\alpha\beta\right)$: Diagonal test.
- $\nabla W((0,0) - (h,-h)) = -\nabla W((0,0) - (-h,h)) = \left(-\frac{1}{h\sqrt{2}}\alpha\beta, \frac{1}{h\sqrt{2}}\alpha\beta\right)$: Diagonal test.
- $\sum_j \nabla W_{ij} = (0,0)$: Sum of gradients from all neighbours should be zero $\rightarrow$ in ideal setting all gradients cancel each other out.

with $\beta = -(3)(2-\sqrt{2})^2$.

6.1.2 Neighbour Search

The neighbour search algorithm is tested using the constructed $11 * 11$ grid. The test focuses on a particle located in the middle of this grid. After setting up the grid, I perform my neighbour search for this central particle and count the number of neighbouring particles detected within 3 different kernel support radii:

- With a kernel support radius of $2.1h$ the central particle should find 13 neighbours.
- With a kernel support radius of $1.9h$, the central particle should find 9 neighbours.
- With a kernel support radius of $1.1h$, the central particle should find 5 neighbours.

If the neighbour search correctly counts the expected number of neighbours for each radius, the test will pass. Otherwise an error message is shown.

6.1.3 Fluid Force Computations

In this section, I show how I tested the functions responsible for computing the density ρ_i and pressure p_i of particles. The tests are again conducted in the ideal $11 * 11$ grid.

Given this setup, we expect that the density ρ_i computed for each particle should match the rest density ρ_0, as the fluid is in equilibrium. Since there are no density deviations in this scenario, the computed pressure p_i should theoretically be zero. This is because pressure forces are only active when there is a deviation from the rest density.

To account for numerical approximations, a small tolerance ϵ is integrated into the tests. If the computed density ρ_i is within ϵ of ρ_0 and the pressure p_i is within ϵ of 0, the test is considered successful.

6.2 Neighbour Search

The first step in every SPH Fluid simulation step is to compute neighbours. In this section want to elaborate how my implementation of index sort works.

I started by defining a class **Grid** which maintains a 2D array `C` that represents the uniform grid. `C` is initialized to zero for all grid cells and is subsequently updated based on the positions of the particles. Also, `C` is initialized with an extra element whose purpose will become clear soon.

For each particle, its position is used to compute the index of the grid cell it occupies. This is done using the following formula:

$$\text{cellIndex} = \left\lfloor \frac{\text{position.x}}{\text{cellSize}} \right\rfloor + \left\lfloor \frac{\text{position.y}}{\text{cellSize}} \right\rfloor \times \text{numCellsX} \tag{6.1}$$

Here:

- `position.x` and `position.y` are the x and y coordinates of the particle.
- `cellSize` is the width (and height) of each grid cell.
- `numCellsX` is the number of cells along the x-axis.

This index determines which grid cell the particle belongs to.

Once the grid cell index is determined, the corresponding element in the `C` array is incremented:

$$\texttt{C}[\text{cellIndex}] + = 1 \tag{6.2}$$

After a full iteration, the `C` array contains the number of particles in each grid cell.

Next, the particle counts in the `C` array are accumulated. Meaning that each entry in `C` now represents the cumulative number of particles up to that grid cell. The accumulation is performed in the following way:

$$\texttt{C}[i] = \texttt{C}[i-1] + \texttt{C}[i] \tag{6.3}$$

After accumulating `C`, the next step is to generate a sorted list of particles, `L`, based on their grid cell indices. This will be the sorted list that will allow efficient neighbour searches.

Using the accumulated `C` array, the particles are placed into the `L` array, which is essentially a list of pointers to the original particles. For each particle we lookup its corresponding cell and insert it into the `L` array at the position given by `C` for that grid cell, in the following way:

$$\texttt{C}[\text{cellIndex} - = 1]\texttt{L}[\texttt{C}[\text{cellIndex}] - 1] = \&\texttt{particle} \tag{6.4}$$

Now we have finished generating our 2 arrays, `L` contains the sorted particles and `C` contains the indexes of the first particle in the respective cell, as illustrated in Figure 5.3. From `C` we can determine the number of particles by subtracting two adjacent values from another. This is also the reason we added an extra element during the initialization of `C`, allowing us to deduce the number of particles located in the last grid cell.

To find potential neighbours, my implementation considers a $3 * 3$ grid of cells surrounding the particle's current cell. We first have to compute the grid cell index **??** of the particle we want to perform a neighbour search on, along with the respective indices of the adjacent cells. The indices of the adjacent cells are computed using an offset, defined in the following way:

$$\text{neighbourOffsets} = \begin{bmatrix} (-1,-1) & (0,-1) & (1,-1) \\ (-1,0) & (0,0) & (1,0) \\ (-1,1) & (0,1) & (1,1) \end{bmatrix}$$

$(0,0)$ represents the current cell our particle is located. The other 8 pairs are the adjacent cells.

With the index of the current cell and the index of a neighbouring cell, we now can access all the potential neighbouring particles located in that cell and perform distance checks. If a neighbouring particles distance is within the kernel support radius, it is added to the list of actual neighbours.

This approach drastically reduces the number of distance calculations required, as particles in distant grid cells are not considered.

6.3 Boundaries

We have not yet discussed how the fluid interacts with boundaries, which is essential for creating realistic fluid behavior. A fluid that does not interact with boundaries will not behave very interestingly. In SPH fluid solvers, boundaries are typically represented as particles, similar to fluid particles, but with a key difference: their positions are static and not advected over time. These boundary particles are included in the SPH sums as neighbours of fluid particles. When computing pressure accelerations my implementation differentiates between fluid particles and boundary particles.

Algorithm 1 Pressure acceleration (Input: Particle i, Neighbours, Pressure p_i)

```
1: for each particle $j$ in Neighbours do
2:     if $j$ is FluidParticle then
3:         $\mathbf{a}_i^p \mathrel{+}= -m_i * (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) * \nabla W_{ij}$
4:     end if
5:     if $j$ is BoundaryParticle then
6:         $\mathbf{a}_i^p \mathrel{+}= -p_i * \frac{2*m_i}{\rho_i^2} * \nabla W_{ij}$
7:     end if
8: end for
9: return $\mathbf{a}_i^p$
```

I use for all boundarys two layers of particles, in this way we can ensure that fluid particles located near a boundary have a complete neighbourhood.

6.4 Surface Tension

In addition to the other fluid forces, I have integrated a simple surface tension force into the solver. This force is computed using only the kernel derivative:

Algorithm 2 Surface Tension (Input: Particle i, Neighbours)

```
1: for each fluid neighbour $j$ of particle $i$ do
2:     sum ← sum + kernelGradient($particle_i.position$, $particle_j.position$)
3: end for
```

If a particle has a complete neighbourhood (i.e., it is surrounded by particles in all directions), the kernel gradients will cancel each other out. This results in no surface tension force on the particle. For particles located at the surface of the fluid, where the neighbourhood is incomplete (i.e., fewer neighbours on one side), the sum of the kernel gradients will not be zero. This results in an additional gradient force that points into the fluid.

6.5 Simulation step

The simulation step is the core of any fluid solver, where all the essential fluid accelerations and forces are computed. In my implementation, these computations are divided into four separate loops. Only the first two loops, which involve finding all neighbours and computing density and pressure, iterate over all particles, meaning both fluid particles and boundary particles. The other two loops focus solely on fluid particles and do not involve the boundaries.

The mass m_i needed for the computations is calculated during the initialization of the solver: $m_i = V_i * \rho_0$ with $V_i = h^2$.

The first step in every SPH fluid solver is to find neighbours. This is done as mentioned earlier.

The second loop is responsible for computing all the local values of particles, density and pressure. The pressure is calculated using the state equation 4.4. These local values are necessary for the third loop, which computes the fluid accelerations.

In the third loop, the accelerations acting on the fluid particles are computed. This includes the SPH sums for pressure acceleration and viscosity, as well as additional forces like gravity and surface tension. For pressure acceleration (4.4) is used. Viscosity is computed by (4.5) and surface tension by 3. At the end of this loop, all accelerations are combined:
$pressureAcceleration + viscosityAcceleration + gravity + surfaceTension = totalAcceleration$

As mentioned earlier, the gravity term does not require SPH computations and is simply defined as a vector $gravity = (0, -9.81)$

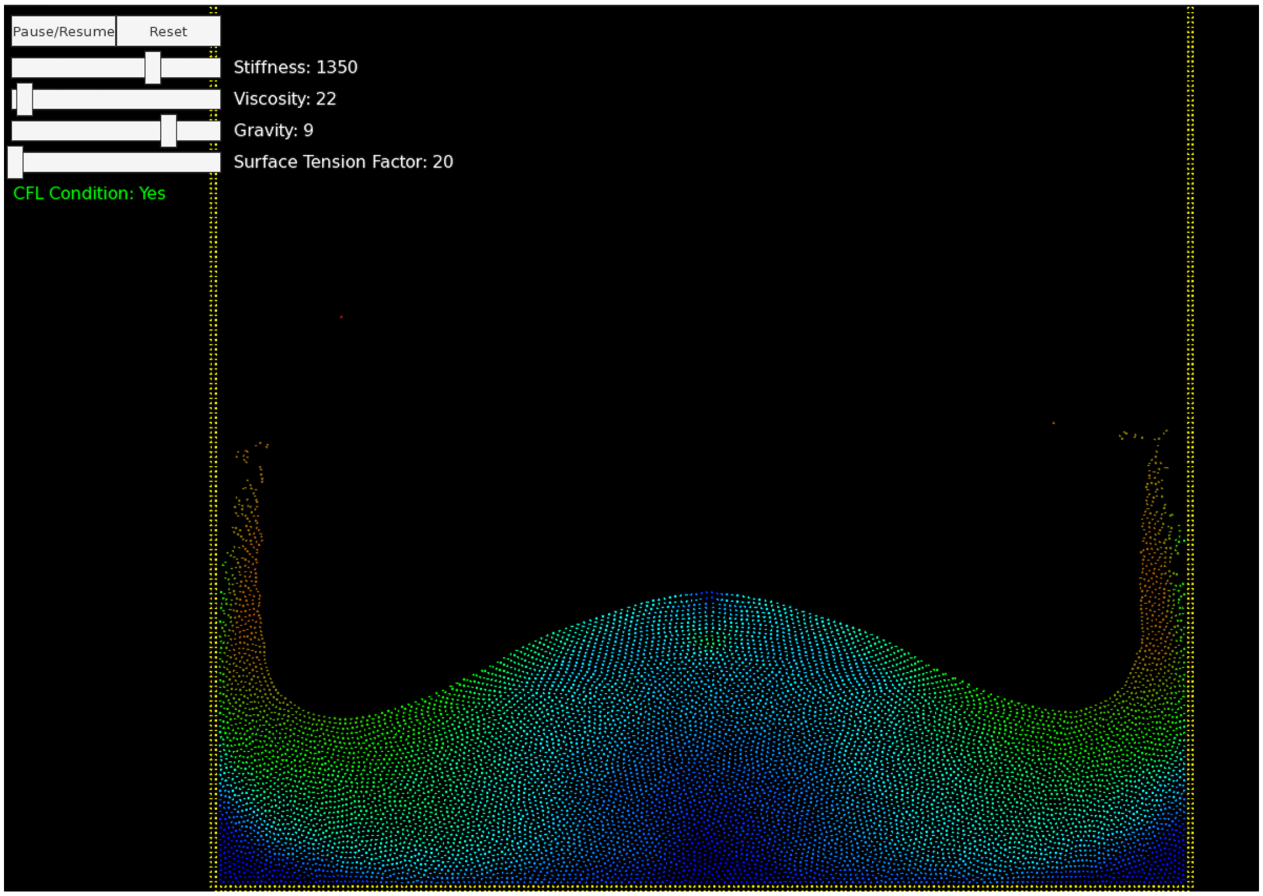
The final loop is the advection step, where the positions and velocities of the particles are updated. I use the Euler-Cromer integration scheme for this purpose. In this scheme, the velocity is updated first, followed by the position:

Algorithm 3 Simulation Step

```
 1: for each particle $i$ do
 2:     find neighbours $j$
 3: end for
 4: for each particle $i$ do
 5:     computeDensity()
 6:     computePressure()
 7: end for
 8: for each fluid particle $i$ do
 9:     computePressureAcceleration()
10:     computeViscosity()
11:     computeSurfaceTension() * surfaceTensionFactor
12:     totalAcceleration = pressureAcceleration + viscosityAcceleration + gravity + surfaceTension
13: end for
14: for each fluid particle $i$ do
15:     particle.velocity = particle.velocity + timeStep * particle.acceleration
16:     particle.position = particle.position + timeStep * particle.velocity
17: end for
```

6.6 Features

Here I will showcase the features I added into my fluid solver.



Stiffness: 1350
Viscosity: 22
Gravity: 9
Surface Tension Factor: 20
CFL Condition: Yes

Figure 6.1: Simulation run of the fluid solver with added features.

6.6.1 User Interface (UI)

In the top left corner of the screen, I integrated an UI that offers the following controls:

1. **Simulation Control**:
   The simulation can be paused and reset at any moment. When reset is pressed, all particles return to their initial positions, and their values and accelerations are reset.

2. **Parameter Adjustment**:
   The four slider below control the parameters of the simulation: stiffness parameter k, viscosity, gravity, and the surface tension factor. These can be changed while the simulation is running. The simulation immediately uses the new values. The numbers behind the slider show the current values.

3. **CFL-Condition Indicator**:

The UI includes an indicator for the CFL-Condition, displayed in green writing. If at any point the CFL condition is not fulfilled, this writing turns red.

6.6.2 CFL-Condition

The CFL-Condition is defined as:

$$\text{timeStep} \leq \lambda \times \left(\frac{h}{\text{getMaxVelocity(particles)}} \right)$$

This condition ensures that the time step is sufficiently small to maintain stability. If the condition fails, the time step must be reduced. It indicates that particles are moving too quickly, potentially leading to some being overlooked in SPH sums, resulting in an unstable and inaccurate simulation. In my implementation, this serves not as an analysis tool, but rather as a safety check to ensure that the simulation is running stable. Here λ is set to a fixed value between 0 and 1. We could also solve the equation 6.6.2 for lambda and get feedback about how stable our simulation is running but I chose this approach.

6.6.3 Additional Features

- **Particle Color Coding**:
  Fluid particles are color-coded according to their velocity, providing a visual representation of speed across the fluid. Boundary particles are colored yellow.
- **Movable Camera**:
  I implemented a camera as well. It can be controlled using the arrow keys, with zooming functionality mapped to the $\boxed{\text{A}}$ (zoom in) and $\boxed{\text{B}}$ (zoom out) buttons.
- **Data Logging for Analysis**:
  To facilitate analysis, each simulation run stores data, such as density values and parameter settings, in a text file. This data can later be used for analysis purposes.

Chapter 7

Analysis of results

In this chapter, I will analyze the parameters of my fluid solver, focusing on the stiffness parameter k and the time step parameter t. The setup used is a pillar of fluid particles inside a narrow tower built with boundary particles, as illustrated in Figure 7.1.

The analysis is done by averaging all density values of all particles. The following graphs will then show the relative error of the average density compared to the rest density. By varying the stiffness parameter k and the time step t, we can evaluate the accuracy of our fluid simulation. As mentioned earlier, our goal is to compute an incompressible fluid. However as our computations base on approximations , some error is inevitable. Visualizing this error allows us to evaluate the accuracy of our simulation. For the stiffness parameter analysis, a fixed time step t was used, and for the time step analysis, a fixed stiffness parameter k was selected.

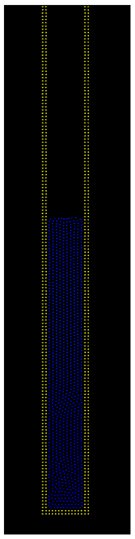


Figure 7.1: Pillar of fluid particles, inside the boundary tower.

7.1 Stiffness Parameter k

The stiffness parameter k directly affects the Equation of State, as discussed in 4.4. In theory, increasing k should result in a less compressible fluid. This behavior is confirmed in Figure 7.2, where the narrow tower setup was simulated with three different values of k.

As seen in Figure 7.2, a higher k value reduces the relative error in density, indicating decreased compressibility. However, this improvement in accuracy comes at a cost, as higher k values make the solver more sensitive to density changes, leading to larger pressure responses. Consequently, smaller time steps t become necessary to maintain the stability of the simulation.

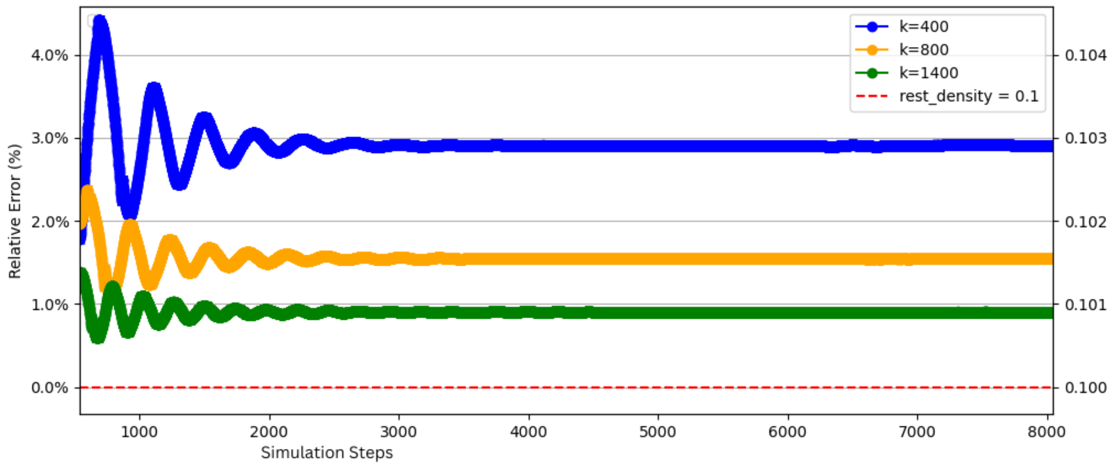


Figure 7.2: Relative Error in tower setup with three different values for k.

7.2 Time Step t

The time step t significantly influences the frequency of the relative error, as seen in Figure 7.3. Smaller time steps reduce the frequency of oscillations, which is expected since a smaller t limits how far a particle can move in a single simulation step, effectively slowing down the simulation. This generally enhances stability but increases computational cost, as the simulation takes longer to converge.

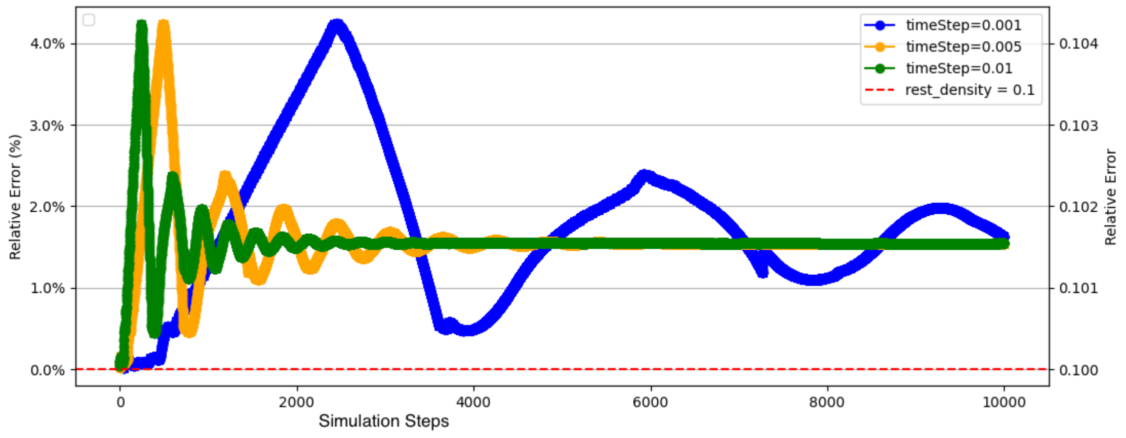


Figure 7.3: Relative error in the narrow tower setup for three different values of t.

On the other hand, larger time steps increase the frequency of oscillations. While this might initially suggest faster convergence, it also introduces a risk of instability, especially with high stiffness parameters k. The results in 7.3 highlight the balance required when selecting t. If t is too large, the simulation may become unstable, leading to inaccuracies of the simulation.

7.3 Neighbour Search

As already mentioned in chapter 5 I implemented Index Search for efficient neighbour detection. Here I compare my Implementation to the Naive approach of always iterating over all particles to detect neighbours. As shown in 7.4 my implementation drastically improves performance as distance checks are only done on particles located in adjacent cells.

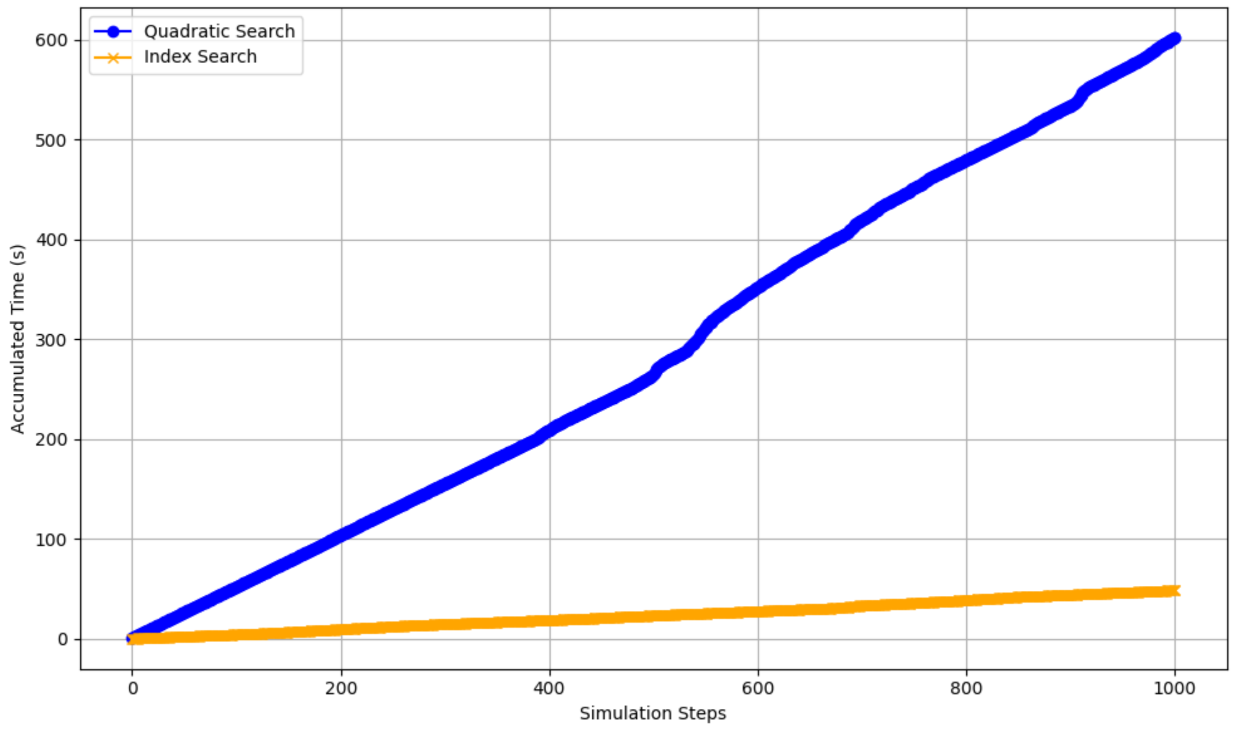


Figure 7.4: Performance comparison of Index Sort versus Naive neighbour search, with 7000 particles.

I have not integrated Space-filling Curves or Z-Index Sort [7] in my implementation, so particles in close cells are not necessarily close in memory. However from 7.4 it is evident that my implementation of Index Search is an improvement in efficiency over quadratic search.

Chapter 8

Conclusion

In summary, I successfully implemented a simple SPH Fluid Solver. I achieved my goal of enhancing neighbour search and enabling real-time interaction with simulation parameters. Analysis demonstrates that with high values of the stiffness parameter k, the solver can simulate incompressible fluids accurately, maintaining an acceptable density deviation, given that it uses an equation of state for pressure computation.

Figure 8.1 showcases a simulation run of my fluid solver with 15,000 particles.

To further enhance this solver, future work could focus on optimizing neighbour search efficiency with techniques like Compressed neighbour Lists [6]. Additionally, implementing more sophisticated pressure solvers, such as the IISPH method, which involves iteratively solving a linear system [8] would improve simulation accuracy and performance as well.

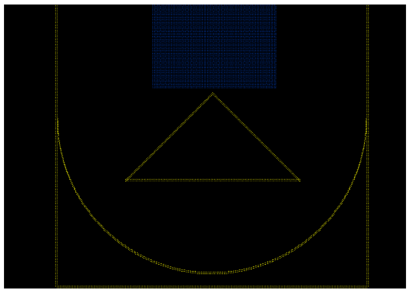


Image 1

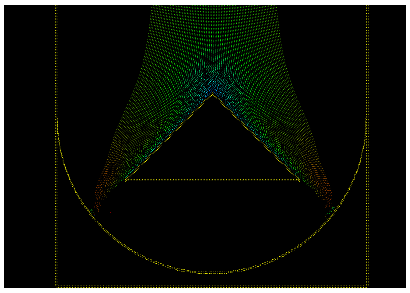


Image 2

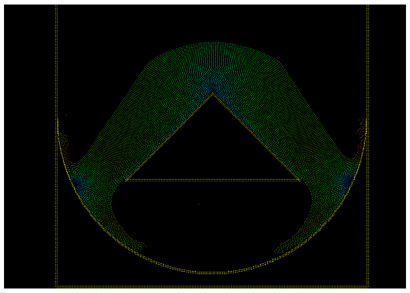


Image 3

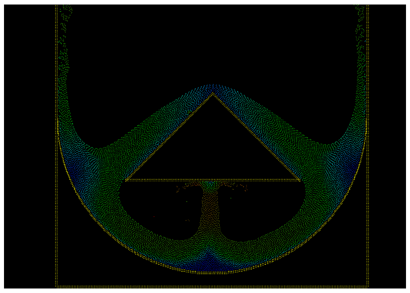


Image 4

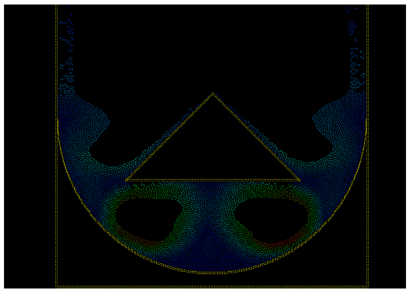


Image 5

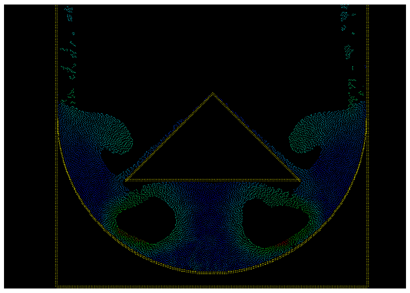


Image 6

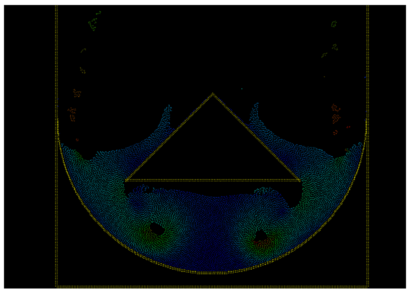


Image 7

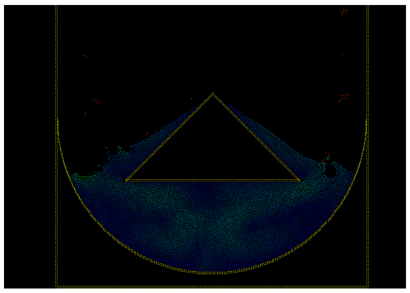


Image 8

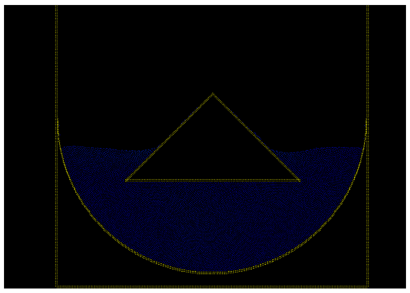


Image 9

Figure 8.1

Bibliography

[1] Robert Bridson. *Fluid Simulation for Computer Graphics.* Taylor & Francis, 2015.

[2] Markus Ihmsen et al. "SPH Fluids in Computer Graphics". In: *Eurographics 2014 - State of the Art Reports.* Ed. by Sylvain Lefebvre and Michela Spagnuolo. The Eurographics Association, 2014.

[3] Dan Koschier et al. "A Survey on SPH Methods in Computer Graphics". In: *Computer Graphics Forum* (2022).

[4] Hanliang Liang et al. "Experimental and numerical simulation study of Zr-based BMG/Al composites manufactured by underwater explosive welding". In: *Journal of Materials Research and Technology* (2019). URL: `https://www.researchgate.net/publication/337968459_Experimental_and_numerical_simulation_study_of_Zr-based_BMGAl_composites_manufactured_by_underwater_explosive_welding`.

[5] G. R. Liu and M. B. Liu. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method.* World Scientific, 2003.

[6] Matthias Teschner Stefan Band Christoph Gissler. "Compressed Neighbors". In: *Computer Graphics Forum* (2019), pp. 1–11. DOI: `10.1111/cgf.13890`.

[7] Matthias Teschner. *Particle Fluids.* Simulation in Computer Graphics, Computer Graphics Faculty of Engineering, University of Freiburg. URL: `https://cg.informatik.uni-freiburg.de/course_notes/sim_03_particleFluids.pdf`.

[8] Matthias Teschner et al. "Implicit Incompressible SPH". In: *IEEE Transactions on Visualization and Computer Graphics* 20.3 (2013), pp. 426–435.