
Bachelor's Thesis:
PBFUnity - Position Based Fluids
Framework in Unity

A thesis submitted in fulfillment of the requirements
for the degree of **Bachelor of Science (B. Sc.)**
to the faculty of **computer science**.

Mert Kaynak
<kaynak@uni-bremen.de>
Matriculation No. 6155675
September 2026

First examiner:	Prof. Dr.-Ing. Gabriel Zachmann
Second examiner:	Dr.-Ing. Robert Porzel
Supervisor:	Prof. Dr.-Ing. Gabriel Zachmann

Abstract

Fluid simulation is expensive, and game engines run at a fixed timestep that is too large for classical Smoothed Particle Hydrodynamics to stay stable. Position Based Fluids avoids this by correcting particle positions against a density constraint instead of integrating pressure forces.

This thesis presents PBFUnity, an implementation of Position Based Fluids in Unity 6 that runs entirely on the graphics card. It consists of a solver of nine compute kernels, a screen space renderer that reconstructs a fluid surface from particle depth, two level of detail systems and a benchmark harness. No particle data is sent back to the processor while a scene is running.

I measured the framework on six machines with graphics cards from an RTX 3060 up to an RTX 4090, across ten scenes ranging from an empty tank to 127,526 particles, each run with the level of detail switched on and off. From the RTX 3080 upward it holds 60 frames per second in nine of the ten scenes, and on the RTX 4090 in all ten. The level of detail improves the result only where frame times already lie close to the target.

Zusammenfassung

Flüssigkeitssimulation ist rechenaufwendig, und Spiel-Engines arbeiten mit einer festen Zeitschrittweite, die zu groß ist, als dass klassische Smoothed Particle Hydrodynamics stabil bleiben könnte. Position Based Fluids umgeht dies, indem die Partikelpositionen gegen eine Dichte-Zwangsbedingung korrigiert werden, anstatt Druckkräfte zu integrieren.

Diese Arbeit stellt PBFUnity vor, eine Implementierung von Position Based Fluids in Unity 6, die vollständig auf der Grafikkarte läuft. Sie besteht aus einem Löser aus neun Compute-Kernen, einem Screen-Space-Renderer, der aus der Partikeltiefe eine Flüssigkeitsoberfläche rekonstruiert, zwei Systemen für Level of Detail (LOD) und einer Umgebung zur Leistungsmessung. Während eine Szene läuft, werden keine Partikeldaten an den Prozessor zurückgesendet.

Ich habe das System auf sechs Rechnern mit Grafikkarten von einer RTX 3060 bis zu einer RTX 4090 vermessen, über zehn Szenen hinweg, die von einem leeren Aquarium bis zu 127.526 Partikeln reichen, jeweils mit ein- und ausgeschaltetem LOD. Ab der RTX 3080 hält das System in neun der zehn Szenen 60 Bilder pro Sekunde, auf der RTX 4090 in allen zehn. Das LOD verbessert das Ergebnis nur dort, wo die Bildzeiten bereits nahe am Zielwert liegen.

Acknowledgements

I would like to thank Prof. Dr. Gabriel Zachmann for supervising this thesis and helping me get through this.

My thanks also go to Amir, Cyprian, Hasan, Talha, Xavier and Yusuf, who ran the benchmark on their own computers and sent me the results. Without them the evaluation would have covered one machine instead of six.

Finally, I would like to thank my family and my friends for supporting me throughout my studies.

List of Abbreviations

- **PBF** = **P**osition **B**ased **F**luids
- **MSS** = **m**ass-**s**pring **s**ystem
- **GPU** = **G**raphics **p**rocessing **u**nit = graphics card chip
- **SSFR** = **S**creen-**s**pace **f**luid **r**enderer(ing)
- **LOD** = **L**evel **o**f **D**etail
- **SPH** = **S**moothed **P**article **H**ydrodynamics
- **XSPH** = **E**xtended **S**moothed **P**article **H**ydrodynamics
- **WCSPH** = **W**eakly compressible **S**moothed **P**article **H**ydrodynamics
- **PCISPH** = **P**redictive-**c**orrective **i**ncompressible **S**moothed **P**article **H**ydrodynamics
- **IISPH** = **I**mplicit **i**ncompressible **S**moothed **P**article **H**ydrodynamics
- **DFSPH** = **D**ivergence-**f**ree **S**moothed **P**article **H**ydrodynamics
- **ISPH** = **I**ncompressible **S**moothed **P**article **H**ydrodynamics

Contents

Abstract	i
Zusammenfassung	i
Acknowledgements	ii
List of abbreviations	iii
1 Introduction	1
1.1 Motivation	1
1.2 Goals	2
1.2.1 Documented and open implementation	2
1.2.2 Smooth performance	2
1.2.3 Visual plausibility	2
1.2.4 Level of Detail	3
2 Fundamentals	4
2.1 Lagrangian vs. Eulerian	4
2.2 Smoothed Particle Hydrodynamics (SPH)	5
2.3 Position Based Dynamics (PBD)	9
2.4 Position Based Fluids (PBF)	11
2.5 Boundary handling	14
2.6 Neighbor search	15
2.7 Screen space fluid rendering (SSFR)	16
2.8 Level of Detail (LOD)	17
3 Related Work	18
3.1 Fluid simulation in Unity	18
3.2 Academic work on fluid simulation in Unity	19
3.3 Alternative approaches to incompressibility	20
4 Implementation	21
4.1 Overview	21
4.2 GPU compute in Unity	21
4.2.1 Compute shaders and buffers	21
4.2.2 Ping-pong buffers	22
4.2.3 No transfer back to the CPU	22
4.2.4 The fixed timestep	23
4.2.5 Spatial grid is Lagrangian	23
4.3 The solver	24
4.3.1 Building the neighbor grid	25
4.3.2 Calculating the density and the multiplier	25

4.3.3	Calculating the position correction	26
4.3.4	Updating the velocity	27
4.3.5	Boundaries and rendering	27
4.4	Deviations from the published method	28
4.4.1	Differences	28
4.4.2	Additions	29
4.5	Boundary handling	30
4.5.1	Geometric boundaries	30
4.5.2	Boundary particles for arbitrary meshes	31
4.5.3	Moving boundaries	32
4.5.4	The wall skin	32
4.6	Screen space fluid rendering	33
4.6.1	Depth and thickness	33
4.6.2	Smoothing	33
4.6.3	Composite	34
4.7	Level of Detail	35
4.7.1	Rendering level of detail	35
4.7.2	Simulation level of detail	36
4.8	Benchmark harness	36
4.9	Build and deployment	38
4.10	Known limitations of the implementation	39
5	Evaluation	40
5.1	Test setup	40
5.2	The scenes	41
5.3	Metrics	42
5.4	Results	43
5.5	Effect of the level of detail	44
5.6	Frame pacing	45
5.7	Limitations of the evaluation	46
6	Future Work	47
6.1	Two-way coupling	47
6.2	Signed distance fields instead of boundary particles	47
6.3	A faster memory layout	48
6.4	Quality driven by the frame budget	48
6.5	Bubbling effect	49
7	Conclusion	50

1 Introduction

1.1 Motivation

Ever since I started programming, I have been interested in shaders and solving physics-related problems. I was and still am inspired by the work of professionals who upload media to the internet, showing visually plausible imitations and replications of real-life elements such as water, cloth and even moving vegetation.

One discovery I have made a few years ago has been a modification for the popular game Minecraft called the *Physics Mod* [1]. It features a lot of new physical-based adaptations from the real world and one of them is an alternative to the usually plain and boring water in the game. Water is flowing more realistically and consists of particles instead of one whole volume.

This modification is highly regarded and it is evidence that the gaming community wants games that introduce plausible and interactive water. The reason why games still don't include this natively into their games is due to the negative performance impact it causes. The more accurate the water, the further the cost moves out of reach of a real-time frame budget.

From all the 3D fluid solutions I've seen available for Unity, none is a documented, open implementation of a published method. For instance, there are *Obi* [2] and *Zibra Liquid* [3] that are both commercial closed-source products whose solvers cannot be inspected or verified against the literature, and the remaining options are 2D or height-field solvers that do not simulate three-dimensional fluid dynamics at all. In addition, none publishes systematic performance data across different hardware.

This is the reason why I have been so eager to implement it myself. I want to demonstrate that it is indeed possible to simulate water in a visually plausible way at acceptable performance while doing so.

1.2 Goals

1.2.1 Documented and open implementation

I want to implement the well-known PBF framework into Unity, while my solver follows Macklin and Müller’s PBF implementation directly [4], with nearly each step traceable to the paper and every deviation from it stated. Unlike the commercial alternatives, my implementation can be inspected and verified against the literature and else.

1.2.2 Smooth performance

It is supposed to run smoothly on mid-range GPUs. I consider GPUs as mid-range that sit between the RTX 3070 and the RTX 5070. And I consider the runtime experience acceptable if at least 90% of measured frames render at 60 FPS or above, i.e. within a frame time of 16.7 ms.

To prove my point, I have implemented a benchmark which consists of ten scenes in total. The first scene resembles an empty aquarium scenario to test the performance at near zero workload. The other nine scenes simulate different amounts of water in different scenarios. All benchmark results are then saved into CSV files and are evaluated by me.

1.2.3 Visual plausibility

My goal is to achieve a visually plausible water simulation. PBF determines how the water moves, but says nothing about how it looks. To achieve the water look, I implemented a screen-space fluid renderer (SSFR) consisting of four shaders in total ensuring the visually plausible water look. Additionally, because the renderer works in screen space, its cost scales with screen resolution rather than with particle count. Therefore, unlike drawing each particle as geometry, adding more particles does not make rendering more expensive.

1.2.4 Level of Detail

Many games feature LOD systems to get more performance by either simulating less or rendering less. I have added both to my project to achieve even more performance when needed.

I'm covering two LOD systems: *Rendering LOD* and *simulation LOD*.

Rendering LOD focuses on lowering details of the water (like resolution and blur) based on how much of the screen the water covers. That means, details that are very small aren't paid for and this equals more performance. This never changes how the water flows because it doesn't influence the physics just the visuals.

Simulation LOD is operating differently. Instead of lowering the details accordingly, it focuses on changing the simulation without influencing what the user sees. It skips the vorticity confinement for particles inside the fluid. Because the swirls of those particles can't be seen by the user, the simulation looks comparable to a simulation without the LOD enabled, and due to the skip of a step, the performance is influenced positively by it. Also, the benchmark result can still be evaluated and taken seriously because every testing system processes the same workload at the end.

2 Fundamentals

2.1 Lagrangian vs. Eulerian

There are two ways to simulate water. Either the Lagrangian way or the Eulerian way. J. F. Price, an American academic and oceanographer, describes both ways [5] in terms of observing water rather than simulating it:

The Lagrangian way to observe is by using a float or similar to let it carry by the water. This float has sensors that measure the flow and its properties. The Eulerian way however is described differently. Instead of relying on the water to carry an instrument, we simply set a fixed instrument at a fixed point in the water and measure the water that flows past it.

We can simply apply this to the simulation of water in real-time applications:

For the Lagrangian way, instead of using a float which gets carried away, we use the water's properties (position, velocity, vorticity, etc.) and attach them to each particle. Those are determined by the movement and interactions of the particles and are altered while the simulation is carrying on.

The Eulerian way is different. Here, the values or properties of the water aren't stored in each particle. Instead, they are stored at fixed grid points. Each grid point represents a position in working space.

Both SPH and the PBF algorithm derived from it use purely the Lagrangian way and not the Eulerian way. This is because the Eulerian grid method struggles with water tracking. The values stored in the grid points don't tell us where the water is currently flowing. For that, we would need an external separate mechanism which would add extra machinery and complexity. And the space complexity would be much higher, because we would have to pre-allocate a grid that stores values for positions with and without fluid.

The Lagrangian way fits perfectly here, because we consider just the positions where the water currently is. Any free space is ignored. Water splashing and re-

merging are much more natural to implement than it would be with the grid method. Therefore, we know where the water is and the implementation is more straightforward. [6]

2.2 Smoothed Particle Hydrodynamics (SPH)

As mentioned before in the previous section, the Lagrangian way stores all properties on the particles themselves. Therefore, we don't have a grid to differentiate against. There is nothing wrong with that, but there needs to be a way to work with those properties. We need to be able to determine how much a quantity changes from place to place. For that, we need density and pressure gradients, and a gradient is a spatial derivative which determines exactly that. But how do we compute those gradients to calculate the necessary physics for the simulation? For that, we need the **SPH method**.

Our goal is to get a property value at some arbitrary point in the scene. This point is usually the position of a particle. We don't have a grid to work with and the particles aren't always perfectly aligned. Therefore, we can use **kernel interpolation** to interpolate a value at that point from the surrounding particles. The idea comes from an exact formula built on an infinitely thin spike, the **Dirac delta** [6], which SPH replaces with a smooth kernel of finite width so it can actually be computed.

One of the most important formulas of the SPH method is the interpolation sum [7] $A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h)$ where $A_S(r)$ is the interpolated value of a field quantity A at position r . The S identifies this as the SPH approximation of A , not the exact value. j iterates over the particles, m_j is the mass of particle j , r_j its position, ρ_j its density, and A_j the value of the field quantity at r_j . m_j/ρ_j is the volume each particle represents. W is the smoothing kernel and h its radius of support.

The radius of support h determines how close a particle must be to another particle to contribute to it. Therefore, a higher h value means that more particles contribute and it results in a smoother simulation but it costs more. Meanwhile, a smaller h value corresponds to fewer particles getting involved and the simulation being sharper in detail and cheaper but noisier.

The smoothing kernel must satisfy five conditions in total [6] but two are the most important. To keep this section at a reasonable length, we will discuss just those important two:

On the one hand, there is the condition *normalized*. Here, the kernel integrates to 1. That means the kernel spreads a total weight of exactly 1 over all space. It acts as a weighted average which ensures that the interpolation reproduces a constant field exactly, instead of scaling everything by some arbitrary factor.

On the other hand, there is the condition called *even* or *symmetric*. Here, the kernel value depends on just the distance and not the direction. This, in simple terms, ensures that neighbors that are symmetrically aligned in space are counted exactly the same. Without this constraint, fluid particles on the one side would have different values than those on the other side although they are influenced by the same conditions.

A smoothing kernel that is both normalized and even makes the interpolation second-order accurate. [7]

While both conditions ensure accuracy, there is another condition which is about speed and performance. Theoretically and practically, all particles outside the h radius range shouldn't contribute to the calculations of the current particle. To ensure that, there is the condition called *compact support*. It sets the kernel value to 0 for all particles outside of h for the current calculations. Therefore, those particles aren't considered, which boosts performance. This is not a hard cutoff. Instead, the kernel is built so that its value and its first two derivatives all reach zero at h [7], so contributions fade out smoothly rather than jumping. The number of particles inside the h range is roughly constant, which makes it possible to reduce the time complexity from $O(n^2)$ to $O(n)$. It is important to remember that we have to select the particles inside for this to work and for that there is *neighbor search*, which we will discuss in Section 2.6.

Following the interpolation sum, there is the density sum [7]:

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W(r - r_j, h) = \sum_j m_j W(r - r_j, h). \quad (1)$$

By replacing the field quantity A with the density ρ , we can see that ρ_j cancels against itself. Therefore, both sums are the same except that here the field quantity A is the density. This formula is evaluated at every simulation step. That means it gets recomputed every time because density must be evaluated every timestep whereas mass is constant throughout [7].

There is something to be careful about: This density sum works well with particles that aren't at the free surface because those have particles all around them. But with those at the surface, it's different. Due to the lack of neighbors at the top and on the sides, the reconstructed density is generally too low [6]. The sum comes up short. The density sum therefore remains an estimation, and one that is biased low at the free surface and this is something any pressure solver has to account for.

Moreover, we need gradients for the calculations of the physics while simulating. The physics needs to know how quantities change from place to place and not just their values. For that, there is a formula [7]:

$$\nabla A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r - r_j, h). \quad (2)$$

It's similar to the interpolation sum but the smoothing kernel W is replaced by its gradient ∇W . It is important to mention that only the kernel is differentiated and the values of the particles are not. The values of particles change but the smoothing kernel does not. Therefore, we only have to derive its gradient once by hand. It still has to be evaluated for every pair of neighboring particles at every step, because the distances between them keep changing. Because we are differentiating the smoothing kernel, we don't need a grid to differentiate against. This solves the problem we had in Section 2.1.

Talking about smoothing kernels, there are two different kernels that we use: the *Poly6 kernel* [7] and the *Spiky kernel* [7]. Both are very important and handle different tasks. The Poly6 kernel is used for values and the Spiky kernel is used for gradients.

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

Why don't we use just one of them for everything? The answer lies in the properties of both kernels:

In the Poly6 kernel r appears only squared. This means that we never need a square root to evaluate it [7]. This saving matters, because it's used in every step for every particle pair. However, there is a catch. When particles are very close to each other, they build clusters under high pressure [7]. The gradient of the kernel, which determines the repulsion force, approaches zero the closer the particles are. To ensure that the gradient doesn't vanish near the center, there is the Spiky kernel, which Müller et al. adopt from Desbrun [7]. It doesn't have this problem but needs square roots for evaluation. Therefore, it is only used for gradients and not values.

$$W_{\text{spiky}}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Now, we have discussed every important part of SPH for the thesis. One question still remains. Why don't we use SPH instead of PBF? To answer this question, we have to point out all the relevant constraints of SPH first:

Water in real life is nearly incompressible, which means that water resists compression. SPH enforces it with a state equation $p = k(\rho - \rho_0)$ [6]. k is the stiffness constant, where a higher value means more stiffness. The k value has to be large for the water to be stiff and resist compression with very large force. Those stiff forces require small timesteps to remain stable. At timesteps that are too large,

the simulation would explode. Now the important part, why it matters for real-time applications:

Game engines like Unity don't give us a small timestep to choose. In Unity it is called *fixedDeltaTime*, which is 0.02 s, and we usually wouldn't shrink it because halving the timestep doubles the number of simulation steps per second, which doubles the GPU work. Therefore, we need an algorithm which works even with bigger timesteps that game engines impose, such as PBF.

2.3 Position Based Dynamics (PBD)

In the SPH section before, we said that we need an algorithm or method that stays stable at a large and fixed timestep. Position Based Dynamics, or PBD for short, is such a method and PBF builds on it.

The core of PBD consists of three steps [8]: *Predict*, *Project*, and *Update*. *Predict* applies external forces (like gravity) to the velocities of the particles and predicts the temporary positions based on them. *Project* goes through a loop a fixed number of times. The loop moves the predicted positions so that the constraints are better satisfied. *Update* determines the actual velocity based on the predicted position and applies both velocity and position: $v \leftarrow (p - x)/\Delta t$ and $x \leftarrow p$. Something to point out is that here velocity derives from the position and not the other way around, unlike force-based methods.

Why is this method stable compared to the SPH algorithm? The PBD solver determines a valid configuration based on the constraints, and then the loop moves vertices to that configuration. This means the vertices aren't extrapolated blindly into the future, as force-based methods do. They are moved to a valid position. Therefore, stability depends on how the constraint functions are shaped instead of the timestep size, and this makes it unconditionally stable [8].

PBD doesn't care what the constraint represents. It could be implemented for cloth or water simulation. The declared constraint function should evaluate to zero, or greater than or equal to zero for inequalities. And it can be based on distances

between particles for cloth, or based on density for fluids. That's the reason why PBF is possible. PBF uses PBD with a constraint based on density.

This method also uses a Newton-Raphson step along the constraint gradient ($\Delta \mathbf{p}$) [8], with an equation for the correction of an individual point ($\mathbf{p}_i$) and its scaling factor (s) [8]:

$$\Delta \mathbf{p} = -\frac{C(\mathbf{p})}{|\nabla_{\mathbf{p}} C(\mathbf{p})|^2} \nabla_{\mathbf{p}} C(\mathbf{p}) \quad (5)$$

$$\Delta \mathbf{p}_i = -s \nabla_{\mathbf{p}_i} C(\mathbf{p}_1, \dots, \mathbf{p}_n) \quad (6)$$

$$s = \frac{C(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_j |\nabla_{\mathbf{p}_j} C(\mathbf{p}_1, \dots, \mathbf{p}_n)|^2} \quad (7)$$

This is exactly the structure of PBF's λ calculation step. We will come back to this later on in Section 2.4.

The PBD method also includes an iteration loop. It does that because constraints are coupled. This is because particles are shared and if we move a particle, it also influences other particles. This means that if a constraint is satisfied, this usually breaks another. We need a loop to repeatedly fix the constraints of all particles. The original PBD method uses the Gauss-Seidel iteration for that [8]. It solves the constraints one after another and each correction becomes immediately visible to the ones that follow. The PBF algorithm, however, uses the Jacobi iteration [4], but we will discuss this in Section 2.4 as well.

2.4 Position Based Fluids (PBF)

This thesis features the Position Based Fluids algorithm, or PBF for short. As mentioned before, it inherits aspects of both SPH and PBD and is stable with bigger timesteps. As the name suggests, it is an algorithm to simulate fluids only.

We learned that SPH is stable with smaller timesteps and that a game engine uses bigger timesteps that make SPH unsuitable for them. PBF fixes this problem. But what exactly allows PBF to work with bigger timesteps? The reason lies in PBD. As mentioned in the previous section, the algorithm uses a loop to move vertices to satisfy a constraint. The algorithm isn't dependent on the timestep size, but rather on the constraint function's shape.

PBF keeps using the same kernels, density estimators, gradients and neighbor search as SPH does, but replaces what's done with the measurement. A density error results in a position correction, without pressure or force.

The constraint that we use in PBF is $C_i(\mathbf{p}_1, \dots, \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1$ [4]. It is satisfied when the particle's density (ρ_i) equals the rest density (ρ_0). If the constraint evaluates to zero, the density is correct. A positive value corresponds to an over-dense condition, while a negative value corresponds to an under-dense condition. The ρ_i value is calculated with this equation $\rho_i = \sum_j m_j W(\mathbf{p}_i - \mathbf{p}_j, h)$, which is the SPH density estimator from Equation (1), evaluated at the position of particle i .

And it's important to mention that the original paper treats all particles as having equal mass and therefore drops the mass term from all following equations.

The same kernels as in SPH are used: Poly6 for density estimation and Spiky for gradient calculation [4].

Now, let's say we want to move a particle. For that we need to know in which direction the density error decreases. This information is given by the gradient of the constraint:

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = i \\ -\nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = j \end{cases} \quad (8)$$

k is an index for either the currently selected particle (i) or one of its neighbors (j). Therefore, there are two cases, depending on which of the two we differentiate with respect to. Here, only the kernel is differentiated, similar to how it is in SPH (2).

Additionally, PBF uses the projection formula from PBD (5) using the density constraint: $\lambda_i = -\frac{C_i(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2 + \varepsilon}$ [4]. ε is a relaxation parameter. It is necessary because the gradient of the density constraint approaches zero at the boundary of the smoothing kernel. That makes the denominator very small when particles are close to separating and this causes instability. With the parameter ε the denominator stays large enough and prevents instability. And it compensates for the density being underestimated at the free surface, where particles have fewer neighbors than they need. As mentioned in Section 2.3, PBF solves the constraints with the Jacobi iteration and ε is also what makes that possible. It acts as a regulator and prevents parallel solvers from oscillating between solutions instead of settling on one.

Now that every particle has its λ multiplier, the next step is the calculation of the position correction [4]:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h) \quad (9)$$

Here, the position correction is calculated from the current particle's multiplier (λ_i) and those of its neighbors (λ_j). The reason why we need both λ_i and λ_j is because we need to be consistent from both the current and the neighbor's side.

SPH has a known problem where particles cluster together instead of spreading out. This problem is called tensile instability [4]. It happens when the number of neighbors isn't enough to reach the rest density. This produces a negative pressure that pulls the neighbors to the current particle. For that there is an artificial pressure

term (s_{corr}) which results from the smoothing kernel itself:

$$s_{\text{corr}} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta \mathbf{q}, h)} \right)^n \quad (10)$$

$\Delta \mathbf{q}$ is a point at a fixed distance inside the kernel radius h . It acts as a threshold that decides at which distance the artificial pressure becomes significant. The paper suggests using a value between $0.1h$ and $0.3h$ for $\Delta \mathbf{q}$, 0.1 for k and 4 for n .

The s_{corr} term is added to the position update equation [4]:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h) \quad (11)$$

Because s_{corr} is purely repulsive, it keeps the density slightly below the rest density. This creates an effect similar to surface tension because the particles pull their neighbors inwards. The paper says that this is a non-physical artifact of the anti-clustering term [4].

Water has a rotational motion that gives it the ability to swirl. Position based methods like PBD introduce additional damping, which negates that ability. PBF adds the energy back with a force called *vorticity confinement*, which is calculated by first estimating the vorticity at each particle. Afterwards, another effect called *XSPH viscosity* is applied. This one helps with neighboring particles moving together. The paper says that it uses a coefficient c that is typically set to 0.01 . [4]

In nearly all equations, we used values of neighbor particles. But to be able to use those, we need to find neighbors first. Those are found with neighbor search outside the solver loop. Inside the solver loop, all constraint values are recalculated each iteration, and usually there are two to four iterations for real-time applications [4].

A serious limitation of standard PBF is that particles can stack along boundaries due to incorrect density estimations when they have contact with solids [4]. In the next section, we will discuss this matter further.

2.5 Boundary handling

As mentioned in the section before, PBF has a problem where particles that are close to a wall sometimes get through it or stack up against it. This happens because there is no fluid on the other side of the wall, and that causes faulty density estimations. It's the same as when particles are at the free surface: Particles appear to be under-dense. The solver then stops pushing those particles away. [6]

We introduce two approaches to prevent this problem: *boundary particles* and *geometric boundaries*.

Boundary particles are particles that sample the surface [9] of a solid and are either immovable or move together with the solid. Otherwise, they act the same as normal particles and contribute to the density sum. They replace the particles that are needed on the other side of the wall. In most cases there should be more than one layer of them because a single layer can leave a gap in the region that the smoothing kernel reaches into. Another limitation is that they occupy memory and take part in every neighbor search.

Geometric boundaries do it differently: They don't sample the surface of solids at all and instead calculate the distance and normal to the wall directly. The distance is used to calculate the missing density, and the normal gives the direction in which the particle is mirrored [10]. The problem with this approach is that it is simple for planes and boxes, but harder for arbitrary meshes.

Both approaches have their use cases: Boundary particles are general but cost memory and neighbor search time, while geometric boundaries are cheap but need a distance function for every shape.

2.6 Neighbor search

Every equation in the previous sections iterates over the neighbors of the currently selected particle. Therefore, we need to find those neighbors and for that there is neighbor search:

The naive way would be to check every particle against every other particle, but this would give a time complexity of $O(n^2)$. Instead, we should remember the term *compact support*, which we introduced in the SPH section. It says that only particles inside the radius h can contribute.

With that being said, let's introduce *the uniform grid*: The uniform grid divides the whole simulation space into cells of equal size, and every particle is assigned to the cell that contains its position. Because the cell size is set to h , no neighbor can be further away than one cell in any direction, so only 27 cells have to be searched instead of the whole simulation. This improves the time complexity from $O(n^2)$ to $O(n)$.

But how exactly do we access those cells? There are multiple approaches and this thesis uses the *spatial hashing* method [11]:

A grid has many empty cells. Those can be ignored, which is more efficient. Spatial hashing features a fixed-size hash table. The three-dimensional cell coordinates are converted into indices and are stored inside the table with the help of a hash function. This way only the cells that actually contain particles take up space. It can occur that two different cells have the same index, which is called a collision.

Also, the grid has to be rebuilt in every simulation step because the particles move. This happens outside the solver loop, so it is done once per step instead of once per iteration.

Lastly, the PBF paper reports that detecting neighbors takes around 28 percent of a frame [4]. And the constraint solve, which itself consists of neighbor loops, takes between 38 and 51 percent. This indicates that the biggest part of a frame is spent on work that involves neighbors.

2.7 Screen space fluid rendering (SSFR)

Everything we discussed in the previous sections is simulation-related, and the simulation only gives us particle positions, which means there is no surface that could be rendered.

Only drawing spheres at the particle positions wouldn't look like water, but more like a pool of balls. One solution to that would be building a real mesh out of the particles, for example with *marching cubes* [12], but this would be expensive because its performance cost scales with the amount of fluid.

However, there is a solution to that: It is called *Screen space fluid rendering*, or SSFR for short. It's a rendering method, which scales not with the amount of fluid, but instead with the number of pixels. Marching cubes, for example, reconstructs a surface in three dimensions. This is needed to render a mesh, but SSFR does this in two dimensions at the resolution of the screen. With that, we can get rid of one dimension in our calculations and gain more performance. This makes it very suitable for real-time applications. Also, the original PBF paper uses it.

Its pipeline consists of five stages [12]. We describe the four that this thesis uses. The fifth is a noise stage that this thesis does not implement. The **first stage** renders all particles as spheres but keeps just the nearest depth value for every pixel. The **second stage** focuses on smoothing out the depth buffer so that the individual spheres merge into one continuous surface. The **third stage** goes through all the pixels again and determines how much fluid lies behind each pixel, which is called the *thickness*. The **last stage** is the reconstruction stage: Here, the surface normal is reconstructed from the smoothed depth from the second stage, and the pixels are shaded by distorting the scene behind them. Moreover, color is absorbed based on the thickness, and reflection and a specular highlight are added.

All stages are important, but the second one is the most important: Without smoothing out the depth buffer, you would still be able to see the spheres instead of a fluid surface. This smoothing is done with a blur, but not a normal one, because a normal blur would blur across the silhouette of the fluid and destroy its outline. To preserve the edges, an edge-preserving blur or a curvature-based method is used [12].

2.8 Level of Detail (LOD)

Level of Detail, LOD for short, is a method that focuses on improving performance while the simulation is running. Its idea is to get rid of small and hidden details that someone most likely wouldn't even perceive.

To achieve that, we use the projected screen size [13], which describes how much of the screen an object covers. Objects that are further away appear smaller and therefore need less detail.

In a fluid, fewer particles are needed in regions where the geometric complexity is low [14], such as deep inside the fluid volume or near a thick flat surface. Therefore, instead of varying the detail continuously, two discrete levels of detail are often already enough [15], because the flow separates into regions with visible small-scale surface effects and regions where large volumes move without producing complex structures.

We cover two types of Level of Detail in this thesis: *rendering LOD* and *simulation LOD*. The rendering system reacts to the position of the camera, while the simulation system reacts only to the state of the fluid. This means the simulation stays identical no matter where the camera is, which is what makes the benchmark results comparable between machines.

3 Related Work

3.1 Fluid simulation in Unity

There are several fluid simulation solutions available for the Unity engine, both as commercial products and as open-source projects.

Obi Fluid [2] is a commercial particle physics package that simulates fluids in two and three dimensions and offers adjustable properties such as surface tension, stickiness and vorticity.

Zibra Liquid [3] is also commercial and combines a custom physical solver with machine-learning-based representations of objects.

Both are closed source, which means their solvers cannot be inspected or verified against the literature, and neither product page states the method it is based on.

The remaining commercial options, such as **FluXY**, **InfiniRIVER**, **Fluid Dynamics** and **Fluid Frenzy**, are two-dimensional or height-field solvers and therefore do not simulate three-dimensional fluid dynamics at all.

There are open-source implementations of SPH for Unity, but they are demonstrations rather than documented frameworks.

The most important gap is that none of these solutions publishes systematic performance data across different hardware. Without such data there is no basis for judging which particle counts are actually achievable on consumer GPUs, which is the question this thesis answers.

3.2 Academic work on fluid simulation in Unity

The closest academic work to this thesis is the framework by Waseem and Hong [16], which implements a GPU-accelerated fluid simulation in Unity using compute shaders. Their setup is very similar to the one we used in this thesis: the same engine generation, the same render pipeline, Direct3D 12 and the same thread group size of 64.

The difference lies in the solver, because they use weakly compressible SPH and derive the pressure from a state equation, while this thesis uses the position-based approach described in Section 2.4.

Their main contribution is not the fluid model but the neighbor search. They combine a *count sort* with a *parallel prefix scan*, and they use a *Structure of Arrays memory layout* that improves the memory access pattern. [16]

The most important difference for this thesis is the evaluation: they measure their framework on a single machine, while this thesis measures several GPUs across ten scenes and two level of detail configurations.

They report up to one million particles [16], which is considerably more than the maximum of 127,526 used in this thesis, but the two numbers are not directly comparable, since a position-based solver performs several constraint iterations per simulation step and therefore does more work per particle than a state equation solver.

3.3 Alternative approaches to incompressibility

PBF is not the only method that enforces incompressibility in a particle-based fluid.

There is **Weakly compressible SPH (WCSPH)**, for example, which derives the pressure from a state equation [17].

Predictive-corrective incompressible SPH (PCISPH) does it differently: It predicts the density error and corrects the pressure iteratively [18].

Implicit incompressible SPH (IISPH) solves for the pressure field implicitly [19].

Divergence-free SPH (DFSPH) enforces a constant density and a divergence-free velocity field at the same time [20].

Incompressible SPH (ISPH) uses a projection method and solves a pressure Poisson equation instead of using a state equation [21].

At first glance, these methods may seem different from each other, but they are less separate than they appear: a recent survey by Koschier et al [6] shows that PBF is equivalent to PCISPH, and that PBF, PCISPH, IISPH and DFSPH can all be understood as Jacobi solvers of the same pressure Poisson equation.

PBF was chosen for this thesis because it remains stable at the large and fixed timestep that a game engine imposes.

4 Implementation

4.1 Overview

In this chapter I will describe what I have been working on for the last eight months:

I developed a framework, PBFUnity, which implements the well-known Position Based Fluids algorithm in Unity. It consists of four parts: the **solver**, the **surface renderer**, **two level of detail systems** and a **benchmark harness**.

The solver works with compute shaders, runs **entirely on the GPU** and never sends particle data back to the CPU.

The whole project uses **Unity 6** with the Universal Render Pipeline and requires **Direct3D 12**, and all compute kernels are dispatched with a thread group size of 64.

One simulation step consists of nine compute kernel dispatches, which are described in Section 4.3. The whole simulation runs in the *FixedUpdate* function with a fixed timestep of 0.02 seconds, which corresponds to the requirement described in Section 2.3.

4.2 GPU compute in Unity

4.2.1 Compute shaders and buffers

For the physics computation, I used the compute shaders that are provided by Unity. With them I was able to run code directly on the GPU, outside the normal rendering pipeline. A compute shader contains one or more kernels, and to start them, a dispatch call from the CPU specifies how many thread groups should run.

The data used by the solver is stored in compute buffers, which are allocated once and stay on the GPU for the whole simulation. I assigned a thread group size of 64 to all kernels in this project. A dispatch launches one thread per particle, so the number of groups is the particle count divided by 64 and rounded up. Threads in the last group that have no particle assigned to them exit immediately: Each kernel

begins with a check that returns immediately if the thread index is greater than or equal to the particle count.

Because all buffers need to be allocated before the simulation starts, I had to specify the maximum number of particles that can be spawned in the scene first. This value doesn't change after the simulation has started, because changing it would mean that I would have to allocate the memory at runtime, which is inefficient.

4.2.2 Ping-pong buffers

When multiple threads are running at the same time and want to access data in a compute buffer, they can get in each other's way. That means if thread A wants to read the content of buffer C, but thread B wants to write to it and goes first, then thread A would read a different value than it should. The reason why this happens is because the order in which threads operate is not defined.

To fix this issue, I implemented so-called **Ping-pong buffers**: Instead of relying on one compute buffer, I used two. One is for reading and the other for writing. After each dispatch both buffers are switched and therefore nothing gets mixed up.

4.2.3 No transfer back to the CPU

I ensured that once the particles are uploaded at the start of a scene, no particle data is ever read back to the CPU. The positions are copied into a separate buffer that is used directly by the renderer, and therefore the data stays on the GPU during the whole simulation.

But this means that the water can never affect other objects in the scene, since nothing on the CPU ever learns where the water is. Nothing can float on the water or be pushed by it. I didn't solve this problem in this thesis and will discuss it further in Section 6. However, the opposite direction does work: Solid objects can push the water, which is described in Section 4.5.3.

4.2.4 The fixed timestep

As mentioned before, the simulation runs in the *FixedUpdate* function. This function is called by Unity at a fixed rate and not based on the actual frame rate. I used a **fixed timestep** of 0.02 seconds, which results in 50 simulation steps per second.

In addition, I set the **maximum allowed timestep** to the same value, so that Unity can never perform more than one simulation step per frame. This ensures that every system simulates the same workload, because a slower machine runs in slow motion instead of skipping simulation steps.

4.2.5 Spatial grid is Lagrangian

The spatial grid described in Section 2.6 stores only particle indices and is rebuilt in every step, so its purpose is to find neighbors and not to store their values. The solver is therefore Lagrangian and not Eulerian, as explained in Section 2.1.

4.3 The solver

I implemented the solver as a single compute shader that contains nine kernels in total. Eight of the nine kernels follow a specific run order and are dispatched every simulation step. While implementing the simulation step, I followed the structure given by Macklin and Müller [4]. I set the fixed number of iterations in the constraint solve loop to three. The solver therefore follows this algorithm:

Algorithm 1 One simulation step of the solver

```

1: for all particles  $i$  do                                     ▷ PredictPosition
2:    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{g}$ ,  $\mathbf{p}_i^* \leftarrow \mathbf{p}_i + \Delta t \mathbf{v}_i$ 
3: end for
4: clear all grid cells                                         ▷ ClearGrid
5: for all particles  $i$  do                                     ▷ BuildSpatialGrid
6:   insert  $i$  into the cell that contains  $\mathbf{p}_i^*$ 
7: end for
8: for  $iter = 1$  to solverIterations do
9:   for all particles  $i$  do                                     ▷ CalculateLambda
10:    calculate  $\rho_i$  from the 27 neighboring cells, then  $\lambda_i$ 
11:   end for
12:   for all particles  $i$  do                                     ▷ CalculatePositionUpdate
13:    calculate  $\Delta \mathbf{p}_i$  including  $s_{corr}$ 
14:   end for
15:   for all particles  $i$  do                                     ▷ UpdatePositions
16:     $\mathbf{p}_i^* \leftarrow \mathbf{p}_i^* + \Delta \mathbf{p}_i$ , then keep it inside the aquarium and outside all solid boxes
17:   end for
18: end for
19: for all particles  $i$  do                                     ▷ UpdateVelocities
20:    $\mathbf{v}_i \leftarrow (\mathbf{p}_i^* - \mathbf{p}_i) / \Delta t$ , apply XSPH viscosity, and vorticity confinement unless skipped by the
    level of detail
21:    $\mathbf{p}_i \leftarrow \mathbf{p}_i^*$ 
22: end for
23: for all particles  $i$  do                                     ▷ ApplyAquariumBoundaries
24:   reflect the velocity of particles that touch a wall or a box
25: end for

```

Two details worth mentioning: The *ClearGrid* kernel is dispatched over grid cells, not particles. That's the reason why I didn't put it in a loop in the pseudocode above. And I added just eight kernels instead of nine to the pseudocode: *CopyPositions* is missing. The reason is that this kernel is dispatched from the render path instead of the simulation step.

4.3.1 Building the neighbor grid

Because the particles move between simulation steps, the grid has to be cleared and rebuilt at the beginning of each step:

To rebuild the grid properly, each particle calculates the cell that contains its predicted position. The three-dimensional cell coordinate resulting from that is then converted into an index by a hash function that multiplies the coordinates by three large prime numbers [11]. The particle then writes its own index into that cell, and an atomic counter makes sure that two particles writing into the same cell do not overwrite each other:

Algorithm 2 Inserting a particle into the spatial hash grid

```

1:  $c \leftarrow \lfloor \mathbf{p}_i^* / h \rfloor$  ▷ the cell that contains the particle
2:  $k \leftarrow ((c_x \cdot q_1) \oplus (c_y \cdot q_2) \oplus (c_z \cdot q_3)) \bmod \text{tableSize}$ 
3:  $n \leftarrow \text{atomicAdd}(\text{cellCount}[k], 1)$  ▷ returns the value before the increment
4: if  $n < \text{MAX\_PER\_CELL}$  then
5:    $\text{cellParticles}[k \cdot \text{MAX\_PER\_CELL} + n] \leftarrow i$ 
6: end if

```

Here, q_1 is 73856093, q_2 is 19349663 and q_3 is 83492791. MAX_PER_CELL is set to 256 and determines how many indices a cell can hold.

4.3.2 Calculating the density and the multiplier

The kernel *CalculateLambda* iterates through the 27 neighboring cells and sums the density contribution of every neighbor within the smoothing radius h . The currently selected particle's contribution is added separately, because the loop skips it.

At the same time the kernel builds up the denominator of the multiplier, which needs both cases of the constraint gradient described in Equation 8. The case for the neighbors is summed inside the loop as a squared magnitude for each neighbor, while the case for the particle itself is collected as a single vector and squared once after the loop has finished.

At the end, the denominator is clamped to a minimum value and the multiplier is calculated.

Algorithm 3 Calculating the density and the multiplier for particle i

```

1:  $\rho_i \leftarrow m_i W(\mathbf{0}, h)$  ▷ a particle contributes to its own density
2:  $D \leftarrow 0, \mathbf{s} \leftarrow \mathbf{0}$ 
3: for all particles  $j$  in the 27 neighboring cells,  $j \neq i$  do
4:    $r \leftarrow \|\mathbf{p}_i^* - \mathbf{p}_j^*\|$ 
5:   if  $r < h$  then
6:      $\rho_i \leftarrow \rho_i + m_j W(\mathbf{p}_i^* - \mathbf{p}_j^*, h)$ 
7:      $\mathbf{g} \leftarrow \nabla W(\mathbf{p}_i^* - \mathbf{p}_j^*, h)$ 
8:      $D \leftarrow D + \|\mathbf{g}\|^2$  ▷ the case  $k = j$ 
9:      $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{g}$  ▷ collected for the case  $k = i$ 
10:   end if
11: end for
12: add the mirror contributions of all walls and boxes to  $\rho_i$ ,  $D$  and  $\mathbf{s}$ 
13:  $D \leftarrow (D + \|\mathbf{s}\|^2) / \rho_0^2$ 
14:  $D \leftarrow \max(D, D_{\min})$  ▷ prevents a division by a very small number
15:  $C_i \leftarrow \max(\rho_i / \rho_0 - 1, 0)$ 
16:  $\lambda_i \leftarrow \text{clamp}(-C_i / D, -1000, 0)$ 

```

4.3.3 Calculating the position correction

The kernel *CalculatePositionUpdate* iterates through the neighbors again and sums the correction from the multiplier of the particle and those of its neighbors. Additionally, the kernel adds the *artificial pressure* term while summing, and the CPU calculates $W_{\Delta \mathbf{q}}$, the kernel value at the reference distance, so that it does not have to be evaluated for every pair.

This results in a position correction, which is stored and applied later by the kernel *UpdatePositions*.

Algorithm 4 Calculating the position correction for particle i

```

1:  $\Delta \mathbf{p}_i \leftarrow \mathbf{0}$ 
2: for all particles  $j$  in the 27 neighboring cells,  $j \neq i$  do
3:    $r \leftarrow \|\mathbf{p}_i^* - \mathbf{p}_j^*\|$ 
4:   if  $r < h$  then
5:      $s_{\text{corr}} \leftarrow -k (W(\mathbf{p}_i^* - \mathbf{p}_j^*, h) / W_{\Delta \mathbf{q}})^n$ 
6:      $\Delta \mathbf{p}_i \leftarrow \Delta \mathbf{p}_i + \frac{1}{\rho_0} (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla W(\mathbf{p}_i^* - \mathbf{p}_j^*, h)$ 
7:   end if
8: end for
9: add the mirror contributions of all walls and boxes to  $\Delta \mathbf{p}_i$ 
10:  $\Delta \mathbf{p}_i \leftarrow \alpha \Delta \mathbf{p}_i$  ▷ under-relaxation
11:  $\Delta \mathbf{p}_i \leftarrow \text{clamp}(\Delta \mathbf{p}_i, -0.3h, 0.3h)$ 

```

The factor α is set to 0.85 and slightly reduces every correction.

4.3.4 Updating the velocity

After the solver iterations are done, the kernel *UpdateVelocities* calculates the velocity for the current particle using the difference between the corrected position and the position at the beginning of the step.

Also, it adds the XSPH viscosity to that velocity so that neighboring particles move more coherently. Afterwards, it calculates the vorticity confinement force and applies it, unless the particle is skipped by the simulation level of detail described in Section 2.8.

Every particle calculates its own vorticity, because the force for a surface particle is calculated from the vorticity of its neighbors, and those neighbors are often inside the water. Only applying the force is skipped.

Algorithm 5 Updating the velocity of particle i

```

1:  $\omega_i \leftarrow$  vorticity of particle  $i$ , calculated from its neighbors
2:  $\mathbf{v}_i \leftarrow (\mathbf{p}_i^* - \mathbf{p}_i) / \Delta t$ 
3:  $\mathbf{v}_i \leftarrow \mathbf{v}_i + c \sum_j \mathbf{v}_{ij} W(\mathbf{p}_i^* - \mathbf{p}_j, h)$  ▷ XSPH viscosity
4: if the level of detail is disabled or  $\lambda_i \geq 0$  then
5:    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_i^{\text{vorticity}}$ 
6: end if
7: clamp  $\|\mathbf{v}_i\|$  to  $v_{\max}$ 
8:  $\mathbf{v}_i \leftarrow 0.995 \mathbf{v}_i$  ▷ damping
9: if the particle moved less than  $0.001 h$  in this step then
10:   $\mathbf{v}_i \leftarrow \mathbf{0}$ ,  $\mathbf{p}_i^* \leftarrow \mathbf{p}_i$ 
11: end if
12:  $\mathbf{p}_i \leftarrow \mathbf{p}_i^*$ , store  $\omega_i$  for the next step

```

4.3.5 Boundaries and rendering

I implemented the kernel called *ApplyAquariumBoundaries* and the kernel called *CopyPositions*. *ApplyAquariumBoundaries* reflects the velocity of particles that have reached a wall, and if the wall is moving, the velocity of the wall is added on top of it.

The kernel *CopyPositions* writes the position of each particle into a separate buffer which the renderer reads directly. The render index is used as an offset so that only the visible subset is copied. *CopyPositions* is dispatched from the render path and

not from the simulation step. That's the reason why it isn't in Algorithm 1.

Algorithm 6 Applying the boundaries to particle i

```

1: touched  $\leftarrow$  false
2: for all axes  $a \in \{x, y, z\}$  do
3:   if  $\mathbf{p}_i$  lies outside the aquarium along  $a$  then
4:     move it to one wallSkin inside that wall
5:      $\mathbf{v}_i[a] \leftarrow \mathbf{v}_{\text{wall}}[a] - (\mathbf{v}_i[a] - \mathbf{v}_{\text{wall}}[a]) \varepsilon_r$ 
6:     touched  $\leftarrow$  true
7:   end if
8: end for
9: for all boxes  $b$  do
10:   $(d, \mathbf{n}) \leftarrow$  distance and outward normal of the nearest face of  $b$ 
11:  if  $d < \text{wallSkin}$  then
12:     $\mathbf{p}_i \leftarrow \mathbf{p}_i + (\text{wallSkin} - d) \mathbf{n}$ 
13:     $\mathbf{v}_{\text{surface}} \leftarrow \mathbf{v}_b + \boldsymbol{\Omega}_b \times (\mathbf{p}_i - \mathbf{c}_b)$ 
14:    reflect the part of  $\mathbf{v}_i - \mathbf{v}_{\text{surface}}$  that points into the box, then add  $\mathbf{v}_{\text{surface}}$  back
15:    touched  $\leftarrow$  true
16:  end if
17: end for
18: if touched then
19:    $\mathbf{v}_i \leftarrow \mu \mathbf{v}_i$  ▷ friction
20: end if

```

ε_r is the restitution, μ the friction, $\mathbf{v}_b$, $\boldsymbol{\Omega}_b$ and $\mathbf{c}_b$ the linear velocity, angular velocity and center of the box.

4.4 Deviations from the published method

4.4.1 Differences

Let's talk about the deviations of my algorithm compared to that of the original PBF paper [4]:

Macklin and Müller define the density constraint as $C_i = \rho_i / \rho_0 - 1$, which becomes negative when a particle has too few neighbors. My algorithm clamps it at zero instead, so that only over-dense particles are corrected. I recognized that there is a cost: The fluid resists compression but never resists expansion.

Moreover, the original method adds a constant relaxation parameter ε to the denominator of the multiplier, which prevents it from becoming very small when a particle is close to separating from its neighbors. I clamp the denominator to a minimum

value instead, which prevents the same division by a number close to zero, but mathematically it isn't the same.

The last difference I recognized is that the paper gives the constant k of the artificial pressure term as a value of 0.1. I use it as a fraction of the density constraint instead, by dividing it by a denominator that is calculated on the CPU. The reason is that s_{corr} is added to the multipliers, and how big those are depends on the smoothing radius h . Therefore, a fixed value for k would perhaps work for one particle size but not for others.

4.4.2 Additions

I added some values that aren't part of the method and only keep the simulation stable at a fixed timestep of 0.02 seconds:

- The velocity of every particle is damped by a factor of 0.995 in every step and clamped to a maximum speed of 15 meters per second. I did this to prevent the water from jumping in some places.
- Particles that moved less than a thousandth of the smoothing radius h are set to rest, so that settled water doesn't keep jittering.
- The *position correction* is clamped to 0.3 times the smoothing radius h , and the multiplier is clamped to the range from -1000 to 0 . The reason is also to prevent the water from jumping in some places.

One addition of mine deviates a lot from the original and its successors: I multiply every position correction by a factor of 0.85. Neither the original paper nor its successor uses under-relaxation of this kind [4], [22]. I added it while fine-tuning the other values, and those were tuned around it. Removing it now would mean repeating the whole tuning process.

4.5 Boundary handling

As described in Section 2.5, a particle close to a wall loses part of its neighbors, because there is no fluid on the other side of the wall. I handle this in three ways, and which one an object uses depends on its shape: The aquarium is described as six planes, a box-shaped solid as the single face nearest to the particle, and an arbitrary mesh is sampled into static boundary particles.

4.5.1 Geometric boundaries

For these shapes I mirror the particle across the wall [10]. The reflection of a particle at distance d sits at distance $2d$, so I treat it as one extra neighbor at that distance. That neighbor contributes to the density ρ_i , to the gradient sum $\mathbf{s}$ and to the denominator D of the multiplier defined in Section 2.4, and the wall normal gives the direction of its gradient. The contribution is skipped once $2d$ exceeds the smoothing radius h , so a particle far from the wall is not affected at all. The same correction is applied in the position update, where the mirrored neighbor uses only the particle's own λ_i , because a wall doesn't have a constraint.

For a solid box I transform the particle into the box's local space, where the box is axis aligned. There I only have to compare the particle's position with the half size of the box along x , y and z . Each of these three comparisons gives the distance to one pair of faces, and the largest of them belongs to the face that the particle is closest to. That face is the plane I mirror the particle across. I use only that single nearest face, because the mirror treats a wall as an infinite plane and near an edge two faces would both claim the particle and count its density twice.

After the constraint solve, any particle that still ended up inside a box is pushed back out along that same normal, because the constraint alone does not guarantee that it stays outside.

Algorithm 7 Mirror contribution of one wall or box face to particle i

```

1:  $(d, \mathbf{n}) \leftarrow$  distance to the wall and its outward normal
2:  $r \leftarrow 2d$  ▷ the distance to the mirrored particle
3: if  $r \leq r_{\min}$  or  $r \geq h$  then
4:   return ▷ out of range, or the particle sits on the wall
5: end if
6:  $\nabla W_{\text{wall}} \leftarrow \nabla W(r, h) \mathbf{n}$ 

  in CalculateLambda:
7:  $\rho_i \leftarrow \rho_i + m_i W(r, h)$ 
8:  $D \leftarrow D + \|\nabla W_{\text{wall}}\|^2$ ,  $\mathbf{s} \leftarrow \mathbf{s} + \nabla W_{\text{wall}}$ 

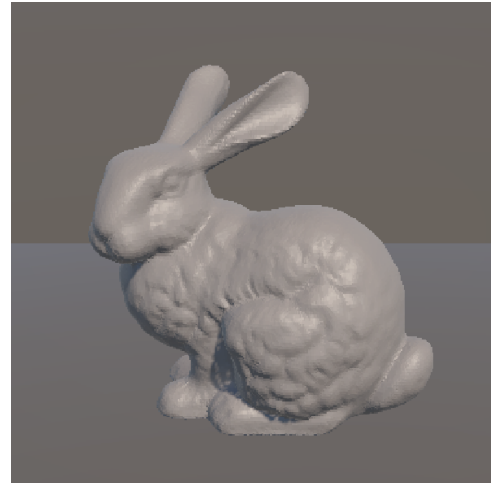
  in CalculatePositionUpdate:
9:  $s_{\text{corr}} \leftarrow -k(W(r, h)/W_{\Delta \mathbf{q}})^n$ 
10:  $\Delta \mathbf{p}_i \leftarrow \Delta \mathbf{p}_i + \frac{1}{\rho_0}(\lambda_i + s_{\text{corr}}) \nabla W_{\text{wall}}$ 

```

4.5.2 Boundary particles for arbitrary meshes

To generate boundary particles, I scatter points over every triangle of the mesh. Bigger triangles need more scattering. This ensures a good coverage over the whole surface. Each point is then rounded to the nearest position on a grid whose spacing is the rest spacing of the fluid. The fluid particles are placed on that same grid, so samples that fall into one cell collapse into a single particle, and a boundary particle never ends up half a spacing away from a fluid particle.

The bunny uses two layers of boundary particles and the armadillo one.

**Figure 1:** Armadillo**Figure 2:** Bunny

Boundary particles share the buffer and the spatial grid from Section 2.6 with the fluid, but an if statement makes every kernel that would move them return early, so they stay where they are and are never rendered.

In the position update only the fluid particle's own multiplier is used. It's the same as with a wall, because a boundary particle has no constraint.

The armadillo alone needs around 7,500 boundary particles, which shows how costly boundary particles can be, as mentioned in Section 2.5.

4.5.3 Moving boundaries

Section 2.5 mentions that boundary particles can move with their solid if implemented, but in my framework only the geometric boundaries move and the boundary particles are static. Geometric boundaries that move have to hand their momentum to the fluid, otherwise they would just displace the water and not push it. I therefore reflect the velocity relative to the wall's own velocity and add the wall's velocity back on afterwards, as shown in Algorithm 5. That means that if the walls aren't moving, the wall's velocity is zero and therefore it reduces to a plain reflection.

Only one direction of interaction works in my framework: solids move water, but water cannot move solids, because no particle data ever returns to the CPU.

4.5.4 The wall skin

During development the fluid formed a film one particle thick that covered the whole aquarium wall, while the water above it dropped off normally. The cause was that particles were clamped exactly onto the wall plane, which puts the distance d at zero. Both mirror functions return early below $r_{\min}$, so a particle sitting on the plane received no wall force at all and remained there.

I added a clamp to half a rest spacing inside the wall, which is the *wallSkin* in Algorithm 6, so the mirror never sees a distance of zero. This didn't change the mirror correction itself. Also, this skin is used when pushing particles out of boxes.

4.6 Screen space fluid rendering

I implemented the four stages from Section 2.7 as four shaders [12]: *depth*, *thickness*, *blur* and *composite*. The first three are recorded into one command buffer and executed before the camera renders. The composite is drawn later, together with the transparent objects of the scene, because Unity renders all opaque objects first and the composite needs that finished image to refract it.

4.6.1 Depth and thickness

Both passes draw every particle as a quad, which is a flat rectangle made of two triangles, and expand it in view space so it always faces the camera. The fragment shader then treats that quad as a sphere: pixels outside the circle are discarded, and the pixels inside get the depth of the sphere's surface, measured from the camera, instead of the flat quad. If several particles cover the same pixel, only the closest one is kept.

I decide which sphere is nearest with min-blending instead of a depth test. The reason is that this pass runs outside any camera, where the platform's reversed depth conventions did not match what camera rendering does, and the depth test kept the farthest surface instead of the nearest, which made the fluid look inside out. Min-blending has no such conventions, so the smallest depth always survives.

The thickness pass uses the same sphere, but outputs the length of the ray through it and blends additively, so overlapping particles add up to a total thickness per pixel.

4.6.2 Smoothing

The depth is smoothed with a bilateral blur, which is the edge-preserving blur mentioned in Section 2.7. Each sample is weighted by its distance and by how similar its depth is to the center pixel. That second weight is what keeps the outline sharp, because samples on the other side of a depth jump contribute almost nothing.

Instead of sampling a square around every pixel, I blur once horizontally and once vertically, which needs far fewer samples. For a Gaussian this gives exactly the

same result as sampling the whole square. For a bilateral blur it does not, because the depth weight is measured against the center pixel, and in the second pass that center pixel is a different one. My smoothing is therefore an approximation.

The thickness is smoothed with an ordinary Gaussian instead, without the depth weight. On the depth, blurring across the outline is a problem, because the surface would smear into the background. On the thickness it is wanted, because it makes the fluid fade out softly at its edge instead of ending abruptly.

The paper I follow for the pipeline smooths with curvature flow [12], while I use a bilateral filter [23] instead.

4.6.3 Composite

The composite runs once for every pixel on the screen. It first reads the smoothed depth and discards the pixel if there is no fluid at that position. It also discards the pixel if an opaque object of the scene stands in front of the fluid, because then the water is hidden behind it.

After that it needs to know which way the surface faces at that pixel. There is no mesh, so there are no normals to read. Instead I reconstruct the direction from the depth texture by comparing the depth of the pixel with the depth of its neighbors on the left, right, top and bottom. The way those values change gives the direction of the surface. At the outline of the fluid, one of the two neighbors is already the background, and using it would tilt the direction the wrong way. Therefore I take the neighbor whose depth is closer to the center pixel for each axis and ignore the other one.

The shading follows from that direction. The scene behind the water is sampled a second time, but shifted a little along the surface direction and shifted more where the water is thicker, which is what makes the water refract what lies behind it. The color is then absorbed with the Beer-Lambert law using the smoothed thickness, so a ray that travels through more water loses more color. Absorbing red and green is what makes the water look blue. On top of that, a Fresnel term [24] makes the

surface reflect more when it is seen from a flat angle and less when it is seen head-on, which is how real water behaves, and a specular highlight [25] from the main light is added.

At the end the composite writes the depth of the fluid into the depth buffer, so transparent objects that are drawn afterwards are sorted correctly against the water.

4.7 Level of Detail

4.7.1 Rendering level of detail

The rendering system measures how much of the screen the fluid covers and lowers the quality of the screen space passes when it covers only a little. The measurement is taken once per frame and gives the height of the fluid on screen as a fraction of the screen height. Instead of the real bounds of the particles I use the bounds of the aquarium. The reason is that the real bounds would have to be read back from the GPU every frame, and nothing in this framework is ever read back. The aquarium is a good enough estimate and costs nothing.

I defined three quality levels. Each of them sets the **resolution of the offscreen buffers**, how many **blur iterations** the depth and the thickness receive, and how many **samples the blur takes**. The blur is what these levels mainly reduce, because it is by far the most expensive of the screen space passes, since it runs over every pixel several times.

A level only changes once its threshold has been passed by a small margin, and the system moves at most one step at a time. Without that margin the quality would flicker back and forth while the camera sits exactly on a threshold, which is very visible and is known as **popping** [26].

The whole system can also be switched off, so that a benchmark run always renders at the highest level and the quality never changes underneath the measurement.

4.7.2 Simulation level of detail

The simulation system **skips exactly one** thing: the **vorticity confinement force** for the particles inside the fluid. Everything else stays the same. The number of solver iterations, the density, the multiplier, the position correction and the boundary handling are all untouched.

The test for which particles are inside is free. The multiplier λ_i is zero exactly when a particle is not over-dense, because I clamp the constraint at zero as described in Section 4.4. A particle at the free surface has neighbors on one side only, so it is under-dense and its multiplier is zero. An interior particle is slightly over-dense, so its multiplier is negative. That gives the test without an extra loop and without an extra buffer.

The vorticity confinement force is also the only term that is safe to leave out. It costs a full walk over all neighbors, so skipping it saves real work, and it only adds back swirls that the solver damped away. Dropping it can therefore calm a particle down, but it can never stir one up.

There is a limit to how far this can go. Every particle still calculates its own vorticity, even the ones that never use it, because the force of a surface particle is calculated from the vorticity of its neighbors, and those neighbors are usually inside the fluid. Only applying the force is skipped, never calculating the vorticity. That is the ceiling of this level of detail.

The two methods this idea comes from also change the size of the particles [14], [15], which needs a different smoothing radius for every particle and therefore a change to the whole solver. I keep one radius for all particles and borrow only their criterion of where detail is needed.

4.8 Benchmark harness

A single run is made of **two parts**: a **short warm-up** that is discarded, and a fixed number of seconds in which **the frames are measured**. The warm-up exists because the first frames after a spawn are always slower while buffers and shader vari-

ants are still settling. I measure differently in each scene, so that it is long enough that the water has finished collapsing and a laptop has had time to throttle, so the result describes a sustained load and not a burst. The number of frames is whatever the machine manages in that window. That is the point of measuring in seconds: every machine is given the same amount of time instead of the same amount of work.

The harness runs ten scenes one after another, beginning with an empty aquarium as a smoke test and ending with the heavy ones.

I report a run in the same terms as Table 1 of the original PBF paper [4]: the **particle count**, the **steps per frame**, the **solver iterations** per step, the **milliseconds per simulation step** and the **frames per second**. An average hides stutter though, and a single dropped frame is exactly what breaks a promise of sixty frames per second. I therefore also report the **frame rate** that only **one frame in a hundred fell below**, the **single worst frame of the run**, and the **share of frames that were fast enough** to reach the target.

The last value is the **simulation speed**. Because the timestep is fixed and steps are never skipped, a machine that cannot keep up does not drop simulation work. It runs the water in slow motion instead. The simulation speed says how much of real time the simulation kept up with, where one means it kept up and a third means the scene played at roughly a third of real speed. Without that value a slow machine could show an acceptable frame rate while the water moves far too slowly.

At the end a **summary screen** shows one line per scene and the mean of their shares on target. I take the mean over the scenes rather than pooling every frame, so that each scene weighs the same and the long ones do not dominate. Every run also writes its own file with one row per measured frame, together with a header that records the **graphics card** and its **memory**, the **graphics API**, the **processor**, the **operating system**, the **Unity version**, the **resolution**, the **fixed timestep**, the **solver iterations**, the **thread group size** and whether the **level of detail** was switched on. That header is what makes two runs on different machines comparable at all.

One limitation is worth stating here rather than leaving it for the reader to find. The time per step is the measured wall clock divided by the steps that completed within it, so it is the cost of the solver only when the simulation is what limits the frame. In a scene where the rendering is the bottleneck, it measures the frame instead.

4.9 Build and deployment

The measurements are taken from a standalone build and not from the Editor, because the Editor adds its own overhead and this influences the test results. The two configurations of a benchmark run, with and without level of detail, are selected by a command line flag when the build is started. That flag switches both level of detail systems at once, and it is written into the file name and into the header of every result file, so that a result can never be confused about which configuration produced it.

The project requires **Direct3D 12**, which I found out the hard way during development. Running it on Direct3D 11 doesn't produce any error message. Every particle position simply comes out as the origin, the floor clamp then lifts them all to the same height, and the solver scatters them around that one point. It looks exactly like the simulation exploding, but the physics is correct and it is the structured buffers that fail.

The second requirement is that the four screen space shaders are listed under the **always included shaders** of the project. The renderer looks them up by name while it runs, and no material asset references them, so Unity treats them as unused and removes them from the build. This resulted in the simulation running and still costing frame rate, but nothing was drawn.

Both of these fail **without an exception**, which is what makes them expensive to find. The **player log** is therefore the first place to look, because it names the shaders that could not be found and prints the values the solver started with.

4.10 Known limitations of the implementation

This section collects the weaknesses of the implementation itself. The places where I deviate from the published method are in Section 4.4 and are intentional, while the ones here are not.

The particles are stored as an array of structures, which means the position, velocity, mass and vorticity of one particle sit next to each other in memory. A kernel that only needs the position therefore still **pulls the whole structure into the cache**. Storing each property in its own buffer suits the way a graphics card reads memory much better, and the literature reports a clear advantage for it at large particle counts [16].

The particles are also **never reordered** after the grid has been rebuilt. Two particles that are neighbors in space can therefore sit far apart in the buffer, so a neighbor loop reads scattered addresses instead of one block. The usual fix is to sort the particles by their cell after every rebuild, so that particles close together in space are also close together in the buffer. I did not measure how much this costs here, because that would require building the sorted version to compare against.

The spatial hash has **two weaknesses** of its own. Each cell **can hold at most 256 particle indices**, and anything beyond that is dropped without any warning, so in a very dense region a particle can miss neighbors that are in fact inside its radius. **Hash collisions** make this worse, because two different cells that receive the same index have to share those same slots. My hash table size is also a power of two, while the paper the hash function comes from [11] reports that a table size which is itself a prime number distributes the entries better. A grid built with a prefix sum would remove the fixed limit per cell entirely.

Two further limitations are already described where they arise. Section 4.2.3 explains that the water **cannot affect any other object in the scene**, because no particle data is ever sent back to the CPU. Section 4.2.1 explains that the maximum **number of particles has to be fixed** before the simulation starts, because the buffers are allocated once and never resized.

5 Evaluation

5.1 Test setup

I benchmarked the framework on **six machines** with different graphics cards, from an **RTX 3060** up to an **RTX 4090**. Every machine ran the same release build with the same settings: 1920 by 1080, vertical synchronization off, Direct3D 12, a fixed timestep of 0.02 seconds and 64 threads per group. Each machine ran the full sequence of ten scenes twice, once with the level of detail switched off and once with it on, which is selected by the command line flag described in Section 4.9.

The runs were carried out by other people on their own computers, so the processors, the amount of system memory and the operating system differ between them. Because there is exactly one machine per graphics card, nothing in this evaluation separates the graphics card from the rest of the computer it sits in. I therefore report results per machine and not as a property of a graphics card in general.

Graphics card	VRAM	Processor
NVIDIA GeForce RTX 3060	12 GB	AMD Ryzen 7 5800X
NVIDIA GeForce RTX 3080	12 GB	AMD Ryzen 9 5950X
NVIDIA GeForce RTX 4070 Ti	12 GB	AMD Ryzen 7 5700X
NVIDIA GeForce RTX 5070	12 GB	AMD Ryzen 7 7800X3D
NVIDIA GeForce RTX 5070 Ti	16 GB	AMD Ryzen 7 5800X
NVIDIA GeForce RTX 4090	24 GB	AMD Ryzen 9 7900X

Table 1: The six machines. Every one ran the same build at 1920 by 1080 with vertical synchronization off.

5.2 The scenes

The benchmark has **ten scenes, from an empty tank to 127,526 particles**. The point of that range is to bracket the limit rather than to show a best case. A benchmark where every machine passes everything would say nothing about where the framework stops working.

The empty aquarium contains no particles at all and serves as a rendering baseline. Two of the scenes contain boundary particles, the bunny with 5,189 and the armadillo with 7,526, and in both of them those particles take part in every neighbor search like any other particle.

Each scene is measured for a fixed number of seconds rather than a fixed number of frames, so that a fast and a slow machine are given the same amount of simulated time. The length differs per scene because the scenes need different amounts of time to develop: the dam takes fifty seconds to drain, while the tsunami takes only eight.

Scene	Particles	Boundary	Seconds
EmptyAquarium	0	0	5
RainyDay	40000	0	15
SimpleSlide	40000	0	12
Tsunami	50000	0	8
Meteoroids	55188	0	8
ScalingAquarium	55188	0	15
FlowingDam	70000	0	50
BunnyInTheMiddle	85189	5189	30
BunnyDrop	79032	0	30
ArmadilloSplash	127526	7526	22

Table 2: The ten benchmark scenes, in the order they are run. The boundary column is the part of the particle count that samples a mesh surface and never moves. The seconds column is the measured time.

5.3 Metrics

The most important number to prove my thesis' claim is the **share of frames that were fast enough for 60 frames per second**, which means a frame time at or **below 16.7 milliseconds**. I chose that rather than an average because an average hides stutter, and a claim of 60 frames per second is broken by single slow frames and not by the mean. Next to it I report the **frame rate** that only **one frame in a hundred fell below**, and the **slowest single frame of the run**.

The second metric is the **simulation speed**. Because the timestep is fixed and steps are never skipped, a machine that cannot keep up does not drop simulation work but runs the water in slow motion instead. A value of one means the simulation kept up with real time, and a value of a third means the scene played at roughly a third of real speed. Without this value a machine could show an acceptable frame rate while the water moves far too slowly, so the two numbers have to be read together.

The headline value for a machine is the unweighted mean of its ten per-scene percentages. It is intentionally not the share over all frames pooled together, because the cheap scenes produce many more frames than the expensive ones and would then dominate the result. On the slowest machine the two differ by fifteen points, which shows that the choice matters.

The empty aquarium always scores 100 percent and makes up a tenth of that mean, so it inflates every result. Leaving it out lowers the numbers by between zero and three points and changes no verdict, and I report it separately as a rendering baseline for that reason.

5.4 Results

On five of the six machines, nine of the ten scenes finish at or very close to 100 percent. Almost all of the spread comes from the armadillo scene at 127,526 particles. This is what I wanted: nine scenes the hardware can handle, and one that shows where the limit is.

Scene	3060	3080	4070 Ti	5070	5070 Ti	4090
EmptyAquarium	100	100	100	100	100	100
RainyDay	78.37 / 80.78	100	100	100	100	100
SimpleSlide	75.05 / 100	100	100	100	100	100
Tsunami	51.70 / 60.62	100	100	100	100	100
Meteoroids	79.38 / 86.41	100	100	100	100	100
ScalingAquarium	44.51 / 52.70	100	100	100	100	100
FlowingDam	77.40 / 78.15	100	100	100	100	100
BunnyInTheMiddle	88.41 / 88.51	96.56 / 96.77	99.99 / 100	100	100	100
BunnyDrop	61.43 / 64.65	99.91	99.93	99.93	99.96	99.97
ArmadilloSplash	5.00 / 11.84	39.06 / 39.47	42.73 / 44.41	86.97 / 94.93	77.39 / 85.28	100

Table 3: Share of frames at or below 16.7 milliseconds, per scene and machine, given as level of detail off / on. A single value means both configurations gave the same result. The machines are those of Table 1.

The verdict separates into three groups. The 4090 machine holds 60 frames per second in every scene, and its slowest single frame in the armadillo scene was 14.08 milliseconds, so it never crosses the line at all. The two 5070 machines pass nine scenes and reach between 85 and 95 percent in the tenth. The 4070 Ti and 3080 machines also pass nine scenes but only reach around 40 percent in the armadillo scene. The 3060 machine misses the target in nine of the ten scenes.

Section 1.2.2 defines mid-range as the graphics cards between the RTX 3070 and the RTX 5070, and sets the target at 90 percent of frames on time. Three of the six machines fall into that range, and all three reach the target as a mean over the ten scenes, even with the level of detail switched off: the 3080 at 93.55 percent, the 4070 Ti at 94.27 percent and the 5070 at 98.69 percent.

The simulation speed sharpens that picture in the one scene where it matters:

Machine	Simulation speed	Frames on target	99th percentile
RTX 3060	0.33	5.00%	11.0 FPS
RTX 3080	0.76	39.06%	34.2 FPS
RTX 4070 Ti	0.84	42.73%	38.7 FPS
RTX 5070	1.00	86.97%	52.7 FPS
RTX 5070 Ti	1.00	77.39%	49.5 FPS
RTX 4090	1.00	100.00%	74.6 FPS

Table 4: The ArmadilloSplash scene with the level of detail switched off. A simulation speed of one means the water kept up with real time. The last column is the frame rate that only one frame in a hundred fell below.

The three slower machines are not only missing the frame rate in that scene, they are not simulating in real time either. The three faster ones hold real time throughout the whole benchmark.

5.5 Effect of the level of detail

The level of detail improves the result on every machine, but by very different amounts:

Machine	LOD off	LOD on	Gain
RTX 3060	66.12	72.36	+6.24
RTX 3080	93.55	93.62	+0.06
RTX 4070 Ti	94.27	94.43	+0.17
RTX 5070	98.69	99.49	+0.80
RTX 5070 Ti	97.73	98.52	+0.79
RTX 4090	100.00	100.00	+0.00

Table 5: Share of frames on target with the level of detail off and on, as the mean over the ten scenes. The gain is the difference between the two, in percentage points.

Four of the six machines gain less than a fifth of a point, so this is not a general speedup and I do not claim it as one. The pattern behind the numbers is that the level of detail only moves frames that already sit close to the 16.7 millisecond line.

The 4090 machine is never above that line, so there is nothing for it to move and the gain is exactly zero. The two 5070 machines sit right at the line in the armadillo scene and gain about eight points there. The 4070 Ti and 3080 machines sit far above it, at

median frame times of 22.8 and 26.1 milliseconds, and removing the vorticity work only brings them to 20.8 and 24.0 milliseconds, which crosses nothing. The 3060 machine gains the most overall, but not because of the armadillo scene. It gains because five of the other scenes sit near the line for it, where every faster machine is already at 100 percent. In one of them, the slide, it goes from 75 percent to 100.

5.6 Frame pacing

One result at first looks like a measurement error. The 5070 Ti machine scores lower than the 5070 machine, 98.52 against 99.49, although it is faster at both ends of the distribution: its median frame in the armadillo scene is 1.65 milliseconds against 10.01, and its slowest frame is 18.23 against 20.17.

The explanation is in the shape of the distribution rather than in the speed.

Machine	Under 5 ms	5 to 16.7 ms	Over 16.7 ms
RTX 3060	6.0	5.8	88.2
RTX 3080	32.2	7.2	60.5
RTX 4070 Ti	37.4	7.0	55.6
RTX 5070	31.9	63.0	5.1
RTX 5070 Ti	72.9	12.4	14.7
RTX 4090	89.1	10.9	0.0

Table 6: How the frame times of the ArmadilloSplash scene are distributed, with the level of detail switched on. The values are the share of frames in each band, in percent.

Five of the six machines split their frames into two groups: **many very fast frames**, plus a **smaller group of much slower ones**. The 5070 machine is the only one that spreads the same work across the middle band, and it happens to spread it just below the threshold. My metric therefore counts those frames as successes, while on the 5070 Ti machine the slower group falls above the threshold and counts as failures.

So in this scene the metric is measuring how the work is distributed over frames and not how much work the machine does. It can rank a faster card lower. Whether the spreading comes from driver queueing, from pipelining inside the graphics card

or from how Unity schedules the fixed update against the render loop cannot be decided from this data.

The 4090 machine shows the same two-group shape with the slower group lying entirely below the line.

5.7 Limitations of the evaluation

There is one machine per graphics card, so the graphics card and the rest of the computer cannot be separated. A difference between two machines is a difference between two whole systems.

The runs were made on other people's computers, so I could not control what else was running on them.

The warm-up window of one second is too short to remove all the slow frames at the start of a run. On some machines the empty aquarium still shows isolated slow frames up to five seconds into the run, at several milliseconds against a baseline below one millisecond. I did not investigate where they come from. Those frames never come close to 16.7 milliseconds, so they do not change any result here, but the window is too short to say they have been excluded.

The time per simulation step is the measured wall clock divided by the steps that completed inside it, so it is the cost of the solver only when the simulation is what limits the frame.

Finally, the scenes are measured for different lengths of time and therefore contribute different numbers of frames. Taking the mean over the scenes instead of over the frames is what keeps that from distorting the result.

6 Future Work

6.1 Two-way coupling

The framework **moves water in one direction only**. Solid objects push the fluid, but the fluid never pushes anything back. The cause is the decision described in Section 4.2.3 to never send particle data back to the CPU, which means nothing outside the solver ever learns where the water is or what force it applies.

Making it work in both directions would mean summing the force each particle applies to a solid on the GPU and reading that sum back once per frame, then handing it to the physics engine as a force on the rigid body. Reading back is what makes this expensive, because the CPU has to wait until the GPU is finished. Unity provides an asynchronous readback for exactly this situation, which delivers the data a few frames later instead of stopping the frame. A delay of a few frames would be acceptable for something floating on the surface, but it would be visible in a fast collision. This is the single largest limitation for anyone who wants to use the framework in a game.

6.2 Signed distance fields instead of boundary particles

Arbitrary meshes are the only shapes that still need boundary particles, and they are the expensive case. The armadillo alone adds 7,526 particles which occupy buffer space and take part in every neighbor search, while a box of any size costs nothing but a few instructions.

A signed distance field would remove them [4]. It stores the distance to the surface of a mesh in a three-dimensional texture, so the solver would look up a distance and a normal instead of summing over thousands of static particles. That is exactly what my mirror correction already expects, because it takes a distance and an outward normal as its input, so the solver itself would barely change. Unity's Visual Effect Graph package can bake a mesh into such a texture inside the editor, which removes the hardest part of the work.

One design question would still have to be answered: A three-dimensional texture cannot be placed in a structured buffer the way the box data is, so either the solver supports one such object at a time or the textures have to be bound as an array. I did not build this during the thesis because changing how boundaries behave would have invalidated every measurement I had already collected.

6.3 A faster memory layout

The three limitations in Section 4.10 are all about how particle data is arranged in memory, and each of them has a known fix. Each property could be stored in its own buffer instead of keeping one structure per particle [16]. The particles could be sorted by their cell after every rebuild of the grid. The fixed limit of 256 indices per cell could be replaced by a grid built with a prefix sum, which has no limit at all.

What makes these attractive is that all three change how the data is stored and not what is computed. The simulation would produce the same result, so the same scenes could be measured before and after the change and the difference would be the cost of the layout alone.

6.4 Quality driven by the frame budget

Both level of detail systems react to something other than the time a frame actually takes. The rendering system reacts to how much of the screen the fluid covers, and the simulation system reacts to the state of the fluid. Neither of them ever measures a frame.

My evaluation shows why that is a limitation. The level of detail only improved the result where frame times already sat close to the target, and it could not help the machines whose frames were far above it. A better system would watch how long each frame takes. When the fluid becomes too expensive, it would lower the solver iterations, the particle count or the resolution of the screen space passes until the frames are fast enough again. Such a system would react to the machine it runs on instead of to the scene. That is also how a game would use this framework, because a game decides the budget first and then fits the workload into it.

6.5 Bubbling effect

While I was testing and adjusting the values for the simulation, I noticed a bubbling effect in the middle of the aquarium when it was halfway filled with water. I suspect that either my solver has a flaw in it or the way the walls of the aquarium reflect the water is not quite right. I tried turning the level of detail on and off, but it didn't change anything, so the level of detail is not the cause. Either way, this needs to be fixed in the future.

7 Conclusion

In this thesis I built PBFUnity, a framework that implements Position Based Fluids in Unity 6 and runs entirely on the graphics card. It consists of a solver of nine compute kernels, a screen space renderer, two level of detail systems and a benchmark harness. Once a scene has started, no particle data is sent back to the processor.

I measured it on six machines with graphics cards from an RTX 3060 up to an RTX 4090. From the RTX 3080 upward it holds 60 frames per second in nine of the ten scenes, and on the 4090 machine in all ten. The limit sits at the armadillo scene with 127,526 particles, and only the two 5070 machines and the 4090 machine simulate that scene at real speed. That clears the target of 90 percent set in Section 1.2.2.

Two of the results were narrower than I expected. The level of detail improves every machine, but four of the six gain less than a fifth of a percentage point, because it can only move frames that already sit close to the target. The share of frames on target also measures how work is spread across frames and not only how much work there is, which is why one machine scored lower than a slower one. That is why I also report the simulation speed.

The framework contributes no new method. The solver follows Macklin and Müller, and the renderer follows van der Laan and his colleagues. What it adds is an implementation of both inside a widely used game engine, together with performance data across a range of consumer hardware, which the existing fluid solutions for Unity do not publish. Its limits are equally clear: the coupling works in one direction only, the particle data is not arranged for the way a graphics card reads memory, and there is one machine per graphics card.

Position Based Fluids can therefore run at interactive rates inside Unity on ordinary consumer hardware, as long as the particle count is chosen for the weakest machine that has to run it.

References

- [1] Physics Mod. “Minecraft physics mod,” Accessed: Sep. 10, 2026. [Online]. Available: <https://minecraftphysicsmod.com>
- [2] Virtual Method. “Obi fluid.” Unity Asset Store, Accessed: Sep. 10, 2026. [Online]. Available: <https://assetstore.unity.com/packages/tools/physics/obi-fluid-63067>
- [3] Zibra AI. “Zibra liquid, smoke & fire, effects,” Accessed: Sep. 10, 2026. [Online]. Available: <https://www.zibra.ai/zibra-liquid-smoke-effects>
- [4] M. Macklin and M. Müller, “Position based fluids,” *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 1–12, Jul. 2013. DOI: 10.1145/2461912.2461984
- [5] J. F. Price, “Lagrangian and eulerian representations of fluid flow: Kinematics and the equations of motion,” Woods Hole Oceanographic Institution, Woods Hole, MA, USA, 2004. Accessed: Sep. 10, 2026. [Online]. Available: https://www.whoi.edu/science/P0/people/jprice/class/ELreps_small.PDF
- [6] D. Koschier, J. Bender, B. Solenthaler, and M. Teschner, “A survey on SPH methods in computer graphics,” *Computer Graphics Forum*, vol. 41, no. 2, pp. 737–760, May 2022. DOI: 10.1111/cgf.14508
- [7] M. Müller, D. Charypar, and M. Gross, “Particle-based fluid simulation for interactive applications,” in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '03, San Diego, CA, USA, 2003, pp. 154–159.
- [8] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position based dynamics,” *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, 2007. DOI: 10.1016/j.jvcir.2007.01.005
- [9] N. Akinci, M. Ihmsen, G. Akinci, B. Solenthaler, and M. Teschner, “Versatile rigid-fluid coupling for incompressible SPH,” *ACM Transactions on Graphics*, vol. 31, no. 4, 62:1–62:8, Jul. 2012. DOI: 10.1145/2185520.2185558

REFERENCES

REFERENCES

- [10] G. Fourtakas, P. K. Stansby, B. D. Rogers, and S. J. Lind, “An Eulerian–Lagrangian incompressible SPH formulation (ELI-SPH),” *Computer Methods in Applied Mechanics and Engineering*, 2018. DOI: 10.1016/j.cma.2017.09.029
- [11] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. Gross, “Optimized spatial hashing for collision detection of deformable objects,” in *Proceedings of Vision, Modeling, Visualization (VMV)*, Munich, Germany, 2003, pp. 47–54.
- [12] W. J. van der Laan, S. Green, and M. Sainz, “Screen space fluid rendering with curvature flow,” in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, Boston, MA, USA, Feb. 2009, pp. 91–98. DOI: 10.1145/1507149.1507164
- [13] J. H. Clark, “Hierarchical geometric models for visible surface algorithms,” *Communications of the ACM*, vol. 19, no. 10, pp. 547–554, Oct. 1976. DOI: 10.1145/360349.360354
- [14] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas, “Adaptively sampled particle fluids,” *ACM Transactions on Graphics*, vol. 26, no. 3, p. 48, Jul. 2007. DOI: 10.1145/1276377.1276437
- [15] B. Solenthaler and M. Gross, “Two-scale particle simulation,” in *ACM SIGGRAPH 2011 Papers*, ser. SIGGRAPH ’11, New York, NY, USA, Jul. 2011, pp. 1–8. DOI: 10.1145/1964921.1964976
- [16] M. Waseem and M. Hong, “Moving towards large-scale particle based fluid simulation in Unity 3D,” *Applied Sciences*, vol. 15, no. 17, p. 9706, 2025. DOI: 10.3390/app15179706
- [17] M. Becker and M. Teschner, “Weakly compressible SPH for free surface flows,” in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA ’07, 2007, pp. 209–217.

REFERENCES

- [18] B. Solenthaler and M. Pauly, "Predictive-corrective incompressible SPH," *ACM Transactions on Graphics*, vol. 28, no. 3, Jul. 2009. doi: 10.1145/1531326.1531346
- [19] M. Ihmsen, J. Cornelis, B. Solenthaler, C. Horvath, and M. Teschner, "Implicit incompressible SPH," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 426–435, Mar. 2014. doi: 10.1109/TVCG.2013.105
- [20] J. Bender and D. Koschier, "Divergence-free smoothed particle hydrodynamics," in *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, ser. SCA '15, New York, NY, USA, Aug. 2015, pp. 147–155. doi: 10.1145/2786784.2786796
- [21] M. Rezavand, M. Taeibi-Rahni, and W. Rauch, "An ISPH scheme for numerical simulation of multiphase flows with complex interfaces and high density ratios," *Computers & Mathematics with Applications*, vol. 75, no. 8, pp. 2658–2677, Apr. 2018. doi: 10.1016/j.camwa.2017.12.034
- [22] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, "Unified particle physics for real-time applications," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 1–12, Jul. 2014. doi: 10.1145/2601097.2601152
- [23] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Sixth International Conference on Computer Vision*, Jan. 1998, pp. 839–846. doi: 10.1109/ICCV.1998.710815
- [24] C. Schlick, "An inexpensive BRDF model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. doi: 10.1111/1467-8659.1330233
- [25] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '77, New York, NY, USA, Jul. 1977, pp. 192–198. doi: 10.1145/563858.563893

- [26] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics*. San Francisco, CA, USA: Morgan Kaufmann, 2002, ISBN: 978-1-55860-838-2.

AI-based tool	Form of use	Affected parts of the thesis	Remarks
DeepL Translator	translation of text	whole thesis	
Claude (Anthropic)	structuring the thesis, answering questions about scientific papers, drafting pseudocode, correcting English, suggesting wording, analyzing the benchmark data, compiling the bibliography, translating the abstract into German	whole thesis	all wording suggestions rewritten in my own words; pseudocode verified against my own implementation
Elicit	finding papers	whole thesis	

Table 7: Use of AI-based tools in this thesis.

Hinweise zu den offiziellen Erklärungen

1. Die folgende Seite mit den offiziellen Erklärungen

A) Eigenständigkeitserklärung,

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten und

C) Einverständniserklärung über die Bereitstellung und Nutzung der Bachelorarbeit / Masterarbeit
in elektronischer Form zur Überprüfung durch eine Plagiatsoftware

ist unverändert im Wortlaut in die Bachelor- oder Masterarbeit zu übernehmen.

Bitte achten Sie darauf die Erklärungen zu unterschreiben. Ein Scan der Unterschrift ist ausreichend.

zu B)

Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden.

zu C)

Das Einverständnis der dauerhaften Speicherung des Textes ist freiwillig.

Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden.

Weitere Informationen zur Überprüfung von schriftlichen Arbeiten durch die Plagiatsoftware sind im Nutzungs- und Datenschutzkonzept enthalten. Diese finden Sie auf der Internetseite der Universität Bremen.

Notes on the official declarations

1. The following pages with the official declarations

A) Declaration of Authorship

B) Declaration on the Publication of Bachelor's or Master's Thesis

C) Declaration of Consent for the Provision and Use of the Bachelor's Thesis / Master's Thesis
in Electronic Form for Review by Plagiarism Software

is to be adopted unchanged in the wording to the bachelor's or master's thesis.

Please be sure to sign each declaration. A scan is sufficient.

Regarding B)

The consent can be revoked at any time with future effect by notifying the University of Bremen.

Regarding C)

Consent for the permanent storage of the text is voluntary. The consent can be revoked at any time with future effect by notifying the University of Bremen.

Further information on the checking of written work using plagiarism software can be found in the data protection and usage concept. This can be found on the University of Bremen website.

**Eigenständigkeits- und Einverständniserklärung zur Überprüfung mit Plagiatssoftware
sowie die Erklärung zur Veröffentlichung bei Bachelor- und Masterarbeiten****Declarations of Authorship and Consent for Checking with Plagiarism Software and the
Declaration of Publication for Bachelor's and Master's Thesis**

Studierenden-Angaben / Student Information:

Matrikelnr. / Student ID **6155675**Nachname / Surname **Kaynak**Vorname / First Name **Mert**

Titel der Arbeit / Title of Thesis

PBFUnity**A) Eigenständigkeitserklärung / Declaration of Authorship**

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge.
Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht.

I hereby affirm that I have written the present work independently and have used no sources or aids other than those indicated. All parts of my work that have been taken from other works, either verbatim or in terms of meaning, have been marked as such, indicating the source. The same applies to drawings, sketches, pictorial representations and sources from the Internet, including AI-based applications or tools. The work has not yet been submitted in the same or a similar form as a final examination paper.

- ☒ Ich habe KI-basierte Anwendungen und/oder Werkzeuge genutzt und diese im Anhang "Nutzung KI basierte Anwendungen" dokumentiert.

I have used AI-based applications and/or tools and documented them in the appendix "Use of AI-based applications".

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten
Declaration regarding the publication of bachelor's or master's thesis

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

Two years after graduation, the thesis is offered to the archive of the University of Bremen for permanent archiving. The following are archived:

- 1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Masterarbeiten
Master's theses with a local or regional focus, as well as per subject and academic year 10 % of all Master's thesis
- 2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.
Bachelor's thesis for the first and last bachelor's degrees per subject and year.

☒ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I agree that my thesis may be viewed by third parties in the university archive for academic purposes.

☒ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I agree that my thesis may be viewed by third parties for academic purposes in the university archive after 30 years in accordance with para. 2 BremArchivG).

☐ Ich bin **nicht** damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I do not consent to my thesis being made available in the university archive for third parties to view for academic purposes.

C) Einverständniserklärung zur elektronischen Überprüfung der Arbeit auf Plagiate
Declaration of consent for electronic checking of the work for plagiarism

Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Masterprüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden.

Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

Submitted papers can be checked for plagiarism using qualified software in accordance with § 18 of the General Section of the Bachelor's or Master's Degree Examination Regulations of the University of Bremen. For the purpose of checking for plagiarism, the upload to the server is done using the plagiarism software currently used by the University of Bremen.

- ☐ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum oben genannten Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

I agree that the work I have submitted and written will be stored permanently on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution accessed only by the University of Bremen), for the above-mentioned purpose.

- ☒ Ich bin **nicht** damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

I do not consent to the work I submitted and wrote being permanently stored on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution accessed only by the University of Bremen), for the above-mentioned purpose.

Das Einverständnis der dauerhaften Speicherung des Textes ist freiwillig. Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden. Weitere Informationen zur Überprüfung von schriftlichen Arbeiten durch die Plagiatsoftware sind im Nutzungs- und Datenschutzkonzept enthalten. Diese finden Sie auf der Internetseite der Universität Bremen.

Consent to the permanent storage of the text is voluntary. Consent can be withdrawn at any time by making a declaration to this effect to the University of Bremen, with effect for the future. Further information on the checking of written work using plagiarism software can be found in the data protection and usage concept. This can be found on the University of Bremen website.

Mit meiner Unterschrift versichere ich, dass ich die obenstehenden Erklärungen gelesen und verstanden habe und bestätige die Richtigkeit der gemachten Angaben.

With my signature, I confirm that I have read and understood the above explanations and confirm the accuracy of the information provided.

10.09.2026

Datum / Date

Mat Kaymak

Unterschrift als Scan/ Signature