# A Complete High Performance Simulation and Rendering Pipeline for Fluid

Quanquan Peng*
bartholomew@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Yiyu Liu*
liu_yiyu@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

## Abstract

Simulating and rendering fluid in real time has a lot of challenges. The simulation and rendering must be very efficient while keeping the quality. On top of that, we need to exploit the computing power as much as possible by parallelizing the calculation. Therefore, We design and implement a complete simulation and rendering pipeline for fluid. We leverage the computational power by means of parallelization with Taichi framework [9–11]. With PBF, screen-based method and Layered Neighborhood Method optimization, we can achieve about 30 FPS with CUDA on a commercial laptop (RTX 3060 Laptop).

## CCS Concepts

• **Computing methodologies** → **Physical simulation**; **Rendering**.

## Keywords

Fluid Simulation, Fluid Rendering, Screen-space Rendering

**ACM Reference Format:**
Quanquan Peng and Yiyu Liu. 2024. A Complete High Performance Simulation and Rendering Pipeline for Fluid. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Simulating fluid is by no means easy — fluid are continuous while computers are only good at discrete things. Generally, we use Naiver-Stokes Equation [15] to describe fluid. However, the way of representing fluid is a great problem that remains to be solved. There are basically two kinds of approaches to represent fluid — *grid-based* [5, 8] and *particle-based* [2, 12–14, 17]. The former one uses girds with attributes of density and velocity that represent all the particles in the square or cube, while the latter one uses particles as elements in free-surface flow simulations. *Grid-based* simulation is better in terms of simulation speed but results in bad fidelity. We will go into details in related works section. In our work, we adopts *particle-based* simulation to better represent the physical property of fluid.

As for the rendering methodology, there are roughly three types of rendering methods – *mesh-based* [1, 18], *ray-casting* [6, 19] and *screen-based* [3, 4, 16], each of which has its pros and cons. In brief, *mesh-based* methods require high-resolution grid and turn out to be computationally expensive; *ray-casting* methods also require enormous computational power; *screen-based* methods is easier for computation but may emit some details. We will discuss about the details later. We adopts *screen-based* method to ensure better rendering speed.

In our work, we design and implement a complete simulation and rendering pipeline for fluid. We leverage the computational power by means of parallelization with Taichi framework [9–11]. With PBF, screen-based method and Layered Neighborhood Method optimization, we can achieve about 30 FPS with CUDA on a commercial laptop (RTX 3060 Laptop). To our knowledge, this maybe one of the few (or even the first) open-sourced and light-weighted full-stack fluid simulation and rendering pipeline which supports GPU. Our source code is publicly available at https://github.com/LauYeeYu/fluid-rendering/.

## 2 Related Works

### 2.1 Fluid Simulation methods

We can generally divide the methodology in fluid simulation into two types — *grid-based* and *particle-based* methods.

**Grid-based Methods.** In grid-based methods, every grid has its own attributes to represent the average property of all fluid in that grid [5, 8]. Fluid is simulated in the view of flow field. The scale of the grid number largely limits accuracy of simulation. Generally, the grid number is less than the particle number in particle-based methods, so grid-based methods are faster in simulation speed but worse in fidelity.

**Particle-based methods.** Fluid is regarded as a collection of particles in particle-based methods, where simulation is achieved by calculating the force on each particle and simulating its movement. Smoothed Particle Hydrodynamics (SPH) [13] is a well-established technique, celebrated for its mass conservation and adaptability in dynamic environments such as video games. Despite its advantages, SPH is susceptible to density fluctuations and the computational cost of enforcing incompressibility due to its unstructured nature. Advancements like Predictive-corrective Incompressible SPH (PCISPH) [17] have enhanced real-time application feasibility by using iterative methods for gradual pressure convergence, allowing for larger time steps. Building on this, our work utilizes the Position Based Fluids (PBF) [12] method, which integrates a position-based dynamics framework to address these challenges.

*Both authors contributed equally to this research.


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

PBF uses an iterative density solver to enforce constant density through positional constraints, combining the benefits of SPH incompressibility with the stability and efficiency of position-based methods, thereby enhancing the realism and performance of fluid simulations in real-time applications.

## 2.2 Fluid Rendering Methods

In the realm of screen-space fluid rendering, a variety of methods have been developed to efficiently render particle-based fluids in real-time applications. We can generally divide these methods into three types — *mesh-based*, *ray-casting* and *screen-based*.

**Mesh-based Methods.** These methods share the same core idea — retrieve the surface mesh from the position of particles in the world-space coordinates. These methods use algorithms like Marching Cubes to get the mesh. These methods can use isotropic kernels [18] or adaptive size kernels [1]. However, retrieving the surface mesh and rendering based on the complicated mesh consumes excessive computing power and therefore not feasible for real-time rendering.

**Ray-casting Methods.** In ray-casting methods, the liquid surface is rendered by employing a GPU-based volume ray-casting of a scalar field defined from the fluid particles. Fraedrich et al. [6] proposed an SPH particle rendering using an adaptive discretization of the view volume performing ray-casting in a perspective grid. With compression on the perspective grid by grouping the voxels, the memory consumption can be reduced [19]. However, although ray-casting methods allow interactive rendering, these techniques are unable to be applied on real-time applications.

**Screen-space Methods.** These methods only cares about the frontmost surface and the distance of light in fluid, which affect the result of render most. In these methods, a scene is calculated into a depth buffer and a thickness buffer with perspective projection. After smoothened, these two buffers are converted into normal buffer and light attenuation value respectively. Traditional screen-space methods [3, 4] rely on smoothing a depth buffer generated from particle data to create the visual representation of a fluid surface. These techniques, while effective in certain contexts, often require extensive computation, especially when using large convolution kernels or multiple filter passes, which can degrade performance significantly. A novel approach presented by Oliveira and Paiva [16] introduces a layered neighborhood method that significantly enhances performance by selectively processing only particles within a narrow band around the fluid's surface. This method not only reduces the computational load but also addresses speed deficiencies inherent in previous techniques that did not employ such an optimization. By focusing on boundary particles and effectively managing particle layers, the layered neighborhood method enables faster rendering times and reduces memory consumption, making it a robust solution for real-time fluid visualization in screen-space rendering frameworks.

# 3 Simulation and Rendering Pipeline Design

## 3.1 Fluid Simulation

Here we show how we do fluid simulation based on PBF method:

**Initialization:** Define and initialize particles that represent the fluid.

**Predict Position:**

- Apply external forces such as gravity to update particle velocities.
- Predict future positions based on current velocities and time step.

**Constraint Enforcement:** Enforce density constraints to maintain fluid incompressibility through iterative position adjustments.

**Velocity Update:** Update velocities based on the corrections made to positions during constraint enforcement.

## 3.2 Fluid Rendering

Since the particle positions are calculated in previous steps, we directly use these values and render the fluid.

*3.2.1 The Layer of each Particle.* Different from solid, fluid will gather together if are very close to each other. In this case, although the number of particles are large, only the outer particles are important for the shape of the fluid. As long as we know the shape of surface, we are able to restore almost every detail of the fluid accurately.

Then the question is, how to tell the outer particles from the inner particles? Since it is hard to exactly pick the nearby particles effeciently, we can divide the space into voxels and consider the voxels instead. The voxels that are likely to carry outer particles are considered as layer-1 [16]. Notice that a voxel is layer-1 only when

(1) it is on the boundary of the space;
(2) it has at least one neighbor voxel without particles.

For layer-2 voxel, a voxel is regarded as layer-2 if and only if

(1) it is has particles in it;
(2) it is not a layer-1 voxel;
(3) it has at least on neighbor layer-1 voxel.

Voxels with higher layer can be calculated similarly.

*3.2.2 Depth buffer.* With the preprocessed layer information, we can identify all the particles that are in layer-1 voxels. With these selected particles, we are able to generate the depth buffer since the frontmost particles must be the outer particles. We can simply map the particles to the place on the screen with perspective projection, and set the depth if the distance from the particle to the camera is less than the current value.

*3.2.3 Filtered Depth Buffer.* Now we've already got the depth information, but it is not enough. Since particles are discrete, the simple representation by spheres of the liquid surface will lead to an unrealistic blobby appearance. We need to smoothen and flatten the surface.

The filter method will be discussed in details in the implementation section.

*3.2.4 Normal Buffer.* With the smoothened normal buffer, we can generate the normal buffer easily. Just consider the change in value from up-down and left-right direction, we can calculate the normal of each screen pixel. The normal buffer is used for calculating the reflection of the frontmost surface.

*3.2.5 Thickness Buffer.* After computing the depth buffer, the inner particles are skipped by the remaining stages of the screen-space rendering pipeline. Therefore, we need to recover the fluid volume. To be specific, we need to estimate the thickness $T_{\mathcal{G}}$ using the voxels of $\mathcal{G}_h$.

Traditionally, we calculate the thickness by computing the contribution of each particle on each pixel. For each pixel $\overline{\mathbf{p}}$, the thickness from a particle set $\mathcal{P}$ is given by:

$$T_{\mathcal{P}}\left(\overline{\mathbf{p}}\right) = \sum_{j=1}^{|\mathcal{P}|} G_\sigma\left(\left\|\overline{\mathbf{p}} - \overline{\mathbf{x}}_j\right\|_2\right), \tag{1}$$

where $\mathbf{x}_j$ is the position of the particle $j$ and $\overline{\mathbf{x}}_j$ is its projection on the screen-space, and $G_\sigma$ is the Gaussian kernel with a radius of influence $\sigma$ as user-defined parameter, such that $G_\sigma(x) = \exp\left(-x^2/2\sigma^2\right)$.

In our case, we use voxels to calculate the thickness. Hence, the thickness is given similarly:

$$T_{\mathcal{G}}\left(\overline{\mathbf{p}}\right) = \sum_{j=1}^{|\mathcal{F}_h|} n_j G_\sigma\left(\left\|\overline{\mathbf{p}} - \overline{\mathbf{c}}_j\right\|_2\right), \tag{2}$$

where $\mathbf{c}_j$ is the centroid of the voxel and $n_j$ is the number of particles in voxel $j$.

*3.2.6 Filtered Thickness Buffer.* Similar to the process on depth buffer, the thickness buffer also suffers from an unrealistic blobby appearance. We need to smoothen the thickness buffer.

*3.2.7 Light Attenuation.* Before calculating the lighting model, we render the fluid volume using some light attenuation model. The loss of light intensity when it propagates in a liquid is directly proportional to the light path length. Therefore, the resulting color is given by $\overline{\mathbf{rgb}} = \mathbf{rgb}\cdot\exp(-\kappa l)$, where $\kappa$ is the attenuation coefficient.

Calculating the exact length of light is impossible under screen-based rendering. A common approach is to use the thickness as an approximation of the length of light in fluid.

## 4 Implementation

Our code is written in Python and Taichi [9–11]. Taichi is a parallel programming language for high-performance numerical computation, which uses multiple parallelization framework, e.g., OpenGL, CUDA, etc., as its backend.

### 4.1 PBF

Based on the pipeline described in Sec. 3.1, we show the code in Alg. 1, where: the density constraint $C_i$ on the $i$-th particle is defined using an equation of state:

$$C_i(\mathbf{p}_1, ..., \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1, \lambda_i = -\frac{C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k \left|\nabla_{\mathbf{p}_k} C_i\right|^2}. \tag{3}$$

Note that we split the 3D space into grids in order to accelerate the process of finding particles' neighbors and used SDF (signed distance field) for boundary detection.

### 4.2 LNM

Layered Neighborhood Method (LNM) [16] is the method used to accelerate the generation of the depth buffer. The key idea is that not all particles contribute to the depth buffer, only the out-layer does. The layers are determined by the k-ring neighborhood $\mathcal{N}_k(V)$ of a voxel $V$, as follows:

- 0-ring is the voxel V itself;
- The $k$-ring (with $k \in \mathbb{N}^*$) of $V$ is the set of adjacent voxels in its $(k-1)$-ring.

Define the distance of two voxels as:

$$\text{dist}(V_i, V_j) = \min\{k \in \mathbb{N} \mid V_i \in \mathcal{N}_k(V_j)\}. \tag{4}$$

Note the set of empty voxels as $\mathcal{E}_h$ and level-set $\mathcal{L}_k := \{V : \ell_i(V) = k\}$, where $\ell(V_i) = \begin{cases} \min_{E\in\mathcal{E}_h}\{\text{dist}(V_i, E)\}, & \text{if } V_i \text{ is not border;} \\ 1, & \text{otherwise.} \end{cases}$.

The pseudocode is described in Alg. 2, here we only consider $\mathcal{L}_1$ particles.

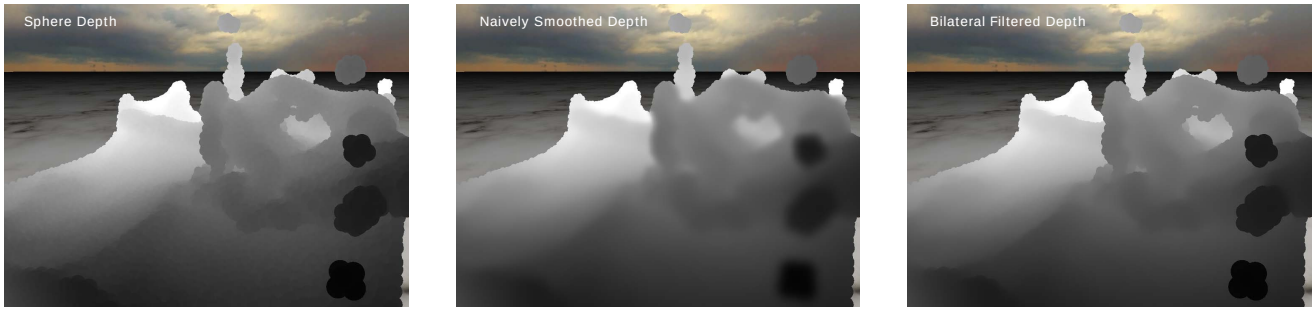
**Algorithm 1** PBF Simulation Loop

```
for all particles i do
    Apply forces v_i ← v_i + Δt f_ext(x_i) ▷ f_ext is the external force
    Predict position x_i^* ← x_i + Δt v_i
end for
for all particles i do
    Find neighboring particles N_i(x_i^*)
end for
while iter < solverIterations do          ▷ solverIterations is a hyper-param
    for all particles i do
        Calculate λ_i
    end for
    for all particles i do
        Calculate Δp_i
        Perform collision detection and response
    end for
    for all particles i do
        Update position x_i^* ← x_i^* + Δp_i
    end for
end while
for all particles i do
    Update velocity v_i ← (1/Δt)(x_i^* − x_i)
    Apply vorticity confinement and XSPH viscosity
    Update position x_i ← x_i^*
end for
```

### 4.3 Smoothening Filters

We have already discussed about the necessarity of smoothening filters in the rendering pipeline. A naive approach is to use Gaussian blur filter. This filter smoothen the values by weighted average of the pixel and its surrounding ones. However, we still want to preserve the silhouette edges in depth image so that particles don't get blended into background surfaces [7]. The soultion is to use biteral filters:

$$I^{\text{filtered}}(x) = \frac{1}{W_p}\sum_{x_i\in\Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|), \tag{5}$$




**(a) Original depth buffer.** **(b) Filtered by Gaussian blur filter.** **(c) Filtered by bilateral filter.**

**Figure 1: The effect of Gaussian blur filter and bilateral filter on depth buffer.**

**Algorithm 2** Layered Neighborhood Method (LNM)

**Require:** $P, G_h$, dim
**Ensure:** narrow-band $B$

$B \leftarrow \emptyset$
$n_{\min} \leftarrow 2^{\text{dim}}$
**for all** voxel $V_i \in G_h$ **do**
  $n_i \leftarrow$ number of particles of $P$ inside $V_i$
  $\ell_i \leftarrow 0$
**end for**
**for all** full voxel $V_i \in G_h$ **do** ▷ first layer
  **if** $|N_i(V_i)| \neq 3^{\text{dim}}$ **then** ▷ $V_i$ is border
    $\ell_i \leftarrow 1$
  **else**
    **for all** voxel $V_j \in \mathcal{N}_i(V_i)$ **do**
      **if** $n_j = 0$ **then** ▷ $V_j$ is empty
        $\ell_i \leftarrow 1$
        **break**
      **end if**
    **end for**
  **end if**
**end for**
**for all** full voxel $V_i \in G_h$ **do** ▷ second layer
  **if** $\ell_i = 1$ **then**
    **for all** voxel $V_j \in \mathcal{N}_i(V_i)$ **do**
      **if** $\ell_j = 1$ and $n_j \leq n_{\min}$ **then**
        $\ell_i \leftarrow 2$
        **break**
      **end if**
    **end for**
  **end if**
**end for**
**for all** full voxel $V_i \in G_h$ **do**
  **if** $\ell_i \neq 0$ **then**
    $P_i \leftarrow$ particles of $P$ inside $V_i$
    $B \leftarrow B \cup P_i$
  **end if**
**end for**
**return** $B$

and the normalization term, $W_p$, is defined as

$$W_p = \sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|), \tag{6}$$

where

- $I^{\text{filtered}}$ is the filtered image;
- $I$ is the original input image to be filtered;
- $x$ are the coordinates of the current pixel to be filtered;
- $\Omega$ is the window centered in $x$, so $x_i \in \Omega$ is another pixel;
- $f_r$ is the range kernel for smoothing differences in intensities (this function can be a Gaussian function);
- $g_s$ is the spatial (or domain) kernel for smoothing differences in coordinates (this function can be a Gaussian function).

Figure 1 shows the result of each filters. From the figure, we can clearly find that particles are blended into background surfaces with Gaussian blue filter, while the silhouette edges are preserved with bilateral filter.

## 5 Result

### 5.1 Experiment Setup

We run simulation and rendering on a commercial laptop equipped with Intel Core i7-11800H and NVIDIA 3060 Laptop. The number of particles is set to 12,000.

### 5.2 Performance

With LNM method, we can achieve about 30 FPS with CUDA backend, and about 2.6 FPS with OpenGL backend. As a comparison, the frame per second with CUDA is only about 20 FPS, and 2.4 FPS with OpenGL.

Figure 2 shows the screenshots of the rendered fluid.

### 5.3 Flaws

Despite the result of render meets the frame per second of a real-time application, we found some flaws in the experiments.

Figure 3 shows the jagged edges in rendering. This results from the bilateral filter. Although the bilateral filter is much better than the Gaussian blur in terms of keeping different surfaces separated, it cannot smoothen the edges well, so the edges will look like a round particle.

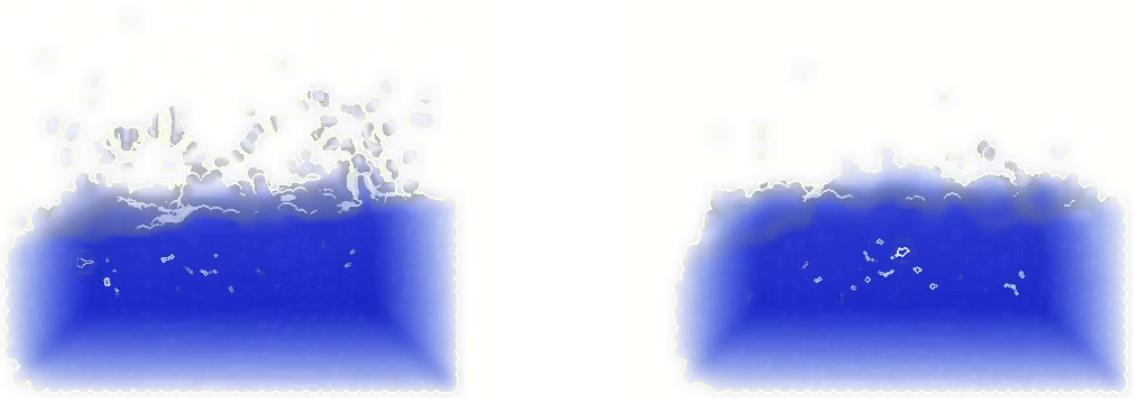


**Figure 2: Screenshots of the rendered fluid.**

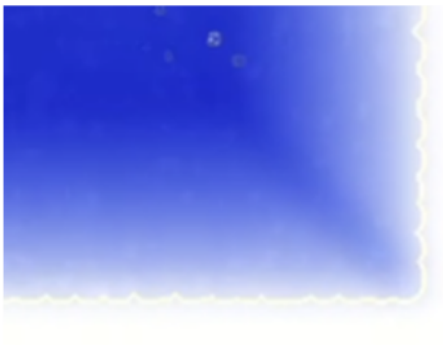


**Figure 3: Jagged edges. This is because the bilateral filter cannot smoothen the edges well.**

**Figure 4: Phantoms. This is because some critical particles are emitted with LNM.**

Figure 4 show the phantoms in rendering. This boils down to the fact that some critical particles are emitted with the Layered Neighborhood Method.

## 6 Conclusion

In brief, we design and implement a complete simulation and rendering pipeline for fluid with both CPU and GPU supported. We use PBF and screen-space method with LNM optimization, which improves the performance dramatically while minimizing the influence on fidelity. With the power of GPU, we make real-time simulation and rendering possible, despite some minor flaws in the final result. Layered Neighborhood method improves the frame per second by 50% with GPU and 8.3% with CPU.

## 7 Future Works

**Avoiding jagged edges.** Although the result has some jagged edges, we need to find a solution of both keeping different surfaces separated while smoothening the edges.

**Improving LNM.** The current LNM largely prune the computation of rendering. However, we think we can further improve the algorithm in terms of its efficiency and accuracy. Figure 4 indicates that the LNM has some inaccurate assertions. On the other hand, we now need to loop through grid and particles for multiple times, which is costly.

**Amending each step in simulation and rendering.** The current simulation and rendering procedure is not perfect and has room for improvement. We believe that with improving it will further increase the number of frames per second.

## Acknowledgments

We sincerely thank Prof. Yang for his invaluable guidance and to TA Shulin Hong for her dedicated assistance throughout this project.

## References

[1] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J Guibas. 2007. Adaptively sampled particle fluids. In *ACM SIGGRAPH 2007 papers*. 48–es.

[2] Carla Antoci, Mario Gallati, and Stefano Sibilla. 2007. Numerical simulation of fluid–structure interaction by SPH. *Computers & structures* 85, 11-14 (2007), 879–890.

[3] Florian Bagar, Daniel Scherzer, and Michael Wimmer. 2010. A layered particle-based fluid model for real-time rendering of water. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 1383–1389.
[4] Hilko Cords and Oliver G Staadt. 2009. Interactive screen-space surface rendering of dynamic particle clouds. *Journal of Graphics, GPU, and Game Tools* 14, 3 (2009), 1–19.
[5] Nick Foster and Ronald Fedkiw. 2001. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 23–30.
[6] Roland Fraedrich, Stefan Auer, and Rudiger Westermann. 2010. Efficient high-quality volume rendering of SPH data. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 1533–1540.
[7] simon Green. 2010. Screen Space Fluid Rendering for Games. http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf.
[8] Francis H Harlow, J Eddie Welch, et al. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of fluids* 8, 12 (1965), 2182.
[9] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).
[10] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 201.
[11] Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Transactions on Graphics (TOG)* 40, 4 (2021).
[12] Miles Macklin and Matthias Müller. 2013. Position based fluids. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.
[13] Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *In: Annual review of astronomy and astrophysics. Vol. 30 (A93-25826 09-90), p. 543-574.* 30 (1992), 543–574.
[14] Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Citeseer, 154–159.
[15] Claude-Louis Navier. 1838. Navier Stokes Equation. *Paris: Chez Carilian-Goeury* (1838).
[16] Felipe Oliveira and Afonso Paiva. 2022. Narrow-Band Screen-Space Fluid Rendering. In *Computer Graphics Forum*, Vol. 41. Wiley Online Library, 82–93.
[17] Barbara Solenthaler and Renato Pajarola. 2009. Predictive-corrective incompressible SPH. In *ACM SIGGRAPH 2009 papers*. 1–6.
[18] Barbara Solenthaler, Jürg Schläfli, and Renato Pajarola. 2007. A unified particle model for fluid–solid interactions. *Computer Animation and Virtual Worlds* 18, 1 (2007), 69–82.
[19] Tobias Zirr and Carsten Dachsbacher. 2015. Memory-Efficient On-The-Fly Voxelization of Particle Data.. In *EGPGV@ EuroVis*. 11–18.

# A Contributions of each Author

The work includes the following tasks:

Task 1. PBF fluid simulation.
Task 2. Basic screen-space fluid rendering.
Task 3. LNM.
Task 3.1. LNM algorithm implement.
Task 3.2. LNM's follow-up rendering.

The contribution is listed below:

- Quanquan Peng (50%): Task 1 & Task 3.1.
- Yiyu Liu (50%): Task 2 & Task 3.2.