Master Informatique

Projet IMA
Final Report
Real Time Fluid Simulations

Supervisor: Daniel Racoceanu
Student: Basci Onur
N°: 21309649

Contents

1 Context

Over the past four decades, fluid simulation has captivated researchers due to its intricate and dynamic nature. These simulations find frequent application in the realm of filmmaking and animation, as showcased in works such as "The Day After Tomorrow" and "The Good Dinosaur." Leveraging high-performance computer systems and offline simulation techniques, these productions can generate highly detailed simulations without time constraints, unlike real-time simulations. Real-time simulations face two primary challenges: time and resources. To meet industry standards for video games, real-time simulations must be calculated swiftly to achieve playback rates of at least 30 or 60 frames per second. This necessitates all computations for each time period of the simulation to be completed within 0.03 or 0.016 seconds. Additionally, real-time simulations for video games must contend with the limitations of consumer-grade hardware, as they are intended for use on personal computers.

Advancements in computer graphics technology, particularly in GPU (Graphics Processing Unit) and parallel computing, are narrowing the gap between real-time and offline simulations. Researchers are delving into innovative methods to optimize these simulations, balancing the constraints imposed while retaining the complex structure of fluids.

Throughout this project, we aim to delve into various methods put forth by researchers and implement one of the latest and most efficient techniques available. Our focus will be on utilizing Unity, a game engine renowned for offering numerous optimization tools, including compute shaders for GPU utilization, the Job System, and the Burst compiler. Additionally, the simulation should be highly parameterizable since we are working on fluids and not only liquids. In fact, a fluid can be water and gas; therefore, we aim to have a simulation model that is capable of simulating phase transitions.

2 Introduction

2.1 Navier-Stokes Equations

A large number of water simulations rely on the Navier-Stokes Equations, which are a set of partial differential equations describing the motion of viscous fluid substances. They are typically written as:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

Where $\vec{u}$ represents the velocity field, p represents the pressure field, $\vec{g}$ denotes external forces (such as acceleration due to gravity), ν represents the kinematic viscosity, and ρ represents the density.

The first equation is essentially derived from Newton's second law, $F = ma$. Here, $\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u}$ represents the material derivative, corresponding to acceleration. The right-hand side of the equation corresponds to F, where ∇p represents the force resulting from pressure; high-pressure regions push on lower-pressure regions. The term $\nu \nabla^2 \vec{u}$ corresponds to the force resulting from viscosity, interpreted as a force attempting to align the particle's velocity with the average velocity of nearby particles. Finally, we can define mass as $m = \frac{\rho}{V}$, thus we get the Navier-Stokes equations with some simplifications.

The second equation pertains to incompressibility. During these simulations, we assume water is incompressible, which is reasonable since water in real life is very close to being incompressible. Unfortunately, we cannot make the same assumption for gas, as it has a much lower density. For water, the density is roughly $1000\,\mathrm{kg/m^3}$, and for air, it's roughly $1.3\,\mathrm{kg/m^3}$, a ratio of about $700 : 1$. [1]

The main idea behind most physics-based water simulations is to solve these equations for the velocity and pressure fields or simulate the effects resulting from these equations. But before we discuss methods of simulation, let's delve into the states of matter, which are crucial for simulating different states in our simulation.

2.2 States of Matters

The primary states of matter are commonly classified into four categories: solid, liquid, gas, and plasma. Each state exhibits unique properties and behaviors, which are governed by the interactions between particles and the environment.

- In a solid, constituent particles (ions, atoms, or molecules) are closely packed together. The forces between particles are so strong that the particles cannot move freely but can only vibrate. As a result, a solid has a stable, definite shape, and a definite volume. Solids can only change their shape by an outside force, such as when broken or cut.
- A liquid is a nearly incompressible fluid that conforms to the shape of its container but retains a (nearly) constant volume. The volume is definite if the temperature and pressure are constant. When a solid is heated above its melting point, it becomes liquid, given that the pressure is higher than the triple point of the substance.

- A gas is a compressible fluid. Not only will a gas conform to the shape of its container, but it will also expand to fill the container. In a gas, the molecules have enough kinetic energy so that the effect of intermolecular forces is small (or zero for an ideal gas), and the typical distance between neighboring molecules is much greater than the molecular size. A gas has no definite shape or volume but occupies the entire container in which it is confined. A liquid may be converted to a gas by heating at constant pressure to the boiling point, or else by reducing the pressure at constant temperature. At temperatures below its critical temperature, a gas is also called a vapor and can be liquefied by compression alone without cooling.

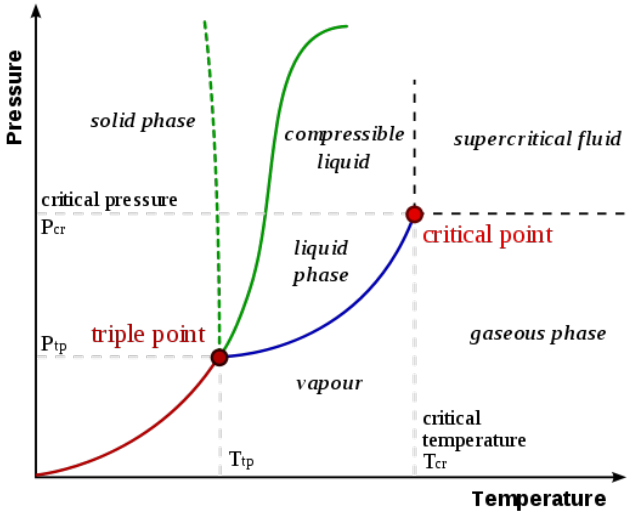

Figure 1: Phase Diagram

So, as evident from the phase diagram, phase transitions primarily depend on two variables: temperature and pressure. Consequently, our simulation should be parametrizable with these two variables.

3 State of the Art

3.1 Procedural Water

A procedural approach directly animates a physical effect instead of simulating its underlying cause. For instance, a common technique to generate the surface of bodies of water like lakes or oceans procedurally involves overlaying sine waves of varying amplitudes and directions. [3] Procedural animation techniques offer limitless creativity, allowing for the creation of desired visual effects without constraints. Controllability is a key advantage of procedural animation, particularly valuable in gaming. However, a drawback of procedurally simulating water is the challenge of accurately depicting its interaction with boundaries or immersed objects.

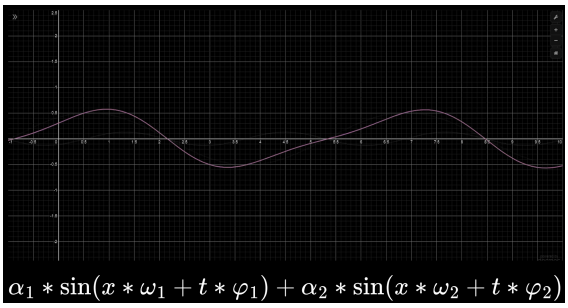

Figure 2: An example of sum of sinus with different amplitude and frequencies

Figure 3: A result obtained from the article A simple Model of ocean waves

Although the sum of sinus method is highly efficient and can simulate large amounts of water, such as ocean waves, it has some drawbacks. Firstly, it only represents the surface of the water and does not account for the volume beneath. Secondly, it does not interact with its environment, such as objects or boundaries within the water. Finally, it is only applicable for liquid simulation and cannot accurately represent other states of matter or their interactions.

3.2 Height Field Approximation

Another method that attempts to simulate the surface of water is the Height Field Approximation. If our interest lies solely in simulating the water surface, simulating the entire three-dimensional body of water would be unnecessary. The concept of the Height Field approach involves representing the water as a set of height blocks. This can be visualized as a 2D array where the index represents positions and the corresponding value represents the height of the block.

To simulate the movement of these blocks, we utilize Archimedes' principle. According to this principle, the upward buoyant force exerted on a body immersed in a fluid, whether fully or partially, is equal to the weight of the fluid displaced by the body. By applying this principle, we can calculate the acceleration of each block based on the difference in height between the block and its neighbors. [1]

This method is pretty simple to simulate and it is quite fast. We can also simulate the interaction with the objects still with the Archimedes's principal. But due to its structure it can not provide overturning waves and splashes.

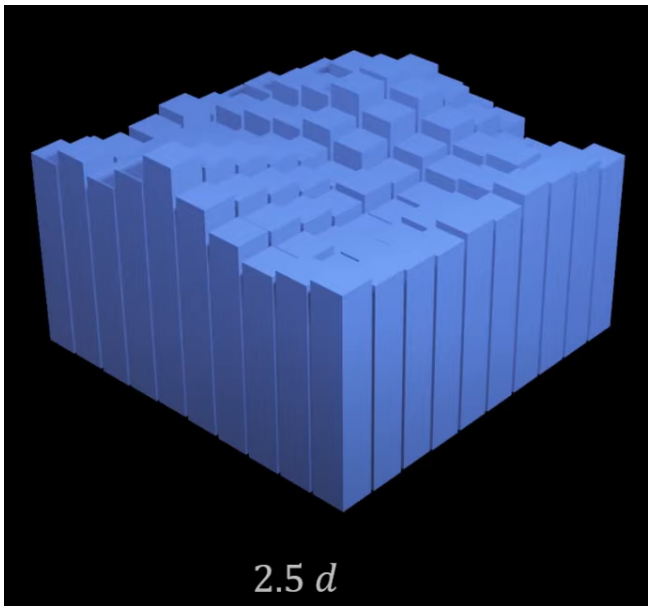

Figure 4: A representation of the height field approach

3.3 Marker and Cell Method

The marker and cell method involves discretizing 2D or 3D space to solve the Navier-Stokes equation. The space is represented as a staggered grid where each block contains information such as pressure, velocity, viscosity, etc. Velocity values are stored on the edges rather than the center of the block. This data structure is typically preferred for vector field representation as it allows for more accurate derivative calculations. The concept is to solve partial differential equations to calculate the velocity and pressure values for each block.

- Start with an initial divergence-free velocity field $\vec{u}(0)$.
- For time step $n = 0, 1, 2, \ldots$
- Determine a good time step Δt to go from time t_n to time t_{n+1}.
- Set $\vec{u}_A = \text{advect}(\vec{u}_n, \Delta t, \vec{u}_n)$.
- Add $\vec{u}_B = \vec{u}_A + \Delta t \vec{g}$.
- Set $\vec{u}_{n+1} = \text{project}(\Delta t, \vec{u}_B)$.

[1]

Here, the "advect" method is utilized to calculate velocity advection. This process involves Semi-Lagrangian advection. In the second step of the for loop, external forces are applied. This step employs a simple Forward Euler update. Lastly, the "project" method is focused on ensuring the water remains incompressible. It achieves this by subtracting the pressure gradient from the intermediate velocity field $\vec{u}$.

$$\vec{u}_{n+1} = \vec{u} - \frac{\Delta t}{\rho} \nabla p$$

With the constraint that

$$\nabla \cdot \vec{u}_{n+1} = 0$$

By using the central difference approximation then putting the velocity update equation on the divergence free equation we obtain the Poisson equation defined as

$$-\frac{\Delta t}{\rho}\nabla \cdot \nabla p = -\nabla \cdot \vec{u}$$

[1]

So the purpose of the project part is to solve the Poisson equation and then use the pressure field to update the velocity to make it divergence free.

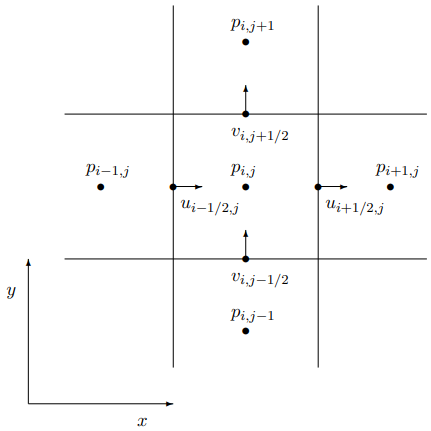

Figure 2.1: The two-dimensional MAC grid.

Figure 5: Staggered grid in 2D

The Marker and Cell method indeed produces realistic results and can interact with the environment by designating certain cells as obstacle solid cells. However, due to the high computational cost associated with calculating pressure and vector fields for each block, this method is not feasible for real-time simulations. This leads us to the question: can we employ a different data structure that requires less computation while still capturing the complex structure of the water? This will be explored in our next section.

3.4 Adaptive Grids

An approach to address the complexity of resolving the Navier-Stokes equation on the grid is to employ an adaptive grid. An adaptive grid dynamically adjusts the resolution of the simulation grid based on the complexity and movement of the fluid flow. This means that regions with high fluid complexity or where fine details are important will have a higher grid resolution, while regions with relatively simple fluid behavior can use a coarser grid. By adaptively adjusting the grid resolution, computational resources can be focused where they are most needed, thereby improving the efficiency of the simulation.

The article "Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid" [2] proposes a state-of-the-art method for adaptive grid-based real-time water simulation. The main contributions of the article are as follows:

- A tall cell grid data structure that allows for efficient liquid simulation within a wide variety of scenarios.
- An efficient multigrid Poisson solver for the tall cell grid. The solver can also be used to accelerate fluid simulations on the commonly employed staggered regular grid.
- Several modifications in the level set and velocity advection schemes that allow both larger time steps to be used and an efficient GPU implementation.

The main idea of the adaptive grid is to integrate a generalized height field representation with a three-dimensional grid layered on top of it. This allows us to approximate the general behavior of the fluid using tall cells while still capturing the interesting features of a full 3D simulation, such as splashes and overturning. Additionally, the article imposes certain constraints to simplify the data structure and algorithm, as well as making the method GPU-friendly.

- Each water column contains exactly one tall cell
- The tall cell is located at the bottom of the water column
- Velocities are stored at the cell center for regular cells and at the top and bottom of tall cells

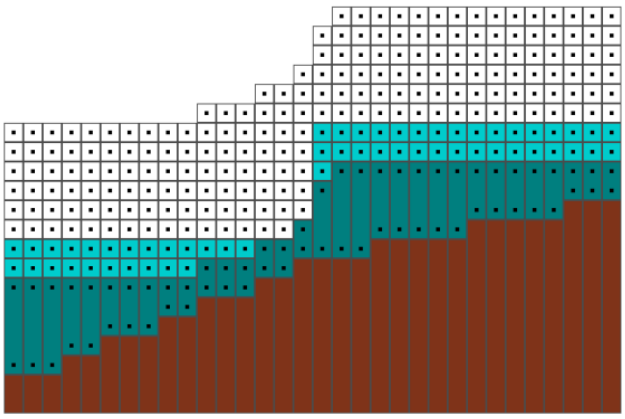

Figure 6: A represantation of the tall cell grid

Even though the article claims to achieve efficient results in various scenarios, maintaining at least 30 frames per second, the method it presents is highly restricted. The version proposed

in the article does not appear to be configurable for parameters such as pressure and viscosity of the liquid. Therefore, we will explore more parameterizable methods, such as Smoothed Particle Hydrodynamics.

3.5 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) represents fluids as a collection of particles distributed throughout the domain, with each particle carrying properties such as position, velocity, density, and other physical quantities. SPH employs kernel functions to evaluate spatially varying quantities at any point within the fluid domain, defining the influence of nearby particles on a given particle based on their distances. Through interpolation, the properties of a fluid at any point are computed by interpolating the values of neighboring particles using these kernel functions. A popular choice for a kernel function is the poly6 kernel. [1]

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

According to SPH, a scalar quantity A is interpolated at location r by a weighted sum of contributions from all particles: [6]

$$A_s(x) = \sum_j m_j \frac{A_j}{\rho_j} W(|x - x_j|) \tag{3}$$

A nice property of this formulation is that the gradient of such a field can easily be computed by replacing the kernel by the gradient of the kernel

$$\nabla A_s(x) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(|x - x_j|) \tag{4}$$

For example, we can calculate the smooth density field from the individual positions and masses of the particles with

$$\rho(x) = \sum_j m_j W(|x - x_j|).$$

Indeed, the main idea behind Smoothed Particle Hydrodynamics (SPH) is to apply the SPH rule to the pressure or viscosity term and update particle properties accordingly. One of the main advantages of this method is that since we work with particles, mass conservation is guaranteed, and as we move the particles directly, all properties are advected automatically.

Additionally, another advantage of SPH is that from the governing equation(3) we can calculate any field that we desire. This makes SPH highly configurable and a promising choice for simulating liquids.

However, SPH is sensitive to density fluctuations resulting from neighborhood deficiencies, and enforcing incompressibility can be computationally expensive due to the unstructured nature of the model. Although recent advancements have improved efficiency by an order of magnitude, small time steps remain a requirement, limiting its applicability in real-time simulations.[5]

3.6 Position Based Fluids

In SPH algorithms, instability can arise if particles lack sufficient neighbors for accurate density estimates. Typical solutions involve taking sufficiently small time steps or ensuring a sufficient number of particles. However, these solutions are often impractical for real-time simulations.

The article "Position Based Fluids" [5] proposes a different approach to maintain stability while ensuring a stable density. This approach introduces a position constraint for each particle. The constraint is defined as:

$$C_i(p_1, \ldots, p_n) = \frac{\rho_i}{\rho_0} - 1 \tag{5}$$

where ρ_0 is the rest density and ρ_i is given by the standard SPH density estimator:

$$\rho_i = \sum_j m_j W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{6}$$

The idea is to find the Δp that satisfies $C(p + \Delta p) = 0$ and than modify the position of the particle with Δp. The method seems to work very efficiently. The article says that it can simulate 128K of particles with the time step of 4.2 ms. Furthermore it still lets us modify the pressure since ρ_0 is configurable.

It seems like this method satisfy all of our requirements and seems applicable during this project.

4 Development Tools

4.1 Unity

During this project I will be working on Unity which is a very popular game engine with many optimization features. One of my objectif is to learn these features. Let's talk about some of them.

- **Compute shaders** are shader programs that run on the GPU, outside of the normal rendering pipeline. They can be used for massively parallel General-purpose computing on graphics processing units algorithms, or to accelerate parts of game rendering.
- **Unity's Job System** is a feature introduced to Unity game engine to enable efficient multi-threaded processing of tasks. It allows developers to take advantage of multi-core processors without the complexity of manual thread management. The Job System works in conjunction with Unity's Entity Component System (ECS) and Burst Compiler to achieve high-performance computing.
- **Unity's Burst Compiler** is a high-performance compiler specifically designed to optimize C code for running on multiple platforms, particularly targeting CPUs. It's primarily used in Unity game development to achieve significant performance improvements, especially for CPU-bound tasks. Burst Compiler works in conjunction with Unity's Job System and Entity Component System (ECS) to maximize performance gains.

5 Implementation

I ran the following implementation on a laptop equipped with a 12th generation Intel i5-12450h CPU and an NVIDIA GeForce RTX 3050 GPU.

5.1 Marker and Cell

To better understand the Eulerian approach in fluid simulation, I began the implementation process with marker and cell (MAC) methods. I worked separately on simulating smoke and water cases. For the smoke simulation, I essentially followed the simulation loop as explained in chapter 3.3. To render smoke, a density field is created where values between 0 and 1 indicate the presence of smoke within a cell. Smoke simulation involves simply advecting the density values using Semi-Lagrangian advection, similar to what we did for the velocity field. Additionally, I incorporated obstacle interaction by adjusting the velocity of cells overlapping with obstacles to the opposite of the obstacle's velocity. This enables us to manipulate the smoke by moving the obstacles.

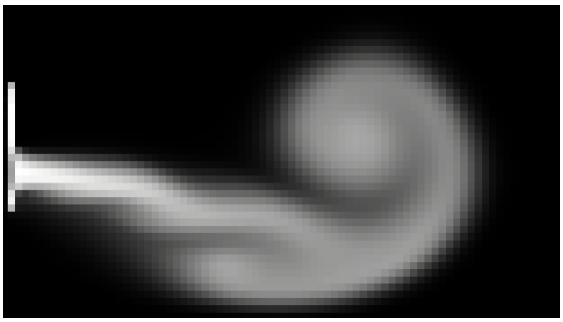

Figure 7: Marker and cell method to simulate smoke

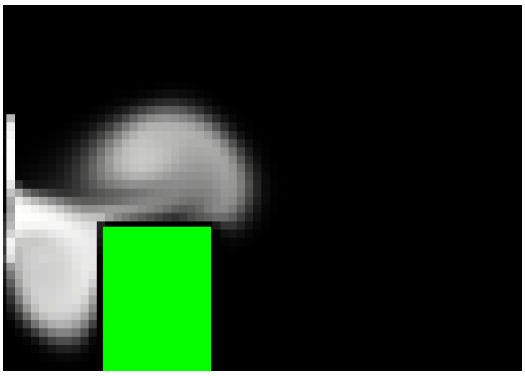

Figure 8: Smoke simulation with an obstacle interaction

To render the water, some adjustments were necessary. Unlike with smoke, we can't represent water presence as a density field (either it's present or it's not). Therefore, we use marker particles to denote where the fluid is: any grid cell containing a marker particle is marked as fluid, while empty cells represent air. The simulation begins by sampling the fluid volume with 4 particles per cell. These particles are then advected using the calculated velocity field. Here, we employ the second-order Runge-Kutta method since the Forward Euler method performs poorly for rotational motion. An additional step for velocity extrapolation is necessary, as the fluid velocity may not be defined if the water splashes into a new region of space. To address this, I utilize a fast sweep method to approximate the velocity by recursively averaging the velocity values of the neighbors closest to the water's surface. The algorithm starts by identifying the surface cell, which is easily done since we mark the fluid with marker particles. Thus, we can find the surface by checking only the neighbors of a water cell. A cell is considered part of the surface if there is an air cell next to it.

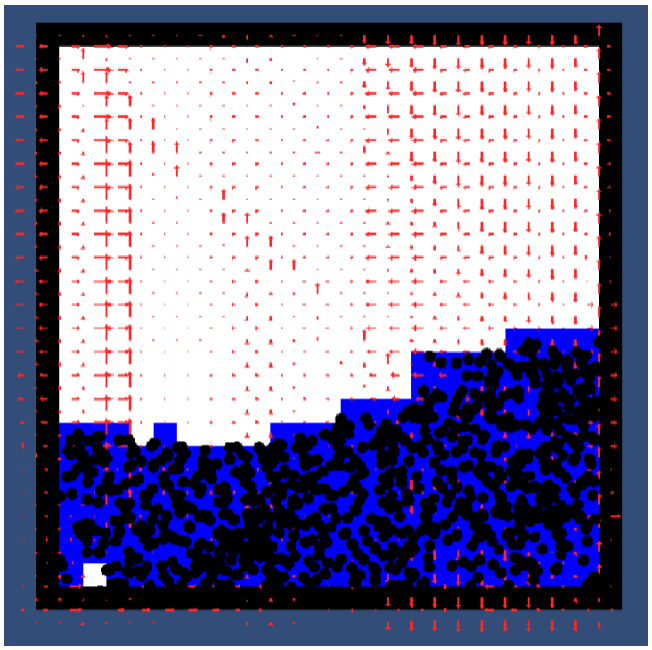

Figure 9: Marker and cell method with Marker Particles

In Figure 9, you can see a simulation frame from the Marker and Cell (MAC) method applied to water. In the image, the black dots represent the marker particles, the blue cells indicate water, the white cells represent air, and the black cells denote solid objects. I have also included some arrows to depict the velocity field. It's worth noting that water cells coincide with the locations of marker particles.

5.2 Smoothed Particle Hydrodynamics

For the Smoothed Particle Hydrodynamics (SPH) implementation, I primarily followed the guidelines outlined in the "Particle-Based Fluid Simulation for Interactive Applications" [6] paper. The main simulation loop is defined as

Algorithm 1 SPH Simulation Loop

for all particles i do **do**
  compute ρ_i
  compute p_i using ρ_i
end for
for all particles i do **do**
  $a_i^{body} = g$
  $a_i^{pressure} = -\frac{1}{\rho_i}\nabla p_i$
  $a_i^{viscocity} = \nu\nabla^2 v_i$
  $a_i(t) = a_i^{body} + a_i^{pressure} + a_i^{viscosity}$
end for
for all particles$i$ do **do**
  $v_i(t + \Delta t) = v_i(t) + \Delta t a_i(t)$
  $x_i(t + \Delta t) = x_i(t) + \Delta t v_i(t + \Delta t)$
end for

I utilized the Poly6 kernel for density estimation, the Spiky kernel for gradient calculation,

as the gradient of the Poly6 kernel diminishes when two particles approach closely, resulting in particle clustering, and the viscosity kernel for viscosity estimation.

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

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

$$W_{\text{spiky}}(r,h) = \begin{cases} \frac{15}{2\pi h^3}\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & \text{if } 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases}$$

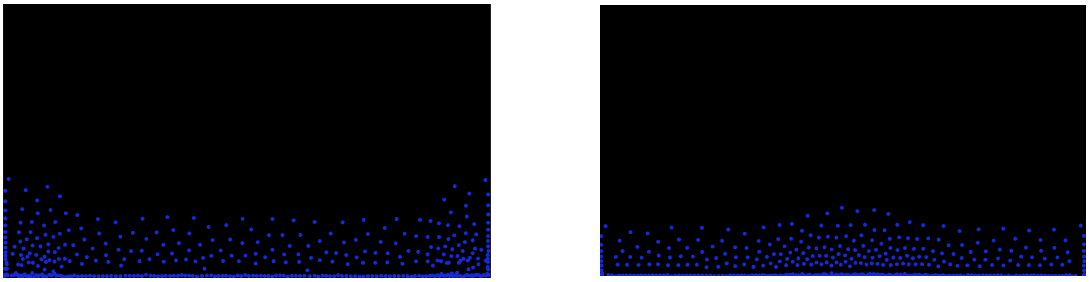

Figure 10: 2 frames from SPH simulation with 400 particles

5.3 Position Based Fluids

For the position based fluids implementation, I followed the Position Based Fluids [5] article. The main simulation loop is defined as:

Algorithm 1 Simulation Loop

for all particles i **do**
  apply forces $\mathbf{v}_i \Leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_{ext}(\mathbf{x}_i)$
  predict position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
end for
for all particles i **do**
  find neighboring particles $N_i(\mathbf{x}_i^*)$
end for
while $iter < solverIterations$ **do**
  for all particles i **do**
    calculate λ_i
  end for
  for all particles i **do**
    calculate $\Delta \mathbf{p}_i$
    perform collision detection and response
  end for
  for all particles i **do**
    update position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i^* + \Delta \mathbf{p}_i$
  end for
end while
for all particles i **do**
  update velocity $\mathbf{v}_i \Leftarrow \frac{1}{\Delta t}\left(\mathbf{x}_i^* - \mathbf{x}_i\right)$
  apply vorticity confinement and XSPH viscosity
  update position $\mathbf{x}_i \Leftarrow \mathbf{x}_i^*$
end for

Figure 11: Postition based Fluids Simulation Loop

In the implementation, I used Poly6 for the density and viscosity calculation, and spikey kernel for gradient calculation. The viscosity is added to the simulation by slightly adjusting the velocity by using the average of the neighbors velocity.

$$v_{\text{new},i} = v_i + c \sum_j v_{ij} \cdot W(p_i - p_j) \tag{7}$$

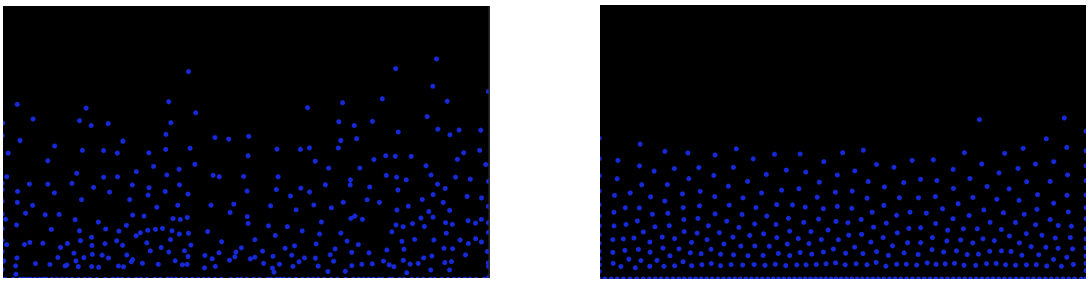

Figure 12: 2 frames from PBH simulation with 400 particles

Notice that when the simulation stabilizes, all of the particles maintain an equal distance, unlike in the SPH method where particle clustering at the borders is observed due to pressure effects. This behavior arises because PBF aims to resolve position constraints without directly utilizing pressure to displace particles.

5.4 Performance Comparison

These basic implementations were great for me to gain a better understanding of various types of Eulerian and Lagrangian-based fluid simulations. At this stage, it was unnecessary to further develop each method. Therefore, I compared these methods and chose the most suitable one for real-time liquid simulation.

For the smoke simulation, I used a grid resolution of 100×50 and for the eularian water simulation I used a grid resolution of 25×25. In both simulations we get more than 30 frames per seconds (fps). However when I doubled the resolution, I observed approximately 15 fps for the smoke simulation and only 0.2 fps for the water simulation. One idea could be to proceed with the Eulerian approach using an adaptive grid, but based on my experience, I've found that the Eulerian approach demands more development time, particularly with the intricate data structure proposed in the "Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid"[2] article. Even though I am interested in further exploring this approach, given the project's time constraints, I have decided to continue with a particle-based approach.
As suggested in the "Position Based Fluids" article, I achieved more efficient performance results with PBF compared to the SPH approach. In a CPU implementation with 400 particles, I obtained approximately 20 fps for the SPH simulation and 35 fps for the PBF simulation. It's worth noting that these fps values vary depending on the density and the smoothing radius, as more computations are performed when there are more neighboring particles.
Given these results, PBF appears to be the preferred option. Therefore, I focused on optimizing the PBF implementation to achieve higher particle counts with improved fps.

6 Optimisation

6.1 Neighbor Search

One of the most computationally expensive parts of particle-based fluid simulations is the neighbor search. Since achieving realistic results requires a large number of particles, we cannot simply loop over each particle every time we need to perform a computation.
My initial approach was to cache the neighbor particles in a list for each particle, thereby needing to calculate the neighbors only once for each iteration. However, this method still maintains a complexity of $O(n^2)$. In fact, here are the time elapsed for major parts of the PBF simulation with 800 particles: External force calculation: 0.038 ms, neighbor search: 16.576 ms, density calculation: 1.617 ms, constraint solver: 35.630 ms. So approximately 30% of the time is consumed by the neighbor search, and this time consumption increases exponentially with the number of particles.
One solution is to employ a spatial grid where each cell contains a list of particles within it. By looping over particles only once with this approach, the complexity reduces to $O(n)$. However, since the size of the lists is not constant, this method is not GPU-friendly. Therefore, I pursued a GPU-friendly approach proposed in the article "Particle Simulation Using CUDA."[4].

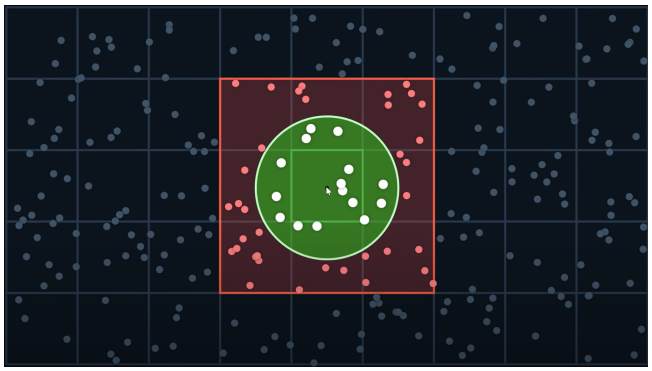

Figure 13: An illustration of the spatial grid

The article primarily utilizes two arrays to find neighbors. The first array is for spatial indexing. For each particle, we calculate a hash value based on the cell index that the particle resides in. Consequently, particles within the same cell share the same hash value. Next, we sort this array by hash value, ensuring that all particles within the same cell are adjacent to each other in the spatial index array. The second array holds the starting index for particles within the same cell. Essentially, it contains the index of variations within the spatial index array.

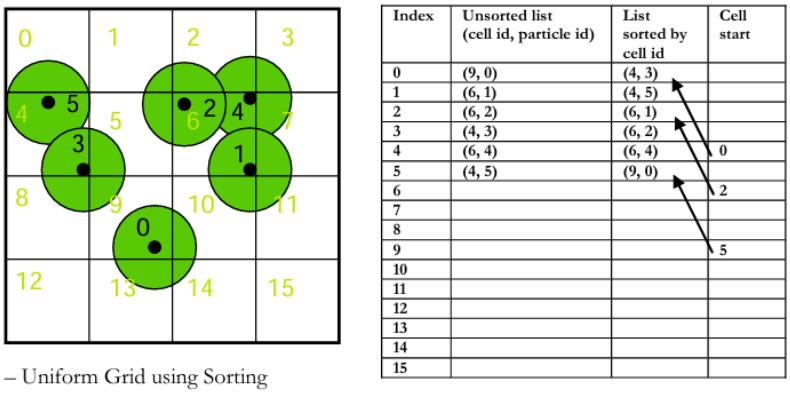

Figure 14: An example to show how the algorithm works

With this method implemented on CPU, the time required for the neighbor search is reduced to approximately 5 ms with 800 particles. With the GPU implementation, this time decreases to 0.046 ms! To sort the spatial index array, I used the bitonic mergesort method, which is a popular parallel sorting algorithm.

6.2 Job System

Unity uses its own Job system, which operates with a specially designed compiler called Burst Compiler. This system allows users to create multithreaded code, enabling applications to utilize all available CPUs on the machine. Additionally, this system utilizes native data structures, making it possible to share data between managed and native environments without incurring marshalling costs.
As a second optimization, I adapted the constraint solver part to utilize Unity's Job system. Consequently, with 800 particles, we achieve around 10 fps with the single-threaded version and 30 fps with Unity's Job system. Moreover, the time required for constraint calculation is four times less than that of the single-threaded version (19.95 ms compared to 81.60 ms).

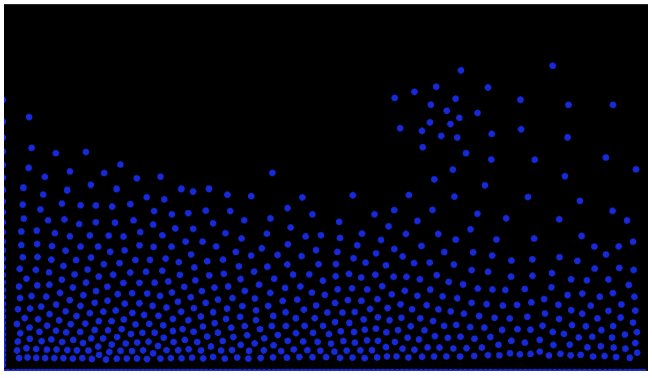

Figure 15: PBF and Unity Jobs, with some mouse interaction

6.3 GPU Implementation

Graphics processing Units (GPU) are designed to handle thousands of threads simultaneously. They consist of many smaller cores designed for handling multiple tasks at the same time, which makes them ideal for parallel processing tasks. Recently, developers start to use gpu,

which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). Unity let the developers to use GPU with Compute Shaders.
My final optimization for the 2D implementation was to convert computations into a compute shader, and the results are astonishing. We achieve 60 fps with 8000 particles, making it 20 times more efficient than the Job system implementation. At this point, I was satisfied with the results and proceeded to the 3D implementation.

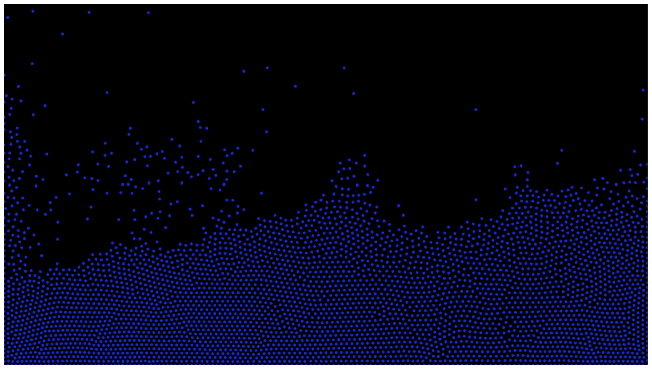

Figure 16: GPU Implementation with 4000 particles

7 3D Implementation

The implementation for the 3D dimension was straightforward. I simply converted the 2D data structures to 3D, adjusted the neighbor search algorithm, and adapted the border collision for the third dimension. However, due to the additional computation required for rendering (especially for lighting calculations), I observed a performance loss in the 3D version. With 4000 particles, we achieve approximately 30 fps with the 3D version compared to 120 fps with the 2D version. For the 3D case, I made one final optimization. In the initial version, I used Unity's game object structure to represent the particles. However, this necessitated separate draw calls for each particle, leading to increased render time. To address this issue, I implemented GPU Instancing, which optimizes draw calls by rendering multiple copies of a mesh with the same material in a single draw call. With this optimization, the fps increased to 45.

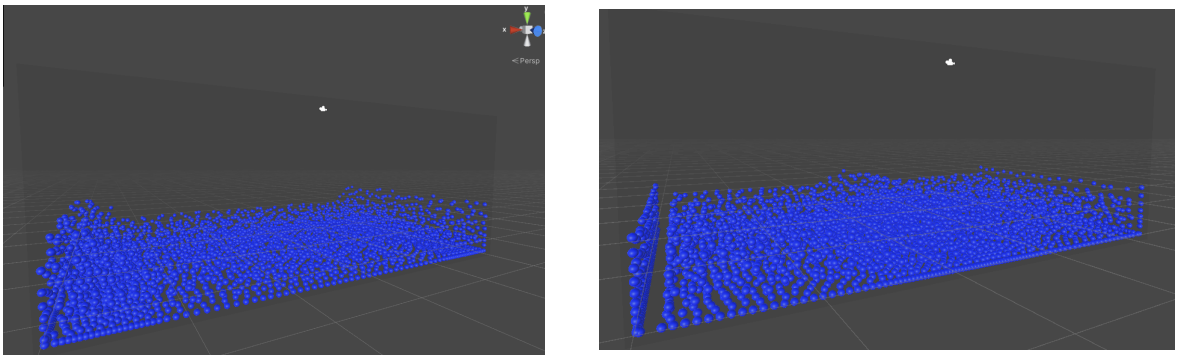

Figure 17: 3D implementation with 4000 particles

8 Results

At the end of this project, I had an optimized fluid simulation based on the Position Based Fluids method. The simulation runs at 45 fps with 4000 particles and 8 constraint solver iterations. More efficient results can be achieved by reducing the number of iterations. However, the incompressibility of the liquid is dependent on the number of solver iterations. Increasing this number results in a more incompressible liquid but requires more computation, leading to a decrease in fps.
The model can simulate both liquid and gas states. Indeed, the state of the simulation depends on the density parameter. As explained in the introduction, in nature, the state of matter is influenced by pressure and temperature. Although we did not have time to incorporate temperature into our simulation, we can simulate phase transitions by modifying density, as changes in pressure lead to variations in the density of matter.

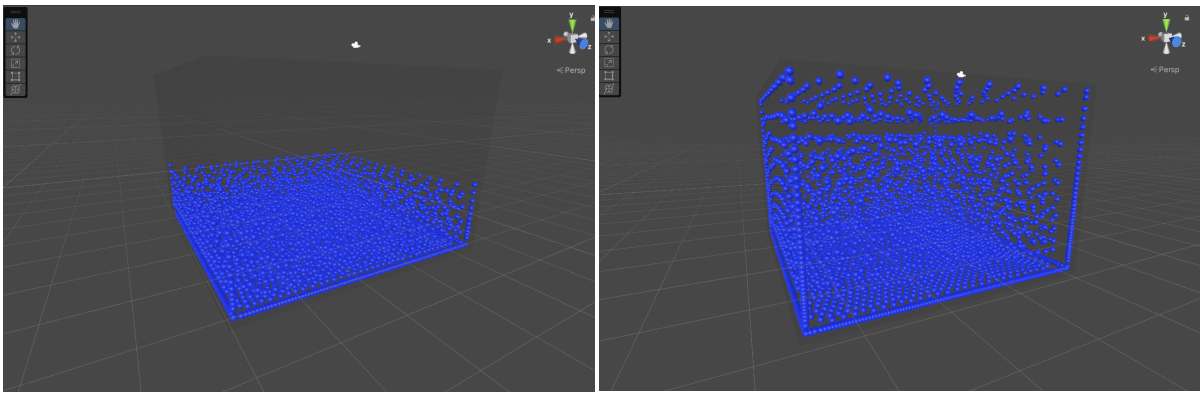

Figure 18: Phase transition by changin the density

In Figure 18, you can observe the transition of matter from liquid to gas by adjusting the density parameter. Notice that in the gas phase, the particles expand to fill the container.

9 Conclusion

During this project, detailed research into fluid simulation methods was conducted. Suitable methods for real-time applications were implemented and compared to select the most efficient method that also meets our requirements. The model was then optimized in both 2D and 3D. It is capable of simulating 4000 particles in real time at 45 fps and can simulate phase transitions by varying the density parameter. However, as future work, further optimizations are possible, as the main article is able to produce 128k particles in real time. For example, we can utilize indirect instancing to update particle positions on the GPU instead of the CPU, by sending translation, rotation, and scaling matrices to the GPU each frame. Additionally, as future work, I would like to explore liquid and gas rendering techniques such as marching cubes. Finally, to create more interactive scenes, I am interested in working with physical object interactions such as boats, buoys, or rubber ducks.

10 Bibliography

References

[1] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: Siggraph 2007 course notes video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 courses*, pages 1–81. 2007.

[2] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. In *ACM Siggraph 2011 Papers*, pages 1–10. 2011.

[3] Alain Fournier and William T Reeves. A simple model of ocean waves. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84, 1986.

[4] Simon Green. Particle simulation using cuda. *NVIDIA whitepaper*, 6:121–128, 2010.

[5] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.

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

11 Appendices

11.1 Semi-Lagrangian Advection

Figure 19: Semi-Lagrangian Advection

In the semi lagrangina advection we linearly approximate a quantity of a cell on a staggered grid, by calculating the gradient of the field and estimating the value at the current position by looking to it's "old" position.