Particle Based Fluids in Rust

Luka Mijalkovic
luka.mijalkovic@mail.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

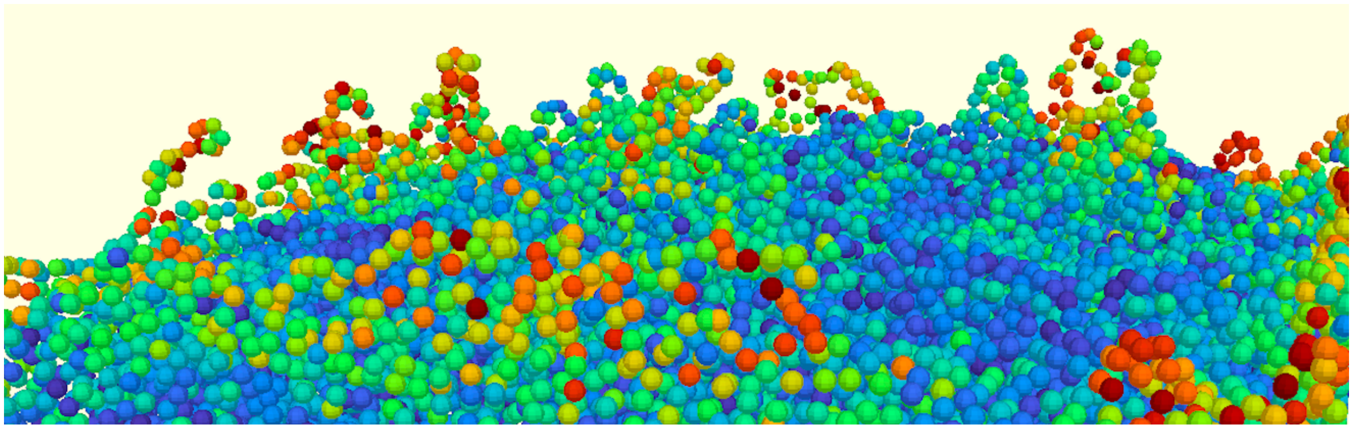


Figure 1. 4000 active particles, method from [Macklin and Müller 2013]

CCS Concepts: • **Computing methodologies → Physical simulation**.

Keywords: fluid simulation, SPH, rustlang, constraint fluids, position based dynamics

1 Introduction

In their paper "Position Based Fluids", [Macklin and Müller 2013] present a constraint based approach to model fluids. They begin by introducing a constraint equation obtained from [Bodin et al. 2012] involving neighboring particles in a fluid simulation. From there, they explain the piecewise development of a constraint gradient initially given by [Monaghan 1992], of which they use to perform a series of Newton steps along the gradient each integration time step.

In this implementation, I make use of the Rust library `rayon` when parallelizing the algorithm, allowing it to utilize multiple CPU threads. Additionally, I make use of an octree data structure to speed up the particle neighbor search. Finally, I decouple the rendering and physics threads to allow for smoother previewing. This is all in done in contrast to [Macklin and Müller 2013], who develop their algorthm to run on the GPU through use of NVIDIA's proprietary `CUDA` architecture. Hence, while my implementation is not as efficient it is far easier to implement, requiring less boilerplate code and no GPU kernels.

2 Constraints

2.1 Constraint Function

Fundamentally, the algorithm by [Macklin and Müller 2013] works by enforcing a constraint equation on each of the particles. Originally from [Bodin et al. 2012], they define a *density constraint function*, C_i, for the i^{th} particle as follows:

$$C_i(\mathbf{p}_1, ..., \mathbf{p}_n) = \frac{\sum_j W(\mathbf{p}_i - \mathbf{p}_j, h)}{\rho_0} - 1 \tag{1}$$

The inputs, $\mathbf{p}_1, ..., \mathbf{p}_n$, are the n positions of all particles neighboring the i^{th} particle within some radius r. The variable ρ_0 represents the rest density of the fluid. The summation in the numerator is often denoted ρ_i, and it is known as the *standard SPH density estimator* for a particle. The summation goes over the n particles in the input, $\mathbf{p}_1, ..., \mathbf{p}_n$.

$W(\vec{\mathbf{v}}, h)$ is a function that takes in an n dimensional vector, $\vec{\mathbf{v}}$, a unit distance, h, and returns a scalar of $\mathbb{R}^{\geq 0}$. It is discussed in more detail in A.

2.2 Constraint Function Gradient

With the constraint function C_i well defined, we can now establish its gradient with respect to any input particle $\mathbf{p}_k$, where $1 \leq k \leq n$. Borrowing notation from [Macklin and Müller 2013], this is denoted as $\nabla_{\mathbf{p}_k} C_i$. The original equation is derived from [Monaghan 1992], after which [Macklin and Müller 2013] have provided a piecewise definition like so:

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j, h) & k = i \\ -\nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_k, h) & k \neq i \end{cases} \tag{2}$$

I have adopted my own notation to clarify some ambiguity in the definition. Again, the summation is only over neighboring particles of particle i.

2.3 Constraint Equation

Using the constraint function C_i and its gradient $\nabla_{\mathbf{p}_k} C_i$ we can develop the most important equation given by [Macklin and Müller 2013]. By trying to enforce uniform density for every single particle, we can make the following equation hold for all particles:

$$C_i(\mathbf{p}_1 + \Delta\mathbf{p}_1, ..., \mathbf{p}_n + \Delta\mathbf{p}_n) = 0 \tag{3}$$

From here, they develop the following first order approximation:

$$\begin{aligned} 0 &= C_i(\mathbf{p}_1 + \Delta\mathbf{p}_1, ..., \mathbf{p}_n + \Delta\mathbf{p}_n) && (4)\\ &\approx C_i(\mathbf{p}_1, ..., \mathbf{p}_n) + \sum_j (\nabla_{\mathbf{p}_j} C_i)^\top \Delta\mathbf{p}_j && (5)\\ &\approx C_i(\mathbf{p}_1, ..., \mathbf{p}_n) + \sum_j (\nabla_{\mathbf{p}_j} C_i)^\top \nabla_{\mathbf{p}_j} C_i \lambda_i && (6)\end{aligned}$$

Where λ_i is some arbitrarily small scalar value that will be solved for.

3 Finding λ and $\Delta\mathbf{p}$

3.1 Deriving λ

Given Equation (6), we can derive an equation for λ_i like so:

$$0 \approx C_i(\mathbf{p}_1, ..., \mathbf{p}_n) + \sum_j (\nabla_{\mathbf{p}_j} C_i)^\top \nabla_{\mathbf{p}_j} C_i \lambda_i \tag{7}$$

$$-\sum_j (\nabla_{\mathbf{p}_j} C_i)^\top \nabla_{\mathbf{p}_j} C_i \lambda_i \approx C_i(\mathbf{p}_1, ..., \mathbf{p}_n) \tag{8}$$

$$\lambda_i \approx -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_j (\nabla_{\mathbf{p}_j} C_i)^\top \nabla_{\mathbf{p}_j} C_i} \tag{9}$$

$$\lambda_i \approx -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_j ||\nabla_{\mathbf{p}_j} C_i||^2} \tag{10}$$

3.2 Relaxation Constant $\mathcal{E}$

Notice that Equation (10) will cause numerical issues when the denominator is close to 0. Hence, [Macklin and Müller 2013] suggest a *relaxation parameter*, $\mathcal{E}$. A concept introduced by [Solenthaler and Pajarola 2009] which will ensure this never occurs:

$$\lambda_i \approx -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_j ||\nabla_{\mathbf{p}_j} C_i||^2 + \mathcal{E}} \tag{11}$$

3.3 Assembling $\Delta\mathbf{p}$

From here, [Macklin and Müller 2013] derived an approximation for $\Delta\mathbf{p}_i$ by assembling all λ values like so:

$$\Delta\mathbf{p}_i \approx \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{12}$$

At this point, the algorithm is functionally finished. With an appropriate time integration scheme, the simulation can be run using $\Delta\mathbf{p}$ updates. The terms introduced in 4 are mostly just visual effects.

4 Stability and Realism

4.1 Artificial Pressure

As noted by [Macklin and Müller 2013], when a particle is left with little neighbors it will struggle to reach rest density. Hence, they follow the method of [Monaghan 2000] by adding a constant, s_{corr}, to account for this:

$$\Delta\mathbf{p}_i \approx \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla_{\mathbf{p}_i} W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{13}$$

Where s_{corr} is defined as:

$$s_{\text{corr}} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta\mathbf{q}, h)} \right)^n \tag{14}$$

This boosts surface tension of the fluid. The values recommended were $k = 0.1$, $n = 4$, and $\Delta\mathbf{q}$ as a vector with magnitude in the interval $[0.1h, 0.3h]$.

4.2 Vorticity

In order to reduce energy loss caused by damping in position based methods, [Macklin and Müller 2013] use vorticity confinement. This is done by calculating the vorticity for each particle. For the proceeding equations, m is the number of neighbors:

$$\boldsymbol{\omega}_i = \begin{cases} \sum_j \mathbf{v}_{ij} \times \nabla_{p_j} W(\mathbf{p}_i - \mathbf{p}_j, h) & m > 0 \\ \vec{0} & m = 0 \end{cases} \tag{15}$$

From here, a *location vector*, $\mathbf{N}_i$, is defined:

$$\mathbf{N}_i = \begin{cases} \frac{\nabla|\boldsymbol{\omega}_i|}{||\nabla|\boldsymbol{\omega}_i||} & |\nabla|\boldsymbol{\omega}_i|| > 0 \text{ and } m > 0 \\ \vec{0} & \text{otherwise} \end{cases} \tag{16}$$

Finally, a vorticity force, $\mathbf{f}_i^{\text{vort}}$, can be calculated using $\mathbf{N}_i$ and ω_i:

$$\mathbf{f}_i^{\text{vort}} = \epsilon(\mathbf{N}_i \times \boldsymbol{\omega}_i) \tag{17}$$

Where ϵ is a small scalar which can be tuned.

4.3 Viscosity

Additionally, [Macklin and Müller 2013] add a simple XSPH viscosity correction step from [Schechter and Bridson 2012]:

$$\mathbf{v}_i^{\text{visc}} = \mathbf{v}_i + c \sum_j \mathbf{v}_{ij} W(\mathbf{p}_i - \mathbf{p}_j, h) \tag{18}$$

5 Algorithm

5.1 Pseudocode

The implementation I used was 1:1 with that which was written by [Macklin and Müller 2013]. However, as this one is CPU based I have notated where the thread spawning/joining must occur.

Algorithm 1 CPU Multithreaded Position Based Fluids

```
1:  spawn_assign_k_threads(particles)    ▷ Split work
2:  for all particles assigned in thread $k$ do
3:      $\mathbf{v}_i^{t+1} \leftarrow \mathbf{v}_i^t + \Delta t \mathbf{f}_i^{\text{ext}}$
4:      $\mathbf{p}_i^{t+1} \leftarrow \mathbf{p}_i^t + \Delta t \mathbf{v}_i^{t+1}$
5:  end for
6:  join_threads()    ▷ Combine all particle calculations
7:  neighbor_search(particles)    ▷ Varies on search
8:  for all solver iteration $q$ do
9:      spawn_assign_k_threads(particles)
10:     for all particles assigned in thread $k$ do
11:         $\lambda_i^q \leftarrow -\frac{C_i(\mathbf{p}_1^{t+1},\ldots,\mathbf{p}_n^{t+1})}{\sum_j ||\nabla_{\mathbf{p}_j^q} C_i||^2+\mathcal{E}}$
12:     end for
13:     join_threads()
14:     spawn_assign_k_threads(particles)
15:     for all particles assigned in thread $k$ do
16:         $\Delta \mathbf{p}_i^q \leftarrow \frac{1}{\rho_0}\sum_j (\lambda_i^q + \lambda_j^q)\nabla_{\mathbf{p}_i^{t+1}} W(\mathbf{p}_i^{t+1} - \mathbf{p}_j^q, h)$
17:         collision_response()
18:         $\mathbf{p}_i^{t+1} \leftarrow \mathbf{p}_i^{t+1} + \Delta \mathbf{p}_i^q$
19:     end for
20:     join_threads()
21: end for
22: spawn_assign_k_threads(particles)
23: for all particles assigned in thread $k$ do
24:     $\mathbf{v}_i \leftarrow \frac{1}{\Delta t}(\mathbf{p}_i^{t+1} - \mathbf{p}_i^t)$
25: end for
26: join_threads()
27: apply_vorticity()    ▷ Use $\mathbf{p}^{t+1}$ not $\mathbf{p}^t$
28: apply_viscosity()    ▷ Use $\mathbf{v}^{t+1}$ not $\mathbf{v}^t$
```

6 Benchmarks

6.1 Neighbor Search

For the neighbor search (each run through of Algorithm 1, line 7), I tested two algorithms. The first was a classic $O(n^2)$ search, in which I checked for every particle i, if another particle j was within radius r. This proved to be very inefficient and greatly slowed down the algorithm. When multithreading this algorithm, the performance actually went down, as can be seen in Table 1. I have a suspicion this has to do with CPU cache misses [Meyers 2014] due to how the data was aligned, but I can't say for certain. Regardless, it wasn't very fast so I didn't see much of a need to intensely profile it. Rather, I just opted for a faster data structure to query points, an octree.

As suggested by [Macklin and Müller 2013], I also dynamically generated an octree each run through in order to query neighbor points. I made use of the open source Rust library `octree` to instantiate the data structure. After creating the octree I queried for the neighbors of all points, using both a multithreaded and a single-threaded query. As can be seen in Table 1, the performance seen when multithreading the point querying gave a big improvement of 250%. This in stark contrast to the naive algorithm which performed approximately 10% worse multithreaded.

Table 1. Runtime of differing neighbor search algorithms, 95% confidence interval. Tested with 5,000 particles on a 13" 2016 Macbook Pro, measured using `criterion`

Algorithm	Lower (ms)	Mean (ms)	Upper (ms)
Naive, Single-threaded	92.423	93.320	94.345
Naive, Multi-threaded	101.67	102.34	103.06
Octree, Single-threaded	50.293	50.646	51.030
Octree, Multi-threaded	20.092	20.318	20.613

6.2 Flamegraphs

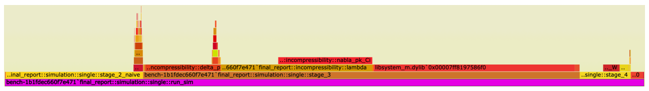


Figure 2. Flamegraph of performance using the single-thread implementation with naive search. Tested with 5,000 particles.

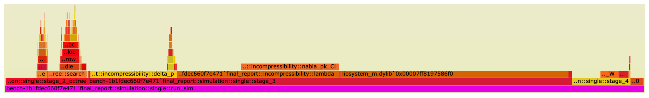


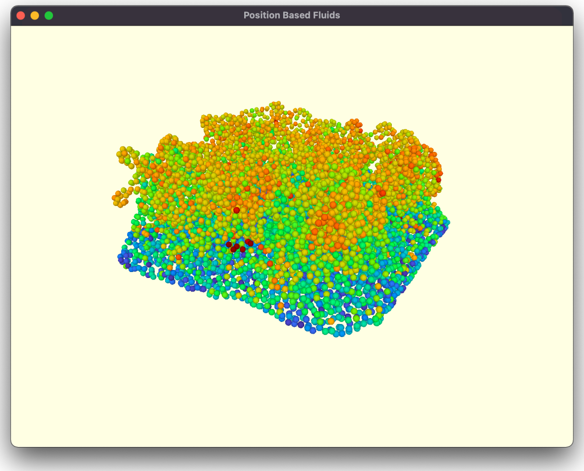
Figure 3. Flamegraph of performance using the single-thread implementation with octree search. Tested with 5,000 particles.

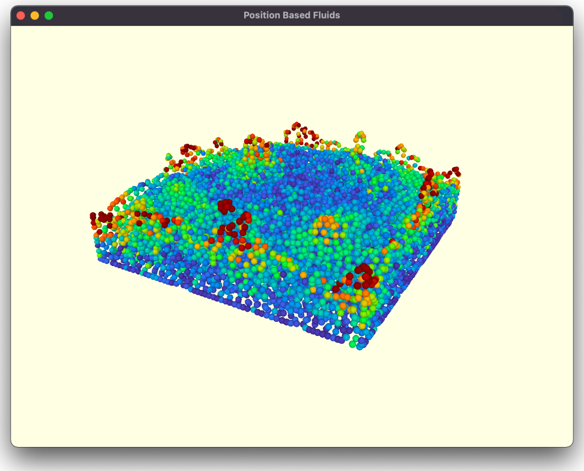
In order to create flamegraphs of the algorithm, I disabled function inlining by leaving `[inline(never)]` above the important sections and ran it on the single-threaded version. This yielded flamegraphs Figure 2 and Figure 3.

As one can see by the flamegraphs, octree performance is indeed better than naive for the single threaded version. Experimentally it is very close to the isolated benchmarks from Table 1. It is also evident that a large amount of time was spent calculating both λ_i^q and $\Delta\mathbf{p}_i^q$ on lines 11 and 16 of Algorithm 1. This makes sense because they are indeed heavy summations.

Interestingly, the algorithm seemed to spend roughly half the time querying the octree, and the other half building.

Creating flamegraphs for the multi-threaded implementation was not practical because `rayon` muddles the call stack when thread switching.





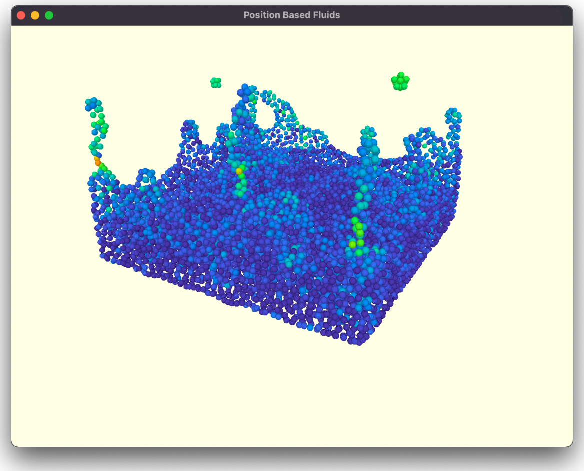


Figure 4. Progression of a simulation with 10,000 particles

6.3 Overall

Table 2. Average simulation step time using differing setups, 95% confidence interval. Tested with 5,000 particles and 3 solver iterations.

Algorithm	Lower (ms)	Mean (ms)	Upper (ms)
Naive, Single-threaded	424.65	426.67	429.17
Naive, Multi-threaded	218.29	219.75	221.44
Octree, Single-threaded	369.56	370.84	372.41
Octree, Multi-threaded	139.78	140.73	142.00

I additionally ran `criterion` on the average runtime of an entire simulation step. This was done with randomly generated locations each time, and the results can be found in Table 2. It might seem like this table contradicts Table 2 due to higher performance with the multithreaded naive search, however this multithreading was not limited to the search algorithm. Every line spawning threads in Algorithm 1 was executed, thus the overall speed up triumphed the worse search algorithm.

The octree search with a multithreaded implementation performed the best overall, as was expected. Steps were calculated roughly 300% faster than the single threaded naive implementation. All testing was performed on a 13 inch Macbook Pro 2016.

7 Rendering

In order to render particles I made use of `glium`, a memory safe wrapper around `OpenGL`. Batch rendering was performed to reduce the amount of VAO binding required. This allowed for thousands of particles to barely affect the GPU rendering times.

8 Conclusions

Due to ease of use with `rayon`, parallelizing the code was relatively straight forward. It mostly consisted of changing top level iterators to parallel iterators, and the performance benefits were enormous, as seen in 6. Furthermore, because this did not use `CUDA` as [Macklin and Müller 2013] did, the implementation is multiplatform.

Perhaps making use of Vulkan or `wgpu` compute shaders may allow for a far faster implementation, however this was untested. This would get around `CUDA`'s NVIDIA only application. However, this could still be impractical for e.g. video games due to GPU/CPU latency per frame. The use of GPU for compute code would also take away from real-time rendering. While this could make for a big leap in performance, testing is and profiling is required.

References

Kenneth Bodin, Claude Lacoursière, and Martin Servin. 2012. Constraint Fluids. *IEEE Transactions on Visualization and Computer Graphics* 18, 3 (2012), 516–526. https://doi.org/10.1109/TVCG.2011.29

Miles Macklin and Matthias Müller. 2013. Position Based Fluids. (April 2013). http://mmacklin.com/pbf_sig_preprint.pdf.

Scott Meyers. 2014. *Cpu Caches and Why You Care.* Nokia. https://www.youtube.com/watch?v=WDIkqP4JbkE&t=239s

J.J. Monaghan. 1992. Smoothed Particle Hydrodynamics. *Annual review of astronomy and astrophysics* 30, 1 (1992), 543–574. https://doi.org/10.1146/annurev.aa.30.090192.002551

J.J. Monaghan. 2000. SPH without a Tensile Instability. *J. Comput. Phys.* 159, 2 (2000), 290–311. https://doi.org/10.1006/jcph.2000.6439

Hagit Schechter and Robert Bridson. 2012. Ghost SPH for animating water. *ACM transactions on graphics* 31, 4 (August 2012), 1–8. https://doi.org/10.1145/2185520.2185557

B. Solenthaler and R. Pajarola. 2009. Predictive-Corrective Incompressible SPH. In *ACM SIGGRAPH 2009 Papers* (New Orleans, Louisiana) *(SIGGRAPH '09)*. Association for Computing Machinery, New York, NY, USA, Article 40, 6 pages. https://doi.org/10.1145/1576246.1531346

A Kernel Functions

A.1 Properties

The function $W(\vec{\mathbf{v}}, h)$ is a unique function known as a *smoothing kernel.* It takes in an n dimensional vector, $\vec{\mathbf{v}}$, and a unit distance, h. It gives the influence of vector a $\vec{\mathbf{v}}$ away from the origin in terms of unit distance h; the farther away a vector, the less influence it has. Hence h acts as a normalization parameter. There is thus flexibility in how a kernel can be defined, however it **must** adhere to the following properties:

$$W(\vec{0}, h) = 0 \tag{19}$$

$$\int_{\mathbb{R}^n} W(\vec{\mathbf{v}}, h) d\vec{\mathbf{v}} = 1 \tag{20}$$

A.2 Poly6 Kernel

For occurrences of $W(\vec{\mathbf{v}}, h)$, [Macklin and Müller 2013] recommended the Poly6 Kernel, of which is defined as follows:

$$W_{\text{Poly6}}(\vec{\mathbf{v}}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |\vec{\mathbf{v}}|^2)^3 & 0 \le |\vec{\mathbf{v}}| \le h \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

The benefit to this equation is that the squared normal can be used, removing the need for a square root calculation.

A.3 Spiky Kernel

For occurrences of $\nabla_{\vec{\mathbf{v}}} W(\vec{\mathbf{v}}, h)$, [Macklin and Müller 2013] recommended the gradient of the Spiky Kernel. The Spiky Kernel is defined as:

$$W_{\text{Spiky}}(\vec{\mathbf{v}}, h) = \frac{15}{\pi h^6} \begin{cases} (h - |\vec{\mathbf{v}}|)^3 & 0 \le |\vec{\mathbf{v}}| \le h \\ 0 & \text{otherwise} \end{cases} \tag{22}$$