Position Based Fluids Implementation Using OpenGL

Justin Yue   K U Minnakan Seral
University of California, Riverside

Abstract

Enforcing incompressibility in fluid simulations is essential for realistic behavior but is often computationally expensive, making real-time applications challenging. In this project, we intend to reproduce an iterative density solver within the Position Based Dynamics (PBD) framework. By defining and solving positional constraints that maintain constant density, our implementation will achieve similar accuracy and convergence to previous works, e.g., Smoothed Particle Hydrodynamics (SPH) solvers while benefiting from PBD's stability. This allows for larger time steps, making real-time simulations more feasible. We incorporated an incompressibility constraint to maintain constant density of the fluids, enforce fluid instability to reduce particle clustering, and add vorticity confinement and viscosity. To demonstrate the effectiveness of our implementation, we perform most of the same evaluation scenarios from the original PBD framework paper.

1 Introduction

Fluid simulation has long been a challenging problem in computer graphics, particularly in achieving both realism and computational efficiency. Traditional Smoothed Particle Hydrodynamics [3](SPH) methods suffer from density fluctuations, requiring smaller time steps or additional particles to maintain stability. To address these limitations, Position-Based Fluids [4](PBF) introduce an alternative approach that integrates an iterative density solver into the Position-Based Dynamics (PBD) framework. This method provides robust incompressibility while allowing larger time steps, making real-time fluid simulation feasible.

This project implements a real-time PBF simulation using OpenGL, following the methodology outlined in the original paper. The implementation will enforce incompressibility constraints, introduce vorticity confinement, and apply viscosity while rendering realistic fluid interactions. To validate the performance of our implementation, we simulate the same scenarios as the original paper.

2 Related Works

Müller [3] demonstrated that Smoothed Particle Hydrodynamics (SPH) can be used for interactive fluid simulations with viscosity and surface tension by employing a low-stiffness equation of state (EOS). However, maintaining incompressibility requires stiff equations, which generate large forces that restrict time-step size that is ultimately computationally expensive. Predictive-Corrective Incompressible SPH (PCISPH) [7] addresses this limitation by using an iterative Jacobi-style solver that gradually accumulates and applies pressure corrections until convergence, avoiding the need for predefined stiffness values. Bodin et al. [1] take an alternative approach by enforcing incompressibility through velocity constraints, solving a linear complementarity problem with Gauss-Seidel iteration.

Hybrid methods, such as Fluid Implicit-Particle (FLIP) [2] combine particle-based and grid-based techniques by solving pressure on a grid and transferring velocity corrections back to particles. This was extended by [8] and [6] to handle incompressible flow with free surfaces and approximate divergence-free velocity field efficiently before an adaptive SPH update.

Position-based dynamics (PBD) has also proven useful for fluid simulation. Most relevant to this paper, Muller et al. [5] introduced a stable alternative using Verlet integration, solving non-linear constraints through Gauss-Seidel iteration. By updating positions directly instead of applying forces, PBD avoids the numerical instabilities of explicit integration.

3 Methodology

3.1 Particle Representation and Fluid Constraints

In our implementation, particles are initialized within a predefined volume, each carrying attributes such as position, velocity, and density. A uniform grid spatial partitioning system has been employed to efficiently perform neighbor searches and reduce computational overhead during constraint resolution.

To maintain fluid incompressibility, we implement a density constraint following the formulation:

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

Here, ρ_i is the computed particle density based on neighboring particles, and ρ_0 is the predefined rest density. This constraint is resolved iteratively through position corrections applied in the solver loop.

To mitigate particle clustering and enhance surface tension, we introduce an artificial pressure term into the solver:

$$s_{corr} = -k \left(\frac{W(p_i - p_j, h)}{W(\Delta q, h)} \right)^n \tag{2}$$

with k and n serving as tuning constants. This correction term helps promote an even particle distribution and stabilize fluid surfaces.

For preserving rotational fluid motion, vorticity confinement is incorporated into our velocity update phase:

$$f_{vorticity} = \varepsilon(N \times \omega) \tag{3}$$

where N is the normalized gradient of the vorticity magnitude and ω represents the curl of the velocity field. Additionally, we implement XSPH viscosity to smooth out velocity differences between neighboring particles, using the relation:

$$\mathbf{v}_i^{new} = \mathbf{v}_i + c \sum_j \mathbf{v}_{ij} \cdot W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{4}$$

ensuring smooth and stable fluid motion.

To compute physical properties like density and inter-particle forces, we utilize common SPH kernel functions. The poly6 kernel is used for density estimation and smoothing operations:

$$W_{\text{poly6}}(r, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3, & 0 \le r \le h, \\ 0, & r > h. \end{cases} \tag{5}$$

For pressure-related forces and gradient computations, the spiky kernel is used:

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

Both kernels ensure smooth and continuous influence over neighboring particles within the smoothing radius h, contributing to the physical realism of the simulation.

3.2 Simulation Loop Implementation

Our implementation of the Position-Based Fluids (PBF) algorithm follows the pipeline proposed by Müller et al. [5], utilizing OpenGL compute shaders to offload most computational tasks to the GPU. The full simulation loop is executed within the `PBFComputeSystem::step()` method and follows the standard steps outlined in the original PBF framework.

Step 1: External Forces and Position Prediction. The `external_forces.comp` shader is responsible for applying external forces (e.g., gravity) to each particle's velocity. Predicted positions are computed using explicit Euler integration:

$$\mathbf{x}_i^* = \mathbf{x}_i + \Delta t \cdot \mathbf{v}_i \tag{7}$$

This step is fully parallelized across all particles.

Step 2: Spatial Hashing and Neighbor Search. To optimize neighbor queries, particles are inserted into a 3D uniform grid structure using `construct_grid.comp` after clearing cell counters with `clear_grid.comp`. The grid resolution is defined by:

$$\text{cellSize} = h \tag{8}$$

where h is the smoothing radius. This reduces the complexity of the neighbor search from $O(N^2)$ to approximately $O(N)$, where N is the number of particles.

Step 3: Iterative Density Constraint Solver. Following the standard PBF methodology [4], we iteratively solve the density constraint over 3 iterations per simulation step:

$$C_i(\mathbf{x}) = \frac{\rho_i}{\rho_0} - 1 = 0 \tag{9}$$

The `calculate_density.comp` shader computes densities and the associated Lagrange multipliers (λ_i), while `apply_position_update.comp` applies position corrections:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{x}_i^* - \mathbf{x}_j^*) \tag{10}$$

where W is the smoothing kernel.

Step 4: Velocity Update. The `update_velocity.comp` shader updates particle velocities according to:

$$\mathbf{v}_i = \frac{\mathbf{x}_i^* - \mathbf{x}_i}{\Delta t} \tag{11}$$

This ensures the velocity is consistent with position corrections.

Step 5: Vorticity Confinement and Viscosity. Finally, the `apply_vorticity_viscosity.comp` shader computes vorticity confinement [4] and XSPH viscosity terms to enhance the fluid's rotational behavior and smooth velocities.

Solver Parameters. The system operates with:

- Smoothing radius $h = 2.5 \cdot r$ (where r is the particle radius).
- 3 solver iterations per timestep.
- Spatial grid with a maximum of 64 particles per cell.
- Rest density $\rho_0 = 125.0$.
- Vorticity confinement coefficient $\epsilon = 0.008$.
- XSPH viscosity coefficient $c = 0.01$.

The complete simulation loop is summarized below:

```
Algorithm 1 GPU-accelerated PBF Simulation Loop
1: for all particles i in parallel do
2:     apply external forces and predict positions (external_forces.comp)
3: end for
4: clear 3D grid cells (clear_grid.comp)
5: insert particles into grid cells (construct_grid.comp)
6: for solver iteration = 1 to 3 do
7:     compute densities and lambdas (calculate_density.comp)
8:     apply position corrections and boundary handling (apply_position_update.comp)
9: end for
10: update particle velocities (update_velocity.comp)
11: apply vorticity confinement and viscosity (apply_vorticity_viscosity.comp)
```

3.3 Rendering Loop Implementation

For visualization, the simulation data is rendered using a standard OpenGL forward rendering pipeline. The `main.cpp` implementation uploads particle data (position and color) to a Vertex Buffer Object (VBO) each frame. Each particle is rendered as a `GL_POINT`, which is shaded in screen space to resemble a sphere.

Rendering pipeline:

1. Extract particle positions and colors from the compute shader simulation (via `downloadParticles()`).
2. Upload the particle data to a dynamic `GL_ARRAY_BUFFER`.
3. Render particles using a custom vertex and fragment shader pair (`sphereShader`), where the vertex shader computes screen-space coordinates and the fragment shader applies Phong lighting.
4. Render the ground plane for visual reference using a separate `planeShader`.

The rendering step is fully decoupled from the simulation step and ensures smooth visualization at each timestep.

Key implementation details:

- A `ParticleVertex` structure stores position and color attributes.
- OpenGL point rendering is used (`glDrawArrays(GL_POINTS, ...)`).
- The ground plane is rendered with simple texture and normal data in a separate draw call.
- Camera control is provided via a free-fly system implemented using keyboard and mouse callbacks.

This approach allows rendering of thousands of particles in real time while maintaining high visual fidelity.

4 Evaluation and Expected Outcomes

To validate the implementation, the simulation is tested in two primary scenarios:

- **Free-fall into an empty container (Dam break scenario):** This setup evaluates large-scale fluid motion by releasing a block of particles into an empty domain, resembling a classic dam break scenario. The goal is to assess splashing behavior, wave formation, and stability under free-surface conditions.
- **Dropping fluid into a pre-filled container:** This scenario analyzes two-fluid interactions, where a falling block merges into an existing fluid body, testing incompressibility enforcement and surface tension representation.

Performance and Fluid Quality Metrics

The simulation is evaluated based on:

- **Computational performance:** Measured via frame time.
- **Density preservation:** Tracking maximum and average particle density relative to the rest density over time.
- **Visual realism:** Evaluating splash, wave, and merging behavior in line with prior PBF-based studies [4].

For visual differentiation, particles are colored using a vertical gradient, helping to track mixing and impact responses.

Free-fall and Dam Break Visualization

This test captures the characteristic fluid behavior expected in dam break simulations. The released fluid column collapses under gravity, producing high-energy splashes and outward-propagating waves upon impact with the container base.

Figure 3 demonstrates the results. The maximum density peaks significantly above the rest density immediately after impact, before gradually relaxing as the fluid stabilizes. The average density oscillates before converging toward a lower value, which is consistent with limitations noted in the original PBF paper [4], where constraint satisfaction remains approximate due to iterative solver limitations.

Minor particle clustering is observed near boundaries during the settling phase, likely due to underresolved artificial pressure effects. Nonetheless, the simulation visually captures realistic large-scale wave propagation and splash formation, with volume preservation largely maintained.

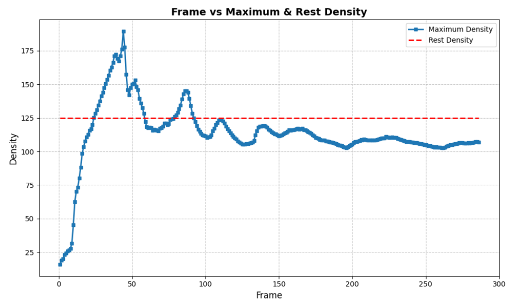

Figure 1: *
(a) Max density over time.

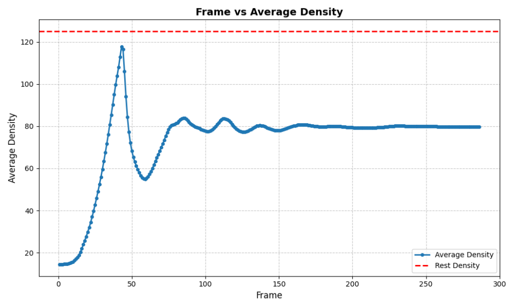

Figure 2: *
(b) Average density over time.

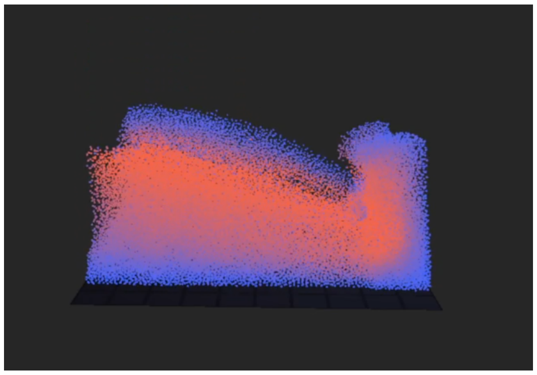

Figure 3: Free-fall (dam break) scenario: (a) Maximum density vs rest density, (b) Average density vs rest density, (c) Visual output showing splash and wave formation.

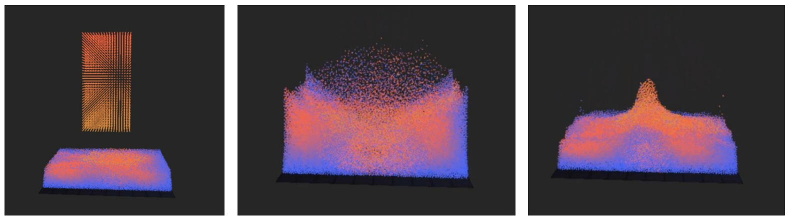

Figure 4: Pre-filled Container Visualization

Pre-filled Container Visualization

The two-fluid interaction scenario, visualized in Fig. 4, demonstrates stable fluid merging behavior, with the falling block smoothly integrating into the pre-existing fluid. The surface remains largely cohesive, with noticeable ripples and secondary waves propagating from the point of collision.

Boundary regions exhibit localized particle accumulation, potentially caused by the Jacobi solver's local nature, limiting the speed of pressure redistribution. However, incompressibility is maintained, with no significant fluid loss or compression artifacts. The interaction aligns with behaviors described in [4], where local position corrections can result in slightly delayed fluid equilibrium.

Density Evolution Analysis

The density plots reveal that while maximum densities exceed the rest density immediately after impact (due to compression), they stabilize over time. Average densities consistently converge below the rest density, indicative of the approximate nature of the constraint resolution in position-based methods. Similar density trends are reported in [4] and are characteristic of real-time PBF solvers under moderate iteration counts.

Overall, the simulation demonstrates high visual fidelity and general alignment with expectations from position-based fluid modeling.

Computational Performance Evaluation

To quantify the computational cost of the simulation, we analyzed the average execution time per stage within a typical frame. As shown in Figure 5, the Density and FindNeighbors stages are the most time-consuming, both around 0.02 ms on average per frame. This is expected, as neighbor searching scales poorly with increasing particle count due to our brute-force search implementation, and density evaluation requires iterating over all neighboring particles per particle.

Stages such as PositionUpdate, ExternalForces, and VelocityUpdate exhibit relatively lower execution times, each averaging under 0.015 ms per frame. The VorticityViscosity stage also shows a modest computational footprint.

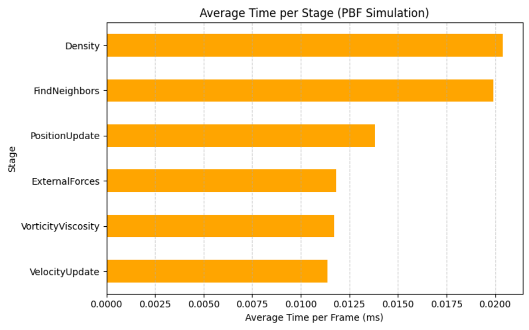

Figure 5: Average time per stage for PBF simulation. Neighbor search and density computation dominate frame time.

This profiling highlights the scalability bottlenecks within the simulation pipeline. The high cost associated with the brute-force neighbor search aligns with the observed frame rate degradation at higher particle counts (notably above 100,000 particles), as discussed earlier. Optimizations such as spatial hashing or hierarchical grids would be crucial to alleviate this limitation, enabling more efficient large-scale simulations.

5 Limitations

While our implementation successfully follows the Position-Based Fluids (PBF) framework, it presents several limitations. Particle stacking near boundaries occurs due to inaccurate density estimations when particles interact with solid objects, a known issue in the literature [5].

The Jacobi-based iterative solver used for enforcing incompressibility limits convergence speed as information propagates slowly across the particle system. Since position corrections use only current neighbor data without leveraging immediate updates, convergence becomes slower as particle count increases. Advanced solvers such as red-black Gauss-Seidel or multi-scale approaches could accelerate convergence [4].

Additionally, the artificial pressure term implemented in our `applyPositionUpdate.comp` shader (with $k = 0.1$ and $n = 4.0$) creates an undesirable coupling between anti-clustering behavior and surface tension effects. Decoupling these parameters would allow more intuitive control over simulation behavior.

Our timing measurements confirm that neighbor finding consumes a significant portion of each frame, becoming a performance bottleneck as the simulation scales beyond 100,000 particles. Furthermore, our current rendering pipeline creates unnecessary overhead by transferring data between GPU and CPU, which could be eliminated with a GPU-only approach.

Future work should focus on optimization techniques like shared memory usage in compute shaders, improved spatial hashing, and extended capabilities for multiphase interactions and complex boundary handling.

6 Conclusion

Overall, we have implemented a successful working implementation of positional-based fluid dynamics for fluid simulations. We evaluated our implementation on two scenarios where the fluid particles interact realistically with an empty container and a pre-filled container. Furthermore, we achieve promising performance in terms of frame time and memory usage. Most importantly, our fluid dynamics implementation using PBF improves on pre-2013 methods with a more stable and more computationally efficient in the compressibility constraint.

References

[1] Kenny Bodin, Claude Lacoursière, and Martin Servin. Constraint fluids. *IEEE Transactions on Visualization and Computer Graphics*, 18(3):516–526, March 2012.

[2] J. U. Brackbill, D. B. Kothe, and H. M. Ruppel. Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48(1):25–38, 1988.

[3] Peter J. Cossins. Smoothed particle hydrodynamics, 2010.

[4] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics*, 32:104:1–, 07 2013.

[5] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

[6] Karthik Raveendran, Chris Wojtan, and Greg Turk. Hybrid smoothed particle hydrodynamics. In Stephen Spencer, editor, *SCA: ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 33–42. ACM, 2011.

[7] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, July 2005.