**Investigating Real-time Fluid Dynamics Systems**

Haico Maters (200972727)

May 2024

BSc Computer Science (Game Eng)

Supervisor: Dr. Richard Davison

Word Count: 14,918

# Abstract

This dissertation delves into fluid simulation by developing a 3D computational fluid simulation in Unity3D, employing Smoothed Particle Hydrodynamics (SPH) methods. The dissertation delves into the SPH method, detailing its formulations aimed at achieving accurate representations of complex fluid behaviour, along with techniques employed to optimize simulation efficiency. The project encompasses a CPU-based implementation in C# as well as a parallelized GPU implementation using HLSL compute shaders. Performance assessments and comparisons between these implementations are conducted to assess the scalability of the SPH approaches. Additionally, potential enhancements to the fluid simulation and avenues for further research are identified.

Acknowledgements

I would like to thank my supervisor, Richard Davison, for being of assistance throughout the project.

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

Contents

[1 Introduction 5](#_Toc165722374)

[1.1 Motivation 5](#_Toc165722375)

[1.2 Aim and Objectives 5](#_Toc165722376)

[1.3 Dissertation Structure 6](#_Toc165722377)

[2 Background 8](#_Toc165722378)

[2.1 Fluid Simulation 8](#_Toc165722379)

[2.1.1 Approaches to Fluid Simulation 8](#_Toc165722380)

[2.1.2 Naiver-Stokes Equations 8](#_Toc165722381)

[2.2 Smoothed Particle Hydrodynamics 9](#_Toc165722382)

[2.2.1 Previous Work 9](#_Toc165722383)

[2.2.2 SPH Theory 9](#_Toc165722384)

[2.2.3 Smoothing kernels 10](#_Toc165722385)

[3 Implementation 12](#_Toc165722386)

[3.1 Particle Representation 12](#_Toc165722387)

[3.2 Calculating and Applying Forces 12](#_Toc165722388)

[3.2.1 Mass-Density 13](#_Toc165722389)

[3.2.2 Internal Forces 14](#_Toc165722390)

[3.2.2.1 Pressure per particle 14](#_Toc165722391)

[3.2.2.2 Pressure Force 15](#_Toc165722392)

[3.2.2.3 Viscosity 16](#_Toc165722393)

[3.2.3 External Forces 18](#_Toc165722394)

[3.2.3.1 Gravity 18](#_Toc165722395)

[3.2.3.2 Buoyancy 18](#_Toc165722396)

[3.2.3.3 Surface Tension 18](#_Toc165722397)

[3.2.4 Applying Forces and Movement of Particles 20](#_Toc165722398)

[3.2.4.1 Acceleration 21](#_Toc165722399)

[3.2.4.2 Velocity 21](#_Toc165722400)

[3.2.4.3 XSPH Particle Velocity Correction 21](#_Toc165722401)

[3.2.4.4 Collision Detection 22](#_Toc165722402)

[3.3 Spatial Hashing 22](#_Toc165722403)

[3.4 CPU and GPU Specifics 24](#_Toc165722404)

[3.4.1 CPU Implementation 24](#_Toc165722405)

[3.4.1.1 Spatial Hashing on the CPU 26](#_Toc165722406)

[3.4.2 GPU Implementation 28](#_Toc165722407)

[3.4.2.1 Spatial Hashing on the GPU 30](#_Toc165722408)

[3.4.2.2 GPU Sort 33](#_Toc165722409)

[3.5 Rendering 34](#_Toc165722410)

[4 Results and Evaluation 35](#_Toc165722411)

[4.1 Testing Environment and Methodology 35](#_Toc165722412)

[4.2 Testing Variable Setup 36](#_Toc165722413)

[4.3 Fluid Analysis 37](#_Toc165722414)

[4.3.1 Fluid Flow 37](#_Toc165722415)

[4.3.2 Comparison of Different Fluids 39](#_Toc165722416)

[4.4 Performance of CPU and GPU Implementations 40](#_Toc165722417)

[4.5 Performance of High Particle Counts on the GPU 44](#_Toc165722418)

[4.6 Known Issues 47](#_Toc165722419)

[5 Conclusions 48](#_Toc165722420)

[5.1 Objectives 48](#_Toc165722421)

[5.2 Skills Learnt 50](#_Toc165722422)

[5.3 Further Work 50](#_Toc165722423)

[Bibliography 52](#_Toc165722424)

# 1 Introduction

Fluid dynamics is the study of how liquids and gases flow and interact with their surroundings. Fluid dynamics simulations are widely adopted across many fields for various applications, including engineering [20], astrophysics [6], computer graphics [4], and medicine [21]. The use of fluid simulation is invaluable as it allows researchers and practitioners to accurately model and understand complex fluid behaviour, leading to advancements in design, analysis, and decision-making processes across these diverse domains.

Smoothed Particle Hydrodynamics (SPH) stands as a robust computational method for simulating fluid dynamics. It adopts a Lagrangian approach to computational fluid dynamics, discretizing the fluid into individual particles, enabling the simulation of fluid behaviour with high fidelity. SPH achieves this by modelling interactions among these particles to interpolate fluid properties, making it a versatile and efficient tool for simulating various fluid phenomena.

However, SPH presents significant challenges, particularly when simulating high-fidelity fluid dynamics with large numbers of particles. This results in substantial computational loads, underscoring the importance of optimizing computational efficiency for accurate and real-time simulations. Traditionally, SPH simulations have primarily relied on CPU-based implementations, which may struggle to deliver the required performance for real-time applications like games, especially as particle counts increase. Leveraging the parallel processing capabilities of modern GPUs offers a solution to these challenges, although it introduces numerous considerations and complexities.

## 1.1 Motivation

As previously mentioned, optimizing SPH poses numerous challenges. However, these challenges offer valuable opportunities to explore optimization methods, including leveraging parallel computing with GPUs. The skills and techniques acquired from optimizing SPH can be effectively applied to various other computing domains, facilitating the attainment of optimal performance across a wide range of applications.

Moreover, with Smoothed Particle Hydrodynamics being integrated into computer graphics, enhancing its performance could open avenues for its inclusion in video games. As SPH becomes more efficient and less resource-demanding, it can be integrated into games alongside other essential components like graphics and AI. This integration would enrich the gameplay experience while maintaining real-time performance.

## 1.2 Aim and Objectives

**T**he aim and objectives for this project are as follows:

**Aim**: To analyse and optimize the performance and computational efficiency of real-time fluid dynamics simulations using Smoothed Particle Hydrodynamics (SPH) on both CPU and GPU architectures.

**Objectives**:

1. Identify key features crucial for accurate fluid simulation using SPH, while emphasizing simplicity to optimize performance.

2. Research existing SPH based fluid dynamics simulation techniques to enhance performance and determine the most effective approaches to employ within the timeframe of the project.

3. Implement the researched techniques and identified features to develop a fluid simulation system using SPH.

4. Implement parallelized SPH algorithms tailored for GPU architectures to leverage their parallel processing capabilities and optimize memory access patterns.

5. Analyse the scalability of the optimized SPH simulations across varying particle counts, examining how computational demands scale with increasing particle count.

6. Compare the performance of CPU and GPU implementations in simulating fluid dynamics across varying particle counts in terms of framerates and memory usage.

Objectives 1 and 2 encompass the research phases of the project, addressing scientific and computational aspects, respectively. The implementation phases are outlined in objectives 3 and 4. Objective 3 entails developing a functional SPH simulation on the CPU to validate its behaviours. Subsequently, objective 4 extends this implementation to the GPU; maintaining identical behaviours with greater performance. Objectives 5 and 6 are aimed at evaluating the performance aspects of implementing SPH. Objective 6 involves a direct comparison of each implementation, showcasing their performance under identical conditions. Objective 5, on the other hand, delves into the scalability analysis of the SPH implementations, aiming to determine the extent to which the implementations can handle increasing computational demands. It also seeks to identify the maximum achievable performance using the GPU, particularly concerning its suitability for real-time applications.

The project objectives have undergone minor adjustments since the project proposal. Originally, Objective 5 encompassed monitoring memory usage as part of the scalability analysis across different particle counts. However, this aspect of memory monitoring has been relocated to Objective 6. This shift ensures that Objective 6 effectively addresses the specific memory usage concerns associated with CPU implementations, particularly relevant due to the computational demands of the Nearest Neighbour search (NNS), which are not as pronounced in the GPU implementation.

## 1.3 Dissertation Structure

The next section is the Background section (Section 2), which provides an overview of fluid simulation methodologies. This includes an exploration of the foundational principles through the Navier-Stokes equations and an examination of Smoothed Particle Hydrodynamics (SPH). Within this section, the historical development, basic theory, and fundamental mathematical approximations of SPH are discussed and explained in detail.

In Section 3, titled Technical Work, the discussion initiates with an overview of the employed technologies and the particle representation within the project. It then delves into the application and calculation of each force, providing specifics on the equations used and the rationale behind the decisions made, in choosing them. Lastly, it explores the optimization and implementation within the contexts of both CPU and GPU

Section 4, Results and Evaluation, the methodology for testing and result acquisition is elaborated upon. It provides an visual analysis of fluid behaviour and flow characteristics, followed by a comprehensive assessment of implementation performance based on scalability. Performance comparisons between the implementations are also included. Additionally, any known issues that exist are documented in this section.

The final section, Section 5 Conclusions, an assessment is made regarding the extent to which the final project aligns with its initial objectives. Furthermore, the acquired skills during the project's execution are reviewed, and potential avenues for future work are explored, taking into account any identified issues or areas for expansion.

# 2 Background

## 2.1 Fluid Simulation

### 2.1.1 Approaches to Fluid Simulation

In computational fluid dynamics simulations, there are three primary approaches: Eulerian, Lagrangian, and Semi-Lagrangian. Eulerian approaches such as that of Richter [30] involve discretizing fluid attributes, such as density and pressure, onto a fixed grid. Each grid cell may contain a portion of the overall fluid, and properties are computed at grid points. In contrast, the Lagrangian method [6] involves representing fluids as individual particles. These particles interact with each other, and their motions are governed by solving inter-particle forces. This approach allows for the simulation of complex fluid behaviours by directly tracking the movement of fluid elements. Semi-Lagrangian methods combine elements of both Eulerian and Lagrangian representations as seen in the paper by Xiu et al. [31]. They utilize particles to track fluid motion while also incorporating a grid-based approach to compute fluid properties efficiently.

This project's primary goal is achieving real-time fluid dynamics simulation using Smoothed Particle Hydrodynamics, a Lagrangian approach offering several advantages. One advantage is that Lagrangian approaches are useful to model free fluid flow, in comparison to an Eulerian approach where the simulation domain is limited to a single grid. Additionally, representing the fluid as particles allows for the simulation of complex fluid phenomena with high fidelity. This high fidelity is achieved because the Lagrangian approach directly tracks individual fluid particles, capturing intricate interactions with high accuracy. In terms of achieving real-time fluid dynamics, Smoothed Particle Hydrodynamics is suitable due to its computational efficiency, which complements its ability to model free fluid flow effectively compared to Eulerian approaches.

### 2.1.2 Naiver-Stokes Equations

The two fundamental equations describing the motion of fluids are the Naiver-Stokes equations, they are used to describe the fluid’s velocity field over time providing a precise mathematical model for most fluids occurring in nature the concept is explored in Stam’s paper from 2003 [14].

For Lagrangian Fluid with weakly compressible flow, the Naiver-Stokes equation takes the form:

represents the density of a particle multiplied by its acceleration . This acceleration is essential for determining the particle's new position at the subsequent time step, often achieved through integration schemes like semi-implicit Euler.

Given this to evaluate the equation to simulate a fluid in motion we require the following terms to be solved. The first term is which is the gradient of the pressure field , this determining the fluid’s movement in response to gradients within the pressure field. Next , accounts for the Laplacian of the velocity field multiplied by the viscosity , and this determines due to the force due to the viscosity of the fluid. The final term encompasses all external forces including gravity and surface tension. Subsequent sections of this dissertation will delve into the specifics of each term, elucidating their significance and methods for resolution.

## 2.2 Smoothed Particle Hydrodynamics

### 2.2.1 Previous Work

Smoothed Particle Hydrodynamics (SPH) was first invented by Gingold and Monaghan [6] in 1977, with the initial purpose of simulating phenomena in astrophysics. The method was further developed by Monaghan [11] in 1989, improving particle methods including particle velocity correction using a technique named XSPH. Monaghan [5] in 1992 introduces many of the first use of SPH approximation formula’s used to solve the Navier-Stokes equation for weakly compressible flow. In 1996, Desbrun and Gascuel[3] extended Monaghan’s paper to incompressible fluid flows. Additionally, this paper proposed a symmetrical way to calculate the pressure force, by combining the ideal gas law with rest density preserving Newton’s third law of motion.

In 2003, Müller et al. [2] proposed enhancements to Newtonian fluid techniques, including the calculation of fluid surface tension by evaluating gradients of a smoothed colour field. A significant optimization occurred with the introduction of Spatial Hashing by Teschner et al. in 2003 [7], providing an efficient solution to the near neighbour search problem.

In 2007 a breakthrough in the parallel processing of SPH was made when Harada et al. [8] managed to develop a method to search for neighbour particles on the GPU allowing for the implementation of SPH simulations entirely on the GPU. This outperformed earlier GPU implementations, such as Amada et al. [9] in 2003, which only utilized the GPU for force calculations. Further advancements in GPU implementations were facilitated by the release of the CUDA programming language by Nvidia in 2007, as discussed by Hérault et al. [10].

This project draws heavily from the works of Micky Kelager (2006) [1] and Müller et al.[2] (2003) [4], providing insights into SPH implementation on the CPU, focusing on key approaches and fundamentals. The GPU implementation is informed by Nvidia's documentation on particle simulation using CUDA during 2010 [12]. While this project utilizes compute shaders in Unity3D for atomic operations, many principles from CUDA implementations remain applicable, specifically the method for neighbour search on the GPU.

The primary components critical for accurate fluid simulation, along with their corresponding calculation methods outlined for this project, include pressure forces[3], viscosity[2], surface tension[2], gravity, buoyancy[1], and particle velocity correction via XSPH[11]. Furthermore, Spatial Hashing [7] emerges as the most efficient technique for enhancing performance in fluid dynamics simulations, to be employed in this project.

### 2.2.2 SPH Theory

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian simulation method for obtaining approximations of the equations of fluid dynamics providing a solution to the Naiver-Stokes equations from section 2.1.2 using these approximations. Widely adopted across various fields such as engineering[20], astrophysics[6], computer graphics[4], and medicine[21], SPH serves as a versatile tool for simulating fluid behaviour.

In SPH the fluid material is represented as discrete particles. Each particle shares many properties in common with regular particle systems, such as positions, velocities and masses, but also contains more specific properties such as density and pressure. Each of these properties evolve over time according the laws of physics.

SPH considers pairwise interactions between each of the particles through forces, like pressure gradients and viscosity. These forces play a crucial role in shaping the movement and behaviour of individual particles within the fluid system, effectively emulating fluid dynamics.

The integral interpolant of any quantity function, is defined by [1]:

Here is a point in , and is a smoothing kernel with as the smoothing radius. The smoothing radius serves as a scaling factor controlling the smoothness or roughness of the kernel, essentially determining the width of the weighting function.

The summation interpolant, along with its gradient and Laplacian, are given by [15]:

It's noteworthy that the gradient and Laplacian operators are applied solely to the kernel function, as per the rules of differentiation, where, becomes zero upon differentiation.

is a representation of an attribute of a certain particle in the position , with denoting the mass. These equations enable specific attributes to be substituted to derive a Smoothed Particle Hydrodynamics (SPH) approximation for that attribute. For instance, the SPH approximation for mass-density can be computed as:

### 2.2.3 Smoothing kernels

In Smoothed Particle Hydrodynamics (SPH), smoothing kernels are essential mathematical functions utilized to spatially smooth or interpolate properties of particles across a continuous domain. Different kernels are often employed at various stages of the simulation to achieve desired effects.

According to Monaghan [5], an appropriate kernel must satisfy two crucial properties:

1. Normalization:

This property ensures that the total influence of the kernel on a particle is properly distributed, preserving mass conservation within the system.

2. Convergence to Delta Function:

is Dirac’s delta function:

This ensures accurate spatial interpolation and differentiation, especially in regions of high particle density or sharp gradients.

Additionally, Kelager [1] introduces further properties:

Non-Negativity:

This property ensures that the kernel always produces positive values, preserving physical consistency.

Symmetry/Evenness:

This property ensures that the kernel is even, which makes sure that symmetry is enforced ensuring invariance under rotations of the coordinate system. The kernel width is used as the compact support radius for all smoothing kernels, which means . Therefore SPH particles can only be affected by other particles within their smoothing radius .

Despite the importance of these properties, there's no consensus on the optimal kernels for SPH simulations, balancing performance and accuracy. As discussed by Ihmsen et al. [4], different kernels may be suitable for different scenarios. In this project, the smoothing kernels utilized include the 6th polynomial, the spiky, and the viscosity kernels used by Kelager [1].

# 3 Technical Work

This section starts by covering the specific SPH approximations and equations used and implemented covering the reasoning behind these decisions, detailing any assumptions that have been made. Also, covered in this section are the specific details of each implementation CPU and GPU, the overall structure of the simulation loop and any optimisation techniques used in detail.

Both implementations are conducted within Unity3D, with the CPU version developed using C# script components and the GPU counterpart executed in DirectX 11, leveraging compute shaders scripted in HLSL.

Given the project focus on fluid dynamics and its objectives, there are no direct ethical considerations to address as there is no involvement of external parties or environmental issues that may be raised by the project.

## 3.1 Particle Representation

The number of particles used in each simulation is predetermined and remains fixed throughout runtime, unable to be dynamically adjusted during the simulation. Therefore, it must be modified via the Unity editor before initiating the simulation. Generally, each particle on the GPU can be conceptualized simply as a position in space, serving as a point from which fluid properties can be interpolated by using the other variables (mass, velocity etc.).

On the CPU, particles are represented as structs comprising an ID, position, velocity, resultant force variables, and a C# list of neighbouring particle ids. However, on the GPU, particles are represented differently, with each variable listed in the CPU representation (except IDs and neighbouring particles) stored in separate buffers. This approach on the GPU is beneficial because it avoids binding each buffer to every compute shader kernel, thus enhancing performance and memory efficiency.

By separating the variables into distinct buffers, GPU compute shaders can efficiently access and manipulate individual properties of particles without needing to access unnecessary data. This streamlined access contributes to faster computation times and overall improved performance during simulations, assisting in achieving real-time fluid dynamics. More specific information regarding these representations will be explored further in the relevant CPU and GPU sections (3.4.1 and 3.4.2).

Additionally, the simulation includes variables shared by all particles, such as mass, smoothing radius, particle radius, and colour. These shared variables ensure consistency across the simulation, simulating a cohesive fluid with uniform properties, such as water. By maintaining uniformity in these properties across all particles, the simulation accurately represents the behaviour of a singular fluid entity.

## 3.2 Calculating and Applying Forces

The main contributor to the overall accuracy of the simulation in the process of calculating and applying forces. Drawing from the methodologies outlined by Kelager [1], Müller et al. [2], and other contributors such as Ihmsen et al. [4], Monaghan [11], and Priscott [13], this project adopts established approaches in force computation and application.

Table 3.1 contains parameters, their corresponding symbols, units, and their representation in both C# and HLSL, which play pivotal roles throughout this section for reference:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Symbol | Code Representation | Units |
| Position |  | Vector3/Float3 |  |
| Velocity |  | Vector3/Float3 |  |
| Acceleration |  | Vector3/Float3 |  |
| Accumulated Force |  | Vector3/Float3 |  |
| Gravity |  | Vector3/Float3 |  |
| Particle Mass |  | Float |  |
| Density |  | Float |  |
| Rest/Reference Density |  | Float |  |
| Pressure |  | Float |  |
| Gas Stiffness (Or Bulk Modulus or Tait’s parameter) |  | Float |  |
| Viscosity |  | Float |  |
| Smoothing Radius |  | Float |  |
| Buoyancy Diffusion Coefficient |  | Float | - |
| Timestep |  | Float |  |
| Surface Tension Threshold |  | Float |  |
| Tension Coefficient |  | Float | - |
| XSPH Smoothing Constant |  | Float | - |
| Collision Dampening Factor |  | Float | - |

Table 3.1. Anything with units as “–“ are dimensionless having no units, usually just being a coefficient or factor.

### 3.2.1 Mass-Density

In a fluid simulation, the mass-density is a fundamental parameter that continuously changes throughout the simulation. It serves as a cornerstone for various calculations and is vital for accurately simulating fluid behaviour.

Densities are computed for each particle in the simulation domain on every iteration. These densities are stored in a cached array, with the lookup for particle corresponding to its position in the array, aligned with the position and velocity arrays for fast lookups.

The density at position  can be found using this SPH approximation by substituting density into the standard SPH approximation [1]:

This simulation assumes that all particles have the same mass, therefore to optimize computation the equation is simplified to:

This simplification has also been applied where applicable in the program for other calculations, such as forces from pressure gradients and viscosity. However, for this dissertation the non-simplified versions of these equations will be used when discussing them to stay in line with standard Smoothed Particle Hydrodynamics (SPH) formulations.

For this simulation I have chosen to use a 6th degree polynomial kernel from [2]. The reason for this decision is while a kernel such as Q-Spline may provide higher computational accuracy, that kernel requires evaluation of the square root, which can be expensive as high particle counts are reached.

The kernel alongside its gradient and Laplacian are defined as follows:

Kernel:

Gradient:

Laplacian:

### 3.2.2 Internal Forces

There are two force contributions that only arise from within the fluid, these are the pressure and viscosity force density fields. The internal fluid force densities are calculated by computing the derivatives of the quantity fields by using SPH.

#### 3.2.2.1 Pressure per particle

Like density, pressure is computed for each particle in every simulation iteration. These values are stored in a cached array, aligned with particle positions, velocities, and densities. This setup ensures efficient access to pressure data during simulation.

One approach to calculate pressure is to use a modified version of the ideal gas state equation proposed by Desbrun and Gascuel [3]:

However, for this project, I have chosen to use Tait’s Equation which also considered to be the preferred SPH approximation according to Ihmsen et al. [4]. Tait’s Equation is formulated by:

,

Where is the desired rest density of the fluid (also known as the reference density), is the gas stiffness that scales the pressure. Higher values of leads to the fluid being less compressible but also however demands lower small integration time steps. For this project the default value for used is 3.

Both equations allow for representing attractive and repulsive pressure forces. If the density of particle is lower than the desired rest density, the pressure value will be negative.

Tait’s Equation was selected for its superior stability and minimal impact on performance compared to simpler approximations. Despite offering greater computational accuracy, the trade-off was deemed worthwhile due to the enhanced stability, which enabled larger time steps. Initially, the simpler approximation was used resulting in instabilities caused by sharp density gradients.

#### 3.2.2.2 Pressure Force

Having calculated pressure and densities, we can now determine the pressure forces exerted by the fluid particles on each other. These forces represent the interactions between particles due to differences in pressure and density, influencing the motion and behaviour of the fluid system.

By simply substituting pressure into the SPH approximation equation we get:

However, this formulation poses some challenges. An essential aspect of pressure forces is that they must obey Newton’s third law of motion, ensuring that two particles exert symmetrical forces upon each other. However, this equation fails to achieve this symmetry because it only considers the pressure of neighbouring particles, neglecting the particle's own pressure. Consequently, if two particles with differing pressure values interact, the resulting forces would not be equal and opposite.

To address this limitation, we adopt a different equation proposed by Monaghan in 1992 [5]:

This equation accounts for the pressure of both the particle itself and it neighbours , scaled by their respective densities. By incorporating the particle’s own pressure, it is ensured that the forces exerted between particles are symmetrical, fulfilling the requirements of Newton’s third law of motion leading to more accurate simulations.

It's crucial to use a different kernel function for smoothing compared to the one employed for density calculations. Using the gradient of the default kernel can lead to particle clustering in high-pressure regions, necessitating an alternative smoothing kernel to mitigate this issue.

In this project, the chosen kernel function aligns with Kelager's [1] formulation:

Gradient:

Laplacian:

The limits have only been written in one dimension only and for simulation purposes, only the gradient of the spiky kernel is utilized in the project, as it's the sole requirement for pressure force calculations.

#### 3.2.2.3 Viscosity

Viscosity, a measure of a fluid's resistance to flow, plays a crucial role in fluid dynamics simulations, influencing flow behaviour significantly. Essentially, viscosity reflects the "thickness" or "stickiness" of a fluid, determining how it flows under different conditions. Viscosity forces emerge from the internal friction between fluid layers as they move relative to each other, impeding deformation or flow.

Through substitution the SPH approximation for the viscosity term is:

However, much like substitution with the pressure approximation this formulation is asymmetric and fails to conserve Newton’s third law of motion. To address this, Müller [2] proposed a solution by using velocity differences instead of absolute velocities:

Here, the coefficient determines the fluid’s viscosity strength. Higher values of result in greater resistance to flow, akin to a thicker fluid. For simulating water-like fluids a value of is used.

To maintain stability in the simulation and avoid introducing energy into the system, it's crucial to ensure that the Laplacian of the smoothing kernel remains positive. This guarantees that viscosity forces act solely as damping terms, opposing the flow direction.

A suitable smoothing kernel, adapted from Kelager [1], is employed:

Kernel:

Gradient:

Laplacian:

The limits have only been written in one dimension only and only the Laplacian of the viscosity kernel is utilized in the project, as it's the sole requirement for viscosity force calculations.

### 3.2.3 External Forces

In Smoothed Particle Hydrodynamics (SPH) simulations, external forces play a crucial role in influencing particle behaviour. These forces, distinct from internal forces, are applied after the calculation of internal interactions, with one notable exception – surface tension.

Typically, external forces, such as gravity or applied forces, are aggregated into a single sum, simplifying their application within the simulation. However, surface tension, despite being categorized as an external force, is computed alongside internal forces. This approach aims to optimize simulation efficiency by minimizing the number of neighbourhood searches required. By integrating surface tension calculations with internal force computations, the simulation process becomes more streamlined, enhancing overall performance without sacrificing accuracy.

#### 3.2.3.1 Gravity

Much like other particle systems the influence of gravity is a critical consideration, given its impact on particle dynamics. Unlike traditional gravity calculations based on mass and weight, SPH employs density rather than individual particle mass to determine gravitational forces.

The force from gravity is calculated as follows [1]:

Here, density takes precedence over mass, as it better reflects the amount of fluid material within a specified volume. By using density, SPH captures the fluid-like behaviour more accurately, ensuring realistic gravitational interactions within the simulation. This approach enhances the fidelity of SPH simulations, aligning them more closely with real-world fluid dynamics.

#### 3.2.3.2 Buoyancy

Buoyancy is a phenomenon crucial for accurately simulating fluid behaviour, particularly in gaseous fluids. While buoyancy traditionally arises from temperature differentials leading to density variations, SPH simulations typically assume isothermal conditions. To emulate buoyancy artificially, an additional term is introduced into the force calculations.

The artificial buoyancy is formulated as follows[1]:

The parameter , known as the buoyancy diffusion coefficient, controls the strength of the buoyancy effect. It ensures that particles become buoyant when their mass-density falls below the desired rest density. However, in non-gaseous fluids like liquids, where temperature differentials are negligible, is typically set to 0, effectively nullifying the buoyancy force.

#### 3.2.3.3 Surface Tension

Surface tension, a prominent phenomenon in fluid dynamics, describes the cohesive forces present at the interface of a liquid, resulting in an inward force experienced by molecules at the surface. This force creates a distinctive behaviour in fluids, akin to an elastic membrane, particularly evident at fluid interfaces with other substances. Accurately incorporating surface tension is crucial for achieving realistic fluid behaviour in simulations.

The forces exerted by surface tension act perpendicular to the fluid surface's inward normals and vary based on the surface curvature. For this project, Müller et al.'s [2] method for calculating surface tension is adopted. The surface tension force at particle is determined by:

At particle , signifies the inward surface normal, denotes the smoothed value of the colour field at that point, and is the tension coefficient. This coefficient is determined by the properties of the interfacing fluids (e.g., water and air), influencing the behaviour and strength of the surface tension effect at the boundary.

The colour field is an additional field quantity with at exactly the particle locations and elsewhere.

The SPH formulation of the smoothed colour field at particle is:

The normal of the inward surface normal of the fluid can be received by evaluating the gradient of the smoothed colour field:

The curvature of the surface depends on the divergence of the surface normal field given by:

The surface traction , is the force per area acting on a given location on the fluid’s surface. Müller [2] uses this equation to calculate the surface traction:

Multiplying the surface traction by a normalized scalar field ensures that the force density is spread onto all potential particles, resulting in the surface tension force. This works as gets smaller the further particle is away from the surface.

This finally results in the original surface tension force equation shown at the beginning of this section. The equation is solved by substituting in both and into the equation with taking the form:

In addition to this one issue must be addressed to ensure that there is no numerical instability. This instability issue occurs as becomes numerically unstable as . To prevent this the force from surface tension is only computed when this condition is fulfilled:

Where is some threshold relating to the particle concentration and must be greater than 0. For the simulation of a water-like fluid a default threshold value of 7.065 is used.

Notably, in this project the surface tension computation is integrated into the internal force calculations, alongside pressure and viscosity, reducing computational overhead by minimizing the total neighbourhood searches carried out each timestep.

### 3.2.4 Applying Forces and Movement of Particles

Now that all the forces have been computed, we can proceed to solve the Navier-Stokes equation for Lagrangian Fluid with weakly compressible flow [14]:

The terms on the right side of the equation can be rewritten as the total force acting upon a particle, which is the sum of all internal and external forces acting upon it:

Here, the internal force covers the and terms of the equation and covers the term in the equation.

After calculating the forces acting on each particle, it's essential to determine their new positions based on these forces. This involves integrating the forces to calculate the particle's acceleration and subsequently updating their velocities and positions. Additionally, any other relevant adjustments to particle movement are made in this stage to ensure an accurate simulation of particle behaviour.

#### 3.2.4.1 Acceleration

Acceleration derives from Newton’s second law of motion: , where the sum of all forces is divided by the density.

Similar to the reasoning explained in section 3.2.3.1 regarding gravity, mass-density is employed instead of mass for consistency and accuracy in simulations.

This can also be derived from the Naiver-Stokes equation explained in the introduction to this section (3.2.4 Applying Forces and Movement of Particles), where the substitution of the right side for is conducted. Then a simple assignment and can be carried out with a simple rearrangement arriving at the acceleration formula.

#### 3.2.4.2 Velocity

For the updating of the velocity and position of the particle, the semi-implicit Euler scheme is used

Where is the velocity and is the position.

The semi-implicit Euler scheme was selected as the integration method to be used in this project due to its stability and energy conservation properties, ensuring that energy remains constant over time without loss or gain. A time step of 0.003 is utilized, as larger time steps (0.004+) resulted in unstable behaviour within this simulation.

#### 3.2.4.3 XSPH Particle Velocity Correction

In addition to standard velocity calculations, an additional particle velocity correction technique called XSPH has been incorporated. This correction aims to adjust the velocities of particles that deviate significantly from an average value. Originally introduced by Monaghan in 1989[11], Priscott (2010) [13] implemented this technique using the equation:

The variable determines the extent of correction applied, typically kept relatively low to maintain the physical accuracy of the simulation; for instance, in this project, it's set to 0.08. According to Priscott, "The effect achieved is that of giving the particles a more orderly flow, especially when dealing with weakly compressible flow" [13]. This velocity correction occurs during each iteration of the simulation, between the steps of integrating the acceleration to compute a new velocity and integrating the velocity to determine a new position.

#### 3.2.4.4 Collision Detection

In this project, collision detection is handled simply, focusing only on collisions between the simulation bounds and individual particles. This detection occurs at the end of each simulation iteration, after integration is complete and particle positions have been updated.

The process begins by determining if each particle lies within the simulation bounds. This is achieved by assessing the particle's local position relative to the bounds, ensuring it falls within the range of 0 to 1 along each axis. Next, the particle's distance from the centre, adjusted by one particle radius, is checked along the x, y, and z axes to verify it remains within the bounds.

If a particle is found to be outside the simulation bounds, its position is adjusted to bring it back within the bounds. This correction ensures that any points outside the bounds are updated to lie within the boundary coordinates, with a slight offset to prevent particles from becoming stuck at the boundary edges.

Additionally, the velocity updates as the particle “bounces” away from the surface by having the velocity some dampening factor .

This dampening factor ensures that particles which fall from a height won’t indefinitely bounce of surfaces. It's akin to the coefficient of restitution in impulse calculations. By default , providing some slight dampening on collision.

## 3.3 Spatial Hashing

The primary optimization technique utilized in both CPU and GPU implementations of Smoothed Particle Hydrodynamics (SPH) is Spatial Hashing. While the core concept remains consistent across both implementations, their actual implementation methods differ significantly.

For the CPU implementation, Spatial Hashing leverages C# collections, specifically Dictionaries, and utilizes additional CPU memory. In contrast, the GPU implementation requires additional steps, notably GPU sorting, to achieve efficient Spatial Hashing.

This section will delve into the fundamental theory and concept of Spatial Hashing, highlighting its importance in optimizing SPH simulations. Subsequent sections (3.4.1.1 and 3.4.2.1) will explore the unique aspects of the CPU and GPU implementations, respectively, shedding light on their specific approaches and challenges.

The most crucial optimization in fluid simulation is the efficient determination of particle neighbours. Using a naïve brute force approach, the time complexity for finding neighbours would be [1], rendering the method impractical for high-fidelity fluid simulations. Spatial Hashing, introduced by Teschner et al. [7], presents a fast Nearest Neighbour search algorithm that preserves the Lagrangian method's flexibility, unlike earlier grid-based methods such as those used by Müller et al. [2] which constrained the fluid simulation to fixed bounds. It reduces the time complexity to where is the average number of neighbours found. Therefore the more uniform the distribution of particles the lower will the faster the simulation will run. In theory Spatial Hashing is bound by .

Conceptually, Spatial Hashing functions like a standard hash table, where scene positions are hashed to create keys that specify positions in the hash table. Positions close to each other should map to the same position in the hash table, resulting in the same bucket. Additionally, space is divided into a grid composed of cells of a certain width, with each hash value representing a specific cell in the grid. To find a particle's neighbours, the key for the cell the particle occupies and the keys for neighbouring cells can be utilized to look up particles in those buckets and identify the neighbours.

The initial step in Spatial Hashing involves determining the cell in which each particle resides. This is achieved using the following equation:

This discretizing function converts a vector with floating-point values into a vector with integer values based on the cell size . For simplicity, the cell size can be assigned as the smoothing length , as particles beyond the smoothing length do not affect force calculations and thus do not need to be neighbours [13].

Once particles are assigned to cells, a hash key is generated for each cell or bucket. The time complexity of the algorithm, as noted by Kelager [1], heavily depends on the hash function's ability to generate unique keys efficiently. The goal is to prevent particles from distant cells hashing to the same cell in the hash table, which would necessitate additional checks for neighbouring particles.

This project adopts the hash function proposed by Teschner et al. [7]:

Where is the size of the hash table and , and are large primes consistent with those used by Teschner [7]:

The performance of the Spatial Hashing algorithm is significantly influenced by the hash table size. Larger hash tables reduce the likelihood of mapping different particle positions to the same cell, resulting in faster operation. However, larger hash tables may incur slight performance decreases due to memory management. Additionally, hash functions perform most efficiently when the hash table size is a prime number [16].

To optimize performance, the function proposed by Kelager [1] is used to determine the table size:

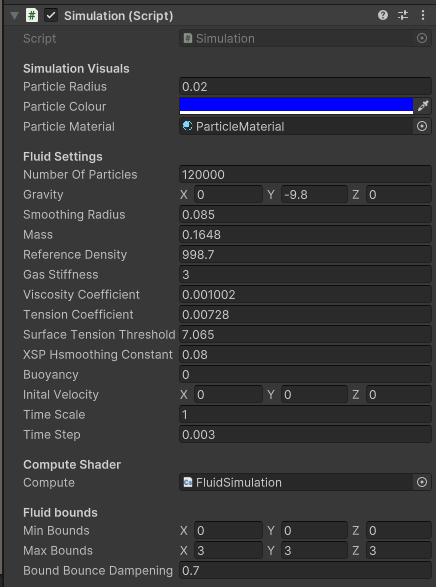
Where is a function which return the next prime and is the amount of particle in the SPH fluid simulation.

The cell retrieval, hash value generation, and table size determination functions described in this section are utilized for both CPU and GPU implementations, with slight variations in the Spatial Hashing process and neighbour search detailed in their respective sections (3.4.1.1 and 3.4.2.1).

## 3.4 CPU and GPU Implementations

This section delves into the comprehensive implementations of Smoothed Particle Hydrodynamics (SPH), which are divided into two subsections: one dedicated to the CPU implementation and the other to the GPU implementation. It explores various aspects such as the overall program structure of the implementations, including simulation loops executed once per frame, as well as the distinctive methods employed for Spatial Hashing. Additionally, it highlights any other noteworthy or unique aspects of each implementation.

Both implementations require direct execution within the Unity editor using a single game object with a script to facilitate variable changes. For instance, in Figure 3.1, highlighting the GPU implementation's C# script component "Simulation.cs," the Unity editor enables variable adjustments during runtime, serving as the preferred method due to the absence of an integrated interactive user interface.

Figure 3.1. Showcases the “Simulation.cs” script in the unity editor (the GPU implementation C# script)

### 3.4.1 CPU Implementation

In the CPU implementation, particles are represented as structs within an array. Each struct contains variables such as id, position, resultantForce, and velocity, along with a list of neighbouring particles. The CPU approach notably utilizes a list for neighbouring particle ids, optimizing the simulation's frames per second at the expense of higher memory usage. This trade-off enhances the feasibility of achieving real-time fluid simulation across a wider range of particle counts. This approach is exclusive to the CPU due to potential performance drawbacks on the GPU caused by scattered memory accesses.

The CPU implementation begins with a brief initialization phase. Here, the table size for Spatial Hashing is computed, and particles are initialized. with all fields set. The particle initialization process begins by uniformly distributing particles within a cubic volume defined by the simulation bounds. Random jitter is added to positions for slight irregularity. Each particle receives an initial velocity, an ID, and a zeroed resultant force. This method ensures a balanced distribution while introducing subtle randomness for natural fluid behaviour. Furthermore, certain preparations are made for rendering of the particles.

After this has occurred the simulation loop and implementation can be described using this pseudocode representing the “ParticleHandler.cs” script:

While simulation running:

Precompute smoothing kernel factors

For every particle :

Set up Spatial Hashing

Parallel for every particle :

NNS (Nearest Neighbour search) to update neighbour list

Calculate density

Calculate pressure

Parallel for every particle :

Calculate internal forces

Calculate external forces

Calculate sum of forces

Parallel for every particle :

Calculate acceleration

Update velocity

Correct velocity with XSPH

Update position

Check boundary conditions updating position and velocity

Render particles on the GPU in parallel

A limited degree of parallelization on the CPU is achieved through "Parallel.For()" in Unity, facilitating certain force calculations concurrently, similar to the multi-core CPU implementation of Wenbo et al.[22]. However, this falls short of the extensive parallelization and performance enhancement achieved on the GPU. The Spatial Hashing setup on the CPU is not parallelized due to the risk of race conditions arising during hash table updates.

The purpose of the precompute smoothing kernel factors function is to minimize the computational load during each simulation step. These factors, like , corresponding to the 6th polynomial kernel, are precomputed to streamline calculations. This optimization ensures that these factors are computed only once per simulation step instead of recalculating them for every kernel function invocation, while allowing for smoothing radius changes during the simulation.

#### 3.4.1.1 Spatial Hashing on the CPU

The CPU implementation of Spatial Hashing leverages C# collections such as Lists and Dictionaries to enhance performance. Dictionaries provide a straightforward method to construct hash tables, with the hash serving as the key and the Dictionary maintaining a list of particles corresponding to each hash value. Lists enable each particle to maintain a list of its neighbouring particles after the initial neighbourhood search, streamlining force and density calculations. This approach significantly improves efficiency compared to the GPU implementation, despite the latter benefiting from parallelization.

The setup of Spatial Hashing involves the following steps. A Dictionary, resembling a multimap as utilized by Priscott [13], is employed. This Dictionary consists of a uint key representing the hash and a list of particles. Each timestep, the Dictionary is cleared, and each particle in the simulation is placed into the dictionary as follows:

Once the setup is complete, the Near Neighbour Search (NNS) is conducted for each particle. Initially, the particle's current neighbour list is cleared, and the cell containing the particle is determined using the method outlined in section 3.4:

Subsequently, all cells that may contain neighbours are identified. Given that the chosen cell length on the grid corresponds to the smoothing radius, neighbours are defined as particles within a distance of the smoothing radius from the particle. As a result, neighbours can be at most one cell away. The simulation is in 3D so there are 27 cells which may contain neighbours ranging from to , with each cell having a different offset of -1, 0, or 1 in each axis relative to the particle's cell.

By hashing each of these values, a key for each cell in the Dictionary can be obtained and used to retrieve the list of particles in that cell:

For each potential neighbouring particle n each list, a comparison is made to determine if it is within range:

Here, the square distance and square of the smoothing radius are employed for computational efficiency, circumventing the need for expensive square root operations. Additional comparisons are made to ensure that the neighbour is not the particle itself and that the particle has not already been considered. If these conditions are met, particle adds the neighbouring particle to its list of neighbours; otherwise, the search proceeds to the next potential neighbour.

This results in the "FindNeighbours()" function depicted in Listing 3.1, which demonstrates how the Nearest Neighbour Search (NNS) is performed for each particle following the completion of the spatial hashing setup on the CPU.

Listing 3.1. “FindNeighbours()” function from “ParticleHandler.cs”



Once this process is complete, each particle is assigned its own list of neighbours, which it can utilize for density, force, and SPH particle velocity correction calculations without requiring another full Near Neighbour search until the next timestep.

### 3.4.2 GPU Implementation

The GPU implementation predominantly relies on a compute shader written in HLSL (“FluidSimulation.compute”) , integrated with Unity to facilitate data transfer from the CPU using a C# script component “Simulation.cs”.

In contrast to the CPU implementation, particles in the GPU version are not represented as a buffer of structs. Instead, they are stored as multiple distinct buffers, each containing the variables present in the CPU implementation's structs (such as position and velocity). This design choice optimizes memory access patterns, aligning with the structure of array (SoA) data format. As according to Cai et al. [19], SoA formats facilitate better-coalesced memory access, unlike array of structure (AoS) formats, which may lead to scattered memory access in GPU implementations.

In the setup phase of the GPU implementation, similar procedures to the CPU version are followed, albeit with additional steps. These include transferring variables like particle mass from the CPU to the GPU. Moreover, due to the Unity framework, proper initialization of Compute Buffers is required, and the "SetBuffer()" function is employed to link the buffers to kernels in the compute shader. Another important step is to calculate the number of sorting steps for the GPU sorting algorithm, a process detailed in subsection 3.4.2.2. Moreover, an additional "factorsBuffer" is established, housing precomputed smoothing kernel factors for GPU access. Hence, these values must also be computed during initialization.

Similarly to the GPU implementation of SPH by Crespo et al. [17], this project leverages the parallel processing power of the GPU, utilizing one thread of execution to compute the resulting force of one particle as it performs all interactions with its neighbours.

A condensed representation of the parallel GPU functions is showcased by the pressure calculation kernel. By following the “CalculatePressures()” Function from Listing 3.2, the overarching concept of how the GPU computes forces using one thread is illustrated. This snippet showcases how each thread, represented by "id", corresponds to a single particle, enabling parallel computation of forces, much like how pressure is computed here. Throughout program execution, the "Dispatch()" function is invoked on the CPU to execute each of these kernels, facilitating parallel calculations on the GPU for every particle in the simulation. Reference density in this context is the same as rest density.

Listing 3.2. CalculatePressures() function in “FluidSimulation.compute”



The simulation loop and implementation can be described using this pseudocode representing the “Simulation.cs” script:

While simulation running:

Pass any updates to settings (e.g. smoothing radius) to GPU

Precompute smoothing kernel factors and update the factorsBuffer

// Spatial Hashing

Update spatial indices on the GPU in parallel

For i < numStages: // GPU Sort Loop

For j < i:

Update Group Width and height and pass to GPU

Parallel bitonic sort dispatch call to GPU

Update particle indices on the GPU in parallel

// Force calculations and application of forces

Calculate particle densities on GPU in parallel

Calculate particle pressures on GPU in parallel

Calculate particle internal forces on GPU in parallel

Calculate particle external forces on GPU in parallel

Calculate velocities, XSPH, and new positions on GPU in parallel

Resolve any boundary collisions in parallel on the GPU

Render particles on the GPU in parallel

Here, all GPU interactions, from updating spatial indices to resolving boundary collisions, are executed through dispatch calls.

Exiting the simulation necessitates clearing any allocated memory due to the use of compute buffers. This task is accomplished simply, as depicted in Listing 3.3.

Listing 3.3. Code from “Simulation.cs” script to clear allocated memory from compute buffers.



#### 3.4.2.1 Spatial Hashing on the GPU

In the GPU implementation of Spatial Hashing, the absence of data structures like Dictionaries and Lists necessitates a different approach. Instead, compute shaders utilize two buffers: "spatialIndices" and "particleIndices" to organize particle data and efficiently access neighbouring particles.

The "spatialIndices" buffer stores pairs of uint2 values, where the x value represents the hash of the particle's position, and the y value indicates the index in the other buffers (position, velocity, resultant forces, densities, and pressures). This buffer takes the same role as the "particleHash" array described by Simon Green[12], with one entry per particle.

On the other hand, the "particleIndices" buffer serves a similar role to the hash table and is sized accordingly. It facilitates efficient storage and retrieval of particle data based on their hash values.

Spatial hashing on the GPU involves three main steps to set up each simulation timestep. First, the "spatialIndices" buffer is updated.

For each particle the buffer is filled as follows:

Here, represents the index particle , aligned with its position in the velocity and other buffers. The buffer comprises uint2 pairs containing the hash and index, where the index is utilized to locate the particle in the other buffers.

Furthermore, the "particleIndices" buffer is configured to be updated, with the buffer being initialized as follows:

Where represents the table size, ensuring that every particle index is lower than the current value at each hash. This setup ensures that the starting index will be correctly replaced later in the process.

Subsequently, the next step involves sorting the entries in the "spatialIndices" buffer based on their hash values. For optimization purposes, it's advantageous to perform this sorting operation on the GPU. Simon Green[12] uses a fast radix sort provided by the CUDPP library, to perform this sort however, since this project utilizes compute shaders written in HLSL within Unity3D, a GPU sorting algorithm needs to be implemented internally. Due to the time constraints of the project, a Bitonic Sort algorithm [18] has been chosen for its simplicity and suitability to implement. More details on this sorting algorithm will be discussed in the next subsection (3.4.2.2).

The final step in preparing for the Nearest Neighbour search involves updating the "particleIndices" buffer. For each entry in the "spatialIndices" buffer, the hash is identified, and the index of the earliest occurrence of that hash value in the buffer is determined and saved as follows:

This operation is executed using the HLSL InterlockedMin() function in the compute shader to ensure thread safety and accurate storage of the minimum value. With this setup complete, the Near Neighbour search can now be conducted.

The Near Neighbour search algorithm follows a similar structure to the CPU version but with some notable differences. It begins by determining the cell of particle and then proceeds to iterate over the hashes of the 27 cells that may potentially contain neighbours, with offsets ranging from (-1, -1, -1) to (+1, +1, +1). The differences begin during the lookup of particles in each cell.

First the algorithm uses the hash as a key to do a look up to find the first index in the “spatialIndices” buffer containing that hashed value as such:

While the value of the start index is less than the number of particles, the neighbourhood search continues as follows:

Retrieve the neighbour index:

Verify if the hash matches:

If false, all particles in the current cell have already been checked, and the loop exits. If true, Increment the start index by one, then check if the neighbour index and skip if so, skip to the next iteration. Finally, much like the CPU implementation compare the square distance to the square of the smoothing radius to determine if the neighbour is within range. This operation is significantly less computationally expensive than using a square root operation. From this point, density, force, or XSPH calculations can be performed by using the neighbour index to retrieve values from the position, velocity, pressure, and density buffers. The loop continues until an exit condition is met. This can be seen in Listing 3.4, when calculating the density for each particle.

Listing 3.4. “CalculateDensities()” from “FluidSimulation.compute” showing a GPU Nearest Neighbour Search (NNS)



Unlike the CPU implementation, the full neighbourhood search is conducted for each of the functions to calculate density, internal forces (plus surface tension), and conduct XSPH particle velocity correction.

In Listing 3.4, the utilization of "kernelFactors," which is the GPU side equivalent of the "factorsBuffer" discussed earlier in this section (3.4.2), is seen on line 111 helping to optimize the calculations of the SPH approximations.

#### 3.4.2.2 GPU Sort

For the Spatial Hashing on the GPU, a Bitonic sorting algorithm [18] is utilized. This choice was made due to its relatively straightforward implementation and parallelized nature, aligning well with the project's timeline.

The implementation comprises two main components: a "GPUBitonicSort()" function on the CPU responsible for calculating parameters for each sorting stage and dispatching the sorting kernel to the GPU, and the sorting kernel itself executed on the GPU.

The number of sorting stages is determined by the formula:

Where is the number of particles in the simulation and “nextPowerOf2()” is a function which finds the power of two after a given value.

Listing 3.5 showcases how the GPU sort is conducted on the CPU side. In this function, the "groupWidth" variable determines the width of each sorting group in the bitonic sort algorithm. As i decreases and j increases, the groupWidth decreases, resulting in the sorting of smaller groups. The "groupHeight" parameter defines the number of elements to be sorted in each group, determining the number of iterations required to sort the elements within each group. The parameters are then passed to the GPU where the sorting kernel on the GPU is then dispatched to complete a sorting stage.

Listing 3.5. “GPUBitonicSort()” function from “Simulation.cs”



Listing 3.6 showcases the GPU side for the sorting. The GPU part of the process executes the sorting operations on the GPU, operating on individual elements of the data in parallel. Each iteration of the GPU sort calculates the indices of the elements to be compared and swapped based on its thread ID and the group width and height parameters received from the CPU. If the value of the cell hash on the left is greater than the value on the right, a swap is performed. This process is repeated until the list is eventually sorted.

Listing 3.6. “BitonicSort” function from “FluidSimulation.compute”



## 3.5 Rendering

Given that the main focus of this dissertation lies elsewhere, rendering will be discussed only briefly in this section. The rendering approach adopted here emphasizes simplicity and efficiency.

A custom shader has been developed specifically for rendering particles, leveraging GPU instanced meshes for efficient rendering. To ensure consistency and facilitate performance comparison between the CPU and GPU implementations, rendering is carried out uniformly across both.

Rendering occurs on the GPU utilizing Unity Engine's support for indirect draw calls. Compute buffers are employed to transmit position data to the shader on a per-frame basis. In the CPU implementation, an additional step is required to update the position data in the buffer each frame. Conversely, in the GPU implementation, the data is updated within the compute shader, eliminating the need for per-frame updates.

For rendering, the Unity function "Graphics.DrawMeshInstancedIndirect()" is utilized, offering efficient instanced rendering capabilities. Meshes are dynamically generated during initialization and are appropriately disposed of upon program termination. To optimize performance, meshes are designed with minimal triangles and vertices.

# 4 Results and Evaluation

This section begins by reviewing the project's outputs, encompassing the testing environment, methodologies, and default testing values. It proceeds to compare the results, including framerates, between the CPU and GPU implementations. Lastly, the GPU implementation's performance at high particle counts is showcased, highlighting the GPU's capability in processing SPH efficiently.

The outputs yielded by this project are two implementations of Smoothed Particle Hydrodynamics (SPH) for both CPU and GPU, prioritizing real-time fluid dynamics simulation. It entails an analysis of these implementations, primarily concerning their ability to achieve real-time frame rates.

## 4.1 Testing Environment and Methodology

The collection of results was conducted on a PC equipped with a 64-bit edition of Windows 10. The system features an Intel Core i7-6820HK 2.70GHz processor, Nvidia GTX 1070 graphics card, and 32GB RAM. The performance tests maintained consistent variable values, such as mass and reference density, which were based on water properties. The only variable adjusted was the particle count, which was the focus of the test.

The fluid analysis section examines how fluid behaviours change when fluid attributes are altered. The approach taken to evaluation of this is based on visual convictions as stated by Kelager[1] “We are computer scientists, thus we are in general pleased if the results are visually convincing, while keeping in mind that the simulations already use as physically correct fluid parameters as possible”.

The primary quantitative focus of the results centres on comparing the performance of CPU and GPU implementations of SPH and assessing whether real-time fluid simulation has been achieved. The methodology for measurement is as follows: For each sampled particle count, 10 simulations are conducted to determine the average frames per second and memory usage. Additionally, between the first two points labelled on the axis, results from 7 evenly distributed particle counts are collected.

The primary quantitative focus of the results centres on comparing the performance of CPU and GPU implementations of SPH and analysing scalability while assessing whether real-time fluid simulation has been achieved. The methodology for measurement is as follows: For each sampled particle count, 10 simulations are conducted to determine the average frames per second and memory usage. Additionally, between the first two points on the axis, results from 7 evenly distributed particle counts are collected, with 3 points between the second and third and 1 point between the third and fourth. This spacing accounts for the time taken for each reading. Since later simulations may run at 30 fps for 10 in-simulation seconds, resulting in about 2 minutes of real-world time per reading with a timestep of 0.003, only values at points on the axis are taken 10 times each to avoid overly lengthy testing durations and completion of results collection within the project’s timeframe. Therefore, the range of results collected and overall evaluation are somewhat limited by this factor.

A lightweight C# script is utilized to calculate both the average frames per second (FPS) and the average memory usage during runtime. This calculation involves dividing the total number of frames by the total time running (in seconds). The script continues to count the total frames and total time until the simulation time, determined by the time step the simulation runs at, exceeds a specific value of 10 in-simulation seconds. This process ensures that the simulations run for the same amount of in-simulation time and are at the same point in simulation time, with enough time for the fluid to settle. This approach is favoured over using real-world time because various frame rates may correspond to different points within the simulation itself.

To maintain result consistency across simulations, it's essential to minimize external factors that could influence performance metrics. By using a fixed simulation time as the basis for measuring FPS and monitoring memory usage, variations caused by system load fluctuations or runtime conditions are minimized. This approach ensures that performance evaluations are conducted under standardized conditions, enhancing the reliability and comparability of the obtained results. Consistency in measurement methodologies facilitates obtaining meaningful insights and drawing accurate conclusions from the simulations.

To maintain result consistency, the main Unity camera is fixed in position (ensuring it remains in the same place for both CPU and GPU tests), and only the game view is open while the scene view remains closed. This precaution ensures that rendering differences do not significantly affect the results and aligns the simulation performance with real-world application scenarios.

## 4.2 Testing Variable Setup

When testing and analysing different fluids, certain parameters are adjusted to represent each individual fluid. Additionally, the particle masses used in these simulations are higher than typical SPH simulations to better enforce incompressibility in the fluid, resulting in lower, more desirable neighbour counts and improving frame rates.

Some parameters remain consistent across all simulations:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Symbol | Value | Units |
| Timestep |  | 0.003 |  |
| Particle Count (For Fluid analysis section) |  | 100,000 |  |
| Gravity |  | (0, -9.8, 0) |  |
| XSPH Smoothing Constant |  | 0.08 | - |
| Smoothing Radius |  | 0.085 |  |
| Collision Dampening Factor |  | 0.7 | - |

Table 4.1. Shared Testing Parameters

One of the most common fluids used in SPH simulations is water. Water is liquid, which has a low viscosity, and therefore it is challenging to keep a realistic water simulation stable.

For simulating a water-like fluid the following parameters are used:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Symbol | Value | Units |
| Reference (Rest) Density |  | 998.7 |  |
| Viscosity |  | 0.001002 |  |
| Gas Stiffness |  | 3 |  |
| Tension Coefficient |  | 0.00728 | - |
| Surface Tension Threshold |  | 7.065 |  |
| Buoyancy Diffusion Coefficient |  | 0 | - |
| Particle Mass |  | 0.1648 |  |

Table 4.2. Water Parameters

Furthermore, for variety it is essential to compare the behaviour with a similar more viscous fluid having a higher surface tension than water. For this parameters similar to a vegetable oil are used:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Symbol | Value | Units |
| Reference (Rest) Density |  | 920 |  |
| Viscosity |  | 0.08 |  |
| Gas Stiffness |  | 3 |  |
| Tension Coefficient |  | 0.022 | - |
| Surface Tension Threshold |  | 7.065 |  |
| Buoyancy Diffusion Coefficient |  | 0 | - |
| Particle Mass |  | 0.1648 |  |

Table 4.3. Vegetable Oil Parameters

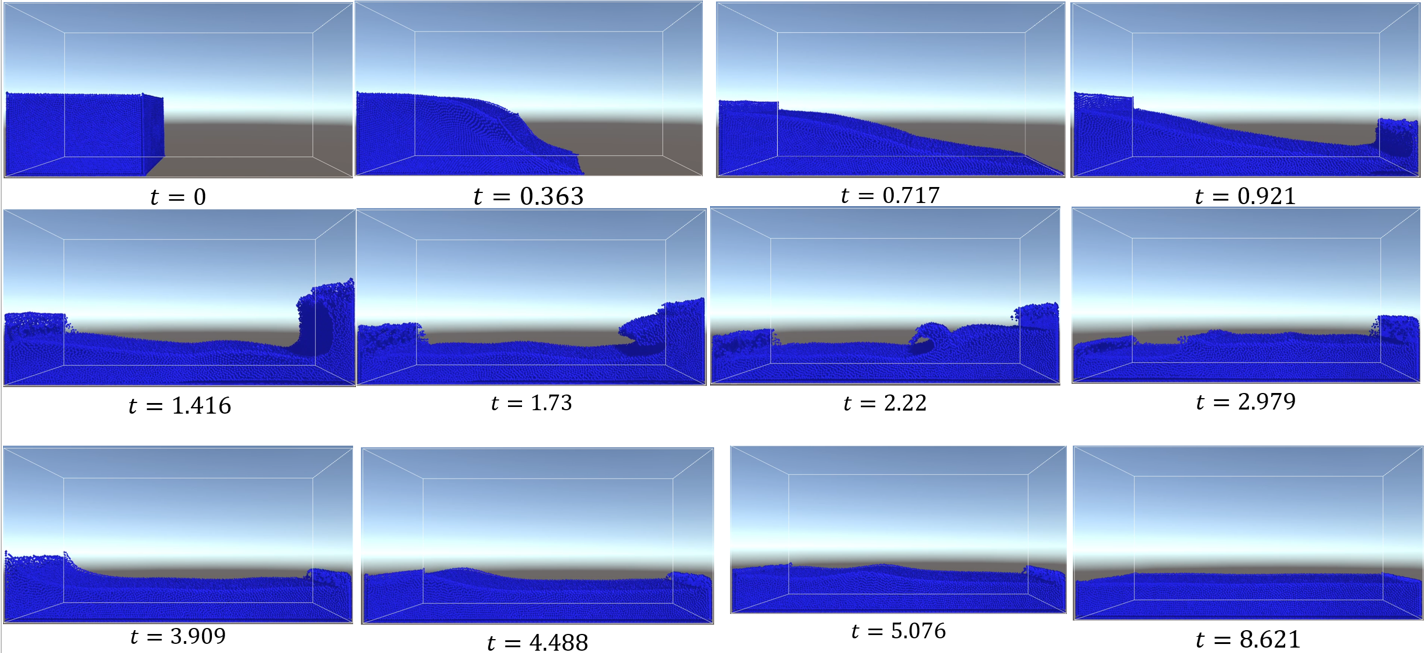
While these parameters, such as reference density, aren't precisely accurate to real-world fluids, they are somewhat close to actual fluid properties. The aim is to observe behaviour rather than achieve 100% accuracy. As mentioned earlier, this is a computer science project, not a physics project. Therefore, for testing, 100% real-world scientific accuracy is not required for the general analysis of the "physical" behaviours exhibited by the simulated fluids. Instead, the goal is to demonstrate that changes in these variables enable the simulation of different fluids, and that each fluid property remains intact as the forces are calculated correctly.

## 4.3 Fluid Analysis

This section is dedicated to analysing the "physical" behaviour and characteristics of various SPH fluids, considering fluids with different properties such as viscosity. All of these simulations have been conducted on the GPU to achieve higher particle counts, enabling more detailed and realistic fluid interactions.

### 4.3.1 Fluid Flow

Our first objective is to determine whether the SPH simulation can generate realistic fluid motion. Therefore, to investigate this, a time-lapse of the fluid flow using the parameters of water (Table 4.2) is presented in Figure 4.1. To demonstrate this, the results for the dam-break problem have been produced. In the dam break problem, the fluid is initially constrained inside a dam. When the fluid is at rest, the dam or barricade that constrains the fluid is removed. Consequently, the fluid flows freely and collides with a vertical wall. In the simulation, this was achieved by waiting for the fluid to reach rest within some fixed bounds and then expanding the simulation bounds in a singular dimension.

Figure 4.1 Dam-break of 100,000 particles of water with timestamps.

Certain moments in time are illustrated from a dam-break of water, simulated by 100,000 particles on the GPU at approximately 50 frames per second, as depicted in Figure 4.1. It's worth noting that if the in-simulation time is not divisible by the time step , this occurs due to the accumulation of floating-point arithmetic errors. Therefore, any evaluations regarding the time may be slightly inaccurate. However, as the timespan of this is rather small, the amount of error in the timestep accumulated should not be too significant, with only a small decimal margin of error.

At , the dam is broken, and the water is sliding down, accelerating due to pressure and gravitational forces. The water continues to flow until it hits a wall at . This results in the water particles being pushed upwards at due to the pressure from the high-speed water. At the water begins to overturn and a wave begins to be formed at due to the water from above falling because of gravity.

At , we witness realistic behaviour as a complex wave emerges through the phenomenon of wave breaking, a well-documented occurrence in fluids studied in prior research [32]. This is the result of two waves meeting: the left wave near the centre at and the wave caused by the water crashing onto the water surface on the right side due to gravity. In the simulation, the wave on the right rises as a result of wave interference. The amplitudes of the two waves form a steeper wave, which then crashes down and creates a wave on the water surface at , forming a smaller wave on the water surface. The exhibited realistic behaviour hints at the simulation's potential applicability across various domains, including computer graphics.

After the wave has propagated against the water surface it collides with the opposite wall at . From here, the water falls, and another wave, which propagates across the water, is formed, as shown at and This process repeats until the fluid eventually comes to rest at .

In general, these results demonstrate realistic fluid behaviours, modelling complex fluid phenomena. However, there is one caveat: a visible issue in the fluid behaviour. This is most easily noticeable at at the left boundary and at the right boundary, where particles seem to slow down near the boundaries and take a long time to move away from them, and they do not seem to stack on top of each other, slowly trickling down away from the boundary. This issue is likely caused by the handling of collisions between particles and the simulation bounds, but a potential fix may also be to increase the XSPH smoothing coefficient for particle velocity correction.

### 4.3.2 Comparison of Different Fluids

To compare the differences in behaviour between different simulated fluids, the parameters for a water (Table 4.2) and the fluid resembling a vegetable oil (Table 4.3) are used. Much like observing flow, a dam break has also been conducted to compare fluid behaviour.

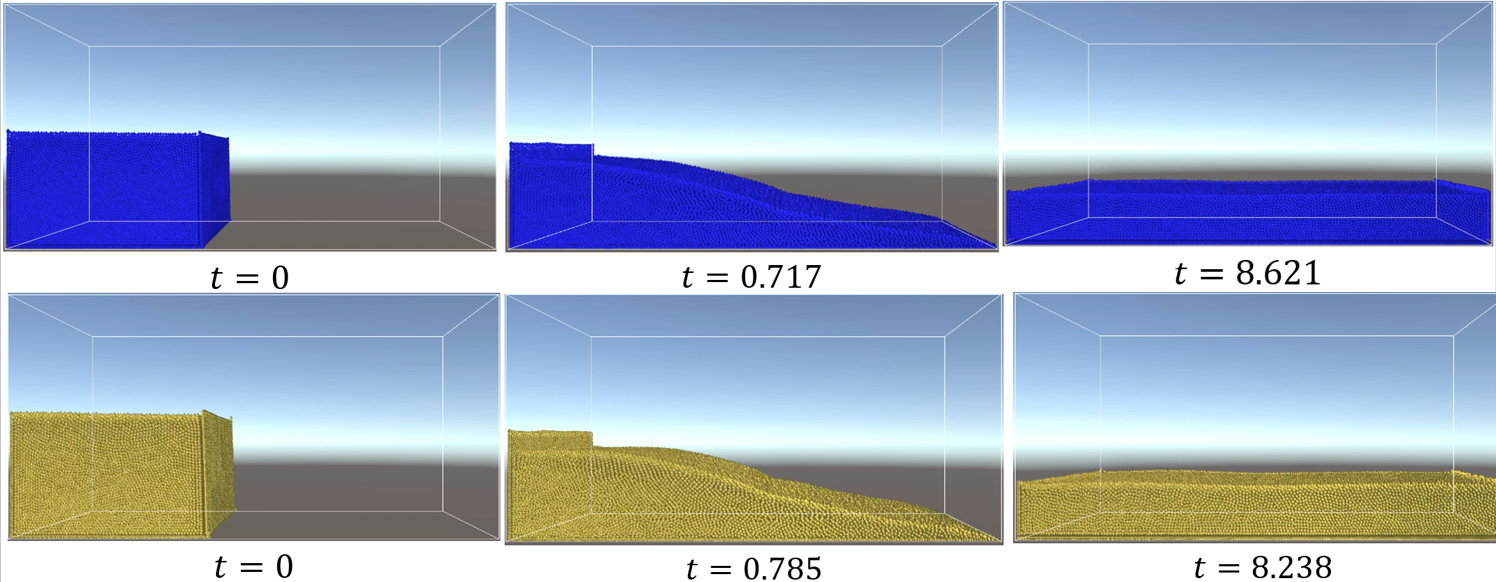


Figure 4.2 Dam-break of 100,000 particles one using the water parameters (blue) and one the vegetable oil parameters (yellow).

The first observation we can make is about the total volumes of each fluid at , where the yellow fluid is slightly taller, taking up more volume than water. Since the particle masses of both fluids are the same and the rest density of the vegetable oil fluid is lower than the rest density of water, using the volume calculation , the volume will be greater the lower the density of the fluid is. Therefore behaviour is as expected, and accurate to real fluids.

Figure 4.2 illustrates the in-simulation times taken for the fluid from rest to reach the boundary on the right and the fluid to reach rest. As expected, the vegetable oil, with its higher viscosity, takes 0.068 in-simulation seconds longer to reach the first boundary as it experiences more resistance. This behaviours is desirable as it suggests that the viscosity is acting well within the simulation. Additionally, the vegetable oil comes to rest 0.383 in-simulation seconds earlier than the water due to more significant loss of velocity, which aligns with the expected behaviour given the higher viscosity.

In terms of the precise times taken, it's important to keep in mind that other factors, such as rest density, also affect the speed of flow, with smaller rest densities generally leading to faster flows. Therefore, the effect from viscosity may be slightly greater than showcased here, and the exact evaluation of how viscosity affects flow is not precise in this case.

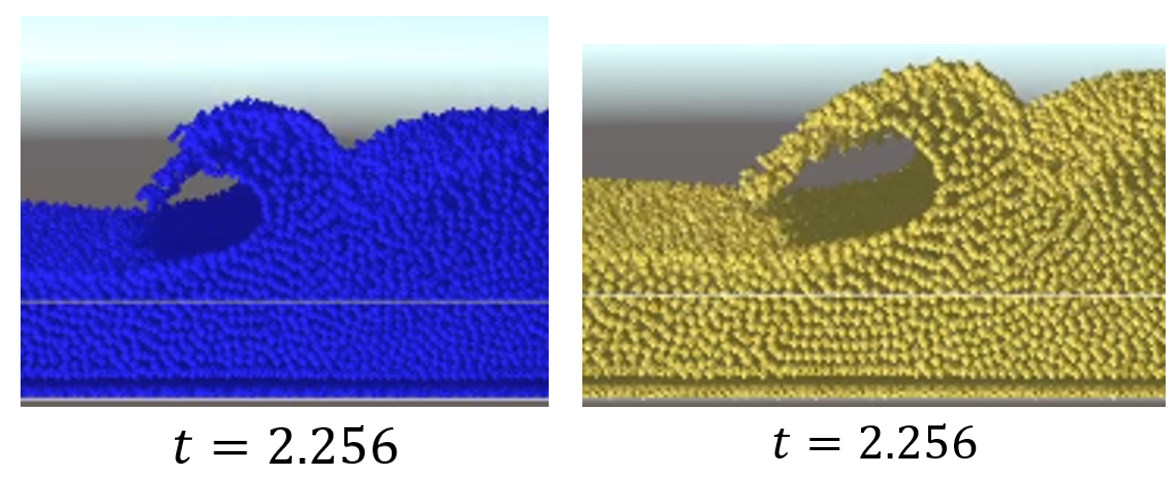


Figure 4.3. Wave breaking occurring at the same in-simulation time for each fluid.

In Figure 4.3, we can observe the effect of surface tension, particularly where wave breaking occurs in the fluid. Increased surface tension in the simulation of the vegetable oil results in a more regular wave shape, with all particles firmly attached to the surface of the wave, resulting in a more cohesive appearance. In contrast, the simulation of the water shows a less smooth shape, with particles tending to be more fragmented, resulting in a less uniform appearance, and some water particles being more detached from the overall wave.

Given this information, the simulation's demonstration of expected fluid behaviours suggests potential utility across specific applications, particularly in fields like computer graphics and video games. However, due to some inaccuracies within the simulation necessitating unusual masses, its usefulness in domains like medicine[21] and engineering[20], which demand higher precision and accuracy, may be limited.

## 4.4 Performance of CPU and GPU Implementations

To compare the results of SPH on the CPU and GPU 160 different readings were taken each for a total of 320 readings, 10 at each point on the graphs, of both the average frame rate (fps) and the average memory used in MB. Results were taken up until a particle count of 4,000 where testing times became too lengthy.

|  |  |  |
| --- | --- | --- |
| Particle Count | CPU FPS | GPU FPS |
| 100 | 300.54 | 467.99 |
| 200 | 250.67 | 467.72 |
| 300 | 215.19 | 466.72 |
| 400 | 186.77 | 466.01 |
| 500 | 165.57 | 465.12 |
| 600 | 146.99 | 466.58 |
| 700 | 128.56 | 463.60 |
| 800 | 111.37 | 465.73 |
| 1000 | 85.26 | 466.19 |
| 1200 | 69.91 | 460.79 |
| 1400 | 59.13 | 464.86 |
| 1600 | 49.25 | 474.23 |
| 2000 | 38.52 | 468.60 |
| 2400 | 31.51 | 464.10 |
| 3200 | 22.71 | 463.88 |
| 4000 | 17.33 | 461.71 |

Table 4.4

Figure 4.4. Graph showcasing how the number of particles in the simulation affects the framerate of CPU and GPU SPH implementations.

Figure 4.5. Same data as Figure 4.4 with a logarithmic y-axis.

The first observation to make is the overall differences in performance of the two implementations. The GPU implementation eclipses the performance of the CPU implementation managing to keep a consistent performance of 460+ frames per second. The flatness of the GPU logarithmic line (Figure 4.5), despite the increase in particle count, indicates that the computational workload remains relatively constant over time. This suggests that the GPU efficiently handles the increasing particle count without a significant drop in frames per second (fps), in the range of thousands of particles.

In contrast, the CPU implementation demonstrates a significant decrease in frames per second as the number of particles in the simulation increases, particularly in the range of hundreds. From the first graph in Figure 4.4, we observe that the fps count on the CPU decreases sharply and then gradually decreases at a slower rate, reaching below 60 fps at approximately 1,400 particles, which is still within the range for a smooth simulation and usability for real-time applications. It reaches below 30 fps somewhere between 2,400 particles and 3,000 particles, impacting overall usability. Additionally, with a timestep for truly real-time fluid dynamics, an FPS count of 333+ would have to be achieved. However, this seemingly is not possible on the CPU using the testing hardware for any particle count particles.

Figure 4.6. Graph showcasing how the number of particles in the simulation affects the memory use in MB of CPU and GPU SPH implementations.

Moreover, it's useful to compare the average memory used for each implementation due to how Spatial Hashing is performed. Figure 4.6 illustrates the average memory used at each particle count. The results here are somewhat inconsistent, caused by the Unity garbage collector and some Unity editor overhead. Generally, all the results for one point were taken at once, so they would share the same error from overhead, slightly limiting this evaluation in terms of identifying any patterns for the implementations, so some assumptions are made. Results may be wrong by up to about 2 MB due to these fluctuations.

From these results, we can see that the overall memory use of the GPU implementation remains generally static at approximately 21 MB. In contrast, the CPU implementation shows an increase in particle memory usage. This can be attributed to each particle holding a list of neighbours. The graph shows a somewhat linear increase in memory usage, indicating that the Spatial Hashing algorithm is resulting in the expected increase, where the number of neighbours is consistent across all particles in the simulation. Also, the initial non-linear start can be attributed to the fixed bounds size where the fluid does not yet fill the area of the floor of the bounds having particles with less average neighbours due to no neighbours from above/below each particle. This behaviour is also discernible in Figure 4.7, where computational time in milliseconds follows a linear trend with a non-linear start.

Figure 4.7. Graph showcasing how the number of particles in the simulation affects the computation time in ms of CPU and GPU SPH implementations.

|  |  |  |
| --- | --- | --- |
| Particle Count | Computation Time CPU (ms) | Computation Time Ratio To Last 800 particles |
| 1600 | 20.31 | 1.1310 |
| 2400 | 31.73 | 1.0415 |
| 3200 | 44.04 | 1.0410 |
| 4000 | 57.69 | 1.0480 |

Table 4.5

Ratio calculated follows, where is the computation time and is the particle count:

From Figure 4.7, the relationship between computation time and particle count appears linear after approximately 1600 particles. Table 4.5 supports this observation, as the ratios after 1600 particles remain approximately constant (around 1.04). This suggests that the simulation has the potential for good scalability; however, it is still limited by the processing power afforded by the CPU's lack of parallel processing. However, this assumption may not accurately reflect the true behaviour of the simulation. As the computation time increases to larger values, surpassing 100 ms, it becomes considerably more challenging to efficiently collect results within a reasonable timeframe.

## 4.5 Performance of High Particle Counts on the GPU

To evaluate the performance of the SPH simulation 210 different readings were taken, 10 at each point on the graphs, of the average framerate (fps). Each of these tests were conducted using fixed bound sizes.

|  |  |
| --- | --- |
| Particle Count | FPS |
| 2500 | 464.78 |
| 5000 | 464.92 |
| 7500 | 462.59 |
| 10000 | 456.44 |
| 12500 | 452.78 |
| 15000 | 441.08 |
| 17500 | 397.87 |
| 20000 | 357.15 |
| 25000 | 257.77 |
| 30000 | 207.35 |
| 35000 | 179.85 |
| 40000 | 156.61 |
| 50000 | 122.31 |
| 60000 | 106.14 |
| 80000 | 71.033 |
| 100000 | 51.35 |
| 120000 | 47.80 |
| 140000 | 40.81 |
| 160000 | 35.43 |
| 180000 | 32.20 |
| 200000 | 30.29 |

Table 4.6

Figure 4.8. Graph showcasing how the number of particles in the simulation affects the framerate of GPU SPH implementation.

From Figure 4.8, it can be observed that the simulation takes on a similar shape to the CPU on Figure 4.4. However, one noticeable difference is the softer gradient observed at the lower particle counts. This initial soft gradient suggests that, during the early stages of particle count increase, the computational workload remains relatively low. Consequently, the system resources may not be fully utilized, leading to stable FPS as the demand placed on the system by the simulation at these early particle counts is not substantial to cause any significant impact on FPS.

The simulation on the GPU seems to achieve truly real-time simulation for particle counts slightly larger than 20,000 and below (real-time accurate to real-world being higher than 333 fps with a timestep of 0.003, where 1 second in-simulation = 1 second real-world). This indicates that the main way the simulation can be improved for real-time fluid simulation is by increasing the timestep. By achieving a larger timestep, for example, 0.01, the frame rate can be reduced greatly to only needing 100 fps for real-time, where on this hardware, it would result in 60,000 particles being supported for truly real-time fluid simulation. However, this introduces more instability into the system and reduces the accuracy of the simulation so a workaround these issues would have to be found.

Another key observation is that the testing hardware supports 60 fps for simulations involving approximately 80,000 to 100,000 particles, making it suitable for real-time applications. Additionally, simulations with 200,000 particles or fewer can achieve 30 fps. Generally, with this SPH implementation, it's reasonable to expect that modern hardware can handle several hundred thousand particles at acceptable frame rates.

This performance seems to be comparable to other similar implementations of SPH utilizing the GPU. For instance, Gunadi et al. [34] presented an SPH implementation in 2018 with similarities in terms of the forces employed for SPH simulation. However, their implementation utilizes Vulkan and OpenGL compute shaders and was tested on hardware featuring a GTX 1070, offering a valuable comparison due to similar hardware. While their implementations initially exhibit higher frame rates—486.80 fps for Vulkan and 602.90 fps for OpenGL—compared to my HLSL implementation's 464.90 fps, my implementation tends to surpass theirs at higher particle counts. For instance, at 10,000 particles, my implementation achieves a frame rate of 456.44 fps, while theirs yields 271.80 fps (Vulkan) and 302.65 fps (OpenGL). This trend continues, as evidenced by the performance at 60,000 particles: my implementation achieves a frame rate of 106.14 fps, whereas theirs achieves 28.40 fps (Vulkan) and 15.25 fps (OpenGL). These results suggest that the implementation holds promise for real-time applications like games, where its performance is comparable to other SPH implementations. The evaluation of the comparison between my implementation and theirs may be somewhat limited, as their implementation is not as highly optimized as it could be especially in terms of the Nearest Neighbor search (NNS).

Figure 4.9. Graph showcasing how the number of particles in the simulation affects the computation time in ms of GPU SPH implementation.

The computation time graph for the GPU Figure 4.9, in contrast to the CPU graph observed earlier (Figure 4.7), appears notably more irregular. This discrepancy could be attributed to several factors, including the scale being halved on the y axis with a larger range of values on the x axis, which might exaggerate variations, or other underlying issues. Potential contributors to this irregularity include instabilities in the simulation, particularly as higher particle counts increase the risk of unstable behaviour, causing particles to behave unnaturally. Additionally, errors in measurement or increased rendering costs could also influence the observed irregularities.

In general, Figure 4.9 still bears resemblance to the CPU graph depicted in Figure 4.7, suggesting a similar type of scalability with linear increments in computation time as the particle count rises. Furthermore, towards the end of the dataset, there seems to be a diminishing gradient, hinting at potentially improved performance. However, the range of values available is insufficient to draw a definitive conclusion on this matter.

In real-time applications like video games, where maintaining a frame rate of 60+ fps (equivalent to a computation time of 16 ms or lower) is crucial for smooth gameplay, the SPH simulation is usable within the range of tens of thousands of particles. The GPU shoulders the bulk of the computational workload, operating at high capacity, while CPU usage remains relatively low. Thus, if the combined computation time of the GPU for simulating a given particle count and the original computation time is lower than the CPU processing time for other tasks, integrating the simulation within this range into the application is feasible without significant performance drawbacks. For example, suppose an application initially requires 1 ms of GPU computation time solely for rendering, and the original CPU processing time is 15 ms. In this scenario, adding an 80,000 particle simulation (with a computation time of 14.08 ms) would incur minimal performance impact.

## 4.6 Known Issues

As mentioned in section 4.2, higher masses than those typically used in SPH were utilized for testing to better enforce incompressibility. Incompressibility is important for many reasons, improving both the performance and accuracy of the simulation. Performance is enhanced as the overall stability of the simulation is improved, leading to smoother, more predictable behaviour where each particle will have the correct number of neighbours, thus enhancing the speed of neighbourhood searches. The simulation becomes more accurate as real-world fluids are incompressible, and incompressibility helps to preserve conservation laws of mass, momentum, and energy in the simulation. In addition, incompressibility leads to a more accurate representation of surface tension effects as a constant density is maintained, making the surface tension more accurate in reproducing real-world phenomena. Therefore, ideally other methods to improve incompressibility could be implemented.

The way particles interact with boundaries is not ideal, and leads to some inaccurate and unstable behaviour. Firstly, as explained towards the end of section 4.3.1, particles at the boundary seem to get somewhat stuck to the boundaries after something like a wave causes them to hit the wall, which is further explained in the relevant section. Secondly, if the simulation bounds are to be made smaller during simulation runtime, the compressed particles lead to high particle densities, causing massive forces to be exerted on each other, resulting in unstable behaviour where particles have massive velocities. Moreover, if particles experience large velocities from instabilities, XSPH particle velocity correction may also lead to other particles gaining large velocities from one unstable particle. This may cause a cascading effect which propagates throughout the entire system, causing more instability compromising the overall integrity of the simulation.

# 5 Conclusions

This conclusion assesses the extent to which each objective was achieved, summarizes the key learnings from the project, and discusses potential avenues for future research and development.

## 5.1 Objectives

**1. Identify key features crucial for accurate fluid simulation using SPH, while emphasizing simplicity to optimize performance.**

The essential components for precise fluid simulation with Smoothed Particle Hydrodynamics (SPH) identified in this project encompass pressure forces, viscosity, surface tension, gravity, buoyancy, and particle velocity correction through XSPH. While these aspects form the cornerstone of accurate SPH simulations, more intricate fluid characteristics like solidification and melting [23] have been excluded due to their potential performance implications.

**2. Research existing SPH based fluid dynamics simulation techniques to enhance performance and determine the most effective approaches to employ within the timeframe of the project.**

During this project, the focus was primarily on two main simulation techniques: Spatial Hashing[7] and parallelization. These methods were selected for their effectiveness in optimizing performance within the project's timeframe. Additionally, minor optimizations were applied, such as rewriting the density calculation in section 3.2.1 (which also applies to other summation calculations involving mass) to reduce the overall computational workload and precomputing smoothing kernel factors for the same effect. In hindsight, while the chosen simulation techniques proved effective within the project's constraints, it's worth noting that additional methods could have been explored for further optimization.

**3. Implement the researched techniques and identified features to develop a fluid simulation system using SPH.**

Each of the features mentioned in objective 1 have been implemented using standard SPH approximations, with force calculations such as pressure taking special considerations to be symmetrical to ensure accurate fluid behaviour as seen in section 4.3. Additionally, the simulation techniques from objective 2 have been implemented with in particular Spatial Hashing being implemented with two different methods suited to the individual implementation CPU or GPU.

Although the implemented features perform as intended, the lack of sufficient incompressibility necessitated the use of larger-than-usual masses to achieve higher levels of incompressibility. This limitation may restrict the accuracy and applicability of the features in scientific domains such as engineering[20] and medicine[21]

.

**4. Implement parallelized SPH algorithms tailored for GPU architectures to leverage their parallel processing capabilities and optimize memory access patterns.**

To meet Objective 4, parallelized SPH algorithms were developed. SPH algorithms were implemented on GPU architectures where each particle's forces were calculated using one thread of execution dedicated to that particle. Furthermore, memory access patterns were optimized by adopting structure of array formats instead of array of structure formats, following recommendations from [19]. Moreover, the GPU's parallel processing capabilities were utilized by employing a parallelized sorting algorithm, specifically bitonic sort [18].

**5. Analyse the scalability of the optimized SPH simulations across varying particle counts, examining how computational demands scale with increasing particle count.**

Sections 4.4 and 4.5 delve into the scalability analysis of the SPH simulations, demonstrating how framerate diminishes with increasing particle count. In the CPU implementation, there's an apparent linear rise in computation time as particle counts escalate beyond a certain threshold, as depicted in Figure 4.7. Similarly, the GPU implementation of SPH, illustrated in Figure 4.9, showcases a comparable trend. However, the GPU behaviour appears more erratic, possibly due to increased instability at higher particle counts.

Nonetheless, it can be generally concluded that SPH simulations demonstrate linear scalability in terms of performance. This likely stems from the Spatial Hashing approach with a computational complexity of , where the number of neighbouring particles remains consistent across all particles. It can be speculated that the observed linear behaviour is influenced by the consistency in neighbouring particle counts, with Spatial Hashing and Nearest Neighbour search playing pivotal roles as the main drivers of the simulation's overall performance and being the main cause of this linear behaviour.

In essence, enhancing the performance of the SPH simulation appears to hinge significantly on finding a more efficient solution for the Nearest Neighbour search problem.

Furthermore, upon comparison, the performance of my GPU implementation appears to be similar to that of other implementations, even surpassing a similar one [34] which also used a GTX 1070.

**6. Compare the performance of CPU and GPU implementations in simulating fluid dynamics across varying particle counts in terms of framerates and memory usage.**

In Section 4.4, the performance of the CPU and GPU implementations is compared. It's evident that the GPU implementation of SPH outperforms the CPU implementation significantly. Increasing particle counts within the usable range of SPH on the CPU, which spans a few thousand particles, noticeably affects the performance of the CPU implementation. In contrast, the GPU implementation remains largely unaffected by such increases in particle counts. Furthermore, Figure 4.6 illustrates how the CPU implementation exhibits worse scalability in terms of memory usage. This is primarily due to the impact of individual lists for each particle to store its neighbours for Spatial Hashing, which leads to a rapid increase in memory usage compared to the GPU implementation.

## 5.2 Skills Learnt

Before researching and I had little knowledge of the subject area of fluid simulation from both a scientific and computing background. Outside of straightforwardly comprehending the fluid dynamics domain, my engagement with the scientific aspect of implementing SPH, which involves utilizing numerical methods to solve the governing partial differential equations, provided me with practical experience in numerical techniques and knowledge of mathematical notations.

Engaging with SPH simulations enabled me to broaden my understanding of simulation physics, with a specific focus on particle systems. I gained insights into how individual particles interact with each other and their surroundings, resulting in behaviour that mirrors real-world phenomena. This knowledge transcends fluid dynamics and finds applications in diverse fields utilizing particle-based simulations, including granular materials [33], and soft body dynamics [7].

Moreover, this project presented opportunities to delve into optimization strategies during development. An essential component of this involves algorithmic thinking, where complex algorithms like Spatial Hashing and Nearest Neighbour Search (NNS) are employed to significantly boost the project's performance and scalability. Additionally, I delved into minor optimizations, such as pre-computing smoothing kernel factors and adapting summation calculations for single-fluid setups, to minimize redundant computations..

Finally, when I began this project, my exposure to parallelization was minimal. I had no prior experience in utilizing compute shaders in Unity for parallel programming, and my familiarity with writing HLSL shaders for computer graphics was limited. Through researching and implementing the GPU simulation, I significantly expanded my skill set. I can now proficiently write compute shaders, and I've acquired numerous optimization techniques for parallel processing. For example, I learned to use structure of array data structures instead of array of structures to optimize memory access patterns, thereby enhancing performance. Understanding how to leverage the massive parallelism offered by GPUs for accelerating computations has broadened my knowledge of high-performance computing architectures and parallel programming paradigms.

## 5.3 Further Work

There are three primary areas for extending the simulation: performance, accuracy, and usability. While performance and accuracy enhancements inherently contribute to usability, in this context, usability refers to expanding the project for purposes beyond accuracy and performance. For example, extending it for educational purposes. Generally, extensions aimed at improving accuracy and usability will inevitably impact performance.

**Accuracy:**

* The primary avenue for enhancing the accuracy of the simulation involves incorporating temperature dynamics to enable non-isothermal fluid simulations. This enhancement holds particular significance in more accurately simulating gaseous fluids, where a temperature-dependent buoyancy effect replaces artificial forces. The inclusion of temperature dynamics facilitates the representation of melting, boiling, and solidification phenomena, accompanied by heat conduction and propagation, as demonstrated in [23]. Such enhancements would render the simulation more applicable for scientific and engineering purposes.
* Improving the simulation's accuracy entails enhancing its incompressibility. Introducing predictive-corrective incompressible Smoothed Particle Hydrodynamics (SPH) methods [26] can address this. Notably, this project currently utilizes weakly-compressible SPH, a factor that limits time step sizes. By transitioning to incompressible SPH methods, larger time steps become feasible, facilitating real-time fluid simulation while maintaining stability.
* An issue observed in the simulation was the unnatural behaviour of particles at the

boundaries. Specifically, particles seemed to become stuck at the simulation bounds. This behaviour may be attributed to inaccuracies in pressure calculations near the boundaries. Introducing dynamic boundary conditions [29] could potentially mitigate this issue by promoting more accurate particle behaviour.

**Performance:**

* In this project, the GPU implementation is tailored for single GPU architectures, given the available resources and timeframe. However, an area for expansion lies in broadening parallel programming capabilities to accelerate for multi-GPU architectures, as demonstrated by Domínguez et al. [27]. This advancement would render the simulation better suited for large-scale simulations on systems like supercomputers, extending its applicability to complex computational tasks.
* In this project, a compromise was made regarding the selection of the sorting algorithm for the GPU implementation. The choice of a bitonic sorting algorithm was favoured for its simplicity in implementation, although faster sorting algorithms are available. Radix sort, in particular, boasts highly optimal complexity, making it a challenging algorithm to surpass on the GPU [28]. Investigating the implementation of radix sort could potentially improve the efficiency and performance of the GPU sorting process.

**Usability:**

* To adapt the simulation for applications in computer graphics and games, enhancing its graphical features is crucial. A significant visual enhancement would involve rendering the simulation as a continuous fluid rather than discrete particles, a feat achievable through surface reconstruction techniques [25].
* The simulation shows promise for educational use, especially in enhancing understanding of fluid dynamics. Fraser et al. [24] found that computer simulations significantly improved the learning experience for engineering students studying fluid mechanics. To maximize educational impact, future efforts could focus on improving accessibility and developing a dedicated educational application. This involves adding user-friendly interface elements for real-time parameter adjustments and integrating visual aids like directional arrows to illustrate fluid flow patterns. Additionally, providing pre-defined simulations of fluid phenomena and educational resources would further enhance learning opportunities.

# Bibliography

[1] Kelager, M., 2006. Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics. Graduate Project. Department of Computer Science, University of Copenhagen.

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

[3] Desbrun, M. and Gascuel, M.-P. (1996). Smoothed particles: A new paradigm for animating highly deformable bodies, In Computer Animation and Simulation 96 (Proceedings of EG Workshop on Animation and Simulation, Springer-Verlag, pp. 61–76.

[4] Ihmsen, Markus & Orthmann, Jens & Solenthaler, Barbara & Kolb, Andreas & Teschner, Matthias. (2014). SPH Fluids in Computer Graphics - Eurographics State-of-the-art report.

[5] Monaghan, J. (1992). Smoothed particle hydrodynamics, Annual Review of Astronomy and Astrophysics, pp. 543–574.

[6] Gingold, R. A. and Monaghan, J. J. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars. Monthly notices of the royal astronomical society, 181(3):375–389.

[7] Teschner, Matthias & Heidelberger, Bruno & Müller, Matthias & Pomeranets, Danat & Gross, Markus. (2003). Optimized Spatial Hashing for Collision Detection of Deformable Objects. VMV’03: Proceedings of the Vision, Modeling, Visualization. 3

[8] Harada, T., Koshizuka, S., Kawaguchi Y. (2007). Smoothed particle hydrodynamics on GPUs. Proc. Comput. Graph. Intl.63–70.

[9] Amada, T., Imura, M., Yasumuro, Y., Manabe, Y., Chihara,K. (2003). Particle-based fluid simulation on the GPU. Proc.ACM Workshop on General-purpose Computing on GraphicsProcessors.

[10] Hérault, A., Bilotta, G. and Dalrymple, R.A., 2010. Sph on gpu with cuda. *Journal of Hydraulic Research*, *48*(sup1), pp.74-79.

[11] Monaghan, J. J. (1989a). On the problem of penetration in particle methods, J. Comput. Phys. 82(1): 1–15.

[12] Green, Simon. 2010. “Particle Simulation Using Cuda” NVIDIA Whitepaper 6:121-128

[13] Priscott, C., 2010. 3D Lagrangian Fluid Solver using SPH approximations. Masters Project. Bournemouth University.

[14] Stam, J. (2003). Real-time fluid dynamics for games., Proceedings of the Game Developers Conference.

[15] Becker, M. and Teschner, M., 2007. Weakly compressible SPH for free surface flows. SCA ’07 Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, 209–217.

[16] T. Cormen, C. Leiserson, R. Rivest, “Introduction to Algorithms,” ISBN 0-262-03141-8, The MIT Press, Cambridge, Massachusetts, 1990.

[17] A.J.C. Crespo, J.M. Domínguez, B.D. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro, O. García-Feal, DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH), Computer Physics Communications, Volume 187, 2015, Pages 204-216,

[18] Batcher, K.E., 1968, April. Sorting networks and their applications. In Proceedings of the April 30--May 2, 1968, spring joint computer conference (pp. 307-314).

[19] Cai, Y., Wei, J., Hou, Q., Gao, R. (2022). An Optimized GPU Implementation of Weakly-Compressible SPH Using CUDA-Based Strategies. In: Lai, Y., Wang, T., Jiang, M., Xu, G., Liang, W., Castiglione, A. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2021. Lecture Notes in Computer Science(), vol 13155. Springer, Cham.

[20] You, Y., Khayyer, A., Zheng, X., Gotoh, H. and Ma, Q., 2021. Enhancement of δ-SPH for ocean engineering applications through incorporation of a background mesh scheme. *Applied Ocean Research*, *110*, p.102508.

[21] Toma, M., 2017. The emerging use of SPH in biomedical applications. *Significances Bioeng Biosci*, *1*(1), pp.1-4.

[22] Wenbo, C., Yao, Y. and Zhang, Y., 2014, December. Performance analysis of parallel smoothed particle hydrodynamics on multi-core CPUs. In *Proceedings of 2014 International Conference on Cloud Computing and Internet of Things* (pp. 85-90).

[23] Farrokhpanah, A., Bussmann, M. and Mostaghimi, J., 2017. New smoothed particle hydrodynamics (SPH) formulation for modeling heat conduction with solidification and melting. Numerical Heat Transfer, Part B: Fundamentals, 71(4), pp.299-312.

[24] Fraser, D.M., Pillay, R., Tjatindi, L. and Case, J.M., 2007. Enhancing the learning of fluid mechanics using computer simulations. Journal of Engineering Education, 96(4), pp.381-388.

[25] Onderik, J., Chládek, M. and Ďurikovič, R., 2011, April. SPH with small scale details and improved surface reconstruction. In Proceedings of the 27th Spring Conference on Computer Graphics (pp. 29-36).

[26] Peng, C., Bauinger, C., Szewc, K., Wu, W. and Cao, H., 2019. An improved predictive-corrective incompressible smoothed particle hydrodynamics method for fluid flow modelling. Journal of Hydrodynamics, 31(4), pp.654-668.

[27] Domínguez, J.M., Crespo, A.J., Valdez-Balderas, D., Rogers, B.D. and Gómez-Gesteira, M., 2013. New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications*, *184*(8), pp.1848-1860.

[28] Singh, D.P., Joshi, I. and Choudhary, J., 2018. Survey of GPU based sorting algorithms. International Journal of Parallel Programming, 46, pp.1017-1034.

[29] Chen, C., Zhang, A.M., Chen, J.Q. and Shen, Y.M., 2021. SPH simulations of water entry problems using an improved boundary treatment. Ocean Engineering, 238, p.109679.

[30] Richter, T., 2013. A fully Eulerian formulation for fluid–structure-interaction problems. Journal of Computational Physics, 233, pp.227-240.

[31] Xiu, D. and Karniadakis, G.E., 2001. A semi-Lagrangian high-order method for Navier–Stokes equations. Journal of computational physics, 172(2), pp.658-684.

[32] Melville, W.K., 1996. The role of surface-wave breaking in air-sea interaction. Annual review of fluid mechanics, 28(1), pp.279-321.

[33] Bell, N., Yu, Y. and Mucha, P.J., 2005, July. Particle-based simulation of granular materials. In Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (pp. 77-86).

[34] Gunadi, S.I. and Yugopuspito, P., 2018, August. Real-time gpu-based sph fluid simulation using vulkan and opengl compute shaders. In 2018 4th International Conference on Science and Technology (ICST) (pp. 1-6). IEEE.