Position Based Fluids

Jacob Chmura*
jacob.chmura@mail.utoronto.ca
University of Toronto
CSC417: Physics Based Animation

Abstract

This document is a write-up of the final project for the Fall 2020 Physics Animation Course course at the University of Toronto. In this project, I implemented a particle-based fluids simulation based on [8]. The code can be found at https://github.com/JacobChmura/PBF.

Keywords: Fluid Simulation, Constraint Fluids, Position Based Dynamics

1 Introduction

Fluid Animation is not doubt one of the most interesting problems in computer graphics, grounded in deep theory from Partial Differential Equations and intricate techniques in numerical analysis. Historically, modelling fluid dynamics have been an active area of mathematical research, and with the evolution of computational methods, simulating such dynamics has become a vital endeavour among the computer graphics community, challenged with jointly optimizing their methods for efficiency and visual appeal.

2 Background

2.1 The Equations of Fluids

The fundamental equations of interest are the *Incrompressible Navier-Stokes*:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \frac{1}{\rho} \nabla p = \mathbf{g} + v \nabla^2 \mathbf{u} \tag{1}$$

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

where $\mathbf{u}$ is the velocity of the fluid, ρ denotes the density of the fluid, $\mathbf{g}$ is the force due to gravity, the scale p described the pressure of the fluid, and v measure how viscous a fluid is. There is not enough space to give a thorough explanation of the richness of these equation here, so instead, we give an intuitive explanation of what they describe. [1]

The first component is typically called the *momentum equation* and it dictates how a fluid accelerates due to the forces acting on it. The forces can be classified as:

- External Forces
  - Gravity
- Internal Forces
  - Pressure (keeps the fluid at constant volume)
  - Viscosity (resist deformation)

[1] A thorough treatment of these equations and how they give rise to numerical algorithms can be found in [1]

With this in mind, the first equation essentially reduces to *Newtons Second Law* and roughly translates to :

Fluid evolves based on the force of gravity, moving areas of high pressure to areas of low pressure, while resisting local changes in shape

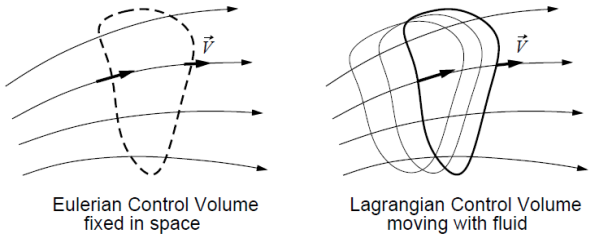
The second equation is an *incomprehensibility constraint* that enforces the fact that the fluid vector field is divergence-free. Intuitively, this asks that locally there are no *sinks* or *sources*.

2.2 Eulerian and Lagrangian Fluids

The Eulerian method to fluid simulation involves a discetization of 3D-space into *grid cells*, each of which store important quantities such as density and volume. For this reason the Eulerian method is often referred to as *Grid-Based*. Simulating the fluid amounts to tracking the variation of these physical quantities over time within these fixed cells and of course the quality of the simulation depends on the granularity of the grid discetization.

The Lagrangian method instead approximates the fluid by tiny atomic elements, or particles, each of which store their own physical quantities such as position, velocity and pressure. For this reason, the Lagrangian method is often referred to as *Particle-Based*. In this case, the quality of the simulation depends on the number of particles in the system. Position Based Fluids models the fluid system using particles, and therefore aligns with the Lagrangian approach.

Figure 1. Euler vs. Lagrangian Fluids



2.3 Smoothed Particle Hydrodynamics

Under the Lagrangian framework of simulating a fluid, it is essential that we can compute scalar and vector quantities between particles in our system - the so called *particle-particle*

interactions:

$$(p_i, p_j) \mapsto F(p_i, p_j) \tag{3}$$

where p_i, p_j are particles in our system and F is some function. When there are N particles, computing particle-particle interactions between each pair of particles in our system is $O(N^2)$ which quickly leaves the realm of possibility of real-time fluid simulation. Because our quantities of interest are fundamentally local, the remedy is to only consider the particle interaction of p_i, p_j if these particles are sufficiently close in space: $||p_i - p_j|| < h, h > 0$.

Next, to solve the *Navier-Stokes* equation on particles, we need to generate smooth fields from discrete quantities given at our particle locations. The principle of *Smoothed Particle Hydrodynamics* [10] is to use some family of *smoothing kernels* $W(\mathbf{r}, h)$ to dissipate discretely sampled fields. More precisely, the influence of particle p_j on particle p_i is weighted by:

$$W(||p_i - p_j||, h) \in \mathbb{R} \tag{4}$$

where $||p_i - p_j|| > h \implies W(||p_i - p_j||, h) = 0$.

Assuming the kernels are normalized so that $\int W(||p - p_j||, h)dp = 1$, this method gives rise to the density of particle p_i:

$$\rho_i = \sum_j m_j W(||p_i - p_j||, h) \tag{5}$$

where m_j is the point mass of particle j, and finally, given the density values of individual particles, arbitrary smooth fields F can be computed via:

$$F_{p_i} = \sum_j m_j \frac{F(p_i, p_j)}{\rho_j} W(||p_i - p_j||, h) \tag{6}$$

This formulation allows one to approximately evaluate the equations of a fluid and solve for an update rule that leads to the simulation of fluid, but there are some limitations. Namely, this method is extremely sensitive to density fluctuations when their are neighborhood deficiencies. Thus, if there are not enough particles in a neighborhood, density estimates become inaccurate, and update equations become unstable. Adding more particles, using smaller time-steps, or explicitly enforcing incompressibility are far from adequate solutions, leading to increased computational costs and leaving the realm of interactive simulation.

2.4 Position Based Dynamics

Whereas most approaches for simulating dynamics involve the integration of velocities due to forces, *Position Based Dynamics* [9] works directly on the positions in the system and omits the integration of velocity altogether; see Table 1.

Given a constraint $C : \mathbb{R}^{3m} \to \mathbb{R}$ as a function of m positions like $C(\mathbf{x}) = K$, position based dynamics aims to find a position correction $\Delta\mathbf{x}$ such that:

$$C(\mathbf{x} + \Delta\mathbf{x}) = 0 \tag{7}$$

Forced Based	Position Based Dynamics
Forces	Constraints
Time Integration	Constraint Projection

Table 1. The Position Based Dynamics Methodology

Linearizing the left hand side and relaxing the constraint with a regularization parameter ϵ to avoid instability yields:

$$\Delta\mathbf{x} \approx \nabla C(\mathbf{x})\lambda \tag{8}$$

$$C(\mathbf{x} + \Delta\mathbf{x}) \approx C(\mathbf{x}) + \nabla C^T \nabla C\lambda + \epsilon\lambda = 0 \tag{9}$$

$$\tag{10}$$

upon which solving for λ becomes the goal.

3 Position Based Fluids

The *Position Based Fluids* [8] framework leverages the earlier components from Smoothed Particle Hydrodynamics and Position based Dynamics. Here we given an outline of the main components of the algorithm.

Let $\{\mathbf{p_i}\}_{i=1}^N$ denote the positions of the N particles in our fluid, $\mathbf{p_i} \in \mathbb{R}^3$. Each particle will store the scalar quantities: mass m_i and density ρ_i. Since the mass is constant throughout simulation, we fix $m_i = 1\ \forall i = 1, ..., N$ and ignore it for the remainder of the document.

3.1 Enforcing Incompressibility

Let ρ_0 denote the rest density of the fluid. [2]. This leads to a constraint for each particle i:

$$C_i(\mathbf{p}_1^{(i)}, \mathbf{p}_2^{(i)}, ..., \mathbf{p}_{n_i}^{(i)}) = \frac{\rho_i}{\rho_0} - 1 \tag{11}$$

$$= \frac{\sum_j W(||p_i - p_j||, h)}{\rho_0} - 1 \tag{12}$$

where $\mathbf{p}_k^{(i)}, k = 1, ..., n_i$ denote the neighbors of particle i and we have plugged in our density estimator with $m_j = 1$.

Applying the Position Based Dynamics linearization to the above constraint yields:

$$\nabla_{\mathbf{p}_k^{(i)}} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_k^{(i)}} W(||\mathbf{p_i} - \mathbf{p_j}||, h) & \text{if } k = i \\ -\nabla_{\mathbf{p}_k^{(i)}} W(||\mathbf{p_i} - \mathbf{p_j}||, h) & \text{otherwise} \end{cases} \tag{13}$$

$$\lambda_i = -\frac{C_i(\mathbf{p}_1^{(i)}, \mathbf{p}_2^{(i)}, ..., \mathbf{p}_{n_i}^{(i)})}{\sum_k |\nabla_{\mathbf{p}_k^{(i)}} C_i|^2 + \epsilon} \tag{14}$$

The total position update for particle p_i becomes:

$$\Delta\mathbf{p_i} = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(||\mathbf{p_i} - \mathbf{p_j}||, h) \tag{15}$$

[2] Choosing ρ_0 can be heuristically motivated given known fluid densities from physics

3.2 Tensile Instability

When a particle has only a few neighbours it is unable to satisfy the rest density constraint and for this reason, an *artificial* pressure term is added to avoid particle clustering and improve fluid distribution. Specifically:

$$s_{corr}(i,j) = -k(\frac{W(||\mathbf{p_i} - \mathbf{p_j}||, h)}{W(\Delta q, h)})^n \tag{16}$$

where $|\Delta q| = 0.1 * h, k = 0.1, n = 4$. The position update becomes:

$$\Delta \mathbf{p_i} = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}(i,j)) \nabla W(||\mathbf{p_i} - \mathbf{p_j}||, h) \tag{17}$$

3.3 Vorticity Confinement

Realistic fluids often display visible *splashing*, but the density constraint disenscitives such behaviour. Introducing turbulent motion is achieved by adding a vorticity force.

An estimation of the vorticity at the location of particle $\mathbf{p_i}$ is given by:

$$\omega_i = \nabla \times \mathbf{v} \approx \sum_j (\mathbf{v_j} - \mathbf{v_i}) \times \nabla_{\mathbf{p}_j^{(i)}} W(||\mathbf{p_i} - \mathbf{p_j}||, h) \tag{18}$$

And the corrective force, controlled by parameter ϵ_{vort} is:

$$\mathbf{f}_i^{vort} = \epsilon_{vort} \cdot (\frac{\nabla |\omega|_i}{||\nabla |\omega|_i||} \times \omega_i) \tag{19}$$

3.4 XSPH Viscocity

Finally, to reduce particle disorder the computed velocity field is smoothed to improve coherence as follows:

$$\mathbf{v}_i^{(new)} = \mathbf{v_i} + c \sum_j (\mathbf{v_j} - \mathbf{v_i}) \cdot W(||\mathbf{p_i} - \mathbf{p_j}||, h) \tag{20}$$

where $c = 0.01$.

4 Implementation

I implement Position based Fluids on CPU using the Eigen library [4] for linear algebra operations and libigl [5] for visualization. Our fluid is constrained to a cube in $\mathbb{R}^3$ under which collision detection becomes straightforward. An overview of the algorithm is found in **Algorithm 1**

4.1 External Forces

The update begins by applying external forces to the system and predicting the position update due to these forces. In our simulation, there are only two external forces. The first, is the force due to gravity, and it remain active on the system throughout. The second, is an optional *user force mode* that can be interactively toggled on program launch. The user force is applied uniformly to all particles in the system and is determined by the location of the user's mouse in world space, and a configurable strength parameter.

Algorithm 1 Position Based Fluids

procedure STEP($\mathbf{p_1}, ..., \mathbf{p_N}$) ▷ Particle Positions
  for *all particles* $i = 1, .., N$ **do**
    $v_i \leftarrow v_i + dt * f_{ext}(\mathbf{p_i})$ ▷ Apply External Forces
    $\mathbf{p_i}^* \leftarrow \mathbf{p_i} + dtv_i$ ▷ Predict Positions
  end for
  for *all particles* $i = 1, .., N$ **do**
    $FindNeighbors(\mathbf{p_i}^*)$ ▷ Get Neighboring Particles
  end for
  for *Jacobi Iterations* **do**
    for *all particles* $i = 1, .., N$ **do**
      $\lambda_i \leftarrow [Equation 14]$ ▷ Compute Constraints
    end for
    for *all particles* $i = 1, .., N$ **do**
      $\Delta \mathbf{p}_i \leftarrow [Equation 17]$ ▷ Position Correction
      Collision Detection and Response
    end for
    for *all particles* $i = 1, .., N$ **do**
      $\mathbf{p_i}^* \leftarrow \mathbf{p_i}^* + \Delta \mathbf{p_i}$ ▷ Update Position
    end for
  end for
  for *all particles* $i = 1, .., N$ **do**
    $v_i = \frac{1}{dt}(\mathbf{p_i}^* - \mathbf{p_i})$ ▷ Update Velocity
    $\mathbf{p_i} \leftarrow \mathbf{p_i}^*$ ▷ Update Position
  end for
end procedure

4.2 Neighbors Search

A critical sub-routine in the simulation procedure is the finding of particle neighbors. To the end, we implement a *Spatial Hash Grid* [2] data-structure that we query and update at every simulation step. We discretize our boundary cube into cells of side length h where h is our kernel radius. The keys in our hash table are an enumeration of the cells which remain fixed during simulation, and our values are a set of particle id's that are currently positioned within that cell. Typically one would use large prime numbers to hash the 3d space to a single integer. However, since our boundary box is fixed, the radius of influence among particles is fixed and common, we directly use the enumerated cell as a key. To find neighbours of a particle, we join the list of particles in the 8 adjacent cells. This conservative query ensures we find all neighbors within distance h. Both insertion and neighborhood finding is $O(N)$. In reality, most particles won't change cells within a given timestep. This is why we do not recompute neighbors after each Jacobi position correction. It's also possible that after an entire time step update a particle does not leave it's hash cell, but for simplicity we re-insert all particles into our table after every time step. Inserting and query of particles is outline in **Algorithm 2**. In our simulation, *lower bound* and *upper bound* denote the edges of our boundary box at ±1.

Algorithm 2 Spatial Hash Grid

```
procedure INSERT(p)                                   ▷ Particle Position
    cellcoord ← (p−lower bound) / (upper bound - lower bound)
    cellcoord ← cellcoord ∗ ((upper bound - lower bound)/h) − 1
    return tuple(cellcoord) ▷ Create a tuple for hash key
end procedure
procedure FindNeighbors(p)                            ▷ Particle Position
    cellcoord ← cellCoord(p) ▷ compute cell coordinate
    neighbors = []
    for dx = −1, 0, 1 do
        for dy = −1, 0, 1 do
            for dz = −1, 0, 1 do
                cellcoord+ = (dx, dy, dz) ▷ adjacent cells
                neighbors.extend(HashGrid[tuple(cellcoord)])
            end for
        end for
    end for
    return neighbors
end procedure
```

Algorithm 3 Collision Detection and Response

```
procedure COLLISION DETECTION AND RESPONSE(p, v)▷ Particle Position and Velocity
    for axis x, y, z do
        if p_axis < lowerbound then
            p_axis ← lowerbound
            if v_axis < 0 then
                v_axis* = −α
            end if
        end if
        if p_axis > upperbound then
            p_axis ← upperbound
            if v_axis > 0 then
                v_axis* = −α
            end if
        end if
    end for
end procedure
```

4.3 Jacobi Loop

The Jacobi loop involves applying the theory derived earlier to iteratively compute particle position correction to satisfy the density constraints. In our simulation we use between 1 and 3 Jacobi Iterations based on the number of particles in our simulation.

4.4 Smoothing Kernels

For density estimation we use the *Poly 6* kernel given by:

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3 & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{21}$$

For gradient estimation we use the *Spiky* kernel whose gradient is given by:

$$\nabla W_{spiky}(r,h) = \frac{-45}{\pi h^6}\begin{cases}(h-r)^2 & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{22}$$

4.5 Collision Detection and Response

Given that our only solid object at this stage is the boundary box, the collision detection and response is quite simple. We check whether our position predictions exceed the fixed boundaries of ±1 on each axis, and if so, we clip the axis position to the boundary, and negate the velocity component in that direction with a *rebound factor* α. Typically $\alpha = 2$ is chosen. The logic is outlined in **Algorithm 3**.

5 Further Work

The current implementation at https://github.com/JacobChmura/PBF is serialized, however it's worth noting that each loop over the particles in **Algorithm 1** can be done in parallel. A fully vectorized implementation using $O(N^2)$ memory can be found in one of my commits however the overhead cost did not beat out a simpler iterative approach once compiler optimization was introduced. Of course, the method should be implemented on GPU hardware, and this will be my next step.

Rendering of the fluid is orthogonal to the contribution of the paper, so it was not prioritized.

6 Acknowledgements

As mentioned earlier, the Eigen Library [4] and the Libigl Library [5] were used throughout the code. Moreover, the project structure was founded upon previous CSC417 assignments [7]. The following papers [10], [9] were important to my understanding of position based fluids [8]. I found these notes [1], [3] to greatly help my understanding of fluid simulation as a general problem. The master thesis [6] provided a detailed overview of the position based fluid method which helped check my understanding. Moreover, I learned about the relevance of Spatial Hash Data structures in animatin from [2].

References

[1] Robert Bridson and Matthias Muller-Fischer. 2007. Fluid Simulation. *SIGGRAPH 2007 Course Notes* (2007). https://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf

[2] Ratan K. Guha Erin J. Hastings, Jaruwan Mesit. 2005. Optimization of Large-Scale, RealTime Simulations by Spatial Hashing. (2005).

[3] M. S. Shadloo et Al. 2016. Smoothed Particle HydroDynamics for Fluid Flows, towards industrial Applications-Motivations, current state, and challenges. (2016). https://www.researchgate.net/publication/303814951_Smoothed_particle_hydrodynamics_method_for_fluid_flows_towards_industrial_applications-Motivations_current_state_and_challenges

[4] Gael Guennebaud, Benoit Jacob, et al. 2010. Eigen v3. (2010). http://eigen.tuxfamily.org.

[5] Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. (2018). https://libigl.github.io/.

[6] Ming-Yen Kuo. 2016. Position Based Fluids. *Bournemouth University* (2016). https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc16/09/mpthesis.pdf.

[7] David Levin. 2021. CSC417 Physics Based Animation: FInite Elements 3d. (2021). https://github.com/dilevin/CSC417-a3-finite-elements-3d.

[8] Miles Macaklin and Matthias Muller. 2013. Position Based Fluids. *ACM Transactions on Graphics* 32, 4 (2013), 1–12. https://doi.org/10.1145/2461912.2461984

[9] Marcus Hennix John Ratcliff Matthias Muller, Bruno Heidelberger. 2006. Position Based Dynamics. *3rd Workshop in Virtual Reality Interactions and Physical Simulation* (2006). https://matthias-research.github.io/pages/publications/posBasedDyn.pdf

[10] J.J. Monaghan. 1992. Smoothed Particle HydroDynamics. *Annual Rev. Astron. Astrophys.* 30 (1992), 543–574. https://doi.org/10.1146/annurev.aa.30.090192.002551