IT UNIVERSITY OF COPENHAGEN

Implementing the Position-Based Material Point Method for Real-Time Liquid Simulation in Unity DOTS

Fabio Mangiameli
fabm@itu.dk
KISPECI1SE

Master's Thesis

M.Sc. Games Tech Track
IT University of Copenhagen
Denmark

1 June 2026

Supervisor
Djordje Grbic

Abstract

Real-time simulation of liquids and highly deformable materials remains difficult in game development because physically plausible behaviour must be balanced against strict technical constraints. This thesis investigates whether the Position-Based Material Point Method can be implemented in a commercial game engine and whether such an implementation is practical for real-time game scenarios. To answer this, I developed a three-dimensional, CPU-based PB-MPM solver in Unity using the Data-Oriented Technology Stack. The implementation maps the main stages of the method, including constraint solving, particle-to-grid transfer, grid update, grid-to-particle transfer, and particle integration, to a parallel data-oriented architecture. The prototype was evaluated from three perspectives: behavioural plausibility, runtime performance, and memory usage. The validation showed that the solver remained stable under practical parameter settings, produced visually plausible liquid-like motion, and interacted reliably with simple box colliders. The performance and memory experiments varied particle count, solver frequency, solver iteration count, and grid resolution in order to examine the main practical trade-offs of the method. In the baseline configuration of 31,250 particles, a solver frequency of 30 Hz, one solver iteration, and 137,700 grid cells, the mean update time of ***PBMPMSolverUpdate()*** was 3.428 ms and the solver-owned memory footprint was 31.03 MiB. The results show that the presented implementation can operate within a practical real-time budget under suitable conditions, but that its feasibility depends strongly on the chosen parameter settings. Particle count and solver iterations were identified as the main runtime factors, while grid resolution proved especially important for solver-owned memory usage. More generally, the performance analysis supports an approximate time complexity of $\mathcal{O}\big(I(P+G)\big)$ per solver update, while the memory analysis indicates an approximate space complexity of $\mathcal{O}(P+G)$. These findings show that PB-MPM is not a universally cheap replacement for simpler liquid effects, but a viable technique when fluid behaviour is central to gameplay and when its technical costs are planned for explicitly. This thesis demonstrates that PB-MPM can be implemented successfully in Unity DOTS and that, under suitable configurations, it provides behaviour, runtime characteristics, and memory characteristics that make its use realistic for interactive game applications.

Contents

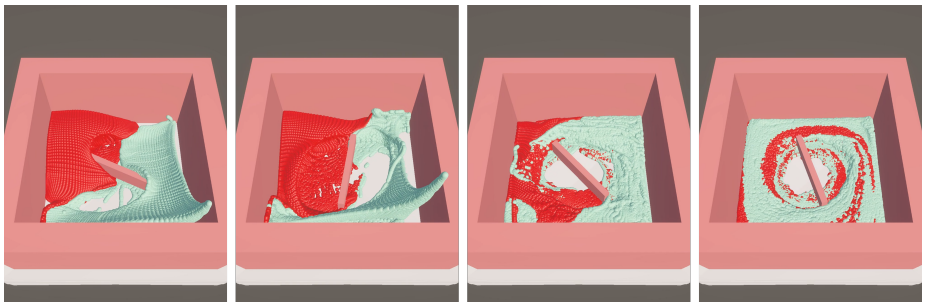

Figure 1: Prototype running with 54,000 particles at a solver step frequency of 60 Hz, using one solver iteration and a grid resolution of 328,032 cells.

1 Introduction

Real-time simulation of liquids and highly deformable materials remains a difficult problem in game development. These materials are visually expressive and can support strong interaction with the game world, but they are also expensive to simulate. In a game, a physically based simulation needs not only to look plausible. It also has to remain stable, fit within a strict frame-time budget, and coexist with rendering, gameplay logic, animation, audio, input, and all other systems that contribute to a frame. For this reason, games often use liquids through authored interaction rules, visual effects, or state-based mechanics instead of fully physically based simulation.

At the same time, liquids in games are often used as gameplay mechanics even when they are not simulated in a physically accurate way. *Timberborn*'s [29] core premise centres on water management through droughts, irrigation, dams, floodgates, canals, and terrain shaping, so water becomes a central gameplay system even though it is stylised and strongly game-driven rather than physically exact. Examples such as these show that liquids can already support meaningful gameplay through visual cues, interaction rules, and state logic alone.

This more authored use of liquids also shows the limitation of simpler approaches. While rule-based and state-based systems can already produce strong mechanics, a more physically accurate liquid simulation opens the door to interactions that are closer to real material behaviour. This makes it possible to design mechanics around flow, accumulation, redirection, pressure, or obstacle interaction in a less scripted and more emergent way. In that sense, the value of a physically based liquid simulation is not only visual. It also lies in the possibility of supporting new gameplay mechanics that follow more naturally from the behaviour of the simulated material itself.

One promising simulation framework for such materials is the Material Point Method (MPM) [1]. MPM combines particles with a background grid: the particles store the material state, while the grid is used for computation and

interaction. This makes the method flexible enough to represent a wide range of materials, including fluids, sand, snow, and elastic bodies. At the same time, traditional MPM formulations can be difficult to use in real-time contexts because stability often depends on small time steps or more expensive implicit solving approaches. Lewin's Position-Based Material Point Method (PB-MPM) is particularly relevant in this context because it combines MPM with concepts from Position Based Dynamics, improving stability while allowing larger time steps [1], [2], [3]. This makes PB-MPM a particularly interesting candidate for interactive applications.

The goal of this thesis is to investigate whether PB-MPM can be implemented in a commercial game engine and whether such an implementation is practical under real-time game conditions. This leads to two central research questions. First, how can the Position-Based Material Point Method be implemented in Unity? Second, how well does such an implementation perform, and is it realistic to use it in an actual game scenario? A simulation may behave plausibly and still be unsuitable for games if it consumes too much frame time, scales poorly in memory, or only remains useful under a very narrow set of parameter choices.

Based on Lewin's PB-MPM work [2] and the existing WebGPU reference implementation [4], I developed a three-dimensional, multithreaded, CPU-based PB-MPM prototype in Unity. The implementation maps the main stages of the method, including constraint solving, particle-to-grid transfer, grid update, grid-to-particle transfer, and particle integration, to ECS components and parallel jobs. In addition, it includes simple collider handling and a visual smoothing pass for presentation. Figure 1 shows a test run of the resulting prototype and in Appendix A the executable build of the prototype can be found.

I chose Unity for this prototype because it is a widely used commercial engine and provides a suitable technical environment for data-oriented simulation work. Unity's 2026 Game Development Report is based on data from nearly five million developers who engaged with the Unity Engine and ecosystem in 2025, which shows the engine's practical relevance for current game development [5]. The thesis focuses on Unity's Data-Oriented Technology Stack (DOTS) [6]. PB-MPM consists largely of repeated operations over large sets of particles and grid elements, which makes memory layout, iteration behaviour, and parallel processing particularly important. Unity's Entity Component System provides compact data layouts through entities and archetypes [7], [8]. The C# Job System allows large parts of the work to be executed in parallel [9], Native Containers provide data structures that can be accessed safely inside jobs [10], and Burst compilation can translate performance-critical C# code into optimised native code [11]. Together, these features make Unity DOTS a useful platform for testing whether PB-MPM can function as more than a stand-alone research prototype.

To evaluate the prototype, I structured the analysis around three complementary perspectives. First, I carried out a qualitative validation step to examine whether the solver behaves in a stable and visually plausible way under practical settings.

Second, I conducted a performance analysis to measure how the runtime of the solver changes when key simulation parameters are varied. Third, I analysed the memory usage of the prototype, with particular emphasis on the memory directly owned by the solver. Across these stages, I varied the parameters that are most relevant to practical game use: particle count, solver frequency, solver iteration count, and grid resolution. This makes it possible to discuss PB-MPM not only as a simulation technique, but as a system that has to fit into the constraints of an actual game engine.

The results of the thesis show that PB-MPM is a viable approach for interactive liquid simulation under suitable conditions, but not a universally cheap one. The implemented solver remains stable and produces plausible liquid-like behaviour in the tested scenes. At the same time, the evaluation shows clear trade-offs between runtime, memory usage, visual quality, and responsiveness. Particle count and solver iterations have the strongest influence on runtime, while grid resolution is especially important for solver-owned memory usage. The practical value of the method therefore depends strongly on how central the simulation is to the intended game experience.

The remainder of this thesis is structured as follows. The Related Work chapter establishes the technical background and introduces Unity DOTS. The Methodology chapter then describes the design of the evaluation. The Implementation chapter explains how the solver was adapted to Unity's data-oriented architecture. The Validation and Behavioural Analysis chapter examines the stability and plausibility of the simulation, as well as the visual effects of different parameter choices. The Performance Analysis and Memory Analysis chapters then investigate the runtime and memory characteristics of the prototype in detail. Finally, the Discussion chapter interprets the results in the context of practical game development, the Future Work chapter outlines potential extensions, and the Conclusion summarises the main findings.

2 Related Work

Simulations for liquids and other highly deformable materials have been an ongoing research topic in computer graphics [1], [2], [12]. Due to their high performance cost, they have mostly been used in film or pre-rendered scenarios. Position-Based Material Point Method (PB-MPM) [2] introduced a way to create stable simulations more efficiently. This section outlines the technologies that led to the development of PB-MPM [2] and explains how they are connected. It also introduces the concept of an Entity Component System, based on the Unity DOTS framework [6], [7], which I used in the implementation of the prototype developed for this thesis.

Figure 2: Visual comparison of PIC and APIC. APIC yields a more energetic simulation than PIC. Adapted from Jiang et al. [12].

2.1 Affine Particle In Cell Method

The Affine Particle-In-Cell method (APIC) was introduced by Jiang et al. [12] as an extension of the original Particle-In-Cell method (PIC) developed by Harlow [13]. Harlow's PIC method was designed for fluid-dynamics simulations involving compressible fluids and large deformations. APIC, by contrast, was introduced in the context of computer graphics to simulate fluids and other highly deformable materials. The APIC method's practical relevance to animation production is illustrated by Disney's film *Moana*, for which an APIC-based solver was used to simulate breaking waves [14].

Both PIC [13] and APIC [12] belong to the class of hybrid particle-grid simulation techniques. In these methods, particles store and transport material quantities, while a background grid is used to compute forces, constraints, and velocity updates. This combination is particularly suitable for fluids and deformable materials: particles can naturally follow large deformations, whereas the grid provides a stable structure for numerical computations.

The main difference between PIC [13] and APIC [12] lies in how information is transferred between particles and the grid. In PIC, each particle stores a position, mass, and velocity. During the particle-to-grid transfer, particle quantities are interpolated to nearby grid nodes. After the grid update, the new grid velocity is interpolated back to the particles. This process can be understood as a

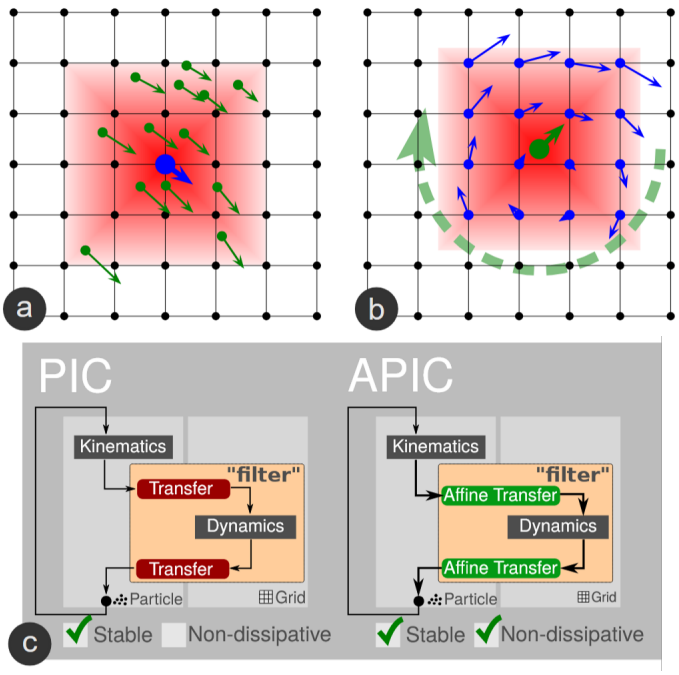

Figure 3: Illustration of PIC and APIC transfer behaviour. (a) PIC uses particle-grid transfers as a stabilising filter, but reduces local motion to a single particle velocity. (b) APIC augments each particle with affine velocity information, allowing local rotational and shearing motion to be preserved more accurately during transfers. (c) The basic data flow of PIC and APIC when doing the particle to grid transfer. Adapted from Jiang et al. [12].

filtering operation. Several particle contributions are averaged on the grid, which improves numerical stability. However, this averaging also removes part of the local motion that existed on the particles before the transfer. As a result, PIC is stable but highly dissipative, meaning that it tends to damp motion and remove energy from the simulation. Figure 2 illustrates the visual difference between PIC and APIC. Compared with PIC, APIC preserves more of the local fluid motion, resulting in a more energetic and visually realistic simulation.

This dissipation is especially visible in rotational motion. Jiang et al. explain that the transfer from particles to the grid preserves angular momentum, but the transfer from the grid back to particles does not. A particle receives information from several grid nodes, but in standard PIC [13] this information is reduced to a single velocity vector. Therefore, local rotational and shearing motion around the particle cannot be represented properly. This loss of information leads to artificial damping and can make simulated materials appear overly viscous or less dynamic.

APIC reduces this loss of information by enriching the particle representation [12]. Instead of storing only a single velocity value, each particle also stores a local affine velocity description. The affine velocity represents a first-order local approximation of the velocity field around each particle, consisting of the particle's translational velocity together with a matrix that describes how velocity changes linearly in its neighbourhood. Conceptually, the particle is no longer treated as an infinitesimally small point with one constant velocity. It can instead be interpreted as a small region of material with a locally varying velocity field [12]. This makes it meaningful for the particle to represent effects such as rotation and local deformation.

Figure 3 illustrates this idea. In the traditional PIC view, particles transfer their data to the grid, where nearby particle contributions are averaged [13]. An equivalent interpretation is that the grid acts as a filter on the particles themselves, so particles can be viewed as having a spatial extent rather than being purely point-like. APIC builds on this interpretation by storing additional affine velocity information on each particle [12]. This allows more of the local velocity structure to survive the particle-grid transfers while still retaining the stabilising filtering behaviour of PIC [13].

For this thesis, APIC [12] is relevant because PB-MPM [2] also relies on repeated particle-to-grid and grid-to-particle transfers. The quality of these transfers has a direct influence on stability, energy preservation, and the visual behaviour of the simulated material. While the implementation in this work focuses on PB-MPM [2], APIC [12] provides important background for understanding why affine velocity information is commonly used in modern Material Point Methods [1] variants. This connection is also visible in the Moving Least Squares Material Point Method (MLS-MPM) [15], where the affine velocity representation is reused to compute velocity gradients and update deformation states [15].

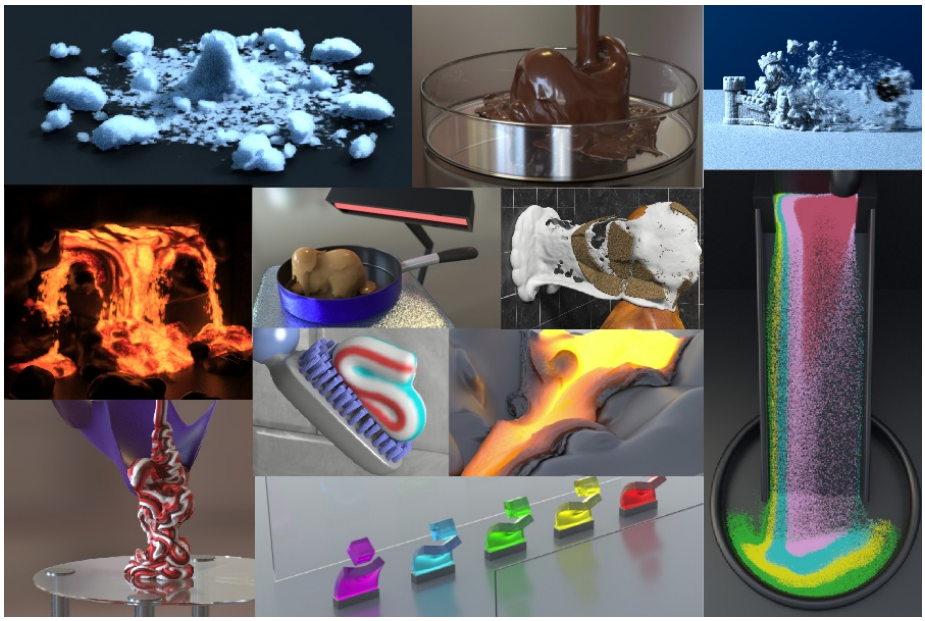

Figure 4: The Material Point Method by Jiang et al. [1]
can be used to simulate a variety of different highly deformable materials, such as snow, lava or tooth paste.

2.2 Material Point Method

The Material Point Method (MPM) [1] is a simulation framework used to model materials such as fluids, snow, sand, or other deformable solids (Figure 4). It is based on a continuum formulation of the governing equations and combines both Lagrangian and Eulerian representations [1]. Material quantities are stored on particles, while a background Cartesian grid is used to compute spatial derivatives and forces. MPM is highly integrated into the production workflows of Walt Disney Animation Studios and has been used in animated movies such as Frozen, Big Hero 6 and Zootopia [1].

This hybrid approach addresses several limitations of traditional methods. Purely Lagrangian methods, such as the Finite Element Method, can suffer from mesh distortion when materials undergo large deformations [1]. Eulerian grid-based methods, on the other hand, handle large deformations well but struggle with tracking material boundaries. MPM combines the advantages of both representations by using particles to track material motion and a grid to perform stable computations. Jiang et al. [1] show that this formulation allows for the natural handling of large deformations, self-collisions, and topological changes such as fracture or merging.

Another important property of MPM is that particles do not represent individual

physical particles such as atoms or molecules [1]. Instead, each particle represents a small portion of a continuous material and stores quantities such as mass, velocity, and deformation. This interpretation follows the continuum assumption, where material properties are defined as continuous fields rather than discrete entities.

MPM simulations are typically structured as a sequence of steps that alternate between particle and grid representations [1]. A common formulation consists of the following stages:

1. **Particle to grid transfer (P2G).**
2. **Compute grid velocities.**
3. **Identify grid degrees of freedom.**
4. **Compute explicit grid forces.**
5. **Grid velocity update.**
6. **Update particle deformation gradient.**
7. **Grid to particle transfer (G2P).**
8. **Particle advection.**

These steps describe how information is transferred from particles to the grid, how physical quantities such as forces and velocities are computed on the grid, and how the updated state is transferred back to the particles. This repeated interaction between particles and grid is a defining characteristic of MPM and directly influences the stability and accuracy of the simulation [1].

2.3 Moving Least Squares Material Point Method

As previously mentioned in the APIC section [12], the affine velocity representation can be used in more advanced MPM [1] formulations such as the Moving Least Squares Material Point Method (MLS-MPM) [15]. MLS-MPM introduces a new discretization of the governing equation, which leads to a more consistent formulation of particle-grid interactions.

One of the main contributions of MLS-MPM is a modified stress divergence discretization. This avoids the explicit computation of interpolation function gradients and instead reuses quantities from the affine velocity [12] representation. As a result, the method simplifies the overall formulation and improves performance, with reported speed-ups of up to two times compared to standard MPM [15].

In the simulation pipeline, MLS-MPM mainly affects the force computation and deformation update steps. In terms of the previously introduced MPM sequence,

this corresponds to the grid force computation and the update of the particle deformation gradient [1], [15]. The overall structure of the MPM algorithm remains unchanged.

Lewin [2] uses the grid transfer scheme derived from MLS-MPM together with quadratic B-Spline weighting functions [15]. In this work, a similar approach is followed by using MLS-MPM-based grid transfer and quadratic B-Spline weighting for the interpolation of particle quantities when transferring them to the cells.

2.4 Position Based Dynamics

Position Based Dynamics (PBD) was introduced by Müller et al. [3] as an alternative to traditional force-based simulation methods. Instead of computing forces and integrating velocities, PBD operates directly on particle positions. This allows constraints to be enforced in a more direct and stable way, which is especially useful for real-time applications.

In contrast to force-based approaches, where forces are accumulated and then integrated over time, PBD modifies predicted positions such that they satisfy a set of constraints [3]. These constraints can represent physical properties such as distances, bending, or collisions. By projecting positions to valid configurations, the method avoids common stability issues such as overshooting that occur in explicit integration schemes.

Another important property of PBD is its controllability. Since constraints are applied directly to positions, it is easy to enforce boundary conditions, attachments, and collision handling [3]. This makes the method well suited for interactive environments and game physics, where stability and robustness are often more important than strict physical accuracy [3].

Due to these properties, PBD has been widely used for real-time simulation of deformable objects such as cloth and soft bodies [3]. It also serves as the foundation for later extensions such as PB-MPM [2], which combines the stability of PBD [3] with the continuum formulation of MPM [1].

2.5 Position-Based Material Point Method

The Position-Based Material Point Method (PB-MPM) was introduced by Chris Lewin as a modification of the traditional MPM framework that improves stability while maintaining a relatively simple implementation [1], [2]. The method combines ideas from MPM and PBD by reformulating the simulation as a constraint-based, semi-implicit process [1], [2], [3]. This allows the simulation to remain stable even when using significantly larger time steps compared to explicit integration schemes.

A key motivation for PB-MPM is the limitation of explicit MPM solvers[1],

[2]. While explicit integration is straightforward to implement, it requires very small time steps to remain stable, especially for stiff materials or highly dynamic scenarios. Implicit methods can address this issue but introduce considerable complexity and additional computational cost [2]. PB-MPM instead adopts a semi-implicit approach based on constraint solving, which avoids many of these drawbacks while preserving robustness.

The overall structure of PB-MPM remains close to the standard MPM pipeline, but introduces an additional iterative constraint solving stage [1], [2]. Instead of performing a single forward simulation step, the method refines a candidate particle state over several iterations. In each iteration, particle displacements are adjusted to satisfy local constraints before transferring data to the grid and back. Only after this iterative process are particle positions and deformation states integrated forward in time [2].

Compared to the previously described MPM sequence [1], the main difference lies in the introduction of an iterative loop [2]. Within this loop, the solver repeatedly adjusts the constraints, bringing them progressively closer to satisfaction with each iteration. This results in a Jacobi-style iteration scheme where constraints are solved locally and in parallel, without immediate global propagation of information. While this can lead to slower convergence and some artificial softness to the material, it significantly increases robustness of the simulation and prevents numerical instabilities that are common in explicit methods [2].

The role of Position Based Dynamics in PB-MPM is central. Similar to PBD, physical behaviour is expressed through constraints rather than forces [2], [3]. These constraints can represent material properties such as incompressibility for fluids or shape preservation for elastic solids. Instead of computing forces and integrating them, the solver directly modifies particle displacements to satisfy these constraints. This leads to a highly stable formulation, as constraint projection inherently avoids the accumulation of numerical errors typical in force-based integration [2], [3].

This design has important implications for real-time applications. PB-MPM allows simulations to run with larger time steps while remaining stable under a wide range of conditions, including extreme or user-driven interactions [2]. Lewin reports that a simulation using PB-MPM remained stable at an update rate of 30 Hz with only a single solver iteration, whereas an equivalent simulation using standard MPM [1] required an update rate of 240 Hz to achieve comparable stability [2]. This makes the method particularly suitable for game development, where performance and robustness are critical. In contrast to traditional MPM approaches, which are often limited to offline or high-performance scenarios, PB-MPM enables the use of continuum-based material simulation in interactive environments [1], [2].

2.6 DOTS - Unity's Data-Oriented Tech Stack

I used the Unity game engine for the implementation of the prototype. The main motivation for this choice was the availability of the Data-Oriented Technology Stack (DOTS) [6], which is well suited for highly parallel implementations. DOTS consists of several technologies designed to support data-oriented programming:

- Entity Component System
- Burst Compiler
- C# Job System
- Native Containers

Together, these technologies enable the efficient implementation of large-scale simulations and were essential for achieving a performant PB-MPM [2] prototype.

In modern software architecture, data-oriented programming patterns are widely used. In Game Engine Architecture, Jason Gregory distinguishes between object-centric and property-centric architectures [16]. The central idea of a property-centric approach is to represent game objects as identifiers, referred to as entities, which are defined by their associated data components. The behaviour of these entities is then implicitly defined by systems operating on specific combinations of components. In the Unity context, a specific combination of components is referred to as an archetype [8].

Unity's DOTS provides an implementation of an Entity Component System that includes tooling, debugging capabilities, and an accessible workflow for development [7]. The ECS architectural design offers both flexibility and performance advantages. Since entities are defined purely through the composition of components, new entity types can be created without modifying existing code. These entities automatically inherit behaviour from systems that operate on their components. Furthermore, ECS is well suited for parallel execution. Systems that operate on independent data can process large numbers of entities concurrently without requiring synchronization of shared state.

From a memory perspective, an Entity Component System can be compared to a relational database [7]. Components correspond to columns, entities to rows, and archetypes to tables. This structure results in a contiguous memory layout, which improves cache locality and significantly increases performance.

The Burst Compiler [11] is a key component of DOTS that translates managed C# code into highly optimised native machine code. It applies aggressive optimizations such as vectorization and low-level instruction tuning, allowing performance comparable to traditional C++ implementations while maintaining the safety and usability of C#. This is particularly beneficial for numerical simulations such as PB-MPM [2], where the same operations are executed repeatedly over large datasets.

The C# Job System [9] provides a framework for parallel execution. It allows computational work to be divided into independent jobs that can be scheduled across multiple CPU cores. In this work, the Job System was used to parallelize operations over particles and grid cells. For example, particle constraint solving and grid transfers can be executed as separate jobs, enabling efficient use of modern multi-core processors.

Native Containers [10] are specialized data structures designed for safe and efficient use in multithreaded environments. They provide deterministic memory management and enforce constraints that prevent race conditions. In the implementation of PB-MPM, Native Containers were used to store particle data and grid cell information. Their compatibility with the Job System allows data to be shared across parallel jobs without introducing unsafe memory access.

DOTS provides a cohesive framework for implementing data-oriented and highly parallel simulations [6]. Its combination of ECS, optimised compilation, parallel job execution, and efficient memory management makes it particularly suitable for real-time implementations of particle-based methods such as PB-MPM [2].

3 Methodology

The methodology of this thesis follows directly from its two main research questions: how PB-MPM can be implemented in Unity, and whether such an implementation is practical under real-time game conditions. Answering the first question requires a technical prototype (Appendix A). Answering the second requires an evaluation that goes beyond implementation alone. A solver may reproduce the algorithm correctly and still be unsuitable for a game if it behaves implausibly, consumes too much frame time, or scales poorly in memory. For this reason, I evaluated the prototype from three complementary perspectives: behavioural validation, runtime performance, and memory usage.

Following the validity concerns discussed in *Methodology of Algorithm Engineering* [17] and the experimental design principles described by Jain [18], I used a controlled evaluation structure throughout the thesis. The same prototype implementation was examined under comparable conditions, a shared baseline configuration was used as a reference, and the most relevant simulation parameters were varied one at a time while the remaining parameters were kept constant. This makes it possible to isolate the effect of each parameter and compare the resulting behaviour across the different analyses.

The first stage of the evaluation was a qualitative validation step. Since this thesis is concerned with a real-time, game-oriented implementation, the purpose of this stage was not to prove exact physical accuracy. Instead, I examined whether the solver behaved in a way that is useful for interactive applications. In practice, this meant checking whether the simulation remained stable, whether the liquid motion appeared visually plausible, whether simple collider interaction

worked reliably, and how the perceived behaviour changed when important simulation parameters were adjusted. This stage therefore acts as a precondition for the later numerical interpretation. Runtime and memory measurements are only meaningful if the tested solver states are also behaviourally usable. The detailed observations from this stage are discussed in Section 5.

The second stage was the performance analysis. Here, the goal was to measure the computational cost of the solver itself rather than the cost of the entire prototype as a game application. For this reason, the profiling focuses primarily on the update work performed by the ***PBMPMSolverSystem***, which schedules the PB-MPM simulation pipeline. The parameters varied in this analysis were particle count, solver frequency, solver iteration count, and grid cell count, because these are the main settings that determine the amount of work performed by the solver and are also the parameters a developer would most likely tune in practice. In most test series, the central metric is execution time, because it gives the clearest indication of how expensive a solver update is. The solver-frequency series is the only exception, because there the more relevant question is how often that cost can be paid over time, which is better reflected by frame rate. The detailed setup and results of this stage are presented in Section 6.

The third stage was the memory analysis. This stage mirrors the controlled structure of the performance study, but shifts the focus from time cost to storage cost. The main concern here is not only how much memory the Unity process consumes overall, but how much of that memory is directly attributable to the PB-MPM solver itself. Since process-level values also include the engine, rendering systems, managed runtime, and other project overhead, I separated the full process memory from the memory explicitly owned by the solver. This distinction is important because the solver-owned values provide a clearer basis for reasoning about how the implementation scales with particle count and grid resolution. The parameters varied in this stage were therefore the two main structural inputs that affect persistent solver storage: particle count and grid cell count. The detailed methodology and results of this stage are described in Section 7.

Taken together, these three stages provide a practical evaluation of the prototype. The validation stage shows whether the solver behaves plausibly enough to be usable. The performance analysis shows how much frame time it requires under different parameter choices. The memory analysis shows how much solver-owned data the implementation requires as the simulation grows. At the same time, the results must be interpreted in light of the main validity limits of the study, especially the use of controlled prototype scenes and measurements taken on a single hardware configuration. This makes it possible to discuss PB-MPM not only as a simulation method, but as a technique that must fit into the technical constraints of a real-time game engine.

4 Prototype Implementation

In this chapter, I describe the implementation of the PB-MPM solver in Unity[1]. The solver follows a data-oriented architecture based on Unity DOTS [6]. Because PB-MPM repeatedly transfers information between particles and grid nodes [2], the implementation benefits from storing the simulation state in component data and processing it using parallel jobs.

This prototype is strongly inspired by Lewin's WebGPU implementation, which demonstrates a 2D example of PB-MPM [4]. However, my approach differs in two key aspects: I use a multithreaded CPU-based implementation, whereas the original project primarily runs on the GPU, and I extend the method to three dimensions instead of two.

While this section focuses on the architectural and algorithmic design of the solver, the finalized executable prototype used for the evaluation is provided as supplementary material in Appendix A. Including the build allows the implemented features and configurable parameters discussed in this chapter to be inspected in a runnable form.

4.1 Software Architecture

At the core of the implementation is the solver loop proposed by Lewin for PB-MPM, shown in Algorithm 1 [2]. The algorithm operates on two main data structures: a set of particles and a background grid. For an efficient implementation, both must be represented in a way that supports frequent iteration and parallel access.

Algorithm 1 Position Based MPM

```
1: function UpdatePBMPM(P) with P as an array of particles pi
2:     for iteration ← 1 to iterationCount do
3:         P ← SolveConstraints(P)
4:         G ← ParticleToGrid(P)
5:         G ← GridUpdate(G)
6:         P ← GridToParticle(P, G)
7:     end for
8:     P ← IntegrateParticles(P)
9: end function
```

In my prototype, the central part of the implementation is a system called ***PBMPMSolverSystem***. Figure 5 provides an overview of the classes used in this prototype. I followed Algorithm 1 closely and implemented a system that schedules each step of the algorithm as jobs [9]. In addition, the system handles

[1]Source Code of the Prototype: https://github.com/MangiameliFabio/Unity-PBMPM

logic for visually smoothing the particles and constructs the colliders required for collision detection within the PB-MPM solver

Another important responsibility of the ***PBMPMSolverSystem*** is advancing the simulation in fixed-size solver steps. A solver step denotes one update of the physical simulation with a fixed time interval

$$\Delta t = \frac{1}{f},$$

where f is the chosen solver update frequency. Within each solver step, the PB-MPM loop can be executed multiple times. These repeated passes are referred to as *iterations*. In other words, the solver step determines how often the simulation advances in time, while the iteration count determines how often the particle-grid transfers and constraint projections are repeated before the particles are finally integrated.

This distinction is reflected directly in the implementation. The system accumulates frame time in every call to ***OnUpdate*** and computes how many fixed solver substeps must be executed based on the configured update frequency. Unity itself also provides a fixed timestep loop. When registering a system with ***FixedStepSimulationSystemGroup***, the update function is executed at fixed time intervals [19]. However, I did not use Unity's built-in physics scheduling for the PB-MPM solver, because the solver frequency should be controlled explicitly and independently from the default physics loop. This makes it possible to choose how many solver steps are executed per second and how many PB-MPM iterations are performed inside each step.

The scheduling of fixed solver substeps is handled in ***OnUpdate***, as shown in Code Listing 1. The method first accumulates the elapsed frame time and stores it in ***_remainingTime***. It then computes the fixed solver timestep from the configured update frequency and determines how many complete solver substeps fit into the accumulated time. Any fractional remainder is kept for the next frame, so the solver frequency remains stable even when render frames and solver updates do not align perfectly. The number of substeps is additionally clamped in order to prevent the simulation from executing an excessively large catch-up loop after a long frame.

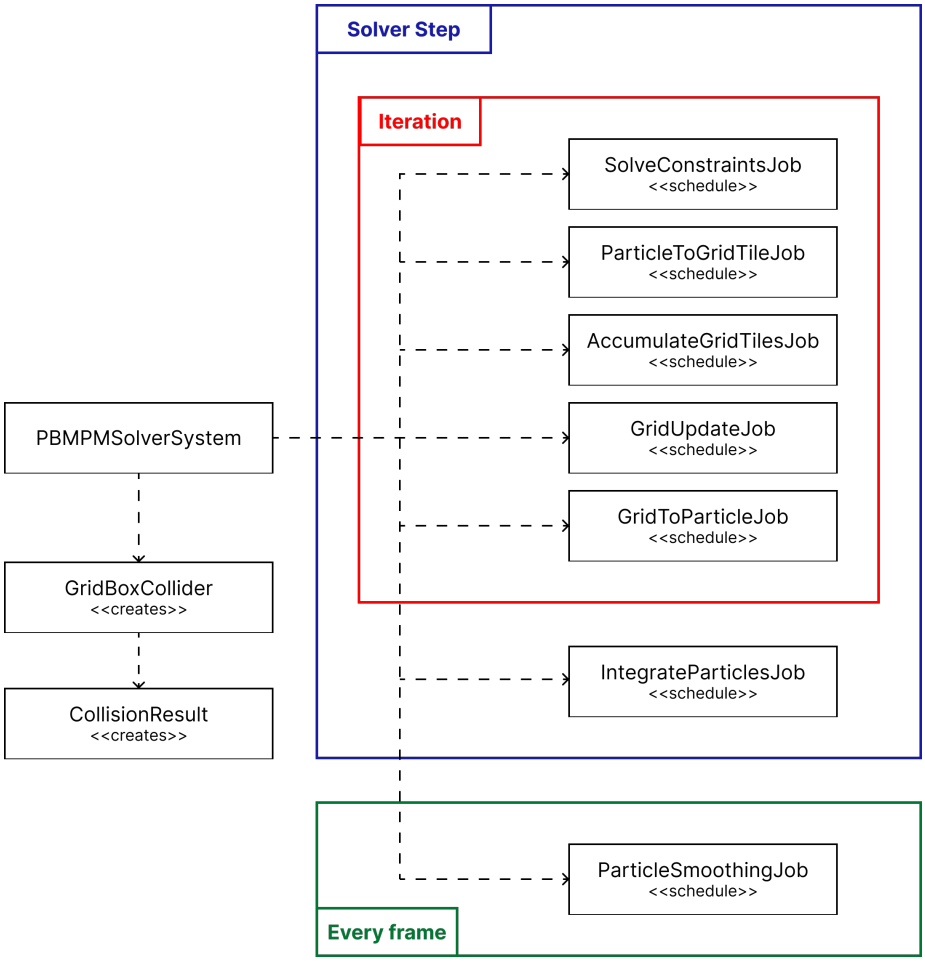

Figure 5: Overview of the software architecture of the PB-MPM solver prototype. The diagram shows the main classes involved in the implementation and their relationships, with ***PBMPMSolverSystem*** coordinating the solver loop, job scheduling, particle smoothing, and collision handling. Jobs running on every solver step are marked blue, jobs running on every solver iteration are marked red and jobs running on every rendered frame are marked green.

```
public void OnUpdate(ref SystemState state)
{
    ...

    _currentTime += SystemAPI.Time.DeltaTime;
    _solverDeltaTime = _currentTime - _lastTime;
    _remainingTime += _solverDeltaTime;

    _fixedDeltaTime = 1f / _config.UpdateFrequency;
    _solverSubsteps = (int)(_remainingTime / _fixedDeltaTime);
    _solverSubsteps = math.min(_solverSubsteps, 10);
    _remainingTime -= _solverSubsteps * _fixedDeltaTime;

    ...

    if (solverRan)
    {
        for (int substepIndex = 0; substepIndex < _solverSubsteps;
            substepIndex++)
        {
            ProfilerMarker.AutoScope solverUpdateScope = default;
            long startTimestamp = PerformanceUtilities.
                BeginSolverUpdateMeasurement(out solverUpdateScope);
            using (solverUpdateScope)
            {
                PBMPMSolverUpdate(ref state, iterationCount,
                    interpolationMode, useGridVolumePreservation);
            }

            solverUpdateTimeMs = PerformanceUtilities.
                EndSolverUpdateMeasurement(startTimestamp);
        }
    }

    ...
}
```

Code Listing 1: Excerpt from ***OnUpdate*** showing fixed solver-step scheduling

Once a solver substep has been scheduled, ***ScheduleParticleJobs***, shown in Code Listing 2, executes the configured number of PB-MPM iterations within that substep and then integrates the particles once. In this way, ***OnUpdate*** determines how often the simulation advances in time, whereas ***ScheduleParticleJobs*** determines how much PB-MPM work is performed during each of those advances. This follows the structure of Algorithm 1 closely.

```
private void ScheduleParticleJobs(ref SystemState state, int iterationCount,
    GridInterpolationMode interpolationMode, bool useGridVolumePreservation)
{
    var gridEntities = _gridQuery.ToEntityArray(Allocator.TempJob);

    for (int iterationIndex = 0; iterationIndex < iterationCount;
        iterationIndex++)
    {
        _gridIterationId++;
        RunSolverIteration(
            ref state,
            gridEntities,
            interpolationMode,
            useGridVolumePreservation,
            applyParticleGravity: iterationIndex == iterationCount - 1);
    }

    ScheduleIntegrateParticles(ref state);

    state.Dependency = gridEntities.Dispose(state.Dependency);
}
```

Code Listing 2: Scheduling PB-MPM iterations within one solver substep

Each stage of a solver iteration is implemented as a separate Unity job (Figure 5). This allows the work to be distributed across multiple threads while preserving a clear correspondence to the conceptual PB-MPM pipeline. The method ***RunSolverIteration***, shown in Code Listing 3, schedules the jobs for one complete iteration: first constraint solving, then particle to grid transfer, grid update, and finally grid to particle transfer. The details of these individual jobs are discussed in the following sections.

```
private void RunSolverIteration(
        ref SystemState state,
        NativeArray<Entity> gridEntities,
        GridInterpolationMode interpolationMode,
        bool useGridVolumePreservation,
        bool applyParticleGravity)
{
    ScheduleSolveConstraints(ref state, useGridVolumePreservation);
    ScheduleParticleToGrid(ref state, gridEntities, interpolationMode);
    ScheduleUpdateGrid(ref state, gridEntities, interpolationMode);
    ScheduleGridToParticle(ref state, gridEntities, interpolationMode,
        useGridVolumePreservation, applyParticleGravity);
}
```

Code Listing 3: Job scheduling for one solver iteration

An additional smoothing pass is executed after the solver substeps. This pass

is only used for visual presentation and does not affect the physical simulation itself. If the solver runs at a low frequency, for example at 15 Hz, the simulation remains stable, but the rendered particles appear to move in visible jumps from one solver state to the next. To reduce this effect, the rendering transform of each particle is updated every frame using its current velocity. This produces smoother motion on screen while leaving the actual solver state unchanged.

Collision handling requires a representation of scene colliders that can be accessed efficiently inside jobs. For this reason, the solver rebuilds a collider cache at the beginning of each update. In the current implementation, only box colliders are considered. Their world-space centre, orientation, and half extents are extracted from Unity Physics and stored in a compact native list, as shown in Code Listing 4. The jobs can then use this cache directly without repeatedly querying the scene.

```
private void RebuildColliderCache(ref SystemState state)
{
    _colliders.Clear();

    foreach (var (collider, transform) in SystemAPI.Query<
                 RefRO<PhysicsCollider>,
                 RefRO<Unity.Transforms.LocalTransform>>())
    {
        if (!collider.ValueRO.IsValid)
        {
            continue;
        }

        ref var sourceCollider = ref collider.ValueRO.Value.As<Collider>();
        if (sourceCollider.Type != ColliderType.Box)
        {
            continue;
        }

        ref var boxCollider = ref collider.ValueRO.Value.As<BoxCollider>();
        BoxGeometry geometry = boxCollider.Geometry;
        float scale = math.abs(transform.ValueRO.Scale);

        GridBoxCollider gridCollider = new GridBoxCollider
        {
            Center = transform.ValueRO.Position +
                math.rotate(transform.ValueRO.Rotation,
                    geometry.Center * scale),
            Rotation = math.normalize(math.mul(transform.ValueRO.Rotation,
                geometry.Orientation)),
            HalfExtents = geometry.Size * (0.5f * scale)
        };
        _colliders.Add(gridCollider);
    }
}
```

Code Listing 4: Building the collider cache for collision handling

The grid, which is needed for classic MPM [1], is itself represented in ECS as an entity with a ***GridComponent*** and a dynamic buffer of ***GridCell*** entries. The component stores the global position and dimensions of the grid, while the buffer stores the per-node state such as displacement, mass, and volume. This design allows the grid topology to be created once during baking and then reused efficiently during simulation. Code Listing 5 shows the helper that initializes the grid-cell buffer together with the baker that creates the grid entity.

Particles are created through a separate spawn pipeline. Its purpose is to define a shape within which a specified number of particles are generated. At the moment, only a box shape is implemented, which spawns particles in a three-dimensional grid pattern.

The process begins with ***SpawnShapeAuthoringBaker***, which uses the user-defined values from ***SpawnShapeAuthoring*** and creates a ***SpawnShapeComponent*** during baking. Instead of storing all particle positions explicitly, this component stores the number of particles along each axis, together with a local start position and a local step size. This compact representation is sufficient to reconstruct a block of particles at runtime.

```
public void RebuildGridCells(DynamicBuffer<GridCell> gridCells)
{
    GridComponent grid = CreateGridComponent();
    int3 cellCounts = GridUtilities.GetCellCounts(grid);
    int3 nodeCounts = cellCounts + 1;

    gridCells.Clear();
    gridCells.EnsureCapacity(nodeCounts.x * nodeCounts.y * nodeCounts.z);

    for (int x = 0; x < nodeCounts.x; x++)
    {
        for (int y = 0; y < nodeCounts.y; y++)
        {
            for (int z = 0; z < nodeCounts.z; z++)
            {
                gridCells.Add(new GridCell
                {
                    Coordinates = new int3(x, y, z),
                    WeightedDisplacement = float3.zero,
                    Displacement = float3.zero,
                    Mass = 0f,
                    Volume = 0f,
                    LastTouchedIteration = 0
                });
            }
        }
    }

    gridCells.TrimExcess();
}

...

public override void Bake(GridAuthoring authoring)
{
    var entity = GetEntity(authoring, TransformUsageFlags.Dynamic);
    AddComponent(entity, authoring.CreateGridComponent());
    AddComponent(entity, new GridAuthoringReference
    {
        AuthoringInstanceId = authoring.GetInstanceID()
    });
    var gridCells = AddBuffer<GridCell>(entity);
    authoring.RebuildGridCells(gridCells);
}
```

Code Listing 5: Selected excerpts from grid baking and grid-cell initialization

The actual particle creation is performed by ***ParticleSpawnSystem***. On its first update, the system iterates over all spawn shapes, creates a particle entity with the ***ParticleComponent***, and initializes its transform and particle data. After that, the system disables itself. The nested loops used for spawning are shown in Code Listing 6.

```
foreach (var shape in SystemAPI.Query<RefRO<SpawnShapeComponent>>())
{
    for (int i = 0; i < shape.ValueRO.SpawnAmount.x; i++)
    {
        for (int j = 0; j < shape.ValueRO.SpawnAmount.y; j++)
        {
            for (int k = 0; k < shape.ValueRO.SpawnAmount.z; k++)
            {
                var localPosition = shape.ValueRO.LocalCenter + new float3(
                    shape.ValueRO.LocalStart.x + i * shape.ValueRO.LocalStep.x
    ,
                    shape.ValueRO.LocalStart.y + j * shape.ValueRO.LocalStep.y
    ,
                    shape.ValueRO.LocalStart.z + k * shape.ValueRO.LocalStep.z
                );

                var globalPosition = localPosition
                    + shape.ValueRO.GlobalPosition;

                var particleEntity = state.EntityManager.Instantiate(
                    config.ParticlePrefab);
                state.EntityManager.SetComponentData(particleEntity,
                    LocalTransform.FromPosition(globalPosition));

                var particleData = state.EntityManager.GetComponentData<
                    ParticleComponent>(particleEntity);
                particleData.Position = globalPosition;
                particleData.Volume = shape.ValueRO.LocalStep.x *
                    shape.ValueRO.LocalStep.y * shape.ValueRO.LocalStep.z;
                particleData.LiquidHydroFactor =
                    shape.ValueRO.LiquidHydroFactor;
                particleData.LiquidViscosityFactor =
                    shape.ValueRO.LiquidViscosityFactor;
                state.EntityManager.SetComponentData(particleEntity,
                    particleData);

                if (state.EntityManager.HasComponent<
                        URPMaterialPropertyBaseColor>(particleEntity))
                {
                    state.EntityManager.SetComponentData(particleEntity,
                        new URPMaterialPropertyBaseColor
                        {
                            Value = shape.ValueRO.ParticleAlbedo
                        });
                }
            }
        }
    }
}
```

Code Listing 6: Core particle creation loop in the spawn system

The software architecture follows the structure of the PB-MPM algorithm closely while adapting it to Unity's data-oriented execution model. The solver logic is

concentrated in a single ECS system, the simulation stages are expressed as jobs, the grid is stored as entity data with dynamic buffers, and particle initialization is handled by a dedicated spawn system. This keeps the implementation modular and makes it easier to reason about the different stages of the simulation.

4.2 Solve Constraints

The solve-constraints step is the part of the algorithm in which the current particle state is locally corrected before it is transferred back to the grid. This is where Position Based Dynamics [3] comes into play, as stated by Lewin [2]. In PB-MPM, this step is executed in every solver iteration before the particle-to-grid transfer. Instead of evaluating forces directly, the method refines a candidate displacement state by applying per-particle constraint projections, which are then reconciled through the grid in the following transfer steps [2].

In my implementation, this stage is scheduled by ***PBMPMSolverSystem*** through ***SolveConstraintsJob***. The job processes each particle independently and updates its deformation displacement tensor. The implementation follows the constraint solving procedure proposed in the PB-MPM paper by Lewin [2].

In the original paper, Algorithm 2 is divided into two branches. One branch handles liquid materials, while the other handles elastic materials. In this thesis, only the liquid constraint has been implemented. This is sufficient for the subsequent performance analysis, since the focus of the evaluation lies on the general behaviour and computational cost of the PB-MPM solver rather than on supporting multiple material models.

Algorithm 2 Solve Constraints

1: **function** SolveConstraints(P) with P as an array of particles p_i
2: **for** each particle p_i in P **do**
3: $c \leftarrow \mathrm{tr}(D_i)$
4: $D_{\mathrm{hydro}} \leftarrow I\,(L_i - 1 - c)$
5: $D_{\mathrm{visc}} \leftarrow -\,(D_i - \mathrm{tr}(D_i)I)$
6: $D_i \leftarrow D_i + \alpha_{\mathrm{hydro}} D_{\mathrm{hydro}} + \alpha_{\mathrm{visc}} D_{\mathrm{visc}}$
7: **end for**
8: **end function**

For each particle, the deformation displacement $\boldsymbol{D}$ is read first, and its trace is computed. This trace corresponds to the volumetric part of the local deformation. Based on this value, two correction terms are constructed: a hydrostatic term, which drives the liquid density towards its rest value, and a viscous term, which damps the non-volumetric part of the deformation.

As discussed by Lewin [2], using only the particle-local deformation history to estimate compression can lead to noticeable volume loss when the iterative solver has not fully converged. To avoid this, the paper proposes using an objective

density measure obtained from the grid transfers. This idea is reproduced in my implementation through the optional use of ***GridMeasuredLiquidDensity***. When grid-based volume preservation is enabled, this value is used as the density target for the hydrostatic correction. Otherwise, the particle's previously stored liquid density is used instead.

The relevant part of the job is shown in Code Listing 7. It closely mirrors the liquid update in Algorithm 2 of the paper.

```
private void Execute(ref ParticleComponent particle)
{
    float3x3 D = particle.DeformationDisplacement;
    float c = D.c0.x + D.c1.y + D.c2.z;
    float deformationVolume = math.max(math.determinant(particle.
    DeformationGradient), 1e-4f);
    float particleLocalLiquidDensity = 1f / deformationVolume;
    float liquidDensity = UseGridVolumePreservation
        ? math.max(particle.GridMeasuredLiquidDensity, 1e-4f)
        : math.max(particleLocalLiquidDensity, 1e-4f);

    float3x3 dHydro = float3x3.identity * (liquidDensity - 1f - c);
    float3x3 dVisc = -(D - c * float3x3.identity);

    D += particle.LiquidHydroFactor * dHydro;
    D += particle.LiquidViscosityFactor * dVisc;

    particle.DeformationDisplacement = D;
    particle.LiquidDensity = liquidDensity;
}
```

Code Listing 7: Constraint solve for liquid particles

The function begins by loading the current deformation displacement tensor and computing its trace. It then selects the density value that should be used for the incompressibility correction. The hydrostatic update is formed as a diagonal matrix, so it only affects the volumetric part of the deformation. In contrast, the viscous update removes the isotropic part and damps the remaining shear component. Both terms are scaled by configurable relaxation factors before they are added to $\boldsymbol{D}$.

Finally, the corrected tensor is written back to ***DeformationDisplacement***. This updated value is then used by the subsequent particle-to-grid and grid-to-particle transfers, so the constraint projection influences the particle motion indirectly through the standard MPM exchange step. In this way, the implementation follows the central idea of PB-MPM: material behaviour is introduced by iterative local constraint corrections, while the grid is used to propagate and combine these corrections across the simulation domain.

4.3 Particle to Grid

The transfer of particle data to the grid requires special handling in order to be executed efficiently in parallel. In this step, each particle contributes mass, volume, and displacement-related quantities to several neighboring grid nodes. Since multiple particles may contribute to the same grid cell, a direct parallel write would lead to write conflicts. To avoid this problem, the particle-to-grid transfer is implemented in two separate steps.

The first step is performed by ***ParticleToGridTileJob***. This job does not write particle contributions directly into the grid. Instead, it determines which grid tiles are affected by each particle and stores a compact particle record for every touched tile. The particle position is used to determine the interpolation support, while the quantities that are later accumulated are the particle mass, volume, displacement, and deformation displacement. At the same time, each touched tile is inserted into a set of active tiles. In this way, the first stage only classifies contributions and groups them by tile, but does not yet modify shared grid memory.

For the interpolation stencil, the implementation uses quadratic B-spline interpolation. This choice follows the MLS-MPM formulation of Hu et al. [15], where the method is explicitly developed around quadratic B-splines. In the supplementary discussion, the authors focus on quadratic B-splines as the smallest kernel that still yields stable constitutive behaviour in practice [15]. Let w_{ip} denote the interpolation weight between particle p and grid node i. In three dimensions, this weight is formed as the product of one-dimensional basis functions,

$$w_{ip} = N(x_i - x_p)\, N(y_i - y_p)\, N(z_i - z_p), \tag{1}$$

where N is the quadratic B-spline basis function. As a result, each particle influences a support region of $3 \times 3 \times 3$ grid nodes, that is, 27 nodes in total [15].

The relevant part of ***ParticleToGridTileJob*** is shown in Code Listing 8.

```
if (!GridUtilities.TryGetQuadraticWeights(
        resolvedGrid,
        particle.Position,
        out QuadraticWeights3D weights))
{
    return;
}

for (int xOffset = 0; xOffset < 3; xOffset++)
{
    for (int yOffset = 0; yOffset < 3; yOffset++)
    {
        for (int zOffset = 0; zOffset < 3; zOffset++)
        {
            int3 supportCoordinates = weights.BaseCoordinate + new int3(
                xOffset, yOffset, zOffset);
            if (!GridUtilities.IsInsideNodeCounts(nodeCounts,
                supportCoordinates))
            {
                continue;
            }

            int tileIndex = GridUtilities.GetTileIndex(tileCounts,
                supportCoordinates, TileSize);
            GridUtilities.AddUniqueTile(ref touchedTiles, tileIndex);
        }
    }
}
```

Code Listing 8: Tile detection for particle-to-grid transfer using quadratic B-spline interpolation

This code computes the quadratic B-spline support of the current particle and iterates over all nodes in its $3 \times 3 \times 3$ neighborhood. For each valid support node, the corresponding tile index is determined and inserted into the local list of touched tiles. Duplicate entries are avoided, so each tile is stored only once per particle. In this way, the job identifies exactly which tiles must later receive contributions from the particle.

The second step is performed by ***AccumulateGridTilesJob***. This job iterates over the set of active tiles and processes each tile independently. At the beginning of the execution, all grid cells inside the current tile are cleared. Afterwards, the job retrieves all particle records that were assigned to that tile in the first step and accumulates their contributions into the tile's grid cells. Since each tile is processed independently, writes to the grid can now be carried out safely in parallel.

The actual accumulation follows the standard MPM transfer notation summarized by Jiang et al. [1], combined with the affine transfer idea used in APIC [12] and MLS-MPM [15]. Using the interpolation weights w_{ip}, the nodal mass is

computed as

$$m_i = \sum_p w_{ip} m_p, \tag{2}$$

and the nodal volume is accumulated analogously as

$$V_i = \sum_p w_{ip} V_p. \tag{3}$$

For the kinematic transfer, the implementation uses an affine particle contribution of the form

$$\mathbf{u}_{ip} = \mathbf{u}_p + \mathbf{D}_p(\mathbf{x}_i - \mathbf{x}_p), \tag{4}$$

where $\mathbf{u}_p$ is the particle displacement, $\mathbf{D}_p$ is the deformation displacement tensor, $\mathbf{x}_p$ is the particle position, and $\mathbf{x}_i$ is the position of the grid node. The quantity accumulated on the grid is therefore the mass-weighted displacement

$$(m\mathbf{u})_i = \sum_p w_{ip} m_p \left(\mathbf{u}_p + \mathbf{D}_p(\mathbf{x}_i - \mathbf{x}_p)\right). \tag{5}$$

This is the quantity stored in ***WeightedDisplacement***. The relevant part of the accumulation routine is shown in Code Listing 9.

```
if (!GridUtilities.TryGetQuadraticWeights(Grid, particle.Position, out
    QuadraticWeights3D weights))
{
    return;
}

for (int xOffset = 0; xOffset < 3; xOffset++)
{
    for (int yOffset = 0; yOffset < 3; yOffset++)
    {
        for (int zOffset = 0; zOffset < 3; zOffset++)
        {
            int3 offset = new int3(xOffset, yOffset, zOffset);
            int3 supportCoordinates = weights.BaseCoordinate + offset;
            if (!GridUtilities.IsInsideNodeCounts(nodeCounts,
                    supportCoordinates) ||
                !GridUtilities.IsInsideTile(tileMin, tileMax,
                    supportCoordinates))
            {
                continue;
            }

            float weight = weights.GetWeight(offset);
            if (weight <= 0f)
            {
                continue;
            }

            int gridIndex = GridUtilities.GetGridIndex(nodeCounts,
                supportCoordinates.x,
                supportCoordinates.y,
                supportCoordinates.z);
            float3 supportPosition = GridUtilities.GetGridPosition(Grid,
                supportCoordinates, InterpolationMode);
            float3 relativePosition = supportPosition - particle.Position;

            GridCell cell = GridCells[gridIndex];
            cell.Mass += weight * particle.Mass;
            cell.WeightedDisplacement +=
                weight *
                particle.Mass *
                (particle.Displacement +
                math.mul(particle.DeformationDisplacement, relativePosition));
            cell.Volume += weight * particle.Volume;
            GridCells[gridIndex] = cell;
        }
    }
}
```

Code Listing 9: Particle accumulation for quadratic B-spline interpolation

The code first reconstructs the quadratic interpolation weights for the current particle. It then iterates over the same $3 \times 3 \times 3$ support region as before. Only support nodes that lie both inside the grid and inside the currently processed

tile are considered. For each valid node, the interpolation weight is evaluated and used to accumulate mass, weighted displacement, and volume into the corresponding grid cell. The affine term $\mathbf{D}_p(\mathbf{x}_i - \mathbf{x}_p)$ allows the transfer to capture local variation within the particle neighborhood and therefore provides a more accurate reconstruction than a purely constant particle contribution.

Together, ***ParticleToGridTileJob*** and ***AccumulateGridTilesJob*** form a two-stage particle-to-grid transfer that is suitable for multithreaded execution. The first stage groups particle contributions by tile without modifying shared grid memory. The second stage processes each active tile separately and performs the actual accumulation only within the tile boundaries. This avoids conflicting writes between parallel workers and makes it possible to schedule the transfer efficiently across multiple threads.

4.4 Grid Update

After the particle-to-grid transfer, each active grid cell stores accumulated particle contributions rather than a final nodal state. In particular, the grid contains the nodal mass, the nodal volume, and the mass-weighted displacement accumulated from neighboring particles. The purpose of the grid update step is to convert these accumulated quantities into a usable grid displacement field, apply external forces, and enforce collisions before the updated values are transferred back to the particles.

In the implementation, this stage is performed by ***GridUpdateJob***, which is scheduled once for each grid cell. Unlike the particle-to-grid transfer, this step does not require any special synchronization between threads. At this point, each cell can be updated independently, since all particle contributions have already been accumulated in the previous stage.

The update follows the usual structure of an explicit MPM grid step as described by Jiang et al. [1]. In standard MPM, nodal momentum is first normalized by nodal mass in order to recover a nodal velocity, after which boundary conditions are applied. In the present implementation, the same idea is used in displacement form instead of velocity form. Let $(m\mathbf{u})_i$ denote the mass-weighted displacement stored for grid node i, and let m_i be the corresponding nodal mass. The nodal displacement is then reconstructed as

$$\mathbf{u}_i = \frac{(m\mathbf{u})_i}{m_i}. \tag{6}$$

This is the displacement-based analogue of the standard explicit grid update in MPM [1].

The implementation also resolves collisions against box colliders on the grid. Let $\mathbf{x}_i$ be the original support position of the grid node and let

$$\mathbf{x}'_i = \mathbf{x}_i + \tilde{\mathbf{u}}_i \tag{7}$$

be its displaced candidate position. If this position lies inside a collider, the displacement is projected along the contact normal in order to remove the penetrating component. In the implementation, this is done using

$$g_i = \min\left(0, \mathbf{n}_i \cdot (\mathbf{x}_{\text{col}} - \mathbf{x}_i)\right). \tag{8}$$

$$p_i = \mathbf{n}_i \cdot \tilde{\mathbf{u}}_i - g_i, \tag{9}$$

$$\lambda_i = \max(p_i, 0), \tag{10}$$

and finally

$$\mathbf{u}_i^{\text{new}} = \tilde{\mathbf{u}}_i - \lambda_i \mathbf{n}_i. \tag{11}$$

Here, $\mathbf{n}_i$ denotes the collision normal and $\mathbf{x}_{col}$ is the corresponding point on the collider surface. This projection removes only the normal component that would move the node into the collider. The general idea of enforcing collision constraints on the grid follows the treatment of collision objects in MPM [1].

The relevant part of ***GridUpdateJob*** is shown in Code Listing 10. The function first checks whether the current cell was touched during the active solver iteration and whether it contains nonzero mass. Cells that were not influenced by any particle are skipped immediately. For active cells, the world-space support position is reconstructed from the grid coordinates. The stored mass-weighted displacement is then divided by the nodal mass in order to obtain the actual nodal displacement.

The second part of the function handles collisions. For each collider, the displaced support position is tested against the box volume. If a collision is detected, the normal component of the displacement is reduced so that the grid node no longer penetrates the collider. Since this correction is applied directly on the grid, the subsequent grid-to-particle transfer automatically propagates the collision response back to the particles.

```
public void Execute(int index)
{
    GridCell cell = GridCells[index];

    if (cell.LastTouchedIteration != CurrentGridIteration || cell.Mass <= 0f)
    {
        return;
    }

    float3 supportPosition = GridUtilities.GetGridPosition(Grid, cell.
    Coordinates, InterpolationMode);

    cell.Displacement = cell.WeightedDisplacement / cell.Mass;

    foreach (var collider in Colliders)
    {
        float3 displacedSupportPosition = supportPosition + cell.Displacement;
        GridBoxCollider.CollisionResult collision = collider.Collide(
            displacedSupportPosition);
        if (collision.Collides)
        {
            float gap = math.min(0f, math.dot(collision.Normal,
                collision.PointOnCollider - supportPosition));
            float penetration = math.dot(collision.Normal,
                cell.Displacement) - gap;
            float radialImpulse = math.max(penetration, 0f);
            cell.Displacement -= radialImpulse * collision.Normal;
        }
    }

    GridCells[index] = cell;
}
```

Code Listing 10: Grid update with gravity and collision handling

4.5 Grid to Particle

After the grid update step, the corrected grid state must be transferred back to the particles. In my implementation, this is done by ***GridToParticleJob***. The job processes each particle independently, reads the updated values from the surrounding grid nodes, and reconstructs the particle displacement, the deformation displacement tensor, and a grid-based estimate of the liquid density. Since each execution only writes to the currently processed particle and only reads grid data, the job is well suited for multithreaded execution.

As in the particle-to-grid transfer, the implementation uses quadratic B-spline interpolation. This follows the MLS-MPM formulation of Hu et al. [15], where particle-grid transfers are based on the same $3 \times 3 \times 3$ support region. Let w_{ip} denote the interpolation weight between grid node i and particle p. The updated particle displacement is reconstructed by weighted interpolation of the nodal

displacements,

$$\mathbf{u}_p = \sum_i w_{ip}\mathbf{u}_i, \tag{12}$$

where $\mathbf{u}_i$ is the displacement stored at grid node i after the grid update step.

In addition to the translational displacement, the implementation also reconstructs an affine local variation from the surrounding grid nodes. Following the APIC and MLS-MPM transfer formulation [12], [15], this affine quantity is computed as

$$\mathbf{C}_p = \frac{4}{\Delta x^2}\sum_i w_{ip}\mathbf{u}_i(\mathbf{x}_i - \mathbf{x}_p)^T, \tag{13}$$

where $\mathbf{x}_p$ is the particle position, $\mathbf{x}_i$ is the support node position, and Δx is the grid cell size. In my implementation, this quantity is stored in displacement form as

$$\mathbf{D}_p = \Delta t\, \mathbf{C}_p. \tag{14}$$

This value is written to ***DeformationDisplacement*** and is later used in the particle integration step to update the deformation gradient.

A further quantity reconstructed in this step is the grid-measured liquid density. This follows the idea proposed in PB-MPM by Lewin [2], where an objective density estimate is obtained from the grid in order to improve volume preservation. In the implementation, this value is computed from the interpolated nodal volumes as

$$L_p^{\text{grid}} = \sum_i w_{ip}\frac{V_i}{\Delta x^3}, \tag{15}$$

where V_i is the volume accumulated at grid node i. To avoid numerical problems, the result is clamped to a small positive lower bound. If grid volume preservation is enabled, the particle liquid density is then updated as

$$L_p \leftarrow L_p^{\text{grid}}. \tag{16}$$

```
for (int xOffset = 0; xOffset < 3; xOffset++)
{
    for (int yOffset = 0; yOffset < 3; yOffset++)
    {
        for (int zOffset = 0; zOffset < 3; zOffset++)
        {
            int3 offset = new int3(xOffset, yOffset, zOffset);
            int3 supportCoordinates = weights.BaseCoordinate + offset;
            if (!GridUtilities.IsInsideNodeCounts(nodeCounts,
                supportCoordinates))
            {
                continue;
            }

            float weight = weights.GetWeight(offset);
            if (weight <= 0f)
            {
                continue;
            }

            int cellIndex = GridUtilities.GetGridIndex(nodeCounts,
                supportCoordinates.x,
                supportCoordinates.y,
                supportCoordinates.z);
            GridCell cell = gridCells[cellIndex];
            if (cell.LastTouchedIteration != CurrentGridIteration)
            {
                continue;
            }

            float3 supportPosition = GridUtilities.GetGridPosition(grid,
                supportCoordinates, InterpolationMode);
            float3 relativePosition = supportPosition - particle.Position;

            displacement += weight * cell.Displacement;
            c += 4f * GridUtilities.OuterProduct(cell.Displacement,
                relativePosition) * (weight * inverseCellSizeSquared);
            gridMeasuredLiquidDensity += weight * cell.Volume *
                inverseCellVolume;
        }
    }
}
```

Code Listing 11: Grid-to-particle transfer using quadratic B-spline interpolation

The relevant part of ***GridToParticleJob*** is shown in Code Listing 11. The code first reconstructs the quadratic B-spline weights for the current particle and then iterates over all nodes in the $3 \times 3 \times 3$ support region. For each valid support node, the interpolation weight is evaluated and the updated nodal displacement is accumulated into the particle displacement. At the same time, the affine term is reconstructed from the outer product of the nodal displacement and the relative position between support node and particle. The same interpolation weights are also used to accumulate the grid-based liquid density estimate from

the nodal volumes.

Only grid cells that were touched during the current solver iteration are considered. This avoids reading stale values from inactive cells. After the loop, the accumulated quantities are written back to the particle state.

```
particle.Displacement = displacement;
if (ApplyParticleGravity)
{
    particle.Displacement += GravityDisplacement;
}

particle.DeformationDisplacement = c * DeltaTime;
particle.GridMeasuredLiquidDensity = math.max(gridMeasuredLiquidDensity,
    1e-4f);
if (UseGridVolumePreservation)
{
    particle.LiquidDensity = particle.GridMeasuredLiquidDensity;
}
```

Code Listing 12: Grid-to-particle adding gravity and displacement update

After the interpolated displacement has been reconstructed, the particle state is updated as shown in Code Listing 12. First, the computed displacement is assigned to the particle. If the current solver iteration is the last iteration of the current solver step, the gravitational displacement is added.

Following Lewin's implementation [4], gravity is applied only during the final solver iteration. If gravity were added during every iteration, its contribution would accumulate multiple times within a single solver step, resulting in unrealistically strong gravitational forces.

Next, the reconstructed affine term is converted into displacement form and stored in ***DeformationDisplacement***. At the same time, the grid-based estimate of the liquid density is written to ***GridMeasuredLiquidDensity***. If grid-based volume preservation is enabled, this value also replaces the particle's current liquid density.

The resulting displacement and deformation displacement are then used by ***IntegrateParticlesJob*** to update the particle position and deformation gradient. In this way, ***GridToParticleJob*** completes the transfer cycle by converting the corrected grid state back into particle data.

4.6 Integrate Particles

After the grid-to-particle transfer, the updated particle values still represent a displacement-based state for the current substep. The purpose of the integration step is to convert these values into the final particle state. In my implementation,

this is done by ***IntegrateParticlesJob***, which is scheduled once after all solver iterations of the current substep have been completed. The job processes each particle independently, updates its position and deformation gradient, performs a final collision correction, reconstructs the particle velocity, and writes the result back to the Unity transform.

The deformation gradient update follows the standard MPM formulation summarized by Jiang et al. [1]. In velocity form, the update is given by

$$\mathbf{F}_p^{n+1} = \left(\mathbf{I} + \Delta t\, \mathbf{C}_p^{n+1}\right) \mathbf{F}_p^n, \tag{17}$$

which corresponds to Equation (181) in the MPM course notes [1]. In the present implementation, the quantity transferred back from the grid is not $\mathbf{C}_p$ directly, but its displacement form

$$\mathbf{D}_p = \Delta t\, \mathbf{C}_p. \tag{18}$$

Therefore, the code applies the equivalent update

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \mathbf{D}_p)\, \mathbf{F}_p^n. \tag{19}$$

This reuse of the affine transfer quantity is one of the characteristic simplifications of MLS-MPM [15]. The matrix $\mathbf{C}_p^{n+1}$ appearing in the deformation update is exactly the affine velocity matrix already used in the APIC transfer formulation [12]. In other words, the same locally affine quantity reconstructed during the grid-to-particle transfer can also serve as the particle-wise velocity gradient needed for the update of $\mathbf{F}_p$. In my implementation, this idea is retained in displacement form by storing $\mathbf{D}_p = \Delta t\, \mathbf{C}_p$ in ***DeformationDisplacement***. This avoids introducing a separate reconstruction of the velocity gradient from interpolation function derivatives and follows the MLS-MPM formulation described by Hu et al. [15].

The particle position is then advanced using the interpolated displacement,

$$\mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \mathbf{u}_p. \tag{20}$$

This is the displacement-based counterpart of the usual APIC advection rule

$$\mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \Delta t\, \mathbf{v}_p^{n+1}, \tag{21}$$

as described by Jiang et al. [1]. Since the implementation stores the particle displacement directly, the particle velocity is reconstructed afterwards as

$$\mathbf{v}_p^{n+1} = \frac{\mathbf{u}_p}{\Delta t}. \tag{22}$$

In addition to these standard updates, the implementation performs a final collision correction on the particle positions. After the position update, each particle is tested against all box colliders. If a collision is detected, the particle is moved out of the collider along the contact normal by the penetration depth.

The same correction is also subtracted from the particle displacement. This step is implementation-specific and serves as a final safeguard against particles remaining inside collision geometry after the grid-based collision handling.

The relevant part of ***IntegrateParticlesJob*** is shown in Code Listing 13.

```
1  private void Execute(ref ParticleComponent particle, ref LocalTransform
       transform)
2  {
3      particle.Position += particle.Displacement;
4
5      particle.DeformationGradient = math.mul(
6          float3x3.identity + particle.DeformationDisplacement,
7              particle.DeformationGradient);
8
9      foreach (var collider in Colliders)
10     {
11         GridBoxCollider.CollisionResult collision = collider.Collide(
12             particle.Position);
13         if (collision.Collides)
14         {
15             particle.Displacement -= collision.Penetration * collision.Normal;
16             particle.Position -= collision.Penetration * collision.Normal;
17         }
18     }
19
20     particle.Velocity = particle.Displacement / DeltaTime;
21     transform.Position = particle.Position;
22 }
```

Code Listing 13: Particle integration after the solver step

The function begins by advancing the particle position by the displacement computed in the previous transfer step. Afterwards, the deformation gradient is updated by multiplying the previous deformation gradient with the incremental deformation term. The collision loop then checks whether the new particle position lies inside any collider and, if necessary, projects the particle back to the surface. Finally, the particle velocity is reconstructed from the displacement and the current time step, and the visual transform is updated to match the simulated position.

5 Validation and Behavioural Analysis

Before looking at the runtime and memory results, I first examined whether the solver behaves in a visually plausible way and whether the resulting motion appears liquid-like. The purpose of this validation was not to prove exact physical accuracy. Instead, I wanted to determine whether the implementation produces stable and believable behaviour under the same kind of real-time conditions that

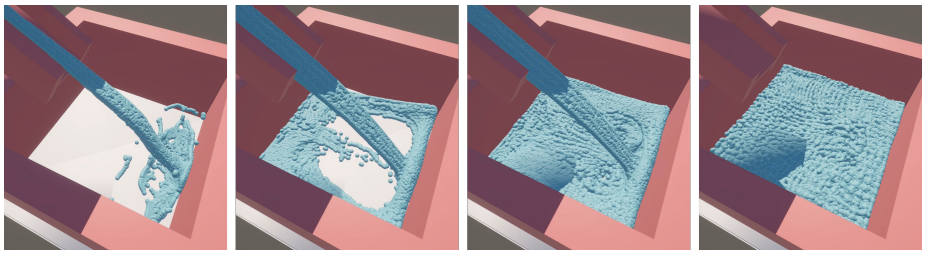

Figure 6: Validation scene used to assess visual plausibility and liquid-like behaviour by showing the flow of the simulated material through and around colliders.

are used in the later performance and memory tests.

To assess this, I used two validation scenes. The first scene was designed to show how the liquid flows through and around colliders. This made it possible to judge whether the motion remains coherent and whether the material behaves in a way that is visually consistent with a liquid. The second scene was used to observe how the simulation responds to a moving collider. This was particularly useful for checking whether the interaction remains stable when the surrounding geometry is not static. Together, these two scenes provide a qualitative way of judging the visual validity of the implementation.

A central requirement for this prototype was stability. In the tested configurations, the solver remained stable even for larger particle counts. This is important because one of the main motivations for using PB-MPM is that it can remain stable at larger time steps than traditional explicit MPM formulations [1], [2]. In this work, I consider the simulation stable when it does not explode numerically, when particles do not receive uncontrolled displacements, and when the liquid remains visually coherent during interaction with the grid and colliders. Figure 6 shows an example of such a test run, and a video of the full experiment can be found in Appendix C. The experiment also shows that higher particle counts, such as 31,250 particles, produce a more visually convincing liquid.

Collider interaction was also examined during testing. The current implementation supports box colliders by projecting grid node displacements out of collider volumes during the grid update step. This allows the particles to react to scene geometry through the grid rather than through direct particle-collider tests. In the prototype scenes, this produced stable obstacle interaction and showed that the solver can be integrated with a simplified game-engine collision representation.

I also considered the visual smoothness of the simulation. The solver can run at a lower frequency than the rendering frame rate, for example at 10 Hz or 30 Hz. In such cases, the physical simulation remains stable, but the movement of the particles can appear less smooth on screen. To reduce this effect, the implementation includes a separate smoothing pass that updates the rendered particle

transforms every frame. This smoothing is only used for visual presentation and does not modify the actual solver state. This distinction is important because the rendered motion can appear smoother than the underlying physical update rate.

To evaluate how different simulation settings affect the visual result, I created an additional test scene, shown in Figure 1, which was later also used for the performance and memory experiments. The main parameters varied during these tests were the following:

- Particle count
- Solver step frequency
- Solver iteration count
- Grid cell count

The particle count has a clear effect on visual plausibility. As already noted, increasing the number of particles generally produces a more convincing liquid. A further showcase can be seen in Appendix E. Lower particle counts appear less fluid and tend to clump together visually. At very high particle counts, such as 250,000 particles at the end of the video, the low frame rate reveals limitations in the simulation, and it becomes visible that the collision of the moving rotor no longer behaves as intended.

The solver step frequency is another important parameter. From a performance perspective, lower frequencies are desirable because they reduce the number of solver updates per second. Visually, however, they introduce clear drawbacks. As shown in Appendix F, a low frequency such as 10 Hz causes noticeable stuttering. The optional smoothing pass can reduce this effect in the rendered particles, as illustrated in Appendix D, but it does not fully compensate for the lower update rate. At very low frequencies, the interaction with the moving collider also becomes less convincing. Conversely, higher frequencies produce smoother motion, but they also make the material appear visually stiffer.

A similar effect can be observed when changing the solver iteration count, as shown in Appendix G. Increasing the number of iterations also tends to make the material appear stiffer. This suggests that both frequency and iteration count influence not only stability and numerical behaviour, but also the visual character of the simulated material.

The final important parameter is the grid resolution, and with it the number of grid cells. Appendix H shows how different resolutions affect the simulation. A fine resolution, such as a cell size of 0.3, makes the material appear more liquid and more bouncy. Increasing the cell size leads to a stiffer appearance, and at very coarse resolutions, such as a cell size of 3, visible pulsating artefacts appear.

These checks indicate that the implementation satisfies the basic behavioural expectations for a real-time PB-MPM prototype. The solver remains stable under

practical parameter settings, produces plausible liquid-like motion, supports simple collider interaction, and can be visually smoothed for rendering. At the same time, this validation is primarily qualitative. The main criteria used here are visual plausibility and the absence of obvious failure cases, such as numerical explosions, uncontrolled particle displacements, or visibly incorrect collider interaction. These observations are useful for judging whether the prototype behaves robustly in a real-time setting, but they do not provide a numerical measure of physical correctness.

6 Performance Analysis

One of the key objectives of this work is to evaluate whether the presented implementation is suitable for real-time game scenarios. To answer this question, I conducted a series of performance tests to determine the execution time of the algorithm and to identify the limits of the implementation under increasing computational load.

6.1 Testing Environment

Based on Unity's profiling recommendations [20], the analysis primarily focuses on execution time rather than frame rate. This metric provides the clearest indication of how changes to the simulation parameters affect performance. The only exception is the test series examining solver frequency. For this series, the average frame rate of the prototype is a more relevant metric, as discussed further in Section 6.5.

The execution time of each solver step was measured using C#'s ***Stopwatch*** class [21]. This made it possible to measure a specific part of the code precisely. Code Listing 1 shows where the measurement is performed. The measured section is the ***PBMPMSolverUpdate()*** method, which rebuilds the collider cache and schedules the jobs required to execute the PB-MPM algorithm.

The individual test series were designed to examine the effect of four parameters:

- Particle count
- Solver step frequency
- Solver iteration count
- Grid cell count

These four parameters were selected because they correspond to the main inputs that determine the computational cost of the PB-MPM solver. In the notation used in the PB-MPM paper by Lewin et al. [2], the most relevant variables are

the number of particles P, the grid G, and the number of solver iterations I. In this work, P is represented by the particle count, G by the grid cell count, and I by the solver iteration count. I additionally examined the solver frequency because it does not change the cost of a single solver update, but it determines how often this cost must be paid during runtime and is therefore important for real-time applications.

In each test series, one parameter was varied while the remaining parameters were kept constant. This makes it possible to isolate the effect of the selected parameter and compare the results across different configurations. The following sections describe the results of each test series.

For each test configuration, the minimum, maximum and mean execution times of the previous 200 solver steps were recorded. This process was repeated five times, resulting in measurements from a total of 1,000 solver steps for each configuration.

All performance tests were carried out under identical conditions. The project was developed in Unity version 6000.3.11f1, and the tests were executed on a system with the following hardware configuration:

- CPU: AMD Ryzen 7 7840HS
- GPU: NVIDIA GeForce RTX 4070 Laptop GPU
- Memory: 32 GB DDR5 RAM (Samsung M425R2GA3PB0-CWMOD)

Following the experimental design principles described by Jain [18], a baseline configuration was defined to provide a consistent reference for all measurements. The baseline consisted of the following parameters:

- Solver frequency: 30 Hz
- Particle count: 31,250
- Solver iterations: 1
- Grid cell count: 137,700

Under these conditions, the mean execution time of ***PBMPMSolverUpdate()*** was 3.428 ms. The complete runtime measurement data for the following test series are provided in Appendix B.

6.2 Effect of Particle Count on Solver Update Runtime

The first test series examines how the particle count affects the runtime of ***PBMPMSolverUpdate()***. To keep the remaining parameters close to the baseline configuration, I set the solver frequency to 30 Hz, used one solver

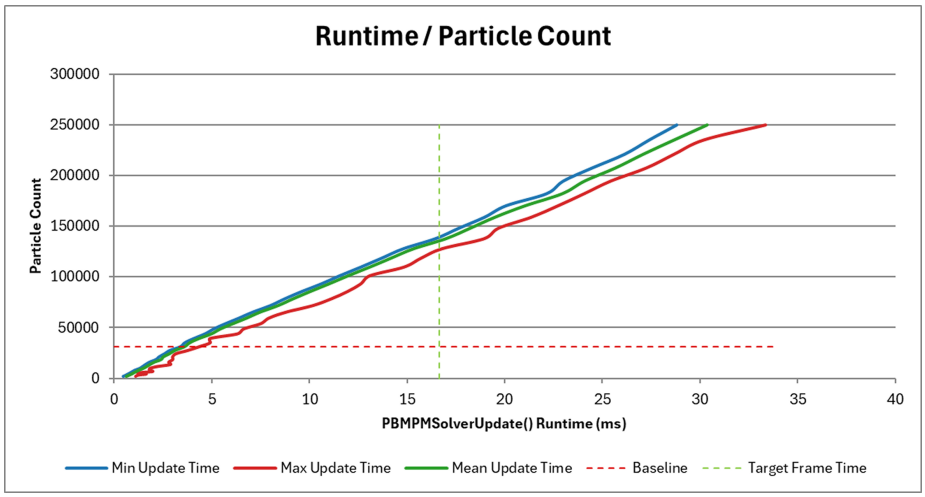

Figure 7: Minimum, maximum and mean runtimes of ***PBMPMSolverUpdate()*** for different particle counts. The solver frequency was set to 30 Hz, with one solver iteration and 137,700 grid cells. The baseline configuration uses 31,250 particles, and the target frame-time marker indicates a frame budget of 16.666 ms.

iteration and kept the grid cell count at 137,700. I only changed the number of particles between the individual test configurations.

For each test configuration, I recorded five measurements. Each measurement contains the minimum, maximum and mean runtimes of the previous 200 solver steps. The particle count depends on the values defined in the particle spawner settings. Each spawner creates the same number of particles along the x, y and z axes. For all tests, I used the same value on each axis. Since the prototype uses two spawners, the total particle count can be calculated as $2P^3$, where P represents the number of particles spawned along each axis. For example, the baseline configuration uses a value of 25, resulting in $2 \times 25^3 = 31,250$ particles.

I started the test series with a value of 10 on each axis, which resulted in $2 \times 10^3 = 2,000$ particles. After each test configuration, I increased the value by one and repeated the five measurements. I continued this process until the value reached 50, which resulted in $2 \times 50^3 = 250,000$ particles. This produced 41 test configurations and a total of 205 measurements.

Figure 7 shows the results for the different particle counts. The baseline marker is set to 31,250, which indicates the particle count used in the baseline configuration. For reference, I also added a target frame-time marker at 16.6 ms. This represents the time budget available for an entire frame when targeting a frame rate of 60 fps. Once the solver update reaches this marker, it consumes the full frame budget on its own.

A clear linear increase in runtime is shown in Figure 7 as the particle count rises.

At the baseline particle count of 31,250, the mean runtime reaches 3.428 ms. The minimum, maximum and mean curves follow a similar shape and increase at a consistent rate across the measured configurations. The mean curve remains closer to the minimum curve than to the maximum curve. This indicates that the higher runtimes represented by the maximum curve occur less frequently.

The linear relationship between particle count and runtime indicates a runtime complexity of $\mathcal{O}(P)$, where P represents the number of particles. Increasing the particle count therefore results in a proportional increase in the runtime of ***PBMPMSolverUpdate()***.

6.3 Effect of Solver Iteration Count on Solver Update Runtime

The second test series examines how the solver iteration count affects the runtime of ***PBMPMSolverUpdate()***. To keep the remaining parameters close to the baseline configuration, I set the solver frequency to 30 Hz, used 31,250 particles and kept the grid cell count at 137,700. I only changed the number of solver iterations between the individual test configurations.

I started the test series with one solver iteration, which matches the baseline configuration. After each test configuration, I increased the iteration count by one and repeated the five measurements. I continued this process until the iteration count reached 10. This produced 10 test configurations and a total of 50 measurements. Each measurement contains the minimum, maximum and mean runtimes of the previous 200 solver steps.

Figure 8 shows the results for the different solver iteration counts. The baseline marker indicates the single solver iteration used in the baseline configuration. I also added a target frame-time marker at 16.666 ms.

A clear linear increase in runtime is shown in Figure 8 as the solver iteration count rises. With one solver iteration, the mean runtime reaches 3.428 ms. At five solver iterations, the mean runtime reaches 16.016 ms, which remains slightly below the target frame-time marker. With six solver iterations, the mean runtime increases to 19.016 ms and exceeds the available frame budget.

The minimum, maximum and mean curves follow a similar shape and increase at a consistent rate across the measured configurations. As in the particle-count test series, the mean curve remains closer to the minimum curve than to the maximum curve. This indicates that the higher runtimes represented by the maximum curve occur less frequently.

The linear relationship between solver iteration count and runtime indicates a runtime complexity of $\mathcal{O}(I)$, where I represents the number of solver iterations. Increasing the solver iteration count therefore results in a linear increase in the runtime of ***PBMPMSolverUpdate()***.

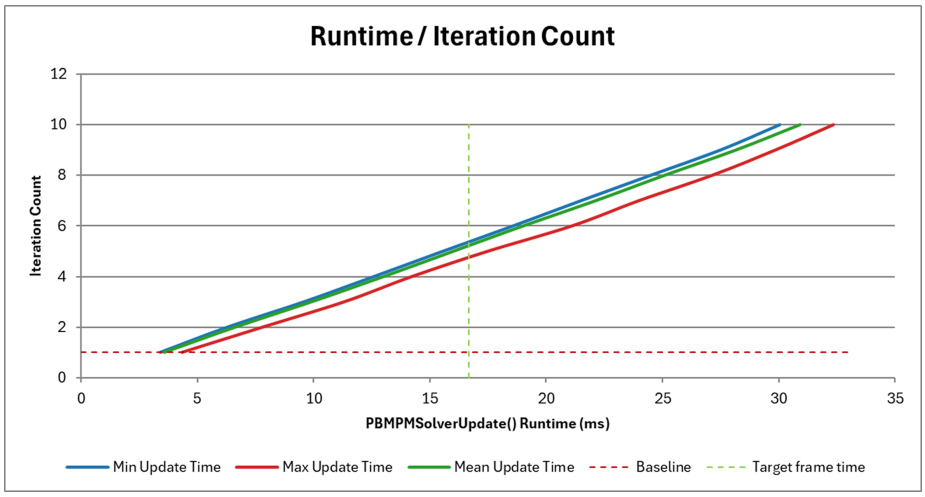

Figure 8: Minimum, maximum and mean runtimes of ***PBMPMSolverUpdate()*** for different solver iteration counts. The solver frequency was set to 30 Hz, with 31,250 particles and 137,700 grid cells. The baseline configuration uses one solver iteration, and the target frame-time marker indicates a frame budget of 16.666 ms.

6.4 Effect of Grid Cell Count on Solver Update Runtime

The third test series examines how the grid cell count affects the runtime of ***PBMPMSolverUpdate()***. To keep the remaining parameters close to the baseline configuration, I set the solver frequency to 30 Hz, used 31,250 particles and kept the solver iteration count at one. I only changed the grid resolution between the individual test configurations.

The grid in the performance test scene has a fixed size of $50 \times 54 \times 51$. The number of grid cells was therefore changed indirectly by varying the cell size. A cell size of 1 corresponds to $50 \times 54 \times 51 = 137{,}700$ grid cells, which matches the baseline configuration. I started the test series with a cell size of 0.3 and increased it in steps of 0.1 until the cell size reached 3.0. This produced 28 test configurations and a total of 140 measurements. As in the previous test series, each measurement contains the minimum, maximum and mean runtimes of the previous 200 solver steps.

The results shown in Figure 9 indicate a clear decrease in runtime as the grid cell count is reduced from very fine grid resolutions towards the baseline configuration. At a cell size of 0.3, the grid contains 5,110,200 cells and the mean runtime reaches 9.931 ms. At a cell size of 0.5, corresponding to 1,101,600 cells, the mean runtime decreases to 4.902 ms. At the baseline cell size of 1, corresponding to 137,700 cells, the mean runtime reaches its lowest measured value of 3.428 ms.

Up to this point, the relationship between grid cell count and runtime is close to

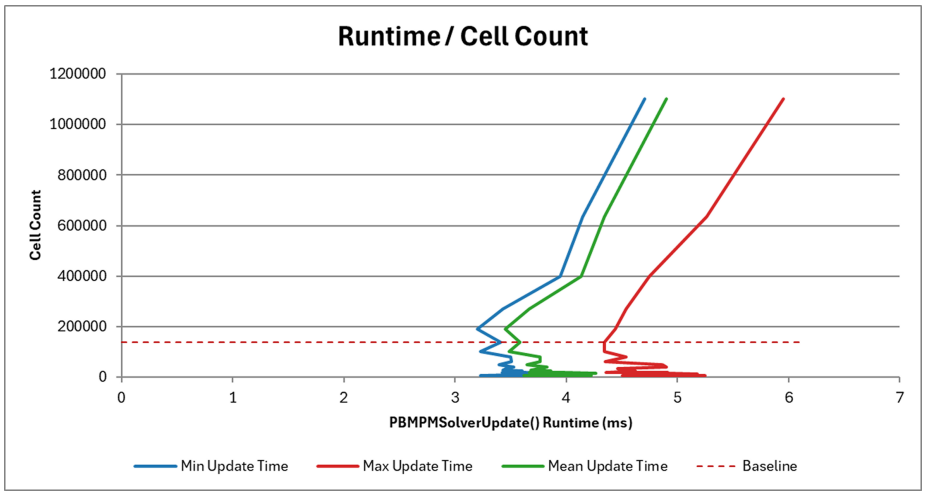

Figure 9: Minimum, maximum and mean runtimes of ***PBMPMSolverUpdate()*** for different grid cell counts. The solver frequency was set to 30 Hz, with 31,250 particles and one solver iteration. The baseline configuration uses 137,700 grid cells, and the target frame-time marker indicates a frame budget of 16.666 ms.

monotonic. This suggests that, for fine grid resolutions, the runtime contribution of the grid grows approximately linearly with the number of grid cells. Since the particle count and iteration count remain constant in this test series, the behaviour in the interval from 5,110,200 to 137,700 cells can therefore be described approximately as $\mathcal{O}(G)$, where G denotes the grid cell count.

Once the cell size exceeds 1, however, the behaviour becomes noticeably less regular. Although the number of grid cells continues to decrease, the runtime no longer decreases accordingly. For example, reducing the grid from 137,700 cells to 101,430 cells increases the mean runtime slightly from 3.428 ms to 3.484 ms. At 40,392 cells, the mean runtime reaches 3.831 ms, at 14,976 cells it reaches 4.270 ms, and at 5,202 cells it still remains at 4.227 ms. This means that coarsening the grid beyond the baseline does not improve performance further, and in several cases it makes the solver slower.

This non-linear behaviour can be explained by the current implementation. First, the grid resolution is derived from a fixed grid size and a variable cell size, which means that the number of cells along each axis must be rounded to integer values. As a result, increasing the cell size does not produce a smooth change in workload, but a sequence of discrete jumps. Second, a coarser grid reduces the number of cells, but it also causes more particles to interact with the same active regions of the grid. This increases the amount of particle-to-grid accumulation and grid-to-particle interpolation work that must be carried out per active region. Third, the current tiled implementation processes the grid in fixed-size active tiles. When the grid becomes too coarse, the number of active tiles decreases,

which reduces the available parallelism and leads to a less even distribution of work across the jobs. The performance gain from fewer grid cells is therefore offset by denser per-tile particle work and less efficient scheduling.

Based on these results, I recommend using a cell size of 1 or lower for this implementation. In this range, the runtime remains lowest and the relationship between grid resolution and runtime is still predictable. Larger cell sizes should not be chosen purely to reduce the number of grid cells, because the measurements show that this does not translate into better overall performance.

6.5 Effect of Solver Frequency on Frame Rate

The fourth test series examines how the solver frequency affects the overall performance of the prototype. In contrast to the previous test series, I measured the average frame rate rather than the runtime of ***PBMPMSolverUpdate()***. This provides a clearer indication of how frequently executing the solver affects the performance of the prototype as a whole.

The solver does not necessarily run during every rendered frame. For example, when the prototype renders at 60 fps and the solver frequency is set to 30 Hz, the solver runs approximately once every two frames. Increasing the solver frequency causes the prototype to execute the solver update more often, which increases the computational load. To measure the effect of this without limiting the results, I tested without a frame-rate limit during the test series.

To keep the remaining parameters close to the baseline configuration, I used 31,250 particles, one solver iteration and 137,700 grid cells. I started the test series with a solver frequency of 10 Hz. After each test configuration, I increased the frequency by 10 Hz and repeated the five measurements. I continued this process until the solver frequency reached 250 Hz. This produced 25 test configurations and a total of 125 measurements.

Figure 10 shows the average frame rate for the different solver frequencies. The baseline marker indicates the solver frequency of 30 Hz used in the baseline configuration.

Figure 10 shows that the average frame rate initially decreases gradually as the solver frequency rises. At 10 Hz, the prototype reaches an average frame rate of 146.267 fps. At the baseline frequency of 30 Hz, the average frame rate reaches 145.172 fps. The frame rate continues to decrease gradually until the solver frequency approaches the average frame rate achieved by the prototype. At 130 Hz, the prototype reaches an average frame rate of 136.709 fps.

Once the solver frequency exceeds the average frame rate, the decrease becomes noticeably stronger. At 140 Hz, the prototype reaches an average frame rate of 134.785 fps. At 200 Hz, the average frame rate drops to 72.479 fps, and at 250 Hz it reaches 27.838 fps.

This behaviour relates to the so-called “spiral of death” in fixed-update simula-

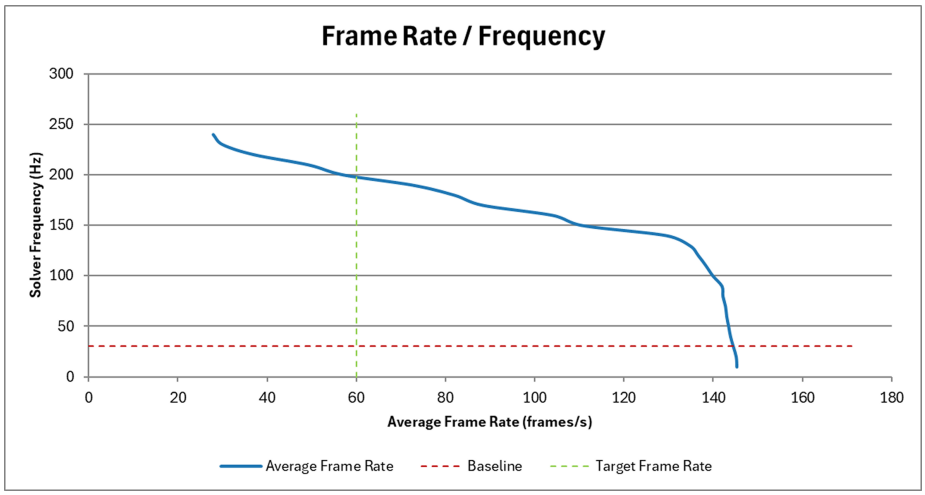

Figure 10: Average frame rate of the prototype for different solver frequencies. The simulation used 31,250 particles, one solver iteration and 137,700 grid cells. The baseline configuration uses a solver frequency of 30 Hz. The prototype ran without a frame-rate limit during the measurements.

tions [22]. When the prototype cannot complete the required solver steps within the available time, the simulation falls behind. The prototype then executes additional solver steps to catch up, which adds further computational load. As a result, increasing the solver frequency beyond the frame rate that the prototype can maintain causes a much stronger decrease in performance.

6.6 Overall Performance Analysis

Taken together, the performance measurements show that the runtime of the solver is primarily governed by three algorithmic inputs: the particle count P, the number of active grid elements G, and the solver iteration count I. Here, G refers to the grid elements that are processed during an update, such as active grid cells, nodes, or tiles, depending on the implementation.

The particle-count test series shows an approximately linear relationship between P and the runtime of ***PBMPMSolverUpdate()***, while the iteration test series shows an approximately linear relationship between I and runtime. The grid-cell test series indicates an approximately linear relationship with G within the practically relevant range of fine to medium grid resolutions, especially for cell sizes between 0.3 and 1. Beyond this range, implementation effects cause the runtime to deviate from a simple monotonic trend.

For a single solver update, the computational cost can be approximated as

$$\mathcal{O}\big(P + I(P + G)\big).$$

The first particle term represents work that is performed once per update, such as particle integration. The remaining particle-related and grid-related work is performed during each solver iteration. Since the solver uses at least one iteration, the expression can be simplified to

$$\mathcal{O}\big(I(P + G)\big).$$

In practice, the particle term is more predictable, while the grid term depends more strongly on the chosen resolution and on how particles are distributed across the active grid tiles.

If the solver frequency is additionally considered as a system-level parameter F, the computational demand per second can be expressed as

$$\mathcal{O}\big(FI(P + G)\big).$$

Increasing the solver frequency does not change the cost of a single update, but it increases how often this update must be executed per second. Once the requested frequency exceeds what the prototype can sustain, the fixed-step update loop falls behind and performance drops sharply.

7 Memory Analysis

In addition to runtime performance, the memory usage of the solver is an important practical consideration. A fluid simulation can fit within the frame budget and still be unsuitable for a game if it requires too much memory or if its memory consumption grows unpredictably as the simulation scales. For this reason, I also analysed how the memory footprint of the prototype changes when the number of particles and the number of grid cells are varied.

7.1 PB-MPM Solver Memory Estimate

Before discussing the captured measurements, I first define the memory terms that are directly controlled by the PB-MPM solver. Following the methodological framing proposed in *Methodology of Algorithm Engineering* [17], I separate the relevant memory footprint into particle state, grid state and persistent cache storage.

A single ***ParticleComponent*** stores one entity reference, three ***float3*** vectors, two ***float3x3*** matrices and six scalar floats. This corresponds to 140 B per

particle. Since the implementation also updates the ECS ***LocalTransform*** of each particle, the full runtime footprint includes an additional 32 B per particle. The total particle-dependent runtime term is therefore $172P$ bytes for P particles, while the solver-owned term used later in this section excludes the transform storage and only counts the PB-MPM data itself.

For the grid, the implementation stores one ***GridComponent*** per grid entity and one ***GridCell*** entry per grid node. The ***GridComponent*** requires 40 B, while each ***GridCell*** requires 48 B. Because the experiments are reported by grid cell count for consistency with the runtime analysis, it is important to note that the memory term itself scales with the corresponding number of grid nodes. In the single-grid test scene used for this section, the direct grid storage is therefore $40 + 48G$ bytes for G grid nodes.

Beyond these direct state objects, the implementation keeps persistent cache containers for the tiled particle-to-grid transfer. The largest of these is ***_tileParticlesCache***, whose capacity grows with up to $8P$ particle-tile records. In the baseline configuration with 31,250 particles and 137,700 cells, the embedded PB-MPM metadata reports 31.03 MiB of solver-owned memory and 31.99 MiB when the particle transforms are included. At this baseline, the tile cache alone accounts for 20.12 MiB, which already shows that particle scaling is influenced not only by the particle components themselves, but also by the cache structures that support the transfer step. In the following analysis, I therefore use the solver-owned value as the primary metric, because it isolates the memory footprint of the PB-MPM implementation more clearly than the whole-process totals alone.

7.2 Testing Environment

All memory measurements were carried out under the same hardware configuration as the runtime measurements reported in Section 6. To record the memory footprint of the running build, I used the Unity Memory Profiler package [23].

The memory test series were designed to examine the effect of two parameters:

- Particle count
- Grid cell count

These two parameters were selected because they correspond to the main structural inputs that determine the size of the solver's persistent data.

To make the collected snapshots comparable across configurations, I exported their summary values to a CSV file after the captures were taken. In addition to the total allocated and resident process memory reported by the Memory Profiler, this export also contains custom PB-MPM metadata written into each snapshot at capture time. The embedded metadata includes the particle count,

the grid cell and grid node counts, the reserved tile-cache capacity and solver-side byte estimates for particle state, grid state and total runtime memory. The process totals therefore show the full memory footprint of the player, while the PB-MPM metadata makes it possible to report the memory used by the solver itself.

This distinction is important for the interpretation of the results. The process totals include the memory used by the engine, the player runtime, the managed environment and other project systems in addition to the solver. The solver-specific values, by contrast, reflect only the data structures accounted for by the PB-MPM metadata. I therefore use the process totals as context, but I base the actual interpretation primarily on the solver-owned values.

To make the results comparable to the runtime analysis, I used the same baseline configuration as before:

- Solver frequency: 30 Hz
- Particle count: 31,250
- Solver iterations: 1
- Grid cell count: 137,700

In each test series, one parameter was varied while the remaining parameters were kept constant. Following the validity concerns discussed in *Methodology of Algorithm Engineering* [17], this makes it possible to isolate the effect of the selected parameter and compare the results across configurations. For each memory snapshot, I recorded three values from the CSV export: total allocated memory, total resident memory and estimated solver memory. Total allocated memory describes the amount of memory that the Unity process has reserved from the operating system, even if not all of it is actively in use at that exact moment. Total resident memory describes the portion of this reserved memory that was actually present in physical RAM at the time of the capture. Estimated solver memory reports the sum of the solver-owned PB-MPM structures recorded in the metadata, namely particle components, grid components, grid cells, the tile-particle cache, the active-tile set and the active-tile list. The complete memory measurement data for the following test series are provided in Appendix B.

7.3 Effect of Particle Count on Memory Usage

The first test series examines how the particle count affects memory usage. To keep the remaining parameters close to the baseline configuration, I set the solver frequency to 30 Hz, used one solver iteration and kept the grid cell count at 137,700. I only changed the number of particles between the individual test configurations.

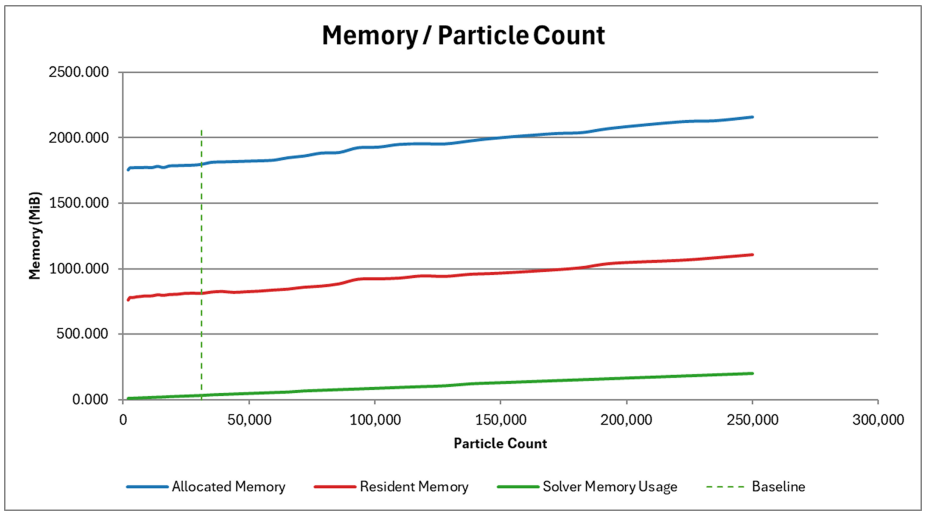

Figure 11: Allocated memory, resident memory and solver-owned memory for different particle counts. The solver frequency was set to 30 Hz, with one solver iteration and 137,700 grid cells. The baseline configuration uses 31,250 particles.

Figure 11 shows a clear increase in all three memory curves as the particle count rises. At 2,000 particles, the player allocates 1751.67 MiB, 761.08 MiB are resident and the solver itself accounts for 8.29 MiB. At the baseline particle count of 31,250, these values reach 1790.67 MiB, 816.90 MiB and 31.03 MiB. At the largest measured configuration with 250,000 particles, they rise to 2157.41 MiB, 1107.27 MiB and 201.08 MiB.

The solver-memory curve follows an almost linear trend across the full test series. This is consistent with the solver-side estimate from the previous subsection. Since the grid resolution remains fixed, the grid-state term stays constant at 6.68 MiB throughout the series. The increase therefore comes mainly from particle state and from the tiled cache storage used during the particle-to-grid transfer. At the baseline configuration, for example, the solver memory is composed primarily of 4.17 MiB for particle components, 6.68 MiB for grid nodes and 20.12 MiB for the tile cache. At 250,000 particles, the tile cache alone grows to 160.96 MiB. The process totals show the same general direction, but the much higher fixed baseline of the player makes the particle-driven growth of the solver easier to see in the solver-specific curve.

The approximately straight solver-memory curve in Figure 11 indicates a space complexity of $\mathcal{O}(P)$, where P denotes the particle count. This follows from the observed near-linear increase in the diagram and from the structure of the implementation itself: with a fixed grid, the dominant changing terms are the per-particle data and the particle-dependent cache capacity. The allocated and resident memory curves also increase with particle count, but their much larger

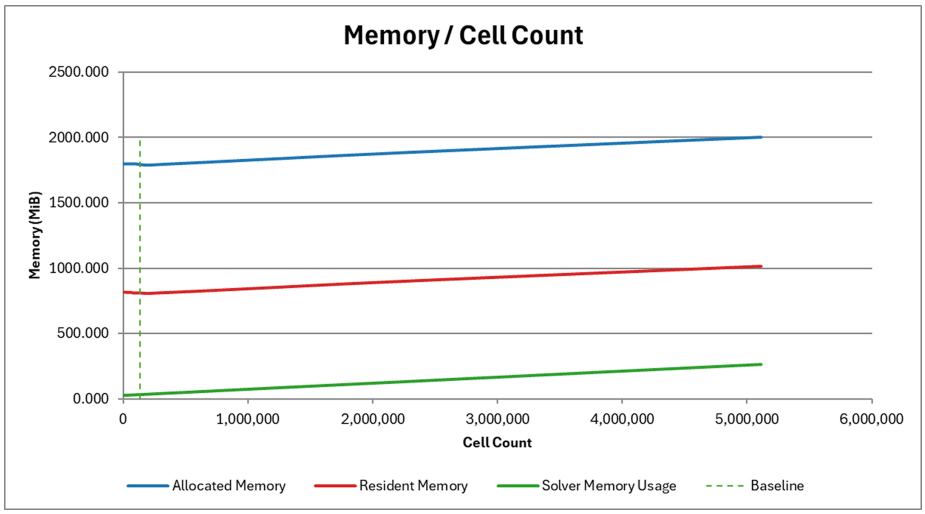

Figure 12: Allocated memory, resident memory and solver-owned memory for different grid cell counts. The solver frequency was set to 30 Hz, with 31,250 particles and one solver iteration. The baseline configuration uses 137,700 grid cells.

constant engine overhead makes them less suitable for complexity discussion than the solver-owned curve.

7.4 Effect of Grid Cell Count on Memory Usage

The second test series examines how the grid cell count affects memory usage. To keep the remaining parameters close to the baseline configuration, I set the solver frequency to 30 Hz, used 31,250 particles and kept the solver iteration count at one. I only changed the grid resolution between the individual test configurations.

Figure 12 shows the same overall pattern, but the cause of the increase is different. At 5,202 cells, the player allocates 1801.00 MiB, 818.50 MiB are resident and the solver itself uses 24.58 MiB. At the baseline with 137,700 cells, these values are 1799.96 MiB, 807.50 MiB and 31.03 MiB. Increasing the grid to 1,101,600 cells raises them to 1833.56 MiB, 849.42 MiB and 77.10 MiB. At the largest measured configuration with 5,110,200 cells, the totals reach 2002.16 MiB, 1014.40 MiB and 264.45 MiB.

In this series, the particle state and the tile cache remain constant because the particle count does not change. The growth is therefore dominated by the grid storage itself. The direct grid-node term rises from only 0.28 MiB at 5,202 cells to 238.03 MiB at 5,110,200 cells, while the particle components remain fixed at 4.17 MiB and the tile cache remains fixed at 20.12 MiB. This produces the

near-linear increase of the solver-memory curve and shows that grid resolution can become the dominant contributor to memory usage once the grid becomes sufficiently fine.

The near-linear solver-memory curve in Figure 12 indicates a space complexity of $\mathcal{O}(G)$, where G denotes the grid cell count. This interpretation is supported both by the diagram and by the implementation, because each increase in grid resolution adds a proportional amount of persistent grid storage while the particle-dependent terms remain fixed. As in the particle-count series, the whole-process curves follow the same direction, but the solver-owned curve gives the clearer basis for describing the scaling behaviour.

7.5 Overall Memory Analysis

Taken together, the two memory test series show that the solver-owned memory of the implementation is mainly governed by two structural inputs: the particle count P and the grid size G, which is reported here by grid cell count and realised in memory through the corresponding grid nodes. The particle-count series shows an approximately linear increase with P, while the grid-cell series shows an approximately linear increase with G over the measured range.

For the solver-owned data of the current implementation, the space requirement can therefore be approximated as

$$\mathcal{O}(P + G).$$

This expression follows directly from the trends in Figures 11 and 12. The particle-dependent part consists of the particle components and the tile-based cache structures, both of which grow with P. The grid-dependent part consists primarily of the persistent grid-node storage, which grows with G. Since both diagrams show near-linear solver-memory curves when one parameter is varied while the other is held constant, the combined solver memory is best described as the sum of these two linear terms.

8 Discussion

Unity recommends defining a frame budget when evaluating the performance of game systems [20]. A common target in the games industry is 60 frames per second, which corresponds to a frame budget of 16.67 ms. Against this background, the results of this thesis show that the presented PB-MPM implementation is practical within a meaningful range of parameter settings, but they also show clearly that runtime and memory usage are controlled by different aspects of the simulation.

The baseline configuration used throughout this work consisted of a solver

frequency of 30 Hz, 31,250 particles, one solver iteration and 137,700 grid cells. Under these conditions, the mean execution time of ***PBMPMSolverUpdate()*** was 3.428 ms and the solver-owned memory usage was 31.03 MiB. Taken on their own, these values indicate that the implementation is within a practical range for real-time use. At the same time, a cost of approximately 3.45 ms is already a meaningful share of a frame budget and does not exist in isolation. It has to coexist with rendering, input, animation, audio, AI, gameplay logic and other engine systems. On lower-end hardware, or in projects that already operate close to their frame-time limit, even this baseline cost can therefore become significant quite quickly.

The qualitative checks from Section 5 add an important behavioural context to these numbers. In the tested scenes, the solver remained stable under practical parameter settings, produced visually coherent liquid-like motion, and interacted reliably with simple box colliders through the grid. This means that the baseline configuration is not merely computationally affordable. It also lies in a parameter range where the simulation already behaves plausibly enough for interactive use, which makes the later performance and memory measurements more meaningful.

At the same time, the performance analysis shows that the baseline should not be interpreted as a guarantee that all parameter combinations will remain equally affordable. The particle-count series shows a clear linear increase in runtime, which is consistent with the $\mathcal{O}(P)$ behaviour identified in Section 6.2. This means that particle count is one of the most direct tuning parameters in the implementation. Reducing the number of particles lowers the solver cost quickly, but it also reduces the visual density of the simulated material. This trade-off can be seen in Appendix E, where the lower particle count produces a noticeably coarser result. Conversely, the behavioural checks also show that higher particle counts generally make the liquid appear more convincing. However, at the highest tested range they also expose the limits of the prototype more clearly, because the lower frame rate makes interaction with the moving rotor appear less reliable.

Solver frequency introduces a different kind of trade-off. The frequency itself does not change the cost of a single solver update, but it determines how often that cost must be paid. The measurements show that the average frame rate remains high while the requested solver frequency stays below the rate that the prototype can comfortably sustain, but the frame rate drops much more sharply once that limit is exceeded. This behaviour is consistent with the fixed-step "spiral of death" described by Fiedler [22]. Lower frequencies can therefore be attractive from a performance perspective, but they also make the simulation less responsive to fast interactions. Appendix F illustrates this clearly, as the reduced update rate causes unreliable interaction with a fast-moving rotor. The validation results further show that the visual smoothing pass can reduce the apparent stuttering of low-frequency simulation, but it cannot fully hide the lower physical update rate because the underlying interaction still becomes less convincing. Another interesting behavioural effect is that higher frequencies

make the material appear stiffer.

The solver iteration count is even more critical for frame time. The iteration test series shows a near-linear increase in runtime, which supports the $\mathcal{O}(I)$ behaviour identified in Section 6.3. In practical terms, this means that solver iterations are expensive very quickly. At five iterations, the mean runtime reaches 16.016 ms and therefore almost consumes the full frame budget of a 60 fps target on its own. At six iterations, the mean runtime rises to 19.016 ms and exceeds that budget.

From a behavioural point of view, however, the iteration count is not only a performance parameter. Lewin et al. [2] note that reducing the number of solver iterations makes the material response softer because the constraints are not resolved as fully during each update step. This is also what can be observed in the present prototype. As shown in Appendix G, lower iteration counts make the material appear more fluid, while higher iteration counts make it appear stiffer. The same general effect can also be seen in the frequency tests. Increasing the solver frequency means that the solver state is updated more often per second, and the behavioural checks show that this too tends to produce a stiffer-looking material. In other words, both more iterations per update and more solver updates per second push the simulation towards a firmer response, even though they affect the computational cost in different ways.

The number of iterations is therefore best understood as a direct quality-versus-cost control. Lower values are cheaper and can produce a softer, more liquid-like appearance, while higher values improve constraint resolution but become expensive very quickly.

The grid-cell results are more nuanced. For fine to medium resolutions, the runtime behaves approximately linearly with the grid size, which is consistent with the $\mathcal{O}(G)$ interpretation in Section 6.4. However, once the cell size becomes larger than the baseline value of 1, the trend is no longer monotonic. In this range, reducing the number of grid cells does not continue to reduce runtime. Instead, the solver becomes slightly slower again. This suggests that the tiled structure of the implementation introduces a practical optimum around the baseline configuration. As described in Section 4.4, the implementation groups work by active tiles before accumulating it on the grid. If the grid becomes too coarse, more particles are concentrated into fewer active regions, which reduces parallelism and increases the amount of work per region. Appendix H also shows that larger cells produce a visibly stiffer material response. The qualitative checks add that very fine resolutions make the liquid appear more bouncy and fluid, while very coarse resolutions can introduce visible pulsating artefacts. Taken together, this means that coarsening the grid beyond the baseline is unattractive from both a runtime and a visual perspective.

The memory analysis adds an important second perspective to these runtime results. The solver-owned memory measurements show that the implementation follows an overall space complexity of $\mathcal{O}(P+G)$. However, the relative importance

of these two terms differs depending on which parameter is being varied. In the particle-count series, solver-owned memory rises from 8.29 MiB at 2,000 particles to 201.08 MiB at 250,000 particles. This increase is caused not only by the particle data itself, but also by the growth of the tile-particle cache. Particle count is therefore a strong driver of both runtime and memory usage.

The grid-cell series shows an even clearer memory effect. At a fixed particle count of 31,250, the solver-owned memory rises from 24.58 MiB at 5,202 cells to 264.45 MiB at 5,110,200 cells. Here, the increase is dominated by the direct grid storage term, because the implementation stores one ***GridCell*** per grid node. The memory results therefore show that grid resolution is a more important parameter for memory planning than for runtime planning. Moderate increases in grid resolution can remain manageable in terms of frame time, but they still raise the persistent memory floor of the solver significantly.

An important consequence of the combined results is that the main runtime parameter is not identical to the main memory parameter. Solver iterations have a strong impact on runtime, but little structural impact on memory because the caches are persistent and reused. Grid resolution, by contrast, can remain moderate from a runtime perspective across part of the tested range while still producing a substantial increase in persistent memory usage. Particle count is the parameter that most strongly affects both perspectives at once. This makes it the most important general-purpose tuning variable in the current implementation.

Taken together, the measurements support the interpretation that the solver update cost is dominated by $\mathcal{O}\big(I(P+G)\big)$, while the solver-owned space requirement is dominated by $\mathcal{O}(P+G)$. The validation results show that these quantitative trends have direct behavioural consequences. Lower particle counts reduce visual density, lower frequencies reduce responsiveness, higher iterations make the material stiffer, and overly coarse grids produce artefacts and less convincing deformation. For practical use, this means that parameter tuning cannot focus on runtime alone. A configuration can look acceptable in frame-time measurements and still be unattractive because it either creates a large persistent memory cost or weakens the liquid-like behaviour that motivates the simulation in the first place.

8.1 Feasibility for Games

From a purely technical perspective, the baseline configuration appears feasible for real-time use. A solver update time of 3.428 ms is comfortably below the frame budget of a 60 fps title and even further below the 33.33 ms budget of a 30 fps title. This remains a relevant reference point in technically demanding console development. Naughty Dog's SIGGRAPH 2020 material [24] on *The Last of Us Part II* [30] describes the challenge of maintaining a target of 30 fps on PlayStation 4 while working within limited processing and memory budgets. Monolith's performance and memory post-mortem [25] for *Middle-earth: Shadow*

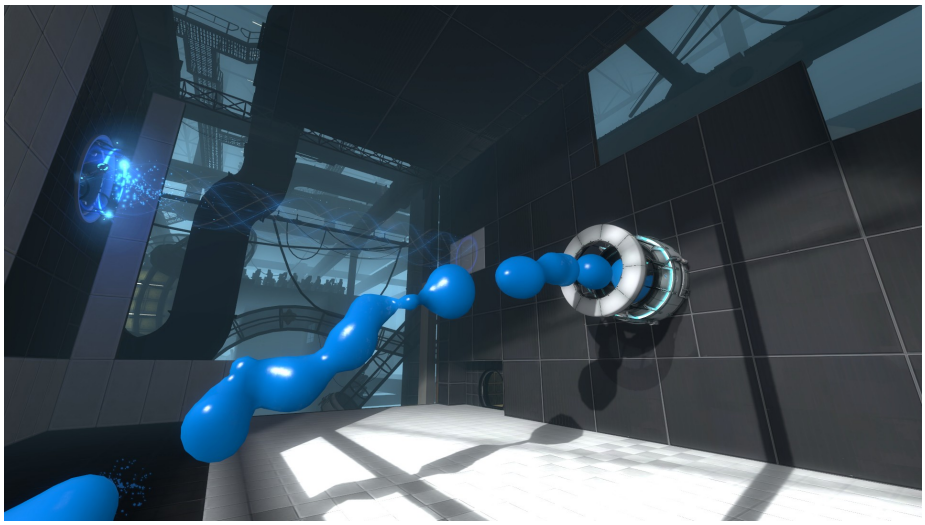

Figure 13: Portal 2 uses gel for a range of gameplay mechanics. In this image, blobs of Repulsion Gel can be seen creating bouncy surfaces in the environment [26].

of War [31] makes this even more explicit by identifying 30 fps, or 33.3 ms per frame, as the target console frame rate used to guide optimisation work. At the same time, the baseline solver-owned memory footprint of 31.03 MiB is modest enough that it would not, by itself, prevent integration into a larger game project.

However, practical feasibility depends less on whether the solver can run in real time in isolation and more on whether the game actually benefits from a physically based liquid simulation. In many games, liquids are mainly decorative. In those cases, shaders or simpler particle systems are usually much cheaper and already sufficient. For such uses, the additional runtime and memory cost of PB-MPM would often be difficult to justify.

At the same time, water and liquids already have a clear role in puzzle design more generally. They are useful because they can be redirected, accumulated, drained, or used to change the movement and state of the player or the environment. In such cases, the liquid is not only decoration, but part of the logic of the puzzle itself.

Portal 2 [32] is a useful example because it turns liquid-like materials directly into puzzle mechanics. Its gel systems change movement and surface behaviour, so that a material spread across the environment becomes part of the solution rather than a purely visual effect (Figure 13). The relevance of this example is not that it tries to simulate water realistically, but that it shows how fluid or fluid-like materials can justify their cost when they define the player's available actions.

The presented implementation becomes much more interesting when liquid behaviour is central to the gameplay. This is the case when the player can directly manipulate flow, pressure, accumulation or redirection and when the physical response of the liquid affects puzzle-solving or interaction design. In that situation, the simulation is no longer only a visual effect, but part of the core mechanic. Under those conditions, the measured baseline and the scaling behaviour identified in this thesis suggest that the implementation lies within a practical range, provided that the parameter settings are chosen carefully. This conclusion is strengthened by the validation results, which show that the prototype already provides stable liquid-like motion and plausible obstacle interaction rather than only acceptable profiling numbers.

For the solver presented in this thesis, the most practical operating range appears to be moderate grid resolutions combined with particle counts that are chosen according to the actual gameplay importance of the fluid. The measurements do not suggest that the solver is universally cheap. Very large particle counts raise both runtime and memory usage significantly, and very fine grids can create a large persistent memory cost even when the runtime remains acceptable. The highest tested grid configuration, for example, requires 264.45 MiB of solver-owned memory, which would already be a substantial commitment for many real projects.

The feasibility of the presented PB-MPM solver should therefore be understood conditionally. If the simulation is peripheral, cheaper methods will often be more appropriate. If the simulation is central to the player's interaction with the world, the measured performance and memory results indicate that the implementation is viable within a well-chosen parameter range for games.

8.2 Limitations

Although the results show that the implemented PB-MPM solver can operate within real-time constraints under practical settings, several limitations still affect how broadly the findings can be interpreted.

One limitation is that the current implementation only supports liquid behaviour. PB-MPM is also used for other material classes, such as elastic solids, snow-like materials or highly deformable bodies, but these models were not part of this thesis. The results therefore show the feasibility of the implemented liquid-focused variant rather than the full range of PB-MPM material behaviour.

The collision handling is also limited. In the current solver, only box colliders are supported, and these colliders are converted into a compact representation for efficient use inside jobs. This is sufficient for evaluating the basic integration of collision handling, but it does not cover more complex cases such as spheres, capsules, triangle meshes or animated collision geometry. A production-ready system would need broader collider support.

Another limitation is that all measurements were taken on a single hardware

configuration. The reported runtimes and memory values therefore show how the implementation behaves on the tested machine, but they do not directly predict its behaviour on lower-end CPUs, consoles, handheld systems or mobile devices. Since the solver is CPU-based and depends on Unity's Job System and Burst compilation, its performance is likely to vary with core count, cache behaviour, memory bandwidth and platform-specific scheduling characteristics.

The evaluation was also carried out in controlled prototype scenes. This was necessary in order to isolate the effects of particle count, solver frequency, iteration count and grid resolution. At the same time, it means that the measurements do not capture the full complexity of a finished game, where rendering, animation, audio, gameplay logic, AI and additional systems compete for the same frame-time and memory budgets. A configuration that is comfortable in the prototype could become less attractive in a production scene with many concurrent systems.

The behavioural validation also has an important limitation of its own. The prototype uses a visual smoothing pass that improves the rendered motion when the solver runs below the rendering frame rate, but this presentation layer does not change the underlying physical state. As a result, the rendered motion can appear smoother than the actual simulation frequency would suggest. This is helpful for real-time presentation, but it also means that visual smoothness alone should not be mistaken for more accurate physical behaviour.

Finally, the thesis focuses primarily on runtime behaviour, memory usage and qualitative plausibility. The solver behaviour was checked visually through stability, collider interaction and the apparent material response, but the work does not include a detailed quantitative validation against analytical solutions, high-accuracy reference simulations or measured physical data. The findings should therefore be understood as evidence that the presented implementation is practically feasible for interactive use, not as proof that it is physically exact.

9 Future Work

While the presented prototype demonstrates that PB-MPM can be integrated into Unity's DOTS framework and executed within a real-time budget, several directions for future work remain. First, the current implementation focuses on liquid behaviour and does not yet cover the wider range of materials that are already supported by Lewin's WebGPU PB-MPM implementation [4]. In particular, extending the solver to support elastic materials or sand would make it possible to evaluate whether the presented architecture remains equally suitable for materials with different constitutive behaviour. This would also allow a more direct comparison between the Unity-based prototype and the existing reference implementation.

Another limitation concerns collision handling. At present, collisions are resolved only against box colliders. While this is sufficient for the controlled test scenes

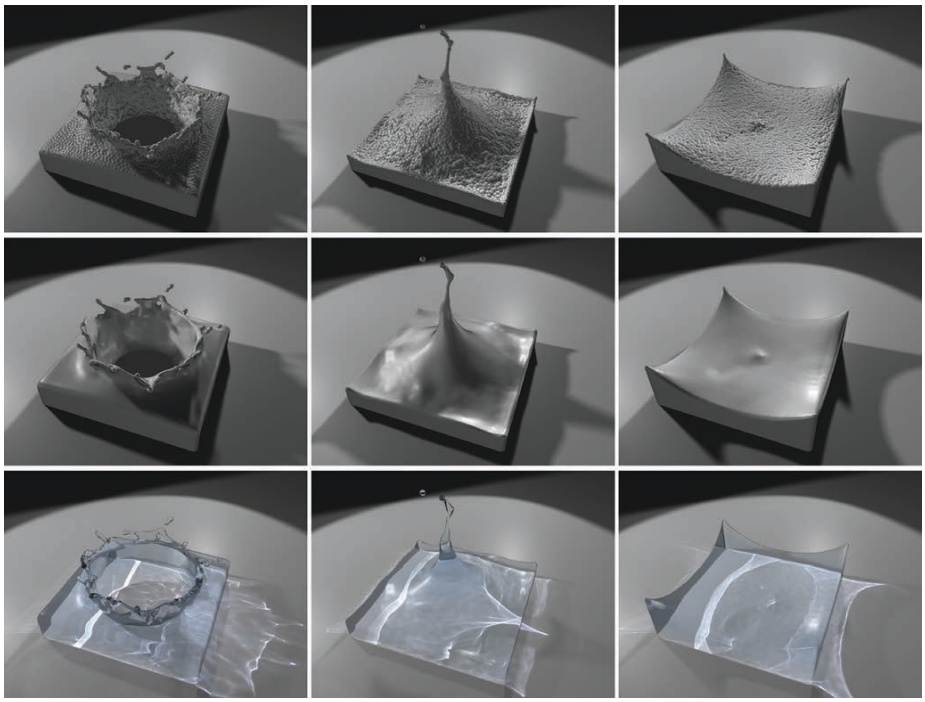

Figure 14: Example of surface reconstruction for particle-based fluids using anisotropic kernels by Jihun Yu [27].

used in this work, it restricts how accurately more complex game worlds can be represented. Support for additional collider types such as spheres, capsules, or compound colliders would make it easier to approximate curved geometry, moving gameplay objects, and container-like environments. This would improve both authoring flexibility and collision fidelity.

The visual presentation of the simulation remains limited. The current prototype renders the simulated material as individual spheres, which is useful for debugging but does not yet produce the continuous appearance typically associated with water or other liquids. Future work should therefore focus on fluid surface reconstruction and dedicated liquid shading. Prior work has shown that particle-based fluids can be rendered more convincingly by reconstructing a smooth surface (Figure 14) from the particle set and by applying screen-space filtering and thickness-based shading [27], [28]. Integrating such a rendering pipeline would improve the visual quality substantially and would bring the presentation of the simulation closer to its intended use in interactive applications.

Further work is also possible in the area of performance optimization. The profiling results in Section 6 showed that particle count is the dominant factor affecting runtime. A particularly promising candidate for GPU acceleration is the

particle-to-grid transfer, together with the accumulation of particle contributions on the grid. In the current implementation, these steps are still executed through CPU jobs, even though they consist largely of repeated local operations over particles and their support nodes. Moving this stage to a compute-based GPU pipeline could increase the feasible particle count and make higher solver frequencies or additional solver iterations more practical for real-time applications. Since the reference PB-MPM implementation by Lewin is executed on WebGPU [4], it also provides a useful baseline for such an extension.

10 Conclusion

This thesis set out to investigate two closely connected questions: how the Position-Based Material Point Method can be implemented in Unity, and whether such an implementation is practical under real-time game conditions. To answer these questions, I developed a PB-MPM prototype in Unity's Data-Oriented Technology Stack and evaluated it from three perspectives: behavioural plausibility, runtime performance, and memory usage.

The implementation shows that PB-MPM can be mapped effectively to Unity's data-oriented architecture. The core stages of the method, including constraint solving, particle-to-grid transfer, grid update, grid-to-particle transfer, and particle integration, were integrated into ECS components and parallel jobs. In this sense, the thesis demonstrates that PB-MPM is not limited to isolated research code or offline simulation contexts, but can be adapted to a production-oriented engine structure.

The behavioural validation showed that the prototype satisfies the basic expectations for a real-time liquid simulation in a game-oriented setting. Under practical parameter settings, the solver remained stable, produced visually plausible liquid-like motion, and interacted reliably with simple box colliders. At the same time, the validation also showed that the visual character of the material changes noticeably with particle count, solver frequency, solver iteration count, and grid resolution. The prototype is therefore not only technically functional, but also sensitive to the same kinds of trade-offs that would matter in an actual game.

The performance analysis showed that the method can run within a practical real-time budget under suitable settings, but also made clear that this feasibility depends strongly on the chosen parameters. In the baseline configuration, the mean runtime of ***PBMPMSolverUpdate()*** was 3.428 ms. This is a useful result, but it does not mean that the solver is universally cheap. Particle count produced a clear approximately linear increase in runtime, solver iterations increased the cost very strongly, and solver frequency determined how often that cost had to be paid during runtime. Grid resolution also influenced the cost, although its effect was more nuanced because the tiled implementation introduces a practical optimum instead of a purely monotonic trend. Taken together, the

runtime results support an approximate time complexity of $\mathcal{O}\big(I(P+G)\big)$ for a single solver update, and $\mathcal{O}\big(FI(P+G)\big)$ when the solver frequency F is considered at the system level.

The memory analysis added an equally important second perspective. In the baseline configuration, the solver-owned memory footprint was 31.03 MiB. The measurements showed that solver-owned memory scales approximately with $\mathcal{O}(P+G)$, where P denotes particle count and G the grid size. Particle count increased memory through both particle state and particle-dependent cache structures, while grid resolution increased the persistent structural memory cost of the grid itself. This means that a configuration can remain acceptable from a runtime perspective and still become unattractive because of its memory demands.

Taken together, these results lead to a more precise answer to the main question of the thesis. PB-MPM is a viable technique for real-time liquid simulation in games, but only under suitable conditions. Its practical use depends on whether the simulation is important enough to justify the required frame time and solver-owned memory, and whether the parameter settings are chosen carefully enough to preserve both behavioural quality and technical feasibility. The method is therefore not a universal replacement for cheaper liquid effects. It is most valuable in situations where fluid behaviour is central to the player's interaction with the game world and where more authored approximations would not provide the same level of responsiveness or plausibility.

In that sense, the most important outcome of this work is not simply that the prototype runs fast enough in one baseline configuration. The more important result is that the thesis identifies the range of trade-offs that determine when PB-MPM is practical in a game engine. Particle count, solver iterations, solver frequency, and grid resolution all influence the method in different ways, and their effects are visible not only in runtime and memory usage, but also in the perceived behaviour of the simulated material. This makes PB-MPM a flexible but demanding technique whose value depends strongly on design context.

I conclude that the Position-Based Material Point Method is a practical and promising approach for interactive liquid simulation in games when the simulation contributes directly to gameplay and when its technical costs are planned for explicitly. The Unity prototype developed in this thesis shows that PB-MPM can be implemented successfully in a commercial engine and can achieve behaviour, runtime characteristics, and memory characteristics that make its use realistic for interactive applications under suitable configurations.

Bibliography

[1] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle, "The material point method for simulating continuum materials," in *ACM SIGGRAPH 2016 Courses*, ser. SIGGRAPH '16, New York, NY, USA: Association for Computing Machinery, Jul. 2016, pp. 1–52, ISBN: 978-1-4503-4289-6. DOI: 10.1145/2897826.2927348. [Online]. Available: https://dl.acm.org/doi/10.1145/2897826.2927348.

[2] C. Lewin, "A Position Based Material Point Method," en, in *ACM SIGGRAPH 2024 Talks*, Denver CO USA: ACM, Jul. 2024, pp. 1–2. [Online]. Available: https://dl.acm.org/doi/10.1145/3641233.3664323.

[3] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109–118, Apr. 2007, ISSN: 1047-3203. DOI: 10.1016/j.jvcir.2007.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1047320307000065.

[4] C. Lewin, *A WebGPU implementation of Position Based MPM*, Oct. 2024. [Online]. Available: https://github.com/electronicarts/pbmpm.

[5] *2026 Unity Game Development Report: Dev insights and trends*, en. [Online]. Available: https://unity.com/resources/gaming-report.

[6] *DOTS - Unity's Data-Oriented Technology Stack*. [Online]. Available: https://unity.com/dots.

[7] *Unity - Manual: Entities*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html.

[8] *Unity - Manual: Archetypes*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-archetypes.html.

[9] *Unity - Manual: C# Job System*, en. [Online]. Available: https://docs.unity3d.com/2021.1/Documentation/Manual/JobSystem.html.

[10] *Unity - Manual: NativeContainer*. [Online]. Available: https://docs.unity3d.com/2021.1/Documentation/Manual/JobSystemNativeContainer.html.

[11] *Unity - Manual: Burst compilation*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/compilation.html.

[12] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin, "The affine particle-in-cell method," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, 51:1–51:10, Jul. 2015, ISSN: 0730-0301. DOI: 10.1145/2766996. [Online]. Available: https://dl.acm.org/doi/10.1145/2766996.

[13] F. Harlow, "The particle-in-cell method for numerical solution of problems in fluid dynamics," vol. Meth Comp Phys 3, Aug. 1962, pp. 319–343.

[14] D. J. Byun and A. Stomakhin, "Moana: Crashing waves," en, in *ACM SIGGRAPH 2017 Talks*, Los Angeles California: ACM, Jul. 2017, pp. 1–2, ISBN: 978-1-4503-5008-2. DOI: 10.1145/3084363.3085056. [Online]. Available: https://dl.acm.org/doi/10.1145/3084363.3085056.

[15] Y. Hu et al., "A moving least squares material point method with displacement discontinuity and two-way rigid body coupling," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, 150:1–150:14, Jul. 2018, ISSN: 0730-0301. DOI: 10.1145/3197517.3201293. [Online]. Available: https://dl.acm.org/doi/10.1145/3197517.3201293.

[16] J. Gregory, *Game Engine Architecture*, 3rd ed. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, ISBN: 978-1-138-03545-4.

[17] J. Mendling, H. Leopold, H. Meyerhenke, and B. Depaire, *Methodology of Algorithm Engineering*, arXiv:2310.18979 [cs.DS], Sep. 2025. DOI: 10.48550/arXiv.2310.18979. [Online]. Available: http://arxiv.org/abs/2310.18979.

[18] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York, NY: Wiley-Interscience / John Wiley & Sons, Jan. 1991, ISBN: 978-0-471-50336-1.

[19] *Unity - Manual: Physics Pipeline*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.physics@1.0/manual/physics-pipeline.html?q=FixedStepSimulationSystemGroup.

[20] *Best practices for profiling game performance*. [Online]. Available: https://unity.com/how-to/best-practices-for-profiling-game-performance.

[21] *Microsoft - .NET Documentation: Stopwatch Class*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-10.0.

[22] G. Fiedler, *Fix Your Timestep!* Jun. 2004. [Online]. Available: https://gafferongames.com/post/fix_your_timestep/.

[23] *Unity - Manual: Memory Profiler*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.memoryprofiler@1.1/manual/index.html.

[24] Naughty Dog, *Naughty Dog at SIGGRAPH 2020*, Feb. 2021. [Online]. Available: https://www.naughtydog.com/blog/naughty_dog_at_siggraph_2020.

[25] P. Mintus, *Performance and Memory Post Mortem for Middle-earth: Shadow of War*, Mar. 2018. [Online]. Available: https://media.gdcvault.com/gdc2018/presentations/Mintus_Piotr_Performance%20and%20Memory.pdf.

[26] Valve Developer Community, *Gel (Portal 2)*. [Online]. Available: https://developer.valvesoftware.com/wiki/Gel_(Portal_2).

[27] J. Yu and G. Turk, "Reconstructing surfaces of particle-based fluids using anisotropic kernels," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 1, 5:1–5:12, Feb. 2013, ISSN: 0730-0301. DOI: 10.1145/2421636.2421641. [Online]. Available: https://dl.acm.org/doi/10.1145/2421636.2421641.

[28] 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*, ser. I3D '09, New York, NY, USA: Association for Computing Machinery, Feb. 2009, pp. 91–98, ISBN: 978-1-60558-429-4. DOI: 10.1145/1507149.1507164. [Online]. Available: https://dl.acm.org/doi/10.1145/1507149.1507164.

Ludography

[29] Mechanistry, *Timberborn*, Sep. 2021.

[30] Naughty Dog, *The Last of Us Part II*, Jun. 2020.

[31] Monolith Productions, *Middle-earth: Shadow of War*, Sep. 2017.

[32] Valve, *Portal 2*, Apr. 2011.

This appendix summarises the supplementary material that accompanies the thesis. Each section briefly explains the purpose of the included file and points back to the relevant discussion in the main text.

A PB-MPM Prototype Build

This appendix includes the prototype build used to evaluate the PB-MPM algorithm. Simulation parameters can be adjusted in the application and applied by pressing the "Reset Scene" button at the bottom of the interface.

Supplementary file: ***Unity_PBMPM.zip***

B Data for Performance & Memory Analysis

This appendix contains the tables with the performance and memory measurements used in Section 6 and Section 7 to analyse the algorithm.

Supplementary file: ***pbmpm_experiment_data.xlsx***

C Liquid Flow Through Colliders

This appendix contains the video material for the first validation scene discussed in Section 5. The video shows how the simulated liquid flows through and around colliders and was used to assess the visual plausibility of the solver. It illustrates that the prototype produces stable, coherent and liquid-like motion under real-time conditions.

Supplementary file: ***pbmpm_liquid_flow_through_colliders.mp4***

D Visual Smoothing

This appendix contains a video comparison of the simulation with visual smoothing enabled and disabled. It is intended to make the visual effect of the smoothing step easier to judge.

Supplementary file: ***pbmpm_visual_smoothing.mp4***

E Effect of Particle Count on Visual Behaviour

This appendix contains the video material for the particle-count comparison discussed in Section 5. The video shows how different particle counts affect

the visual appearance of the simulation. In particular, it illustrates that higher particle counts produce a more coherent and more convincing liquid, while lower particle counts make the material appear less fluid and more prone to visible clumping. The video also shows that extremely high particle counts reveal practical limitations in the current implementation, especially during interaction with moving geometry.

Supplementary file: ***pbmpm_particle.mp4***

F Effect of Solver Frequency on Visual Behaviour

This appendix contains the video material for the solver-frequency comparison discussed in Section 5. The video shows how different solver frequencies affect the smoothness and overall appearance of the simulation. Lower frequencies lead to visible stuttering and less convincing interaction with moving colliders, while higher frequencies produce smoother motion. At the same time, the material also appears visually stiffer as the solver frequency increases.

Supplementary file: ***pbmpm_frequency.mp4***

G Effect of Solver Iteration Count on Visual Behaviour

This appendix contains the video material for the solver-iteration comparison discussed in Section 5. The video shows how different iteration counts affect the visual response of the simulated material. Increasing the number of iterations makes the material appear stiffer, while lower iteration counts preserve a softer and more fluid-looking motion. This appendix therefore complements the qualitative observations made in the main text.

Supplementary file: ***pbmpm_iterations.mp4***

H Effect of Grid Resolution on Visual Behaviour

This appendix contains the video material for the grid-resolution comparison discussed in Section 5. The video shows how different cell sizes, and with them different grid cell counts, affect the visual behaviour of the simulation. Finer grid resolutions make the material appear more liquid and more bouncy, while coarser grid resolutions lead to a stiffer response. At the coarsest tested settings, visible pulsating artefacts can be observed.

Supplementary file: ***pbmpm_grid_cells.mp4***