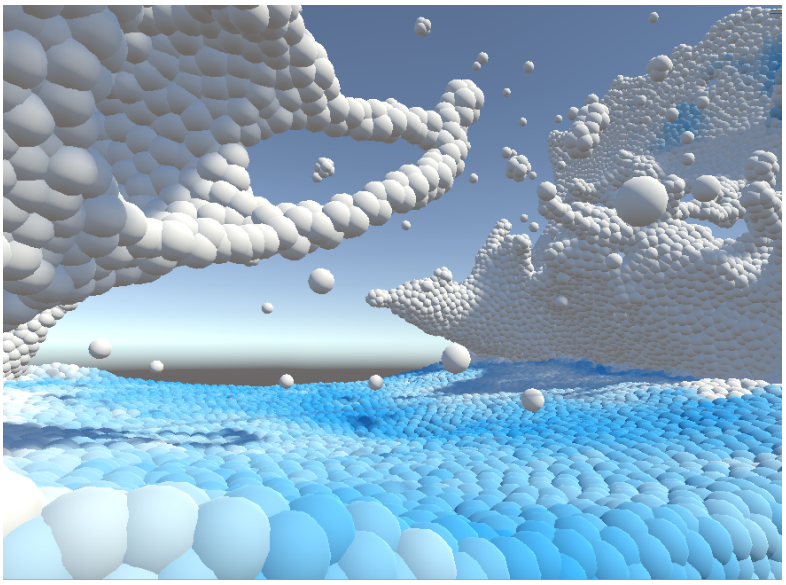
UNIVERSITAT JAUME·I

DySAQUASIMs
Dynamic Aquatic Systems And Advanced Simulations



Jaime Pérez Villena

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 19, 2024

Supervised by: Rafael Fernández Beltrán

To my dear parents, Amparo and Jaime,

whose infinite love, patience, and dedication have guided me to become the person I am today. Thanks to their relentless support, I have achieved dreams that I once thought were unattainable.

***

To my brother,

who not only shared those afternoons playing video games with me but also awakened my passion for this world.

Acknowledgments

I would like to thank my Final Degree Work supervisor, Rafael Fernández Beltrán, for his guidance through this project.

Abstract

This document introduces DySAQUASIMs, a tool developed for the Bachelor's Degree in Video Game Design and Development at Jaume I University. Implemented as a plugin for the Unity graphics engine, DySAQUASIMs facilitates the realistic simulation of fluids by leveraging advanced particle systems. This allows for the creation of interactive and visually compelling fluid dynamics within virtual environments, catering especially to dynamic aquatic systems integral to video game simulations.

A key objective of DySAQUASIMs is to simplify the integration of complex fluid behaviors into graphic applications, enabling even those with minimal knowledge of fluid dynamics to achieve professional results in game environments. This project explores and addresses the challenges of replicating intricate fluid behaviors using digital technologies, particularly in enhancing the realism and interactivity of video game water scenarios.

By integrating Unity's simulation techniques, DySAQUASIMs achieves high-quality visual effects without taxing computer resources excessively. This advancement not only serves educational purposes in game design but also pushes the boundaries of simulating natural elements within interactive media, setting a new standard in the visual and interactive quality of video games.

Keywords:

- Fluid Simulation
- Smoothed Particle Hydrodynamics (SPH)
- Unity
- Virtual Environments
- GPU-Accelerated Computing

Contents

CHAPTER 1

INTRODUCTION

Contents

This project delves into the intricate world of fluid dynamics simulation, a cornerstone in both video game development and physics research. By bridging the gap between theoretical physics and practical application, this work seeks to enhance the realism and interactivity of virtual environments. To ensure the system is accessible to all users, including those with limited knowledge of fluid dynamics, it simplifies usage mechanisms and optimizes the efficient use of physical computer resources. This dual focus not only advances the fidelity and interactivity of virtual scenarios but also democratizes the technology, making advanced simulations more accessible and practical in a wide range of applications.

1.1 Work Motivation

My fascination with physics has driven my academic pursuits and hobbies, culminating in this project. While the theoretical aspects of fluid dynamics were engaging and comprehensible, the real challenge lay in their implementation. Transforming complex equations and simulation techniques into a working model within a video game engine environment proved to be a formidable task. This project was born out of a desire to conquer these practical hurdles and to fulfill a long-standing aspiration.

Another significant motivation is to bring users and developers closer to experimenting with and leveraging these types of simulations in their projects. By demonstrating

how to implement extensions or integration within a graphics engine, this project aims to provide practical insights and tools. These are designed to make it easier for others to incorporate sophisticated fluid dynamics simulations into their own work, thereby expanding the practical applications and impact of this field.

1.2 Objectives

The overarching goal of this project is to bridge the gap between theoretical knowledge and practical application in the realm of fluid dynamics. By harnessing the power of modern computational methods, aiming to transform abstract concepts into tangible, interactive simulations that both educate and enhance user experience in digital environments.

The primary objective is to develop a robust graphical tool that provides realistic simulations of fluid behavior. This tool will be tailored specifically for integration within the Unity engine, making it accessible not only to students and professionals in the field of video game design but also to those with limited technical background in fluid dynamics. The plugin will serve dual purposes: it will be an educational resource, elucidating the principles of fluid behavior, and a practical application that enhances the realism of game environments, thereby elevating the overall gaming experience.

Furthermore, this project aims to establish a foundation for ongoing research and development. It will explore the simulation of complex physical phenomena in virtual spaces, paving the way for future innovations that could transform how interactive environments are created and experienced. By making the tool intuitive and simple to use, it encourages users and developers to incorporate advanced simulations into their projects, promoting a broader understanding and application of dynamic systems within the gaming industry.

1.3 Environment and Initial State

Ever since I was gifted a book on astronomy as a child, I have held a profound interest in the sciences. This fascination has significantly shaped my academic and personal pursuits, especially my attraction to physics and understanding the mechanics of the natural world. My particular affection for aquatic scenarios—where water is not only a visual element but also a key mechanical player—has greatly influenced my academic path. When tasked with selecting a thesis topic, I brainstormed various ideas, which led me to explore the complexities of liquid simulation online, a subject known for its formidable challenges.

My initial exploration into existing technologies and methods used for fluid dynamics simulation in video games and other interactive applications highlighted the inherent challenges in this field. The complexity of fluid simulations stems primarily from the need to solve intricate partial differential equations, such as the Navier-Stokes equations,

which describe fluid behavior. These equations are not only fundamental to understanding fluid motion but are also computationally demanding, requiring significant resources for accurate and real-time simulations.

Achieving a balance between computational efficiency and the realism of fluid simulations is crucial in video games and interactive environments. The adoption of methods like SPH (Smoothed Particle Hydrodynamics) has provided a framework for interactive and visually faithful simulations. However, these methods still demand careful optimization of performance to run smoothly on standard gaming hardware without compromising the interactive quality or visual fidelity of the simulations.

This project aims to address these challenges by developing a water simulation system that not only meets the visual and physical realism expected in modern video games but is also optimized for better performance. This involves minimizing the computational load while ensuring that the simulations can be easily integrated and utilized within a game engine by developers, regardless of their expertise in fluid dynamics. By enhancing the accessibility and usability of such sophisticated simulations, the project seeks to empower more developers to incorporate dynamic aquatic systems into their projects, enriching the overall gaming experience and broadening the scope of interactive game design.

My initial project ideas ranged from creating a fishing game, where water would play a significant role in gameplay, to developing an interactive fluid simulator. After presenting these ideas and discussing with my mentor, Rafael Fernández Beltrán, I decided to pursue the latter based on the potential for broader educational and developmental impact. This choice was further solidified by the foundational insights gained from influential tutorials on how SPHs work.

Thus began the development of this project with a strong foundation in Unity. The concept was clear, though the process was daunting. My enthusiasm was boundless, and despite having limited resources, the depth of the available materials within the Unity environment allowed for a thorough understanding and addressing of technical limitations. This solid base paved the way for innovative solutions in simulating complex fluid dynamics.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter outlines the most technical aspects of the project, detailing the comprehensive time planning and resource allocation required to achieve the project objectives. Each task is broken down by hours allocated and specific goals.

2.1 Planning

Detailed planning for the project. Below is a breakdown of all tasks and sub-tasks, along with the estimated time required for each.

- **Task 1 (30 hours):** Study the theory behind Particle Simulation using CUDA, Particle-based Viscosity Fluid Simulation, Particle-based Fluid Simulation for Interactive Applications, and Smoothed Particle Hydrodynamics Techniques for the Physics-Based Simulation of Fluids and Solids.
- **Task 2 (10 hours):** Engage with tutorials to apply theories learned in Task 1 within Unity environment.
- **Task 3 (5 hours):** Define the objectives of the simulation, identify desired outcomes, and research efficient shaders that align with the project requirements.
- **Task 4 (20 hours):** Integrate fundamental mathematical operations essential for the simulation.

- **Task 5 (10 hours):** Apply the mathematical models to default-generated Unity spheres and test their interactions.

- **Task 6 (15 hours):** Optimize particle systems to reduce computational load and enhance rendering efficiency for multiple instances in the simulation.

- **Task 7 (35 hours):** Learn about compute shaders and compute buffers to optimize and better integrate with Unity.

- **Task 8 (30 hours):** Integrate and distribute kernels for parallel calculations to avoid race conditions when modifying variables simultaneously.

- **Task 9 (20 hours):** Develop methods to iterate through large numbers of particles efficiently, avoiding unnecessary iterations over isolated particles.

- **Task 10 (20 hours):** Implement and refine a Bitonic Merge Sort algorithm for the project.

- **Task 11 (25 hours):** Compile and review the outcomes and processes from Tasks 1 to 10.

- **Task 12 (40 hours):** Integrate shaders to create a meta-ball effect on particles, enhancing their visual similarity to liquids. If integration faces technical limitations, document the challenges and propose the development as an extension for future enhancements.

- **Task 13 (10 hours):** Make final adjustments to increase user interaction and preconfigure settings.

- **Task 14 (30 hours):** Documentation and Presentation

Total hours: 300

This plan outlines the intended sequence of operations; however, due to the project's complexity, actual progress varied. The following sections detail the execution and adjustments made during each task.

In Figure 2.1, we will present a detailed Gantt chart outlining each task allocation from the project's inception, starting with the presentation of the technical proposal, through to its conclusion with the preparation of documentation and presentation.

Task	Estimated Duration (h)	January	February	March	April	May	June
Study Theoretical Frameworks	**30**						
Research CUDA and Particle Simulations	7,5						
Learn Smoothed Particle Hydrodynamics	7,5						
Study Fluid Simulation Techniques	7,5						
Review Physics-Based Simulation of Fluids and Solids	7,5						
Engage with Unity Tutorials	**10**						
Setup Unity for Particle Simulations	1						
Implement Basic Particle Systems in Unity	4						
Apply Theoretical Concepts Practically	5						
Define Project Objectives	**5**						
Define Simulation Objectives	2,5						
Research and Identify Efficient Shaders	2,5						
Mathematical Operations Integration	**20**						
Develop Fundamental Math Operations for Simulation	10						
Integrate Operations into Unity	10						
Apply Mathematical Models	**10**						
Implement Models in Unity Spheres	5						
Test Interactions between Spheres	5						
Optimize Particle Systems	**15**						
Reduce Computational Load	7,5						
Enhance Rendering Efficiency	7,5						
Learn Compute Shaders and Compute Buffers	**35**						
Study Compute Shaders	10						
Implement Compute Buffers in Unity	25						
Parallel Calculations Integration	**30**						
Develop Kernel Distribution Strategy	15						
Implement and Test Kernels	15						
Develop Particle Iteration Methods	**20**						
Design Efficient Iteration Algorithms	10						
Implement Algorithms in Unity	10						
Bitonic Merge Sort Implementation	**20**						
Study Bitonic Merge Sort	10						
Integrate Algorithm into Project	10						
Review Project Outcomes	**25**						
Compile Results from Tasks 1-10	5						
Review and Adjust Processes	20						
Shader Integration for Meta-ball Effect	**40**						
Develop Meta-ball Shaders	20						
Integrate and Test Shaders	20						
User Interaction and Settings Adjustments	**10**						
Enchance User Interface	5						
Configure Predefined Settings	5						
Documentation and Presentation	**30**						
Document Project Findings	15						
Prepare and Rehearse Presentation	15						

Figure 2.1: Gantt Chart of the Project Timeline

2.2 Resource Evaluation

This section evaluates the necessary resources, both hardware and software, required for the development and implementation of the project. It also includes an estimation of costs for these resources and human labor.

2.2.1 Hardware Resources

The hardware selected ensures optimal performance for software development and simulation tasks. The detailed specifications of the hardware used are outlined in Table 2.1.

Component	Specification
Laptop	GAMING Laptop with 12th Gen Intel(R) Core(TM) i7-12700H (2.30 GHz), 32.0 GB RAM, 64-bit OS on x64 processor, NVIDIA GeForce RTX 4060 GPU

Table 2.1: Hardware used for development and testing

2.2.2 Software Resources

The software tools are essential for developing, testing, and managing the project. The primary software resources employed are summarized in Table 2.2.

Software	Description
Unity 3D 2022.3.11f1 (Free)	Primary development platform used for creating interactive game simulations.
Visual Studio Code 2024 (Free)	IDE with robust support for C#, used primarily for programming within Unity.
GitHub Desktop (Free)	Provides version control to manage coding changes and maintain a stable development environment.
Wolfram Alpha (Free)	Employed for validating complex mathematical formulations and computations in fluid dynamics.

Table 2.2: Software resources used in the project

2.2.3 Tools and Other Resources

These tools support various non-development aspects of the project, from project management to content creation. A comprehensive list of these tools is available in Table 2.3.

Tool	Use
Trello	Online project management tool used to track progress, assign tasks, and manage deadlines effectively.
Unity Asset Store	Accessed for obtaining both free and paid assets to enhance the visual and functional capabilities of the game.
Google Meet	Utilized for tutor assistance and team meetings.
Streamlabs Desktop	Used for capturing high-quality video recordings of simulations for analysis and presentation purposes.
StackOverflow	Resource for code suggestions and community support to solve programming issues and enhance code quality.
Google Sheets	Used for creating the Gantt chart to visually represent the project timeline and task allocation.

Table 2.3: Tools and other resources used in the project

2.2.4 Estimated Costs

The primary costs associated with the project were categorized into human resources, hardware, and software. Each of these cost categories is detailed below.

Human Resources

Human resources constituted a significant portion of the project's costs. The project involved a junior game developer working for a total of 260 hours. The cost for human resources was calculated at an average rate of 10.77 EUR per hour, resulting in an estimated total of 2800 EUR. These figures are summarized in Table 2.4. The average rate was determined based on the salary information for junior developers in Spain [10].

Hardware Costs

The project required robust hardware to ensure optimal performance for software development and simulation tasks. A high-performance gaming laptop was used, with the following specifications: - 12th Gen Intel(R) Core(TM) i7-12700H (2.30 GHz) - 32.0 GB RAM - 64-bit OS on x64 processor - NVIDIA GeForce RTX 4060 GPU

The cost of this gaming laptop was estimated at approximately 1500 EUR, as detailed in Table 2.4.

Software Costs

The project utilized various software tools, all of which were accessed in their free versions. The main software tools used included Unity 3D, Visual Studio Code, GitHub Desktop, and Wolfram Alpha. The use of free versions of these tools ensured that there were no additional software costs incurred for the project.

Cost Category	**Details**
Hardware Costs	1500 EUR.
Software Costs	0 EUR.
Human Resources	2800 EUR.
Total	4300 EUR.

Table 2.4: Estimated financial costs for the project

CHAPTER 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Theoretical Framework

Fluid simulation is grounded in the principles of Computational Fluid Dynamics (CFD), a well-explored and established field. The complexity of liquid behavior emerges from the interaction of various phenomena such as convection, diffusion, turbulence, and surface tension. Accurate simulation of these behaviors is crucial for a wide range of applications, from aerodynamics to entertainment technologies.

To begin, we turn to the Navier-Stokes equations, fundamental equations for the dynamics of moving fluids. These describe the conservation of momentum along with continuity equations for mass conservation and state equations for energy conservation. They are the cornerstone of fluid simulation, enabling the fluid model to flow and adapt its behavior under varying conditions.

The Navier-Stokes equations in the SPH context are represented as follows:

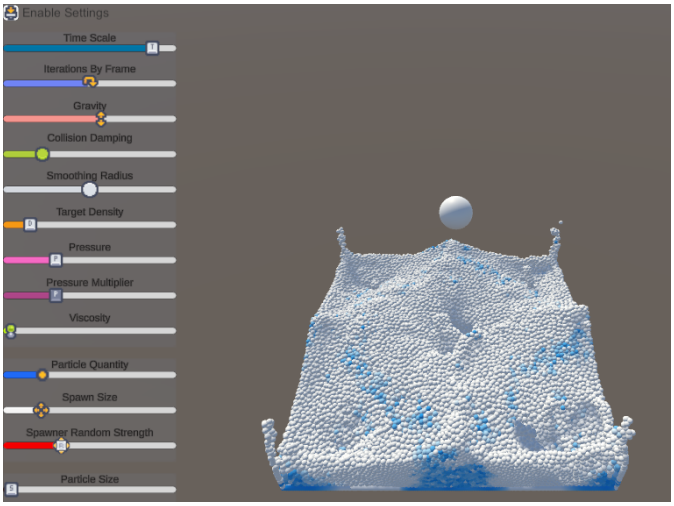


Figure 3.1: Simulation UI

1. **Stress Tensor for Incompressible Flow:**

$$T = -p\mathbf{I} + \mu(\nabla v + \nabla v^T) \tag{3.1}$$

Where p is the pressure, μ is the dynamic viscosity, and v is the fluid velocity.

2. **Incompressible Navier-Stokes Equation:**

$$\rho\left(\frac{Dv}{Dt}\right) = -\nabla p + \mu\nabla^2 v + f_{\text{ext}} \tag{3.2}$$

With ρ being the fluid density, and f_{ext} representing external forces.

These formulations detail how the stress tensor for incompressible fluids is defined and how it is used within the Navier-Stokes equation to describe fluid motion.

3.1.1 Eulerian and Lagrangian Methods

The Eulerian method uses a regular grid to solve the 3D Navier-Stokes equations, reproducing the visual properties of dynamic fluids. This technique has been expanded to include simulations of dynamic gases, liquids, and highly viscous or viscoelastic fluids, integrating behaviours like elasticity and plasticity, thus enabling realistic simulations of melting objects and complex fluid properties.

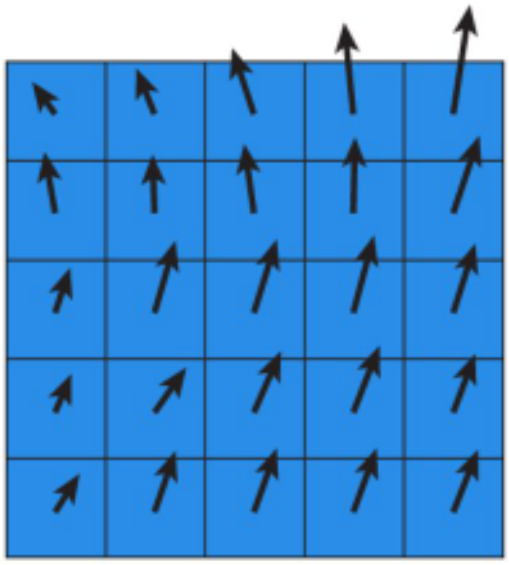


Figure 3.2: Illustration of an Eulerian grid in fluid simulation.

In contrast, the **Lagrangian method**, exemplified by Smoothed Particle Hydrodynamics (SPH), uses particles to sample space non uniformly. These particles carry properties such as mass density and velocity, allowing for the evaluation of field quantities anywhere in space through radial smoothing kernels, offering a distinct approach to fluid simulation.

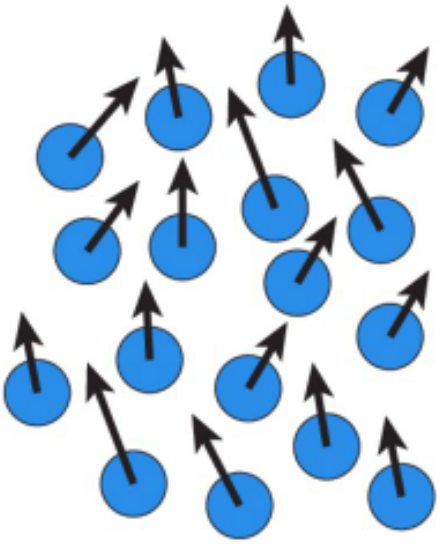


Figure 3.3: Smoothed Particle Hydrodynamics (SPH) illustrating particle distribution in fluid simulation.

3.1.2 Smoothed Particle Hydrodynamics (SPH)

SPH (Smoothed Particle Hydrodynamics) [6] is an advanced computational technique employed for simulating the dynamics of continuous media, such as fluids, gases, and even solid materials, in a wide array of scientific and engineering applications. This mesh-free, Lagrangian method models the medium by discretizing it into a collection of particles. Each particle represents a fluid element that carries essential physical properties, including mass, momentum, and energy.

SPH was originally developed for astrophysical problems but has since been adapted and expanded for use in various fields due to its flexibility and robustness. Unlike grid-based methods (Eulerian), SPH handles deformable bodies and interfaces naturally, making it particularly useful for simulating complex free-surface flows and multi-phase interactions.

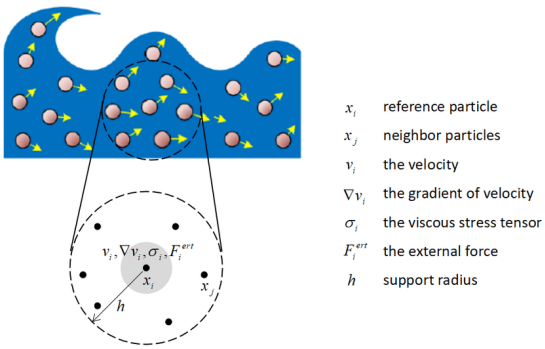


Figure 3.4: Smoothed Particle Hydrodynamics (SPH).

Using the SPH algorithm, these particles interact with one another based on their proximities and a smoothing kernel function, which accurately approximates the physical quantities across the fluid domain. This approach facilitates the simulation of complex fluid behaviours and interactions, offering remarkable flexibility and adaptability to model various physical scenarios without the constraints of a fixed grid structure

Key Components of SPH

Kernel Functions: The core of SPH involves kernel functions, which are used to approximate field quantities and their spatial derivatives. A commonly used kernel is the cubic spline kernel, which is compact and efficient. The choice of kernel affects the accuracy and stability of the simulations.
Density Estimation: The density at a particle's location is estimated by summing the contributions from neighboring particles, weighted by the kernel function. This local density estimation allows for capturing detailed variations within the simulated medium.
Pressure Calculation: SPH computes pressure forces based on the density field. Two main approaches exist: using a state equation (relating pressure to density) or solving the Pressure Poisson Equation (PPE) to enforce incompressibility.

Viscosity and Surface Tension: Additional physical phenomena such as viscosity and surface tension are modeled using SPH by incorporating terms that account for these forces. Viscosity is often treated using formulations that ensure smooth transitions and stable simulations.

SPH Algorithms

Neighbor Search: Efficient neighbor search algorithms are crucial for SPH as they determine which particles influence each other. Commonly, a uniform grid or spatial hashing techniques are used to expedite the process of finding neighboring particles within the kernel support radius.

Applications of SPH

Fluid Simulation: SPH is widely used for fluid simulations in computer graphics, particularly for visual effects in movies and video games. It captures the dynamics of water, smoke, fire, and other fluids with high fidelity.
Engineering: In engineering, SPH is employed for simulating phenomena such as erosion, sediment transport, and fluid-structure interactions. Its ability to handle complex boundaries and interfaces makes it ideal for these applications.
Astrophysics: The method's origins in astrophysics involve simulating the formation and evolution of stars, galaxies, and other celestial phenomena. SPH's particle-based approach is well-suited for the vast, dynamic systems found in space.

Advantages and Challenges

Advantages:

1. SPH naturally handles large deformations and complex boundary interactions.
2. The method is flexible and can be adapted to various types of continua, including fluids, gases, and solids.
3. SPH is highly parallelizable, making it suitable for modern GPU-based computations.

Challenges:

1. The accuracy of SPH depends heavily on the choice of kernel and the resolution of particles.
2. Boundary handling can be complex and requires special techniques to ensure physical accuracy.
3. Computational cost can be high for large-scale simulations due to the need for neighbor searches and kernel evaluations.

3.2 Requirement Analysis

In this section, the functional and non-functional requirements of the developed work will be explored. However, before delving into the content, it is worth noting that the significance of the project lies in the development of a system that implements the Smoothed Particle Hydrodynamics (SPH) method to simulate fluid dynamics in an interactive and efficient manner.

Requirement analysis involves identifying and documenting the necessary functional requirements for the system. This process ensures that the system's functionality aligns with user needs and project goals. The requirements are analyzed to determine how they will be implemented and how they will interact with other system components.

- **R1: Pause Simulation**
- **R2: Frame Navigation**
- **R3: Reset Simulation**
- **R4: Modify Simulation Speed**
- **R5: Modify Iterations per frame**
- **R6: Modify Gravity**
- **R7: Modify Collision Damping**
- **R8: Modify Smoothing Radius**
- **R9: Modify Target Density**
- **R10: Modify Pressure**
- **R11: Modify Pressure Multiplier**
- **R12: Modify Viscosity**
- **R13: Modify Particle Quantity**
- **R14: Modify Spawn Size**
- **R15: Modify Spawn Random**

3.2.1 Functional Requirements

A functional requirement defines a system function that will be developed. It describes the set of inputs, the behavior, and the outputs of that function [11]. Functional requirements might include calculations, technical details, data manipulations, and other specific functionalities that define what the system is supposed to achieve.

In general, the functional requirements refer to what the system should do and can be expressed in the form "the system will do <requirement>". The plan to implement these will be detailed in the system design section (see Section 3.3).

The functional requirements in this project are:

R1: Pause Simulation	
Input:	Pressing the spacebar
Output:	Simulation is paused or resumed
When the user presses the spacebar, the simulation will toggle between paused and running states. This allows users to pause the simulation to analyze specific moments in detail.	

Table 3.1: Functional requirement «R1: Pause Simulation»

R2: Frame Navigation	
Input:	Pressing left or right arrow keys
Output:	Simulation moves backward or forward by one frame
Pressing the left arrow key will rewind the simulation by one frame, and pressing the right arrow key will advance it by one frame. This enables precise control over the simulation for detailed analysis.	

Table 3.2: Functional requirement «R2: Frame Navigation»

R3: Reset Simulation	
Input:	Pressing the 'R' key
Output:	Simulation restarts
When the user presses the 'R' key, the simulation will reset to its initial state, allowing the user to start the simulation from the beginning under the same conditions or new ones adjusted via the UI.	

Table 3.3: Functional requirement «R3: Reset Simulation»

R4: Modify Simulation Speed	
Input:	User adjusts the simulation speed via a UI slider
Output:	Simulation speed is adjusted accordingly, and simulation restarts.
Adjusting the simulation speed slider allows the user to control how fast the simulation runs, making it possible to slow down for detailed observation or speed up for quicker results.	

Table 3.4: Functional requirement «R4: Modify Simulation Speed»

R5: Modify Iterations per frame	
Input:	User adjusts number of iterations via a UI slider, and simulation restarts.
Output:	Iterations setting is updated in the simulation
By adjusting the iterations slider, the user can increase or decrease the number of iterations applied to particles, affecting how they interact within the simulation environment.	

Table 3.5: Functional requirement «R5: Modify Iterations per frame»

R6: Modify Gravity	
Input:	User adjusts gravity via a UI slider
Output:	Gravity setting is updated in the simulation
By adjusting the gravity slider, the user can increase or decrease the gravitational force applied to particles, affecting how they interact within the simulation environment.	

Table 3.6: Functional requirement «R6: Modify Gravity»

R7: Modify Collision Damping	
Input:	User adjusts gravity via a UI slider
Output:	Collision Damping setting is updated in the simulation
By adjusting the collision damping slider, the user can increase or decrease the loss of force applied to particles when a collision is made with the walls or items of the simulation.	

Table 3.7: Functional requirement «R7: Modify Collision Damping»

R8: Modify Smoothing Radius	
Input:	User adjusts smoothing radius via a UI slider
Output:	Smoothing radius setting is updated in the simulation, and simulation restarts.
By adjusting the smoothing radius slider, the user can increase or decrease the interaction sphere applied to particles, affecting how they interact within the simulation environment.	

Table 3.8: Functional requirement «R8: Modify Smoothing Radius»

R9: Modify Target Density	
Input:	User adjusts density via a UI slider
Output:	Target density setting is updated in the simulation, and simulation restarts.
By adjusting the target density slider, the user can increase or decrease the desired density applied to particles, affecting how they interact within the simulation environment.	

Table 3.9: Functional requirement «R9: Modify Target Density»

R10: Modify Pressure	
Input:	User adjusts pressure via a UI slider
Output:	Pressure setting is updated in the simulation, and simulation restarts.
By adjusting the pressure slider, the user can increase or decrease the desired pressure applied to the simulation environment, forces help maintain fluid volume and handle interactions between fluid particles. It ensures that particles repel each other to mimic the incomprehensibility of liquids.	

Table 3.10: Functional requirement «R10: Modify Pressure»

R11: Modify Pressure Multiplier	
Input:	User adjusts pressure multiplier via a UI slider
Output:	Pressure multiplier setting is updated in the simulation, and simulation restarts.
By adjusting the pressure multiplier slider, the user can increase or decrease the desired pressure multiplier applied to the simulation environment, such as its comprehensibility and the speed at which pressure equilibrates across the fluid. It allows to control how aggressively the simulation responds to density changes, thereby affecting the overall dynamics and visual appearance of the fluid.	

Table 3.11: Functional requirement «R11: Modify Pressure Multiplier»

R12: Modify Viscosity	
Input:	User adjusts viscosity via a UI slider
Output:	Viscosity setting is updated in the simulation, and simulation restarts.
Changing the viscosity slider modifies the fluid's resistance to gradual deformation by shear or tensile stress, directly impacting the fluid flow behavior in the simulation.	

Table 3.12: Functional requirement «R12: Modify Viscosity»

R13: Modify Particle Quantity	
Input:	User adjusts particle quantity via a UI slider
Output:	Particle count is updated in the simulation, and simulation restarts.
Adjusting the particle quantity slider increases or decreases the number of particles in the simulation, allowing for denser or sparser fluid representations.	

Table 3.13: Functional requirement «R13: Modify Particle Quantity»

R14: Modify Spawn Size	
Input:	User adjusts spawn size via a UI slider
Output:	Spawner size is updated in the simulation, and simulation restarts.
Adjusting the spawner size slider increases or decreases the size of the box that spawns particles in the simulation.	

Table 3.14: Functional requirement «R14: Modify Spawn Size»

R15: Modify Spawn Random	
Input:	User adjusts random spawn via a UI slider
Output:	Spawner randomness is updated in the simulation, and simulation restarts.
Adjusting the spawner randomness slider increases or decreases the randomness value of the box that spawns particles in the simulation, making it easier to control how the particles spawn if in an oriented 3x3x3 array or randomly.	

Table 3.15: Functional requirement «R15: Modify Spawn Random»

3.2.2 Non-functional Requirements

Non-functional requirements define the system attributes such as security, reliability, performance, maintainability, scalability, and usability. These requirements ensure that the system meets certain standards and operates efficiently under various conditions.

The non-functional requirements for this project are as follows:

- **R1: Performance.**
  - The simulation should update in real-time with a refresh rate of at least 30 40 frames per second (fps) under standard conditions.

  - The system should handle up to 91125 particles without a significant drop in performance (no more than 10% reduction in fps).

- **R2: Usability.**

  - The user interface should be intuitive and easy to navigate, allowing users to control the simulation without extensive training.
  - Controls for adjusting simulation parameters should be clearly labeled and easily accessible within the UI.

- **R3: Reliability.**

  - The system should have an up-time of 99.9%, ensuring high availability and minimal downtime.
  - Any changes in simulation parameters should be applied consistently without causing the system to crash or produce errors.

- **R4: Scalability.**

  - The system should be scalable to handle increased numbers of particles and more complex simulations as required by future updates or user needs.

3.3 System Design

This section must present the (logical or operational) design of the system to be carried out.

In the following pages are defined the cases of use taken from the functional requirements, a case use diagram, a class diagram, and a activities diagram. The plugin is really simple, the inputs are clear what they do in the simulation UI, but the output is not known.

3.4 Functional Requirements and Use Cases

The functional requirements and use cases in this project are based on simulation control operations performed by the user through the user interface.

Requirement:	R1: Pause Simulation
Actor:	User
Description:	The user can pause the simulation by pressing the spacebar.
Preconditions:	The simulation must be running.
Steps (Normal Sequence):	1. The user presses the spacebar. 2. The simulation pauses.
Alternative Sequence:	None.

Table 3.16: Use case of R1: Pause Simulation

Requirement:	R2: Frame Navigation
Actor:	User
Description:	The user can move forwards or backwards one frame at a time using the keyboard arrows.
Preconditions:	The simulation must be in a paused state.
Steps (Normal Sequence):	1. The user presses the left arrow key to move one frame back. 2. The user presses the right arrow key to move one frame forward.
Alternative Sequence:	None.

Table 3.17: Use case of R2: Frame Navigation

Requirement:	R3: Reset Simulation
Actor:	User
Description:	The user can reset the simulation to its initial state by pressing the 'R' key.
Preconditions:	The simulation can be in any state (running, paused, etc.).
Steps (Normal Sequence):	1. The user presses the 'R' key. 2. The simulation restarts to initial conditions.
Alternative Sequence:	None.

Table 3.18: Use case of R3: Reset Simulation

Requirement:	R4: Modify Simulation Speed
Actor:	User
Description:	The user can adjust the simulation speed through a UI slider.
Preconditions:	The simulation must be initialized.
Steps (Normal Sequence):	1. The user accesses the speed setting through the UI. 2. The user adjusts the speed slider. 3. The simulation speed changes according to the slider position.
Alternative Sequence:	None.

Table 3.19: Use case of R4: Modify Simulation Speed

Requirement:	R5: Modify Iterations per Frame
Actor:	User
Description:	The user can adjust the number of iterations per frame through a UI slider.
Preconditions:	The simulation must be initialized.
Steps (Normal Sequence):	1. The user accesses the iterations setting through the UI. 2. The user adjusts the iterations slider. 3. The simulation updates the iterations per frame according to the slider position.
Alternative Sequence:	None.

Table 3.20: Use case of R5: Modify Iterations per Frame

Requirement:	R6: Modify Gravity
Actor:	User
Description:	The user can adjust gravity settings through a UI slider.
Preconditions:	The simulation must be initialized.
Steps (Normal Sequence):	1. The user accesses the gravity setting through the UI. 2. The user adjusts the gravity slider. 3. The gravity effect in the simulation changes according to the slider position.
Alternative Sequence:	None.

Table 3.21: Use case of R6: Modify Gravity

Requirement:	R7: Modify Collision Damping
Actor:	User
Description:	The user can adjust collision damping through a UI slider.
Preconditions:	The simulation must be initialized.
Steps (Normal Sequence):	1. The user accesses the collision damping setting through the UI. 2. The user adjusts the damping slider. 3. The collision damping effect in the simulation changes according to the slider position.
Alternative Sequence:	None.

Table 3.22: Use case of R7: Modify Collision Damping

Requirement:	R8: Modify Smoothing Radius
Actor:	User
Description:	The user can adjust the smoothing radius through a UI slider.
Preconditions:	The simulation must be initialized.
Steps (Normal Sequence):	1. The user accesses the smoothing radius setting through the UI. 2. The user adjusts the smoothing radius slider. 3. The simulation updates the smoothing effect according to the slider position.
Alternative Sequence:	None.

Table 3.23: Use case of R8: Modify Smoothing Radius

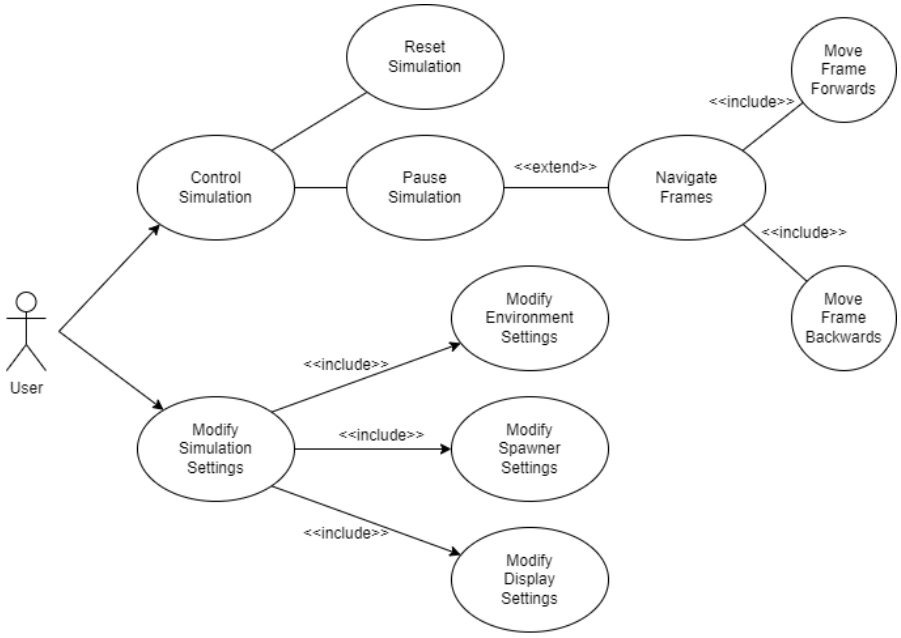


Figure 3.5: Use Case Diagram

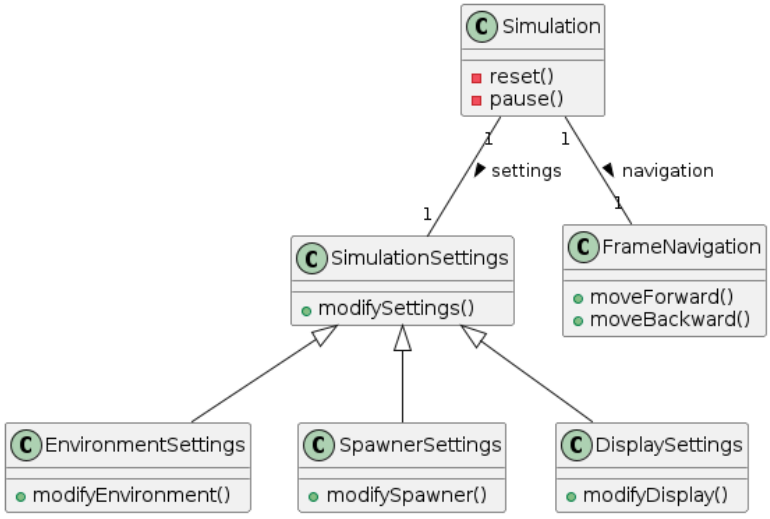


Figure 3.6: Class Diagram

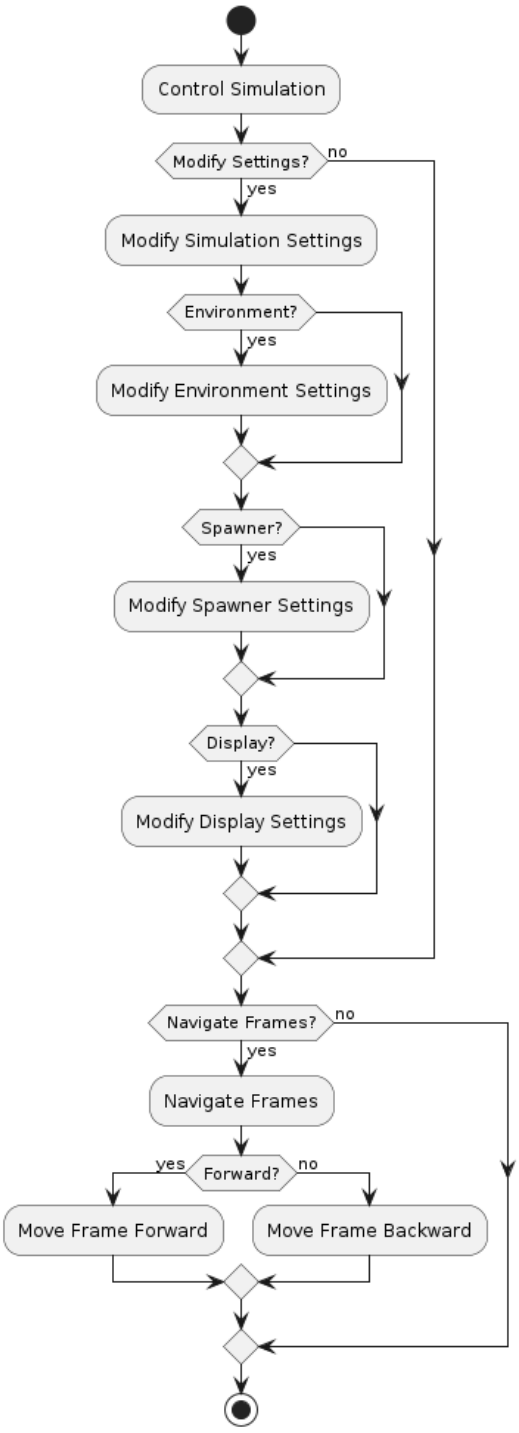


Figure 3.7: Activities Diagram

3.5 System Architecture

In this section, we shall delineate the minimum and recommended requisites for the projected system. While parallelism techniques have been implemented to augment the simulation's capabilities, it becomes imperative to possess a GPU with adequate processing capacity. Additionally, ensuring the flawless execution of the simulation involves meeting specific hardware and software requirements. The architecture of the system is designed to leverage modern computational resources effectively, particularly in the context of fluid simulations using the Smoothed Particle Hydrodynamics (SPH) method. The use of parallel processing and GPU acceleration is central to achieving the desired performance and accuracy.

3.5.1 Minimum System Requirements

The simulation requires the following minimum system specifications to run:

- **Operating System:** Windows 10 (64-bit)
- **Processor:** Intel Core i5-7500 or AMD Ryzen 3 1300X
- **RAM:** 8 GB
- **Graphics Card:** NVIDIA GeForce GTX 1050 Ti or AMD Radeon RX 560 with 4 GB VRAM
- **Graphics API:** DirectX 11
- **Storage:** SSD of 256 GB or HDD of 1 TB (SSD recommended for less load time)
- **Internet Connection:** Required for updates and online features
- **Additional Requirements:** Hardware vendor officially supported drivers. For development: IL2CPP scripting backend requires Visual Studio 2015 with C++ Tools component or later and Windows 10+ SDK.

3.5.2 Recommended System Requirements

For better performance, particularly in scenarios requiring intensive computation, such as the intense use of particle systems and parallel processes, the following system specifications are recommended:

- **Operating System:** Windows 10 or higher (64-bit)
- **Processor:** High-performance CPU with advanced multi-threading capabilities, such as Intel Core i7-9700K or AMD Ryzen 5 3600
- **RAM:** At least 16 GB to handle large simulations efficiently

- **Graphics Card:** High-end GPU with a significant number of CUDA cores or equivalent, such as NVIDIA GeForce RTX 2070 or AMD Radeon RX 5700 XT with 8 GB VRAM, supporting DirectX 12
- **Storage:** Solid State Drive (SSD) for faster data access and improved performance
- **Internet Connection:** High-speed internet connection for seamless updates and online functionalities
- **Additional Requirements:** Latest drivers and system updates for optimal performance and stability.

Why Cores are important?
The number of particles affects the number of threads required for the simulation. Each thread handles computations for a subset of particles, and the number of threads per group is fixed. Consequently, the number of thread groups ($\lceil \text{numParticles/NumThreads} \rceil$) determines whether all CPU/GPU cores can be fully utilized or if they will need to time-slice the workload.

- **Fully Utilized:** numCores $\geq$ threadGroupCount
- **Time-Sliced:** numCores $<$ threadGroupCount

This relationship helps optimize performance by balancing the particle count with the available core resources. Fully utilizing the cores ensures maximum efficiency, while time-slicing can lead to reduced performance due to increased context switching and overhead.

These specifications ensure that the simulation can run efficiently with optimal rendering speeds and computational accuracy. Leveraging modern hardware capabilities, especially with CPUs and GPUs designed for parallel processing, is crucial to achieving high performance.

3.6 Interface Design

Since this project is a Unity Plugin, the user interface (UI) displayed in the program is designed for better interaction with users interested in utilizing the plugin. The final product of the interface design should not necessarily include a UI for the end player, as it is up to the programmer or user developing the game to configure the desired water simulation settings. In any case, a programmer is fully capable of making the UI accessible or hidden for the players, depending on the intended use.

The programmer who wants to implement the plugin will just need to download it, assign the scripts correctly that come with the package (or create their own scripts since the system is scalable), and configure the desired properties and settings in the Unity Inspector.

The UI for the simulation control is designed to be intuitive and user-friendly, allowing users to easily interact with and manipulate the simulation parameters. The design includes various sliders and buttons for controlling different aspects of the simulation, which are described in detail below.

3.6.1 Control Panel

The control panel allows users to adjust various simulation parameters using sliders [7]. Each slider is labeled with its corresponding function for ease of use. These controls are illustrated in Figure 3.8.

3.6.2 Simulation View

The simulation view displays the particles in real-time based on the parameters set in the control panel. Some changes in the control panel reset the simulation to enhance performance. It provides a visual representation of how changes to the controls affect the simulation. The view is designed to be responsive, updating dynamically as the user adjusts the settings.

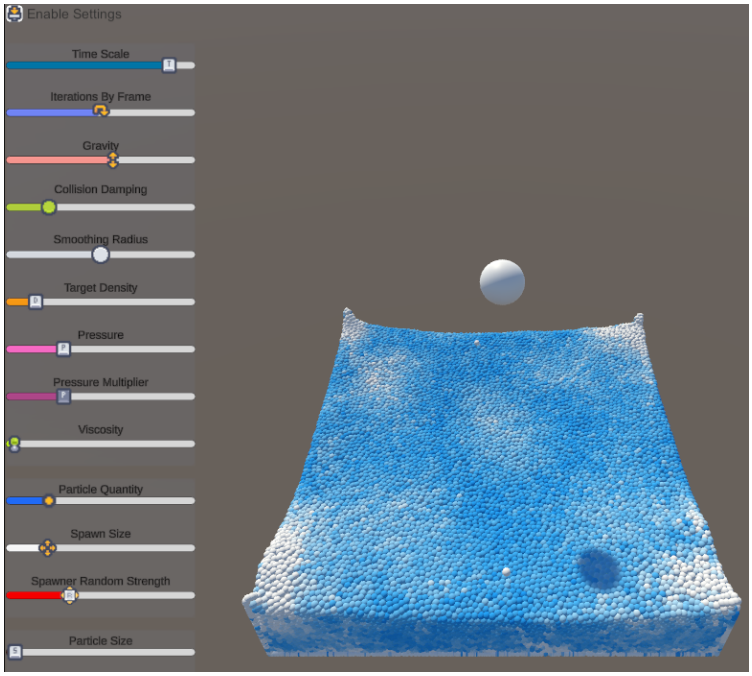


Figure 3.8: Mock-up of the Control Panel and Simulation View

By providing an intuitive and adjustable interface, this plugin ensures that users can customize the simulation to meet their specific needs. Whether for development or testing purposes, the UI facilitates easy manipulation of simulation parameters to achieve the desired outcomes.

CHAPTER 4

Work Development and Results

Contents

This chapter provides a comprehensive overview of the project's progression from its inception to its completion, encompassing deviations from the initial planning, errors encountered, and resultant outcomes.

4.1 Work Development

In this section, work carried out will be presented and the decisions made as the project progressed. This includes the research and design of the simulation, the mathematical operations involved, and the implementation as a Unity plugin.

4.2 Research and Simulation Design

To initiate the development of this project, extensive research was conducted, focusing on tutorials and documentation relevant to simulating interactions between objects to mimic liquid behavior. Leveraging existing materials and content, meticulous investigation led to the identification of several studies that yielded satisfactory results, forming the foundation upon which this project is built.

The primary aim is to simulate a collection of particles within a three-dimensional rectangular environment, where these particles collectively emulate a fluid with specified properties. Each particle is endowed with distinct physical attributes, including position, velocity, and mass, to accurately replicate fluid dynamics.

During the research, Smoothed Particle Hydrodynamics (SPH) was identified as the best implementation method for the desired outcome [8]. SPH is a computational method used for simulating fluid flows, where the fluid is represented by particles [2]. Each particle is influenced by forces such as pressure, viscosity [2], and surface tension, derived from the Navier-Stokes equations [6]. This method assigns an influence radius to each particle, within which it exerts or is subjected to forces from neighboring particles, aiding in calculating the density and pressure at a point by considering the contributions of nearby particles [6].

One significant advantage of using SPH over traditional grid-based methods is its capacity to handle complex, dynamic interfaces and free surfaces naturally [9]. Unlike grid-based methods, SPH does not require a fixed grid, simplifying the simulation of complex boundary interactions and fluid interfaces [4]. However, a notable challenge with SPH involves calibrating the interaction forces among particles. This process necessitates the fine-tuning of smoothing kernels used for density and pressure calculations to ensure realistic fluid behavior [6].

4.2.1 Fluid Density

To begin with SPH, the general idea is to determine the densities of each of the particles in the simulation environment. This is achieved by considering a sphere of influence around each particle [11]. The influence radius defines the extent to which each particle affects its neighbors. The interaction force within this radius is stronger when the neighboring particles are closer to the central particle and diminishes towards the periphery [11].

The density ρ_i at a particle i is calculated by summing the contributions of all neighboring particles within the influence radius. This is mathematically expressed using a smoothing kernel function W, which weights the contributions based on the distance between particles. The general formula for computing the density is:

$$\rho_i = \sum_j m_j W(r_{ij}, h)$$

where:

- ρ_i is the density at particle i,
- m_j is the mass of neighboring particle j,
- $W(r_{ij}, h)$ is the smoothing kernel function,
- r_{ij} is the distance between particles i and j,

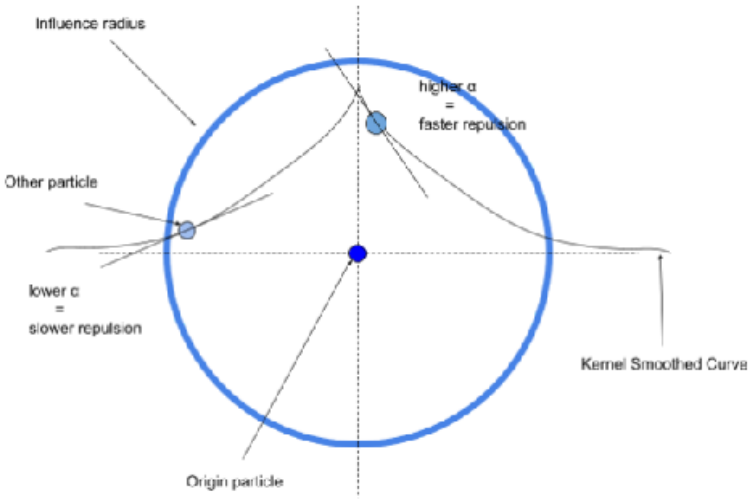


Figure 4.1: Illustration of particles within a sphere of influence for density calculation in SPH.

- h is the smoothing length, which defines the radius of influence.

The smoothing kernel W is a function that ensures particles closer to the center have a higher influence on the density calculation, while those at the edge of the influence radius have a reduced effect. Various forms of smoothing kernels can be used, such as the poly6 kernel, spiky kernel, and viscosity kernel, each having specific properties suitable for different aspects of the simulation [2, 6, 8].

Poly6 Kernel

The poly6 kernel [8] is commonly used for density estimation due to its smooth, even distribution of influence and its ability to avoid numerical instabilities. The poly6 kernel, as implemented in the code, is defined as:

$$W_{\text{poly6}}(r,h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3 & \text{if } 0 \leq r \leq h, \\ 0 & \text{if } r > h, \end{cases}$$

where r is the distance between particles and h is the smoothing length. This kernel ensures a smoothly varying density field.

Spiky Kernel

For pressure calculations, the spiky kernel is often used because it provides a strong repulsion force when particles come very close to each other, preventing clustering. The spiky kernel, as implemented in the code, is defined as:

$$W_{\text{spiky}}(r,h) = \begin{cases} \frac{15}{\pi h^6}(h-r)^3 & \text{if } 0 \le r \le h, \\ 0 & \text{if } r > h, \end{cases}$$

The spiky kernel has a sharp peak near $r = 0$, which ensures strong repulsive forces to maintain particle separation [2].

Density Calculation Process

The process of calculating the density at each particle i involves iterating over all neighboring particles j within the influence radius h and summing their contributions using the chosen kernel function. This iterative process ensures that the density field is accurately represented, reflecting the local particle distribution [6].

- **Step 1: Initialize Density** - Set the initial density ρ_i to zero for each particle.
- **Step 2: Sum Contributions** - For each neighboring particle j within the influence radius, add the weighted mass contribution $m_j W(r_{ij}, h)$ to the density ρ_i:

$$W(r,h) = \frac{15}{2\pi h^5}(h-r)^2 \quad \text{if } 0 \le r \le h.$$

- **Step 3: Normalize Density** - Normalize the density value if necessary, depending on the specific formulation and requirements of the simulation.

By accurately calculating the density of each particle, we lay the groundwork for other computations such as pressure forces, viscosity forces, and surface tension [9].

4.3 Fluid Pressure

Now that we have defined the interactions between particles to determine their densities, we need to ensure that particles strive to reach a defined target density. To achieve this, we implement pressures within the system, which will provide velocity and movement to our particles.

The pressure force on a particle is calculated using the following approach [6]. A straightforward way to introduce mobility to the particles is by calculating the pressure based on the current density of the particle, the target density, and a pressure multiplier. This approach is a simplified method and is not entirely accurate for liquids, as it more closely simulates the behavior of gases. However, its simplicity makes it useful for our purposes.

```
float PressureFromDensity(float density)
{
    return (density - targetDensity) * pressureMultiplier;
}
```

In the SPH method, pressure forces are derived from the density of particles. Each particle exerts a pressure based on its density relative to the target density. The pressure force on a particle is calculated using the following approach:

1. Density Calculation: As described previously, each particle's density is computed based on the contributions of neighboring particles using the smoothing kernel functions.

2. Pressure Calculation: The pressure of a particle is computed from its density. If the density is higher than the target density, the particle exerts an outward force to reduce the density, and vice versa. This pressure force is applied to neighboring particles to maintain an even distribution of particles and prevent clustering.

3. Pressure Force Formula:

$$\mathbf{f}_{\text{pressure}} = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(r_{ij}, h)$$

   where:

   - $\mathbf{f}_{\text{pressure}}$ is the pressure force.
   - m_j is the mass of neighboring particle j.
   - p_i and p_j are the pressures of particles i and j, respectively.
   - ρ_i and ρ_j are the densities of particles i and j, respectively.
   - $\nabla W(r_{ij}, h)$ is the gradient of the smoothing kernel function with respect to the distance between particles i and j.

4. Implementation of Pressure Forces: The pressure force on each particle is calculated by summing the contributions from all neighboring particles within the influence radius. The calculated pressure forces are then used to update the velocities and positions of the particles, ensuring realistic fluid movement.

4.4 The implementation on Unity

The implementation phase translates the theoretical framework and research outcomes into a functional Unity plugin that simulates fluid dynamics using Smoothed Particle Hydrodynamics (SPH). This section delves into the specifics of how the SPH method was integrated into Unity, detailing the technical steps and coding practices employed to achieve the desired fluid simulation.
The following subsections will discuss the key components of the implementation process, including the setup of the simulation environment, the integration of SPH algorithms, and the optimization techniques used to ensure efficient and realistic fluid behavior.

4.4.1 Simulation3D

The Simulation3D class is the core component responsible for managing and executing a fluid simulation using Smoothed Particle Hydrodynamics (SPH) in Unity. It integrates various parts of the system, including particle spawning, rendering, and the simulation logic itself, executed through compute shaders. Below is a detailed explanation of its components and functionality.

References The following references are used within the Simulation3D class:

- `computeShader`: The compute shader that performs the simulation calculations.
- `spawner`: The Spawner object responsible for generating the initial particle positions and velocities.
- `display`: The Display object responsible for rendering the particles.
- `floorDisplay`: A transform representing the floor in the simulation.

Compute Buffers The compute buffers store particle data, including positions, velocities, densities, and spatial indexing information:

- `positionBuffer`, `velocityBuffer`, `densityBuffer`, `predictedPositionsBuffer`, `spatialIndexes`, `spatialOffsets`

Kernel Indices These integers represent the indices of different kernels within the compute shader, each responsible for a specific step in the simulation process:

- `externalForcesKernel` (0)
- `spatialHashKernel` (1)
- `densityKernel` (2)
- `pressureKernel` (3)
- `viscosityKernel` (4)
- `updatePositionsKernel` (5)

Initialization Methods

- **InitializeComputeBuffers()**: Sets up the fixed time step, retrieves spawn data, initializes buffers, configures the compute shader, initializes GPU sorting [4], and initializes the display.
- **InitializeBuffers(int particleQuantity)**: Initializes the compute buffers with the required size based on the number of particles.

- **ConfigureComputeShader()**: Configures the compute shader with the initialized buffers and sets the number of particles.
- **SetBuffersInShader()**: Sets the various buffers in the compute shader.
- **InitializeGPUSort()**: Initializes the GPUSort instance and sets its buffers.

Update and Simulation Methods

- **Update()**: Manages the simulation with a variable time step, handles the pause state, adjusts the floor and simulation space display scale, and processes input.
- **SimulationStep()**: Dispatches each kernel in the compute shader to perform the simulation steps, including external forces, spatial hashing, density calculation, pressure forces, viscosity, and position updates.

Data Management Methods

- **SetInitialBufferData(Spawner.SpawnData spawnData)**: Sets the initial data in the compute buffers based on the spawn data.
- **SetShaderParameters(float deltaTime)**: Sets various parameters in the compute shader for each frame, including time step, gravity, collision damping, smoothing radius, target density, pressure multipliers, viscosity strength, and simulation bounds.

The Simulation3D class is the central component for managing and executing a 3D fluid simulation in Unity. It initializes the simulation, manages the execution of simulation steps, and handles the rendering of particles. The class uses compute shaders to perform physics calculations efficiently on the GPU, ensuring real-time performance even with a large number of particles. The integration with the Spawner and Display classes allows for flexible initialization and dynamic visualization of the simulation.

GPU Sort and BitonicMergeSort

The GPUSort class and the BitonicMergeSortEnhanced shader are designed to perform efficient data sorting on the GPU using the bitonic merge sort algorithm. This approach leverages the parallel processing capabilities of modern GPUs to handle large datasets efficiently. The main enhancements in BitonicMergeSort include better index management and optimized thread utilization.

GPUSort Class The GPUSort class orchestrates the sorting process using the bitonic merge sort algorithm. It initializes the necessary compute shaders, sets up the buffers, and manages the sorting operation.

SetBuffers Method The `SetBuffers` method initializes the buffers used by the compute shader:

- **indexBuffer**: Holds the data to be sorted.
- **offsetBuffer**: Used for spatial hashing after sorting.

Sort Method The `Sort` method performs the sorting using the bitonic merge sort algorithm:

- Calculates the number of stages and steps required.
- Sets the necessary parameters.
- Dispatches the compute shader for each sorting step.

SortAndCalculateOffsets Method The `SortAndCalculateOffsets` method combines sorting and offset calculation for spatial hashing:

- First calls the `Sort` method.
- Then dispatches the compute shader to run the `CalculateOffsets` kernel.

BitonicMergeSortEnhanced Shader The BitonicMergeSortEnhanced shader performs the sorting operations on the GPU. It consists of two kernels:

- **Sort**: Handles the sorting process.
- **CalculateOffsets**: Calculates offsets for spatial hashing.

Detailed Explanation of Bitonic Merge Sort

- **Overview**: Bitonic merge sort is a parallel sorting algorithm well-suited for GPU implementation. It sorts data by creating bitonic sequences, which are sequences that are monotonically increasing and then monotonically decreasing, or vice versa.
- **Stages and Steps**:
  - **Stages**: The number of stages is determined by the nearest power of 2 greater than or equal to the number of elements (`numEntries`). This is calculated using the base-2 logarithm.
  - **Steps**: Each stage involves a series of steps where elements are compared and swapped to create bitonic sequences.
- **Sorting Process**: In each step, the kernel computes the indices of elements to be compared and possibly swapped. The group width and height determine the range of elements to be compared in each step. Elements are swapped if they are out of order, progressively sorting the entire dataset.

- **Thread Utilization**: The `numthreads(128, 1, 1)` directive specifies that each thread group contains 128 threads. This high level of parallelism ensures that the sorting operation is performed efficiently on the GPU.
- **Handling Non-Power-of-Two Sizes**: The implementation can handle arrays whose sizes are not powers of 2 by using the `NextPowerOfTwo` function. This ensures that the sorting algorithm operates correctly regardless of the input size.

Example: Sorting Process Consider the following example to illustrate the sorting process:

- **Initial Data**: A dataset with indices [0, 1, 2, 3, 4].
- **Hash Calculation**: For a given point, the cell coordinate is calculated and hashed.
- **Bitonic Merge Sort**: The data is sorted through stages and steps, where elements are compared and swapped if necessary.
- **Index Management**: The `startIndices` array indicates the starting index of each hash key in the sorted list, enabling efficient lookup of points in the same grid.

The GPUSort class and BitonicMergeSortEnhanced shader implement an efficient GPU-based sorting algorithm using bitonic merge sort. This approach leverages the parallel processing capabilities of modern GPUs to handle large datasets quickly. Key features include enhanced index management, optimized thread utilization, and the ability to handle non-power-of-two array sizes. The combination of the `Sort` and `CalculateOffsets` kernels enables both efficient sorting and spatial hashing, making this implementation highly suitable for performance-critical applications [1].

4.4.2 DySAQUASIMs

The DySAQUASIMs.compute shader script is designed to simulate fluid dynamics in a 3D environment using Smoothed Particle Hydrodynamics (SPH). It includes several compute kernels that handle various stages of the simulation, from applying external forces to updating particle positions.

Constants and Buffers

- **Constants**: These define the number of threads, gravity, delta time, particle mass, smoothing radius, target density, pressure multipliers, viscosity strength, bounds size, and other simulation parameters.
- **Buffers**: These define read-write buffers for particle positions, predicted positions, velocities, densities, spatial indices, and spatial offsets.

numthreads Directive In DirectCompute (part of the DirectX API) and HLSL (High-Level Shading Language), `numthreads` is a key directive used to define the size of the thread groups for compute shaders. Compute shaders are used for general-purpose computing tasks, such as physics simulations, image processing, and in this case, fluid dynamics simulations.

- **numthreads**(X, Y, Z): This directive specifies the number of threads per thread group in the X, Y, and Z dimensions.
- **NumThreads**: Constant defining the number of threads in the X dimension.
- **1, 1**: These specify that there is only one thread in the Y and Z dimensions.

Purpose of numthreads The `numthreads` directive defines how work is divided among the GPU's cores. Each thread within a thread group executes the same compute shader code but operates on different data. This parallel execution allows for significant performance gains, especially for tasks that can be subdivided into smaller, independent operations.

How numthreads Works in This Script In DySAQUASIMs.compute, the `numthreads` directive is used in several kernels. Let's expand on how it operates and its specific purpose in this context [8].

Purpose in Compute Shaders

- **Parallel Processing**: The primary purpose of `numthreads` is to enable parallel processing. Each thread in a thread group processes a separate particle, allowing the simulation to handle many particles concurrently.
- **Efficient Resource Utilization**: By defining a large number of threads (64 in this case), the GPU can efficiently utilize its cores, reducing idle time and improving overall computation speed.
- **Scalability**: `numthreads` makes the simulation scalable. Whether the number of particles is large or small, the workload is distributed across multiple threads, ensuring efficient computation.

Detailed Breakdown

- **Kernel Execution**: When a compute shader is dispatched, it is divided into thread groups, each containing a number of threads as specified by `numthreads`.
- **Thread Group Example**: For `numthreads(64, 1, 1)`, each thread group contains 64 threads along the X dimension. If the total number of particles (*numParticles*) is 1024, the GPU will create 16 thread groups (1024 / 64 = 16).

- **Thread Dispatch**: The `SV_DispatchThreadID` system value provides a unique ID for each thread within the dispatch. This ID is used to determine which portion of the data each thread will process.

Compute Kernels The DySAQUASIMs.compute script consists of multiple compute kernels, each responsible for a specific aspect of the simulation process [5]:

1. **ExternalForces**:
   - Applies external forces such as gravity and object interactions to the velocities of particles.
   - Predicts the next positions of particles based on their velocities.
2. **UpdateSpatialHash**:
   - Updates the spatial hash for each particle to facilitate efficient neighbor searches.
   - Converts particle positions to cell coordinates, hashes these coordinates, and generates keys.
3. **CalculateDensities**:
   - Calculates the density and near density for each particle based on its neighbors within the smoothing radius.
   - Uses the spatial hash to quickly find neighboring particles.
4. **CalculatePressureForce**:
   - Calculates the pressure forces acting on each particle based on the density and near density.
   - Uses the spatial hash to find neighbors and compute pressure forces, updating particle velocities accordingly.
5. **CalculateViscosity**:
   - Calculates the viscosity forces between particles to simulate fluid viscosity.
   - Uses the spatial hash to find neighbors and compute viscosity forces, updating particle velocities.
6. **UpdatePositions**:
   - Updates particle positions based on their velocities.
   - Resolves collisions with the bounding box to ensure particles stay within bounds.

The DySAQUASIMs.compute script offers a comprehensive solution for simulating fluid dynamics in a 3D environment. By employing SPH techniques, it efficiently calculates forces, densities, and updates particle positions. The script is organized into multiple compute kernels, each handling a specific aspect of the simulation process. This modular approach ensures the simulation is both efficient and scalable. The specific use of `numthreads(64, 1, 1)` ensures that each thread group processes 64 particles concurrently, making full use of the GPU's capabilities.

Fluid Mathematics

This subsection elaborates on the mathematical functions implemented for fluid simulation in 3D space within Unity, specifically focusing on the computation of various kernel functions and their derivatives.
The following functions are defined to support the DySAQUASIMs.compute code.

- SmoothingKernelPoly6: Calculates the Poly6 kernel value, which is used for smoothing particle interactions over a defined radius. This function returns a value based on the distance between particles (dst) and the smoothing radius (radius).
- SpikyKernelPow3: Computes the Spiky kernel to the power of 3, emphasizing the effect of closer particles. This is used primarily in pressure force calculations.
- SpikyKernelPow2: Similar to SpikyKernelPow3 but to the power of 2, providing a different scale of influence.
- DerivativeSpikyPow3: Calculates the derivative of the Spiky kernel to the power of 3. Used for determining the rate of change in the kernel, which is necessary for force calculations in SPH simulations.
- DerivativeSpikyPow2: Computes the derivative of the Spiky kernel to the power of 2, used similarly in force calculations.
- DensityDerivative: Returns the derivative value for the density calculation using the derivative of the Spiky kernel to the power of 2.
- NearDensityDerivative: Provides the derivative value for the near density calculation using the derivative of the Spiky kernel to the power of 3.

These functions are directly related to the computations discussed in Section 4.2.1, where we detailed the process of calculating particle densities and pressures using SPH.

3D Spatial Hash

The SpatialHash3d.hlsl script is used to efficiently manage and query the positions of particles within a 3D grid. The main goal is to group particles into grid cells and enable fast retrieval of particles that reside within the same or neighboring cells. This technique significantly speeds up the process of finding nearby particles that might influence each other.

Hashing and Key Generation For each particle:

1. Convert its position to a cell coordinate using `GetCell3D`.
2. Hash the cell coordinate using `HashCell3D` to obtain a unique hash value.
3. Generate a cell key using `KeyFromHash` to map the hash value to an index in the lookup table.

Consider there are 5 particles in the scene with indices [0, 1, 2, 3, 4]. For particle 0, the cell coordinate is (2, 1, 5) based on its position and the smoothing radius. To obtain the cell hash, multiply each coordinate by a prime number and sum them: $2 * 1301 + 1 * 5449 + 5 * 14983 = 82966$. The cell key is $82966\%5 = 2$.

Building the Lookup Table The process involves:

1. Computing the hash values for all particles and storing them.
2. Sorting these hash values to group particles in the same grid cell together.
3. Generating an array (`startIndices`) that indicates the starting index of each hash key in the sorted list. This array enables quick access to all particles within a specific cell.

For the entire scene:

- `spatialLookup` array: [2, 2, 8, 1, 3]
- `Sorted pointsHashKey`: [1, 2, 2, 3, 8]
- `startIndices` array: [0, 1, 1, 3, 4]

The `startIndices` array helps to look up all points in the same grid. For example, `startIndices[0] = 2` means all points with hash key 0 start from lookup array index 1, so points [1, 2] are in the same grid with hash key 2.

Querying Nearby Particles To find nearby particles for a given sample point:

1. Determine the sample point's cell coordinate and hash key.
2. Use the `offsets3D` array to iterate through the 27 neighboring cells.
3. For each neighboring cell, compute its hash key and use the `startIndices` array to find all particles within that cell.
4. Check if each particle within these cells lies within the smoothing radius of the sample point. If it does, update the properties of these particles accordingly.

This structured approach ensures efficient organization and retrieval of particle data, optimizing the performance of the fluid simulation.

4.4.3 Display

The Display class in Unity is responsible for rendering particles as spheres in a 3D space using a GPU-based approach. The class interacts with the Simulation3D class to retrieve particle data and utilizes a custom shader for rendering.

Overview of Display Class The Display class handles several key tasks to manage the rendering process in real-time:

- **Initialization**: Sets up materials, generates the sphere mesh, and prepares the necessary buffers for drawing.
- **Material Properties**: Updates material properties dynamically to reflect changes in particle states.
- **Gradient Textures**: Manages gradient textures to enhance the visual representation of particles given a speed.
- **Efficient Rendering**: Ensures efficient rendering through instanced indirect drawing, leveraging the GPU for high performance.

Sphere Mesh Generator

The SphereGenerator class is responsible for creating a sphere mesh based on a given resolution. This mesh is used to represent particles visually as spheres in the 3D space.

Functionality of SphereGenerator Class The SphereGenerator class performs the following tasks:

- **Vertex Generation**: Generates vertices for the sphere mesh based on the specified resolution.
- **Triangle Creation**: Creates triangles from the generated vertices to form the surface of the sphere.
- **Mesh Construction**: Constructs the complete 3D sphere mesh, ready to be used for rendering particles.

Usage in Display Class The sphere mesh generated by the SphereGenerator class is utilized by the Display class to render particles as spheres. This involves:

- **Mesh Integration**: Integrating the generated sphere mesh into the rendering pipeline of the Display class.
- **Buffer Setup**: Setting up vertex and index buffers for efficient rendering on the GPU.
- **Real-time Rendering**: Using instanced indirect drawing to render a large number of particles dynamically, ensuring high performance.

Summary The Display class [3], in conjunction with the SphereGenerator class, provides a robust solution for rendering particles as spheres in a 3D space. By leveraging GPU-based techniques and efficient mesh generation, this approach ensures high-performance visualization of fluid simulations. The use of instanced indirect drawing further enhances rendering efficiency, allowing for real-time updates and dynamic visual representation of particle data.

4.4.4 Spawner

The Spawner class generates a grid of particles with specified initial positions and velocities. It includes functionality for adding random jitter to particle positions and visualizing the spawn bounds in the Unity editor. This class is essential for initializing the particle system in a controlled, customizable manner.

4.5 Troubles and Solutions

4.5.1 Excessive Triangles in Default Unity Spheres

Problem: The default Unity sphere mesh contains a high number of triangles, which can be computationally expensive and negatively impact the performance of the simulation, especially when rendering a large number of particles.

Solution: Replace the default Unity sphere mesh with a custom sphere mesh generated at a desired resolution using the SphereGenerator class. This approach allows for control over the number of vertices and triangles, optimizing performance.

4.5.2 Non-Functional Particle Interactions

Problem: Interactions between particles, such as pressure and viscosity [2] forces, were not functioning correctly, leading to unrealistic simulation behavior.

Solution: Ensure that the compute shader kernels responsible for calculating particle interactions are correctly implemented and that the buffers are properly set. Verify the sequence of kernel dispatches to make sure each step in the simulation is executed in the correct order.

4.5.3 Performance Issues with Increased Particle Count

Problem: Increasing the number of particles to enhance the simulation may lead to performance issues due to the higher computational load, particularly when using default Unity spheres.

Solution: Optimize the simulation by using lower-resolution meshes for individual particles and leveraging the GPU for parallel processing. Ensure efficient memory management by using structured buffers and minimizing unnecessary data transfers.

4.5.4 ComputeBuffer and ComputeShader Management

Problem: Issues with ComputeBuffer and ComputeShader usage can lead to runtime errors, incorrect data processing, and simulation instability.

Solution: Verify that ComputeBuffer objects are correctly initialized and released, and ensure that data is correctly transferred between the CPU and GPU. Confirm that the compute shader code correctly handles buffer operations.

4.5.5 Challenges with Unity Metaballs

Problem: Implementing metaballs in the simulation significantly reduced performance, making it challenging to achieve real-time interactivity and visualization, particularly with a high particle count.

Solution: The complexity and computational expense of generating and rendering metaballs are substantially higher than those of simple sphere rendering. Metaballs require evaluating a scalar field for each particle and extracting an isosurface, typically using algorithms like marching cubes, which are computationally intensive.

Given the real-time performance constraints of our fluid simulation, we opted to use simpler spheres for particle visualization. This approach ensures that the simulation remains interactive and performant, even with a large number of particles. Metaballs, while visually superior for certain applications, were deemed impractical for our specific performance requirements. By focusing on optimizing sphere rendering, we achieve a balance between visual fidelity and real-time performance, which is crucial for the intended use case of our simulation.

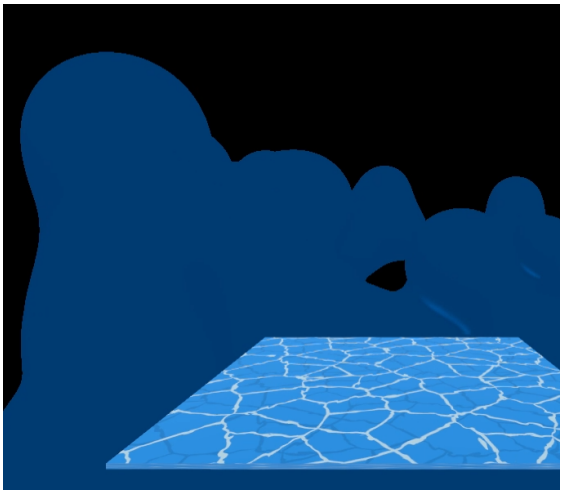


Figure 4.2: A try with metaballs in Simulation.

4.6 Results

In this section, the results of the project are detailed, aligning with the initial objectives. The focus is on the milestones achieved without delving excessively into technical details. Additionally, the potential applications and uses of the developed work, as well as any releases made or planned, are discussed.

After numerous iterations involving code refinements, computeBuffers, computeShaders, and SPH (Smoothed Particle Hydrodynamics) interactions in the simulation, the system simulations reached a highly satisfactory level. The primary objective of achieving realistic fluid behavior was accomplished by adjusting various parameters, leading to successful outcomes in multiple aspects of the simulation.

The simulation effectively replicated the behavior of different types of fluids through parameter adjustments. The interactions between particles, walls, and various scenarios were accurately modeled, resulting in realistic simulations of forces, pressures, and densities. These realistic interactions are a significant success of the project, showcasing the robustness of the SPH method in simulating fluid dynamics.

Although the visual outcome of scenarios filled with spheres may not appear ideal, the importance lies in the accurate interactions among these spheres [9]. Achieving realistic forces and behaviors was a major accomplishment, demonstrating the system's capability to simulate real-world dynamics effectively. Below are some images showcasing the various results obtained:

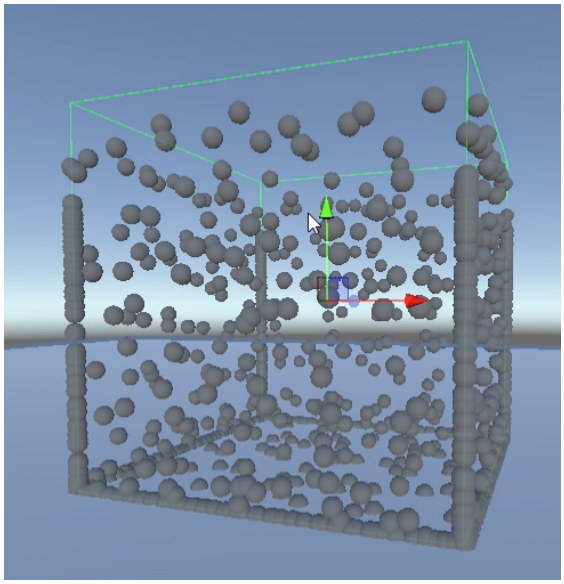


Figure 4.3: First looks of the simulation.

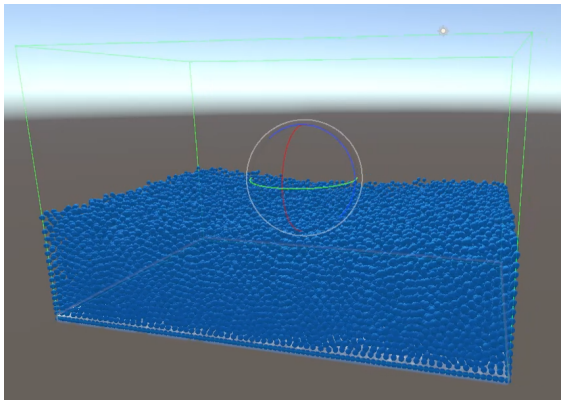


Figure 4.4: Another version of the simulation.

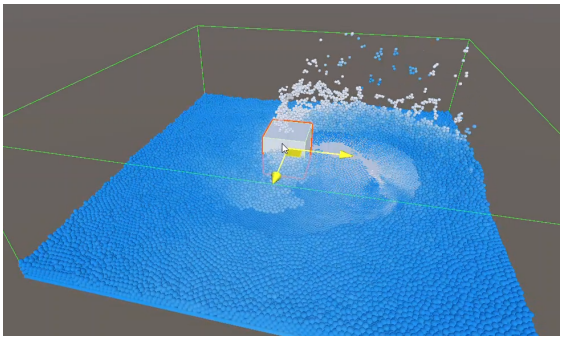


Figure 4.5: Collisions with objects

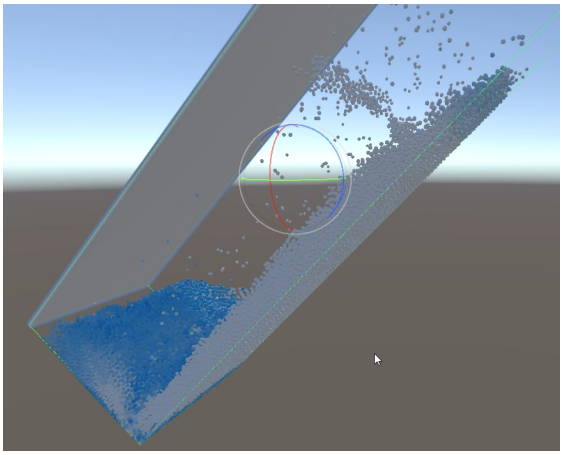


Figure 4.6: Almost final result.

The primary goal of this project was to develop a user-friendly interface for Unity, enabling users to interact with and simulate fluid dynamics and particle behavior using the SPH method. This application allows users to set up scenarios, adjust physical properties, and visualize the simulation results in real-time. By making complex fluid simulations accessible without requiring in-depth knowledge of the underlying mathematical models, the application broadens its utility in education, research, and entertainment. Future releases will include software repositories and web pages to facilitate broader access and usability.

In conclusion, the project successfully achieved its objectives, resulting in a robust simulation system capable of realistic fluid dynamics. The development of a user-friendly interface for Unity further enhances the application's accessibility and potential uses. Future improvements and releases will continue to build on this foundation, expanding the application's reach and utility.

Chapter 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

This project has been incredibly challenging, but from the beginning, I was determined to complete it and maximize its potential. Studying so many aspects for a project that has reached its current state fills me with enthusiasm and motivation to continue working in this field and advancing in my career as a video game designer and developer. This work demonstrates the effort one can achieve after countless hours of dedication, which may not seem apparent at first glance, but once you look behind the scenes, you see the true extent of the work involved.

It has been a tough journey with many challenges along the way, including significant changes in my social and professional life. I have learned an immense amount about Unity and the behavior of fluids in real life, a subject I have always been eager to explore. This has been one of the best experiences I have had in designing a project of this magnitude and caliber.

5.2 Future work

Regarding the future of this project, I do not plan to continue its development. I am satisfied with its current state, although there could be improvements in particle appearance or scenario visuals. However, the workload has been so intense that I feel a need to move on to new horizons. This project might not be included in my future endeavors, as my current goal is to develop a video game and advance my professional career.

Bibliography

[1] Carnegie Mellon University. Particle-based fluid simulation. http://15462.courses.cs.cmu.edu/fall2018/lecture/pdes/slide_024, 2018.

[2] S. Clavet, P. Beaudoin, and P. Poulin. Particle-based viscoelastic fluid simulation. In K. Anjyo and P. Faloutsos, editors, *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 219–228, 2005.

[3] Computer Animation - RWTH Aachen. Weighted laplacian smoothing for surface reconstruction of particle-based fluids. https://www.youtube.com/watch?v=2bYvaUXlBQs, September 21 2023. [Video].

[4] S. G. Green. Particle simulation using cuda. https://web.archive.org/web/20140725014123/https://docs.nvidia.com/cuda/samples/5_Simulations/particles/doc/particles.pdf, 2012.

[5] JetsonHacks. Cuda particle simulation on jetson tk1. https://youtu.be/9teSvCL0NX0?si=SXeDUveOoTiofKvX, July 13 2014. [Video].

[6] D. Koschier, J. Bender, B. Solenthaler, and M. Teschner. Smoothed particle hydrodynamics techniques for the physics-based simulation of fluids and solids. In W. Jakob and E. Puppo, editors, *EUROGRAPHICS 2019 Tutorial*, 2019.

[7] Sebastian Lague. Coding adventure: Simulating fluids. https://www.youtube.com/watch?v=rSKMYc1CQHE, October 08 2023. [Video].

[8] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In D. Breen and M. Lin, editors, *Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 154–159, 2003.

[9] M. S. Shadloo, D. Le Touzé, and G. Oger. Smoothed particle hydrodynamics method for fluid flows, towards industrial applications-motivations, current state, and challenges. *Computers & Fluids*, 136:11–34, 2016.

[10] Talent.com. Salario programador junior en españa. https://es.talent.com/salary?job=programador+junior, 2023. Accessed: 2023-05-15.

[11] Wikipedia contributors. Smoothed-particle hydrodynamics. https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics. [Accessed on date].

APPENDIX A

Source Code

This chapter presents key parts of the source code developed for the fluid simulation project using Smoothed Particle Hydrodynamics (SPH) in Unity. Only critical code snippets are included and discussed to highlight important aspects of the implementation. The full source code is provided in the appendix for reference.

Here is the link to my final degree project: DySAQUASIMs on GitHub.

A.1 Simulation3D Class

The `Simulation3D` class is the core component responsible for managing and executing the fluid simulation using SPH in Unity.

A.1.1 Initialization and Buffer Setup

The initialization of compute buffers is crucial for setting up the simulation:

```
void InitializeComputeBuffers() {
    SetFixedTimeStep();
    RetrieveSpawnData();
    InitializeBuffers(spawnData.points.Length);
    SetInitialBufferData(spawnData);
    ConfigureComputeShader();
    InitializeGPUSort();
    display.Init(this);
}

```

```
void InitializeBuffers(int particleQuantity) {
    positionBuffer =
    ComputeHelper.CreateStructuredBuffer<float3>(particleQuantity);
    predictedPositionsBuffer =
    ComputeHelper.CreateStructuredBuffer<float3>(particleQuantity);
    velocityBuffer =
    ComputeHelper.CreateStructuredBuffer<float3>(particleQuantity);
    densityBuffer =
    ComputeHelper.CreateStructuredBuffer<float2>(particleQuantity);
    spatialIndexes =
    ComputeHelper.CreateStructuredBuffer<uint3>(particleQuantity);
    spatialOffsets =
    ComputeHelper.CreateStructuredBuffer<uint>(particleQuantity);
}
```

These methods set up the fixed time step, retrieve initial particle data, and initialize the necessary buffers.

A.1.2 Compute Shader Configuration

Configuring the compute shader involves setting the buffers and shader parameters:

```
private void ConfigureComputeShader() {
    SetBuffersInShader();
    computeShader.SetInt("numParticles", positionBuffer.count);
}

private void SetBuffersInShader() {
    ComputeHelper.SetBuffer(computeShader, positionBuffer,
    "Positions", externalForcesKernel, updatePositionsKernel);
    ComputeHelper.SetBuffer(computeShader, predictedPositionsBuffer,
    "PredictedPositions", externalForcesKernel, spatialHashKernel,
    densityKernel, pressureKernel,
    viscosityKernel, updatePositionsKernel);
    ComputeHelper.SetBuffer(computeShader, spatialIndexes,
    "SpatialIndices", spatialHashKernel, densityKernel, pressureKernel,
    viscosityKernel);
    ComputeHelper.SetBuffer(computeShader, spatialOffsets,
    "SpatialOffsets", spatialHashKernel, densityKernel, pressureKernel,
    viscosityKernel);
    ComputeHelper.SetBuffer(computeShader, densityBuffer,
    "Densities", densityKernel,
    pressureKernel, viscosityKernel);
    ComputeHelper.SetBuffer(computeShader, velocityBuffer,
```

```
    "Velocities", externalForcesKernel, pressureKernel,
    viscosityKernel,
    updatePositionsKernel);
}
```

These methods ensure the compute shader is correctly configured with all necessary buffers and parameters.

A.1.3 Simulation Execution

The main simulation steps are executed in the `SimulationStep` method:

```
void SimulationStep() {
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: externalForcesKernel);
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: spatialHashKernel);
    gpuSort.SortAndCalculateOffsets();
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: densityKernel);
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: pressureKernel);
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: viscosityKernel);
    ComputeHelper.Dispatch(computeShader, positionBuffer.count,
    kernelIndex: updatePositionsKernel);
}
```

This method dispatches each kernel in the compute shader to perform the necessary simulation steps.

A.2 Compute Shaders

The compute shaders handle the core calculations for the SPH simulation. Key functions and kernels include:

A.2.1 Kernel: External Forces

This kernel applies external forces such as gravity and predicts the next positions of particles:

```
[numthreads(64,1,1)]
void ExternalForces (uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

```

```
    // Apply gravity
    Velocities[id.x] += float3(0, gravity, 0) * deltaTime;

    // Predict next positions
    PredictedPositions[id.x] = Positions[id.x] + Velocities[id.x] * deltaTime;
}
```

A.2.2 Kernel: Update Spatial Hash

This kernel updates the spatial hash for each particle to facilitate efficient neighbor searches:

```
[numthreads(64,1,1)]
void UpdateSpatialHash (uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

    // Reset offsets and update index buffer
    SpatialOffsets[id.x] = numParticles;
    uint index = id.x;
    int3 cell = GetCell3D(PredictedPositions[index], smoothingRadius);
    uint hash = HashCell3D(cell);
    uint key = KeyFromHash(hash, numParticles);
    SpatialIndices[id.x] = uint3(index, hash, key);
}
```

A.2.3 Kernel: Calculate Densities

This kernel calculates the density and near density for each particle based on its neighbors:

```
[numthreads(64,1,1)]
void CalculateDensities (uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

    float3 pos = PredictedPositions[id.x];
    int3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;
    float density = 0;
    float nearDensity = 0;

    // Neighbor search and density calculation
    for (int i = 0; i < 27; i++) {
        uint hash = HashCell3D(originCell + offsets3D[i]);
```

```
        uint key = KeyFromHash(hash, numParticles);
        uint currIndex = SpatialOffsets[key];

        while (currIndex < numParticles) {
            uint3 indexData = SpatialIndices[currIndex];
            currIndex++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;

            uint neighbourIndex = indexData[0];
            float3 neighbourPos = PredictedPositions[neighbourIndex];
            float3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour, offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;

            float dst = sqrt(sqrDstToNeighbour);
            density += DensityKernel(dst, smoothingRadius);
            nearDensity += NearDensityKernel(dst, smoothingRadius);
        }
    }

    Densities[id.x] = float2(density, nearDensity);
}
```

A.2.4 Kernel: Calculate Pressure Forces

This kernel calculates the pressure forces acting on each particle:

```
[numthreads(64,1,1)]
void CalculatePressureForce (uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

    // Calculate pressure and pressure force
    float density = Densities[id.x][0];
    float densityNear = Densities[id.x][1];
    float pressure = PressureFromDensity(density);
    float nearPressure = NearPressureFromDensity(densityNear);
    float3 pressureForce = 0;

    float3 pos = PredictedPositions[id.x];
    int3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;

```

```
    // Neighbor search and pressure force calculation
    for (int i = 0; i < 27; i++) {
        uint hash = HashCell3D(originCell + offsets3D[i]);
        uint key = KeyFromHash(hash, numParticles);
        uint currIndex = SpatialOffsets[key];

        while (currIndex < numParticles) {
            uint3 indexData = SpatialIndices[currIndex];
            currIndex++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;

            uint neighbourIndex = indexData[0];
            if (neighbourIndex == id.x) continue;

            float3 neighbourPos = PredictedPositions[neighbourIndex];
            float3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour, offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;

            float densityNeighbour = Densities[neighbourIndex][0];
            float nearDensityNeighbour = Densities[neighbourIndex][1];
            float neighbourPressure = PressureFromDensity(densityNeighbour);
            float neighbourPressureNear =
            NearPressureFromDensity(nearDensityNeighbour);

            float sharedPressure=(pressure + neighbourPressure) / 2;
            float sharedNearPressure=(nearPressure + neighbourPressureNear) / 2;

            float dst = sqrt(sqrDstToNeighbour);
            float3 dir = dst > 0 ? offsetToNeighbour / dst : float3(0, 1, 0);

            pressureForce += dir * DensityDerivative(dst,
            smoothingRadius) * sharedPressure / densityNeighbour;
            pressureForce += dir * NearDensityDerivative(dst,
            smoothingRadius) * sharedNearPressure / nearDensityNeighbour;
        }
    }

    float3 acceleration = pressureForce / density;
    Velocities[id.x] += acceleration * deltaTime;
}
```

A.2.5 Kernel: Calculate Viscosity

This kernel calculates the viscosity forces between particles:

```
[numthreads(64,1,1)]
void CalculateViscosity (uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

    float3 pos = PredictedPositions[id.x];
    int3 originCell = GetCell3D(pos, smoothingRadius);
    float sqrRadius = smoothingRadius * smoothingRadius;

    float3 viscosityForce = 0;
    float3 velocity = Velocities[id.x];

    // Neighbor search and viscosity calculation
    for (int i = 0; i < 27; i++) {
        uint hash = HashCell3D(originCell + offsets3D[i]);
        uint key = KeyFromHash(hash, numParticles);
        uint currIndex = SpatialOffsets[key];

        while (currIndex < numParticles) {
            uint3 indexData = SpatialIndices[currIndex];
            currIndex++;
            if (indexData[2] != key) break;
            if (indexData[1] != hash) continue;

            uint neighbourIndex = indexData[0];
            if (neighbourIndex == id.x) continue;

            float3 neighbourPos = PredictedPositions[neighbourIndex];
            float3 offsetToNeighbour = neighbourPos - pos;
            float sqrDstToNeighbour = dot(offsetToNeighbour,
            offsetToNeighbour);
            if (sqrDstToNeighbour > sqrRadius) continue;

            float dst = sqrt(sqrDstToNeighbour);
            float3 neighbourVelocity = Velocities[neighbourIndex];
            viscosityForce += (neighbourVelocity - velocity) *

            SmoothingKernelPoly6(dst, smoothingRadius);
        }
    }
    Velocities[id.x] += viscosityForce * viscosityStrength * deltaTime;
```

```
}
```

A.2.6 Kernel: Update Positions

This kernel updates the positions of particles and resolves collisions:

```
[numthreads(64, 1, 1)]
void UpdatePositions(uint3 id : SV_DispatchThreadID) {
    if (id.x >= numParticles) return;

    Positions[id.x] += Velocities[id.x] * deltaTime;
    ResolveCollisions(id.x);
}
```

A.3 Key Helper Functions

A.3.1 Density and Pressure Kernels

These functions define the SPH kernel functions and their derivatives:

```
float SmoothingKernelPoly6(float dst, float radius) {
    if (dst < radius) {
        float scale = 315 / (64 * PI * pow(abs(radius), 9));
        float v = radius * radius - dst * dst;
        return v * v * v * scale;
    }
    return 0;
}

float SpikyKernelPow3(float dst, float radius) {
    if (dst < radius) {
        float scale = 15 / (PI * pow(radius, 6));
        float v = radius - dst;
        return v * v * v * scale;
    }
    return 0;
}

float SpikyKernelPow2(float dst, float radius) {
    if (dst < radius) {
        float scale = 15 / (2 * PI * pow(radius, 5));
        float v = radius - dst;
        return v * v * scale;
```

```
    }
    return 0;
}

float DerivativeSpikyPow3(float dst, float radius) {
    if (dst <= radius) {
        float scale = 45 / (pow(radius, 6) * PI);
        float v = radius - dst;
        return -v * v * scale;
    }
    return 0;
}

float DerivativeSpikyPow2(float dst, float radius) {
    if (dst <= radius) {
        float scale = 15 / (pow(radius, 5) * PI);
        float v = radius - dst;
        return -v * scale;
    }
    return 0;
}
```

A.3.2 Spatial Hashing Functions

These functions handle the conversion of positions to grid cells and hashing:

```
int3 GetCell3D(float3 position, float radius) {
    return (int3)floor(position / radius);
}

uint HashCell3D(int3 cell) {
    cell = (uint3)cell;
    return (cell.x * hashK1) + (cell.y * hashK2) + (cell.z * hashK3);
}

uint KeyFromHash(uint hash, uint tableSize) {
    return hash % tableSize;
}
```

A.4 Summary

This chapter has presented key parts of the source code for the fluid simulation project. The Simulation3D class and associated compute shaders form the backbone of the simulation, leveraging the GPU for efficient and real-time particle physics calculations. The full source code is available in the appendix for further reference and study.