Volume 0 (1981), Number 0 pp. 1–11 COMPUTER GRAPHICS *forum*

Narrow-Band Screen-Space Fluid Rendering

Felipe Oliveira and Afonso Paiva

ICMC-USP, São Carlos, Brazil

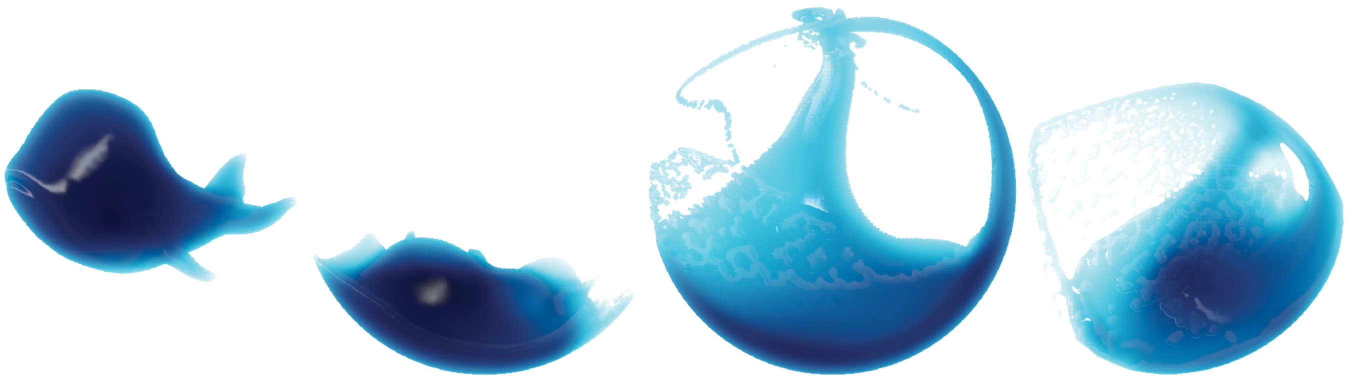


Figure 1: An SPH simulation of the Happy Whale liquid drop in a spherical container rendered with the novel narrow-band screen-space method. On average, our method is 2.4× faster than screen-space rendering using the same narrow-range filter [TY18], also reducing the memory footprint by 44%.

Abstract

This paper presents a novel and practical screen-space liquid rendering for particle-based fluids for real-time applications. Our rendering pipeline performs particle filtering only in a narrow-band around the boundary particles to provide a smooth liquid surface with volumetric rendering effects. We also introduce a novel boundary detection method allowing the user to select particle layers from the liquid interface. The proposed approach is simple, fast, memory-efficient, easy to code, and it can be adapted straightforwardly in the standard screen-space rendering methods, even in GPU architectures. We show through a set of experiments how the prior screen-space techniques can be benefited and improved by our approach.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Physical simulation; Rendering;***

1. Introduction

In graphics, particles are ubiquitous elements in free-surface flow simulations, frequently used to track the liquid interface in Lagrangian methods such as Smoothed Particle Hydrodynamics (SPH) [IOS*14] and Position Based Dynamics (PBD) [MM13] or hybrid Lagrangian-Eulerian methods, such as Fluid-Implicit Particle (FLIP) [ZB05] and Material Point Method (MPM) [HZGJ19]. Although these methods have been successfully applied to real-time applications (e.g., games) due to modern GPUs' parallel opportunities, the high-quality rendering of intricate liquid animations with a high number of particles at real-time frame rates (i.e., rendering at least 30 frames per second) remains a challenge in computer animation.

Traditionally the rendering of liquids in particle-based fluids consists of two steps: first, splatting the level-set function defined by the particles to a regular grid. Second, the liquid surface reconstruction is given by the isosurface extracted from a discrete level-set in world-space using a polygonization algorithm, such as Marching Cubes [LC87], where a polygonal mesh represents the isosurface. However, this entire process is computationally expensive since the surface's smoothness and topological coherence between consecutive frames require a high-resolution grid, mainly in large-scale domains. An alternative is to compute the surface rendering directly from the particles in the screen-space, without mesh generation. Screen-space techniques consist in computing the volume rendering directly from geometric primitives (spheres or ellipsoids) representing the particles, generating a depth buffer from the frontmost surface along a viewing direction.

This paper presents a novel and practical screen-space liquid rendering for particle-based methods implemented on GPU for real-time applications. Our method performs particle filtering only in a narrow-band around the boundary particles to provide a smooth liquid surface. We also introduce an efficient boundary detection that performs a particle peeling from the free-surface to produce the narrow-band. Therefore, we show the effectiveness of the proposed method through a set of comparisons against prior standard screen-space rendering pipelines. The excellent performance is attested in a set of practical applications where our approach can be adapted and incorporated in the existing screen-space methods in the literature, improving their performance significantly. Figure 1 shows our method in action.

In summary, the contributions of our method are:

- a novel screen-space fluid rendering that uses only the particles in a narrow-band of the liquid interface;
- a novel boundary detection method that allows the user to select particle layers from the free-surface;
- our approach speeds up and reduces memory consumption of the prior screen-space methods regardless of the choice of depth buffer filtering technique;
- our method is simple and easy to code, even in GPU architectures.

2. Related Work

In order to better contextualize our approach and highlight its properties we organize the existing methods for particle-based fluid rendering into three main groups: *mesh-based*, *ray-casting* and *screen-space* methods.

Mesh-based methods. The main goal of these methods is to extract a smooth polygonal surface mesh from the particle positions in the world-space coordinates using Marching Cubes (MC) like algorithms. Typically, the surface is represented implicitly by the zero level-set of a signed distance field computed from a weighted sum of kernel evaluations from the particles' distances. These methods can use isotropic kernels [MCG03; ZB05; SSP07], adaptive size kernels [APKG07] or anisotropic SPH kernels [YT13]. Despite the existence of parallel implementations of these methods [AIAT12; YG20; CZZ21], if the liquid spreads more over the computational domain, the underlying MC grid and its resulting surface mesh become very large, causing excessive memory consumption. Recently, Sandim et al. [SCN*16] proposed an alternative framework for surface reconstruction. Their framework relies on a level-set definition using the *boundary particles*, i.e., particles located at the liquid interface. Firstly, this method computes the boundary particles and their normals in parallel using OpenMP. Then, the surface is extracted fitting the Hermite data (particle positions and normals) using Screened Poisson surface reconstruction [KH13]. However, this method also suffers the same problem of the kernel-based methods.

Ray-casting methods. This class of view-dependent methods renders the liquid surface employing a GPU-based volume ray-casting of a scalar field defined from the fluid particles. Fraedrich et al. [FAW10] proposed an SPH particle rendering using an adaptive discretization of the view volume performing ray-casting in a perspective grid. Zirr et al. [ZD15] improved the previous work's memory consumption compressing the perspective grid by grouping its voxels. Goswami et al. [GSSP10] performed a ray-casting of a distance field computed in a narrow-band near the boundary particles detected using the distance between each SPH particle and the centroid of its particle neighborhood. Later, Xiao et al. [XZY18] also used ray-casting in a narrow-band around the isosurface defined by an SPH density field. Recently, Biedert et al. [BSS*18] replaced the polygonization scheme used to compute the isosurface of the scalar field derived from the anisotropic SPH kernels provided by Yu and Turk [YT13] by a direct ray-casting. However, the methods above mentioned losing the liquid surface depth information because it discards internal fluid particles' contribution. Similar to screen-space methods, Reichl et al. [RCSW14] presented a rendering based on sphere ray-casting, storing the distances of the intersection points between rays and particles (spheres) to the viewpoint in the depth buffer. Then, the depth buffer is smoothed using a total-variation denoising filter [Cha04]. Although ray-casting methods allow interactive rendering, these techniques are not feasible for real-time applications.

Screen-space methods. As mentioned, this category of methods performs in 2D image space using a smoothed depth buffer from the visible liquid surface defined by the particles, where the resulting surface is represented by a triangle mesh using Marching Squares [MSD07] or without mesh generation by using rasterization techniques [CS09; LGS09; Gre10; BSW10; IKM16; NA17; TY18]. The liquid surface's visual quality relies on the depth buffer's image-based filtering process, which may demand large convolution kernels and perform multiple passes (filter iterations). Thus, several improvements have been made in this direction with many filter flavors: binomial filters [MSD07; CS09], curvature flow filters [LGS09; BSW10], and bilateral Gaussian filters [Gre10; NA17]. Imai et al. [IKM16]. proposed another alternative using Moving Least-Squares plane fitting for filtering. This strategy improves the liquid surface's quality compared to the bilateral Gaussian filter, avoiding the liquid surface's flattened appearance in regions near discontinuities. Recently, Truong and Yuksel [TY18] presented a remarkable screen-space filtering known as *narrow-range filter*. Their method smooths the foreground and background surface's depth values separately according to a narrow depth range using a Gaussian filter with adaptive kernel size. However, beyond the screen-space size, these methods' efficiency also depends on the number of filter iterations and the number of particles in the world-space. Our method addresses the particle issue, improving the previous methods' performance considerably regardless of the screen-space filter's choice.

3. Screen-Space Fluid Rendering in a Nutshell

The key idea behind the screen-space method is to estimate the liquid surface by the particle projections. By getting the depth values from the distances between the viewer and particles, we can determine a surface by smoothing this depth distribution. It is possible to reconstruct geometric properties such as surface normals and po-

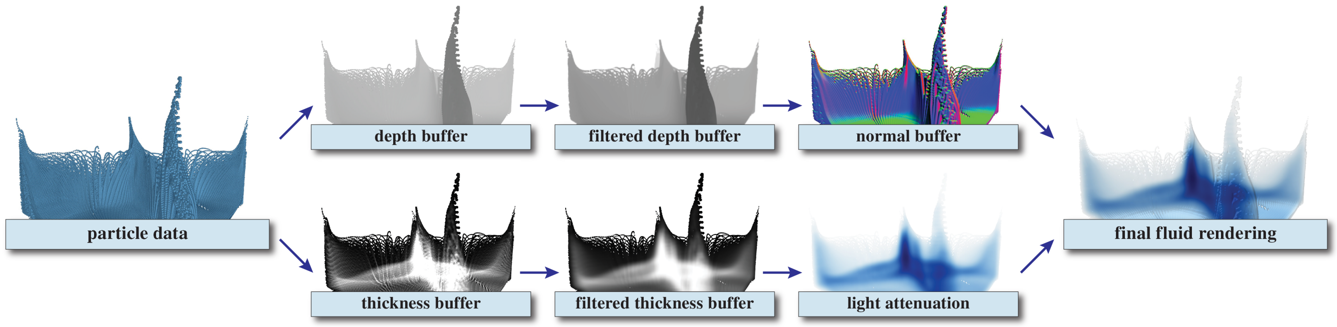


Figure 2: Screen-space fluid rendering pipeline.

sition from the smoothed depth values and perform a complete rendering pipeline on the input depth buffer. Most screen-space fluid rendering methods are based on the pipeline introduced by Van der Laan et al. [LGS09], which consists of several render passes, as illustrated in Figure 2. In this section, we give a high-level overview of each pipeline's stages.

Depth buffer. In this first stage, the generation of depth values from particle data is entirely solved by rasterizing all particles, represented by spheres of diameter h, using point sprites (i.e., screen-oriented quads). As mentioned before, we store the particle's depth value concerning the camera's viewpoint for each pixel on the screen-space. In Figure 2, we can see darker pixels for closer points from the camera and lighter pixels for points further away.

Filtered depth buffer. Once the depth buffer is generated, we have a simple representation by spheres of the liquid surface, resulting in an unrealistic blobby appearance due to the cusp's aspect resulting from the intersection of spheres. To produce a smooth, flat surface from the particle positions, we perform a two-dimensional smooth filter in the depth buffer to remove the curvature's high variation between the particles.

Normal buffer. With the resulting smoothed depth buffer, we calculate the surface normals required for the lightning model. The normals are estimated from the derivatives of the depth field using the finite difference method and stored in a normal buffer. The colors in Figure 2 encode the normal directions.

Thickness buffer. Another essential component in the final rendering image is the liquid volume. The volume is represented by the *thickness* value, it controls the color attenuation and the transparency of the liquid rendering, affecting the visibility of objects immersed or occluded by a liquid. The thickness is achieved by computing the portion of liquid between the viewpoint and a specific location inside the liquid. For each pixel $\bar{\mathbf{p}}$ in the screen-space, the thickness from a particle set $\mathcal{P}$ is given by:

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

where the operator $|\cdot|$ denotes the set's cardinality, $\mathbf{x}_j$ is the position of the particle j and $\bar{\mathbf{x}}_j$ is its projection on the screen-space, and G_σ is the Gaussian kernel with a radius of influence σ as user-defined parameter, such that $G_\sigma(x) = \exp(-x^2/2\sigma^2)$. In practice, the thickness buffer is generated by rendering G_σ for each point sprite, replacing the summation of the Equation (1) by an additive alpha blending to accumulate the amount of liquid at each pixel of the output buffer. As shown in Figure 2, lighter and darker pixels represent high and low thickness values, respectively.

Filtered thickness buffer. Again, due to the particle representation by spheres, the alpha blending operation's output needs to be smoothed to generate the final liquid volume. Usually, the thickness values are smoothed by using the bilateral Gaussian filter [Gre10].

Light attenuation. Before performing the lighting model, we rendered the liquid volume using some light attenuation model. Given the light path length l and the light-transmitting color $\mathbf{rgb}$ in the liquid, we can estimate each pixel's output color regarding the liquid's light absorption using the Beer-Lambert law [IKM16]. This law stated that the loss of light intensity when it propagates in a liquid is directly proportional to the light path length. Thus, the resulting color influenced by the Beer-Lambert law is given by: $\overline{\mathbf{rgb}} = \mathbf{rgb} \cdot \exp(-\kappa l)$, where κ is the attenuation coefficient. In particular, the thickness provides a good approximation for the light path length.

Finally, after computing the light attenuation and normal buffer, we make a composition pass to perform the lighting model using the simple Blinn-Phong scheme. Moreover, we can enhance the final rendered image's quality by integrating some lighting effects in the scene, such as reflections/refractions, shadows, and caustics.

4. Narrow-band Screen-Space Fluid Rendering

To improve the performance of the screen-space fluid rendering for large-scale particle-based simulations, we reduce the number of particles required to perform the rendering pipeline by using the particles in a *narrow-band* (NB) of the liquid interface.

Since the *boundary particles* [SPd20] give a good sampling of the free-surface, it is a straightforward choice to use them to build an NB. Given a spherical particle set $\mathcal{P}$, the boundary particle definition is based on a sphere covering analysis, checking whether a given particle is covered by its neighbor particles. In mathematical terms, let $B_h(\mathbf{x})$ be an open ball of radius h centered at a point $\mathbf{x} \in \mathbb{R}^{dim}$. A particle j in $\mathcal{P}$ is *interior* whether $\partial B_h(\mathbf{x}_j) \subset \cup_i B_h(\mathbf{x}_i)$, where $\partial B_h(\mathbf{x}_j)$ is the boundary of $B_h(\mathbf{x}_j)$; otherwise, j is classified as *boundary*. The value h corresponds to the particle resolu-

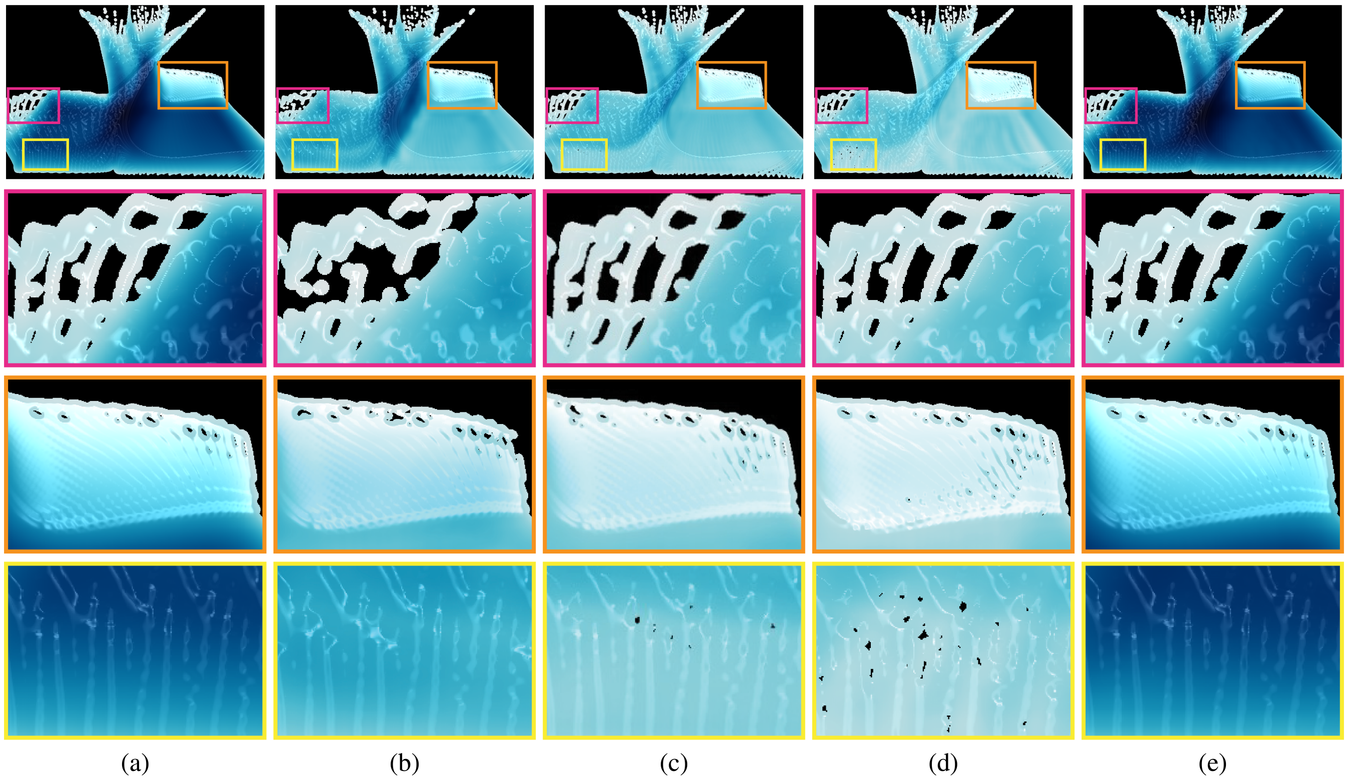


Figure 3: Comparison with different approaches for the NB screen-space fluid rendering using the narrow-range filter in a liquid splash with 405K SPH particles: (a) reference image without NB, (b) NB with Müller et al. [MCG03], (c) NB with He et al. [HLW*12], (d) NB with Sandim et al. [SCN*16], and (e) with our NB. Note that our NB rendering preserves the details of the liquid surface and the volume.

tion, i.e., a value usually close to the initial particle spacing. For instance, in SPH simulations, this value is known as *smoothing length* [IOS*14].

Some methods can be used to detect boundary particles, some employing threshold of SPH gradient norm of a color field [MCG03], based on the distance of a particle and the centroid of its neighbors [HLW*12], or using visibility test on the particles [SCN*16]. However, despite being accurate, these methods have some issues (see Figure 3) when applied straightforward in the screen-space rendering pipeline: holes and loss of details (drops and thin-sheets) of the liquid interface and lack of volume. The first issue is caused by the drawbacks associated with the corresponding boundary detection method, while the second one is generated by removing the internal particles. In the following sections, we will present strategies to solve these issues.

4.1. Layered Neighborhood Method

To eliminate the spurious results on the liquid surface, we need to build an NB using a boundary detection method suited to screen-space fluid rendering, satisfying the following requirements:

R1 – Covering and details: the method should cover any possible holes and also capture high-quality details in the liquid surface regardless of the particle distribution;

R2 – Simplicity: the method must be simple to be coded and inserted into a parallel particle-based fluid solver on both CPU and GPU architectures;

R3 – Efficiency: since performance is our primary goal, the detection should be faster in both loading and rendering particle data.

To fulfill **R1**, we create an NB around the liquid surface by increasing the boundary particle layer thickness, just adding a few *false positives*, i.e., interior particles classified as boundary.

Firstly, we build a regular grid $\mathcal{G}_h$ covering the computational domain $\Omega \subset \mathbb{R}^{dim}$ with a voxel size of $2h$. A voxel is labeled *empty* if it contains no particles in its interior, otherwise, V is labeled *full*. We denote by $\mathcal{E}_h$ and $\mathcal{F}_h$ the set of empty and full voxels of $\mathcal{G}_h$, respectively. Also, we can define a hash table associated with $\mathcal{G}_h$, where a voxel $V \in \mathcal{F}_h$ that contains a particle j in its interior can be retrieved by a hash function H based on the particle positions, that is $H(\mathbf{x}_j) = V$. For SPH solvers, we can avoid the creation of an additional grid $\mathcal{G}_h$ by reusing the neighborhood search grid [IOS*14].

We design a greedy algorithm for boundary detection based on particle layers generated from voxels' adjacency relation. For instance, boundary particles appear in full voxels adjacent to an empty voxel or lying at the grid border (see Figure 4). In our case, two voxels are *adjacent* if they share at least one vertex, edge, or face. The layers are determined by the *k-ring* neighborhood $\mathcal{N}_k(V)$ of a voxel V, as follows:

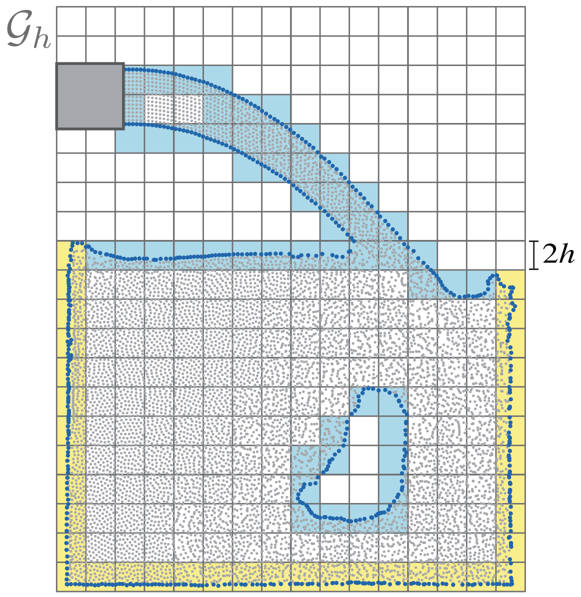


Figure 4: The boundary particles (dark blue dots) are inside of full voxels of $\mathcal{G}_h$ that are adjacent to an empty voxel (light blue) or located at grid border (yellow).

- 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.

This definition also allows us to identify voxels in the grid border by examining whether their neighborhood is incomplete, i.e., $V \in \mathcal{G}_h$ is a *border voxel* whether $|\mathcal{N}_1(V)| \neq 3^{dim}$.

Another important ingredient for our boundary detection is the *k-ring distance* between voxels. Given two voxels V_i and V_j, the distance is defined as:

$$\mathrm{dist}(V_i, V_j) = \min\{k \in \mathbb{N} \mid V_i \in \mathcal{N}_k(V_j)\},$$

The distance of a full voxel $V_i \in \mathcal{F}_h$ to the set $\mathcal{E}_h$ provides a labeling scheme for V_i as follows:

$$\ell(V_i) = \begin{cases} \min_{E \in \mathcal{E}_h}\{\mathrm{dist}(V_i, E)\}, & \text{if } V_i \text{ is not border} \\ 1, & \text{otherwise} \end{cases}. \quad (2)$$

For sake of simplicity, we denote the label of V_i by $\ell_i = \ell(V_i)$. The colored voxels illustrated in Figure 4 receive label $\ell_i = 1$ and contain all boundary particles (dark blue dots) even computed by a sophisticated and accurate method as proposed by Sandim et al. [SCN*16].

From the particle's point of view, a particle j belongs to a *k-layer*, $\mathcal{L}_k \subset \mathcal{P}$, if $H(\mathbf{x}_j) \in \ell^{-1}(k)$, where $\ell^{-1}(k) = \{V_i \in \mathcal{F}_h \mid \ell_i = k\}$. In other words, the particle receives the label of the voxel that contains it Figure 5 shows the particle layers according to the Equation (2). Effectively, we use some of these layers to build our narrow-band.

The set $\mathcal{L}_1$ is a natural choice to represent the narrow-band $\mathcal{B}$. However, a particle deficiency may occur in voxels with the label $\ell_i = 1$. In this case, we need to include some extra particles of $\mathcal{L}_2$ in the narrow-band. Let $n_{\min}$ be the minimum number of particles allowed within a voxel and $\mathcal{P}_k$ denotes the particle subset of $\mathcal{P}$ inside a voxel V_k. To determine the particle subset $\tilde{\mathcal{L}}_2 \subseteq \mathcal{L}_2$ from a voxel V_i with $\ell_i = 2$, we create a rule for its neighbors $V_j \in \mathcal{N}_1(V_i)$:

$$\text{if } \ell_j = 1 \text{ and } n_j \leq n_{\min} \;\Rightarrow\; \text{insert the particles of } \mathcal{P}_i \text{ in } \tilde{\mathcal{L}}_2,$$

where $n_j = |\mathcal{P}_j|$. Thus, the NB is defined by $\mathcal{B} = \mathcal{L}_1 \cup \tilde{\mathcal{L}}_2$. This procedure reduces the number of particles in $\mathcal{B}$, avoiding holes in the rendered surface. We call this approach the *Layered Neighborhood Method* (LNM).

In terms of implementation, we process the layers $\mathcal{L}_1$ and $\mathcal{L}_2$ using the 1-ring neighborhood of the full voxels. A full voxel V_i is $\ell_i = 1$ if there is an empty voxel in its neighborhood $\mathcal{N}_1(V_i)$. Once assigned the voxels in the first layer, we process the second layer analogously. Thus, a full voxel $V_j \in \mathcal{N}_1(V_i) \setminus \ell^{-1}(1)$ with $\ell_i = 1$ has the label $\ell_j = 2$. Algorithm 1 summarizes our LNM, taking into account the particles deficiency in the first layer voxels by setting the parameter $n_{\min} = 2^{dim}$. Note that LNM is simple because it relies on the voxels' adjacency relation and easy to code in GPUs

Algorithm 1: Layered Neighborhood Method (LNM)

```
Input: $\mathcal{P}, \mathcal{G}_h, dim$
Output: narrow-band $\mathcal{B}$
$\mathcal{B} \leftarrow \emptyset$
$n_{\min} \leftarrow 2^{dim}$
foreach voxel $V_i \in \mathcal{G}_h$ do
    $n_i \leftarrow$ number of particles of $\mathcal{P}$ inside $V_i$
    $\ell_i \leftarrow 0$
end
foreach full voxel $V_i \in \mathcal{G}_h$ do                /* 1st layer */
    if $|\mathcal{N}_1(V_i)| \neq 3^{dim}$ then              /* $V_i$ is border */
        $\ell_i \leftarrow 1$
    else
        foreach voxel $V_j \in \mathcal{N}_1(V_i) \setminus V_i$ do
            if $n_j = 0$ then                               /* $V_j$ is empty */
                $\ell_i \leftarrow 1$
                break
            end
        end
    end
end
foreach full voxel $V_i \in \mathcal{G}_h$ do                /* 2nd layer */
    if $\ell_i \neq 1$ then
        foreach voxel $V_j \in \mathcal{N}_1(V_i) \setminus V_i$ do
            if $\ell_j = 1$ and $n_j \leq n_{\min}$ then
                $\ell_i \leftarrow 2$
                break
            end
        end
    end
end
foreach full voxel $V_i \in \mathcal{G}_h$ do
    if $\ell_i \neq 0$ then
        $\mathcal{P}_i \leftarrow$ particles of $\mathcal{P}$ inside $V_i$
        $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{P}_i$
    end
end
return $\mathcal{B}$
```

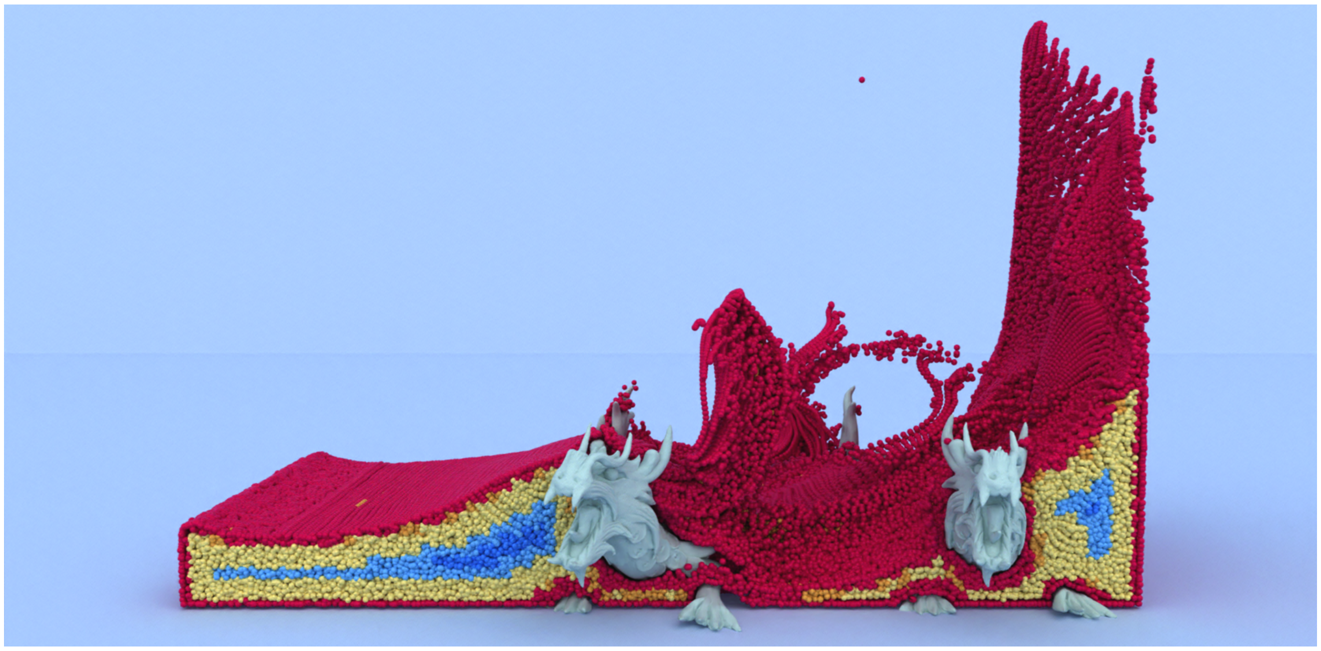


Figure 5: A cutaway view shows the classification of the particle layers: $\mathcal{L}_1$ (🟥), $\mathcal{L}_2$ (🟧), $\mathcal{L}_3$ (🟨), $\mathcal{L}_4$ (🟦), and $\mathcal{L}_5$ (🟦).

Table 1: Average computational times of the boundary detection methods and their fulfillment requirements.

Methods	$\lvert\mathcal{B}\rvert$	**time**	**ratio**	**R1**	**R2**	**R3**
Müller et al. [MCG03]	198K	3.77	10.77	✗	✓[†]	✗
He et al. [HLW*12]	147K	2.90	8.29	✗	✗	✗
Sandim et al. [SCN*16]	89K	8.27	23.63	✗	∂	✗
Our LNM	198K	0.35	1.00	✓	✓	✓

since each voxel is processed independently, fulfilling the requirement **R2**. Besides, an LNM implementation in CUDA is available on GitHub (click the top-right icon on Algorithm 1).

Table 1 shows the performance of LMN against other boundary detection methods in the experiment illustrated in Figure 3 using a single-core CPU. The column $|\mathcal{B}|$ is the number of particles in NB. The column **time** is the average computational times (in seconds) per frame. The column **ratio** of the average time of each method to LNM. As can be seen, our method generated the NB at least eight times faster than the previous methods, satisfying the requirement **R3**. Also, the table summarizes the self-fulfillment requirements of each method. The symbol $\boldsymbol{\partial}$ denotes a partial fulfillment of some requirement.

In our screen-space rendering, we use the NB particles $\mathcal{B}$ to compute the depth buffer instead of the entire particle system $\mathcal{P}$. This strategy alleviates the computational efforts of the screen-space rendering because the relation $|\mathcal{B}| \ll |\mathcal{P}|$ usually occurs in particle-based fluid simulations. In particular, this is the only stage of our rendering that handles particle computations.

† Yang and Gao [YG20] presented a GPU version of [MCG03].

4.2. Volume Restoration

After computing the depth buffer using the NB represented by the particle set, the particles are skipped by the remaining stages of the screen-space rendering pipeline. Therefore, to recover the fluid volume, we need to estimate a thickness $T_{\mathcal{G}}$ using the voxels of $\mathcal{G}_h$. To accomplish this task, we splat the voxels instead of the particles, i.e., we need to rewrite the Equation (1) for voxels.

For each pixel $\bar{\mathbf{p}}$ in the screen-space, we compute $T_{\mathcal{G}}$ taking into account the contribution of the grid voxels. Given $V_j \in \mathcal{F}_h$, the thickness is defined regarding the occupancy and the distribution of the particles $\mathcal{P}_j$. We estimate the occupancy using the number of particles $n_j = |\mathcal{P}_j|$, while the particle distribution is represented by the particle centroid $\mathbf{c}_j = \sum_i \mathbf{x}_i / n_j \in \mathbb{R}^3$, where each particle i belongs to $\mathcal{P}_j$. Firstly, we render the projected centroid $\bar{\mathbf{c}}_j$ on the screen-space as a point sprite, as shown by Figure 6a. Then, the thickness $T_{\mathcal{G}}$ is given by a local average of the occupancy using the Gaussian convolution in the screen-space as follows:

$$T_{\mathcal{G}}(\bar{\mathbf{p}}) = \sum_{j=1}^{|\mathcal{F}_h|} n_j \, G_\sigma(\|\bar{\mathbf{p}} - \bar{\mathbf{c}}_j\|_2)\,. \tag{3}$$

Each parcel of Equation (3) is computed individually as a point sprite. Then, we perform accumulative alpha blending operations to obtain the resulting thickness of the sum of all point sprites' contributions, as illustrated by Figure 6b. After splatting all voxels, we move on to the next step of the rendering pipeline, where the thickness buffer is smoothed by using the bilateral filter.

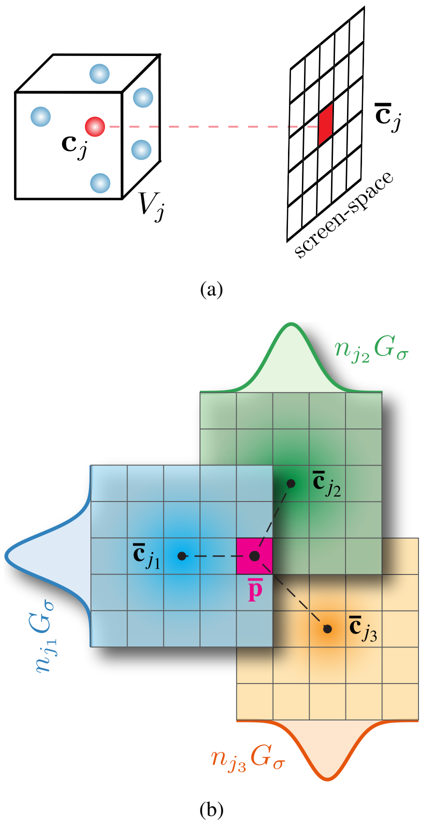


Figure 6: The thickness $T_\mathcal{G}$ defined from the grid voxels V_j. (a) The centroid $\mathbf{c}_j$ (red dot) of the particles $\mathcal{P}_j$ inside V_j (blue dots) is rendered as a point sprite (red quad) with center at the projected centroid $\bar{\mathbf{c}}_j$ with size h. (b) For each point sprite (colored quad) associated with V_j, we evaluate its pixels by the Gaussian G_σ centered at $\bar{\mathbf{c}}_j$ multiplied by $n_j = |\mathcal{P}_j|$. Finally, the resulting value of $T_\mathcal{G}$ for a pixel $\bar{\mathbf{p}}$ overlaid by the point sprites is achieved by enabling the accumulative alpha blending.

For large-scale particle-based fluid simulations, our method considerably reduces the number of Gaussian evaluations in Equation (3) w.r.t. Equation (1) because the number of full voxels is much less than the number of particles, i.e., $|\mathcal{F}_h| \ll |\mathcal{P}|$. As an implementation remark, the centroid computation in the object-space can be performed in GPU during the last loop of Algorithm 1.

5. Results

We implemented our approach in C++ and OpenGL. The particle-based fluid simulations were produced using predictive-corrective incompressible SPH (PCISPH) [SP09] parallelized on GPU using CUDA. All results have been achieved using a computer equipped with a processor Intel i7-7700HQ with four 2.8GHz cores and 16GB RAM, and an NVIDIA GeForce GTX 1050Ti with 768 CUDA cores and 4GB of RAM. Table 2 shows the computational times and some statistics for a set of experiments presented in this section. The column **res** is the resolution of $\mathcal{G}_h$ and the column **mem** is the memory saving (in percentage) provided by LNM, i.e., how much our NB reduces memory storage in the rendering. In our application, we incorporate the LNM implemented in GPU into the PCISPH fluid solver. Note that the NB computation spends less than 1.2% of the overall PCISPH time on average and a memory saving of 44% in the worst case and 66% in the best case.

Table 2: Average statistics and computational times (in milliseconds) per frame.

	PCISPH		$\mathcal{G}_h$	**LNM**		
Experiment	$\lvert\mathcal{P}\rvert$	**time**	**res**	$\lvert\mathcal{B}\rvert$	**mem**	**time**
Happy Whale (Fig. 1)	1.12M	8257	$110 \times 110 \times 110$	528K	44%	17.20
Liq. sloshing (Fig. 7)	1.14M	8398	$75 \times 75 \times 75$	302K	66%	12.93
Water drop (Fig. 8)	722K	4352	$70 \times 57 \times 70$	206K	59%	9.50
Dam breaking (Fig. 9)	572K	3919	$91 \times 57 \times 54$	142K	66%	14.38

Figures 1, 7, 8, and 9 show our NB screen-space fluid rendering applied in different simulations using many sorts of depth-map filters, such as narrow-range filter [TY18], bilateral filter [Gre10], plane fitting filter [IKM16], and curvature flow filter [LGS09]. In our experiments, we use a screen size in full HD (1920×1080) resolution, a filter size of 7×7, and 4 iterations for all filters. Except for the curvature flow filter, we apply 30 iterations.

Figure 10 shows the boxplots of the computational times for different strategies of screen-space fluid rendering. As can be seen, our NB approach improves the performance considerably regardless of the filter's choice, which demonstrates our method's efficiency.

Figure 11 provides a visual comparison between regular screen-space fluid rendering and our NB approach, and as can be seen, our method delivers results nearly indistinguishable from regular ones. Our NB method increases the frame rates in fps (frames per second) at least 15 fps in the worst case (with curvature flow filter) and 51 fps in the best case (with narrow-range filter). Note that our NB achieves a speed-up of $\approx 2\times$ for the screen-space renderings that require few filter iterations.

6. Discussion

Performance profiling. Figure 12 presents the computational timing and the performance profiling of each stage of the rendering pipeline (as shown in Figure 2) using the narrow-range filter for a liquid splashing produced during a quadruple dam breaking simulation with 900k SPH particles. Our NB method improves the rendering performance considerably by reducing the execution time of the depth map and thickness stages. On the other hand, the rendering bottleneck is the filtering processes performed on screen-space, where the image resolution directly impacts its performance.

Thickness on voxels. The choice of the particle centroids instead of voxel centers in thickness computation, given by Equation (3), avoids artifacts caused by the pixels' thickness displacement that occurs when the particles are all located on one side of the voxels. Figure 13 shows the volume silhouettes computed with Canny edge detection applied on $T_\mathcal{G}$ using the two approaches: particle centroids and voxel centers. Note that $T_\mathcal{G}$ with particle centroids affords the best approximation regarding the thickness reference $T_\mathcal{P}$ given by Equation (1).

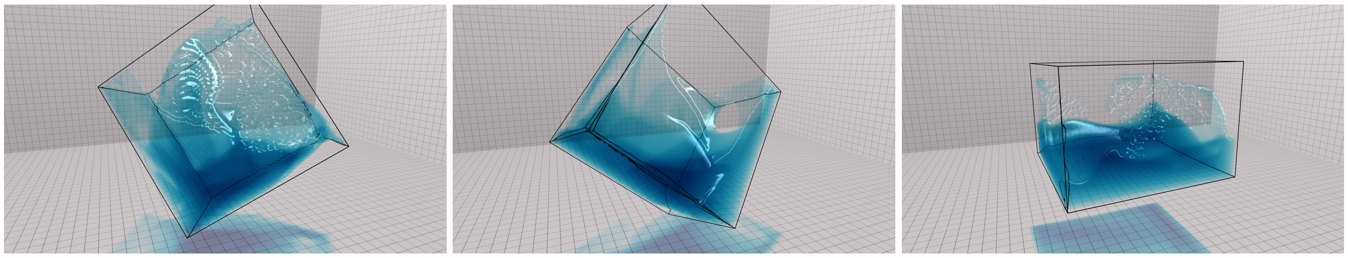


Figure 7: Liquid sloshing simulation rendered using our NB screen-space method with plane fitting filter. On average, our method is 1.7× faster than the same rendering without NB.

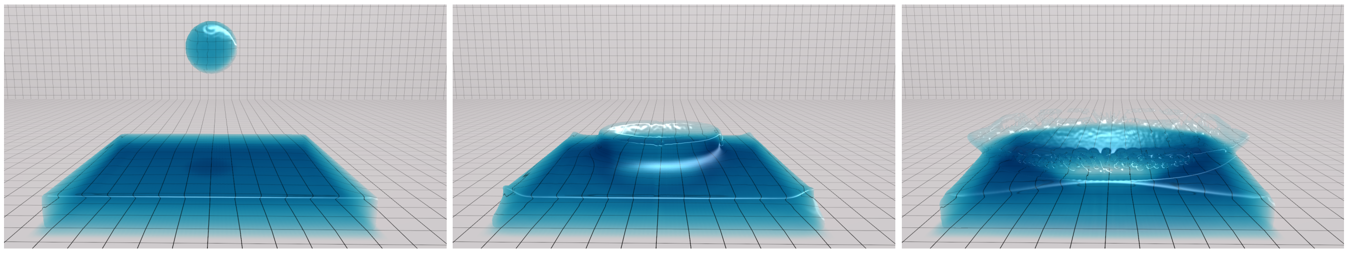


Figure 8: Water drop simulation rendered with our NB screen-space method using curvature flow filter. On average, our method is 1.5× faster than the same rendering without NB.

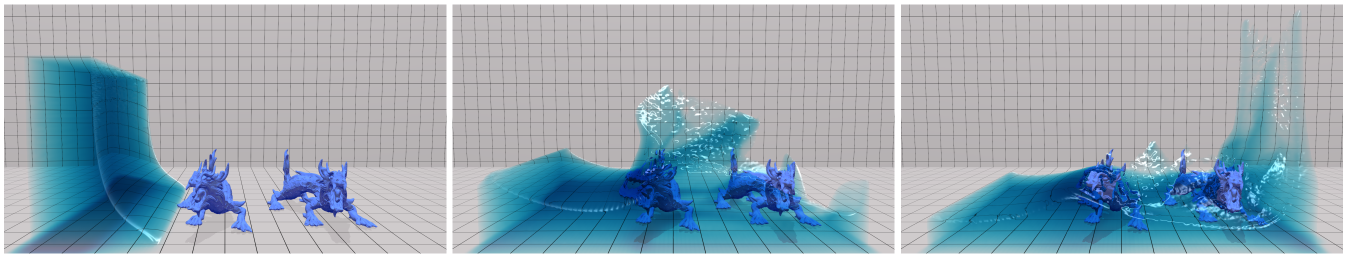
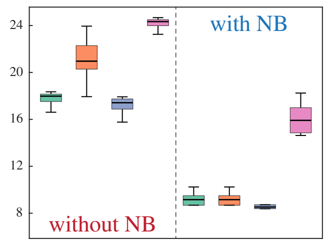
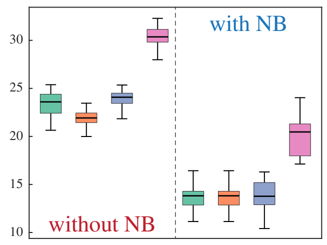


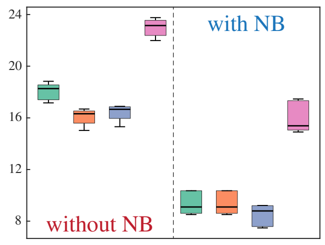
Figure 9: Dam breaking with obstacles of dragons rendered using our NB screen-space method with bilateral filter. On average, our method is 2× faster than the same rendering without NB.



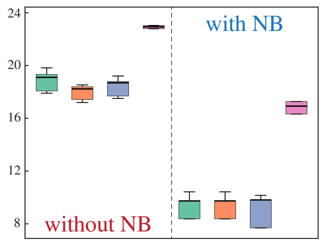
(a) Happy Whale (Fig. 1)



(b) Liquid sloshing (Fig. 7)



(c) Water drop (Fig. 8)

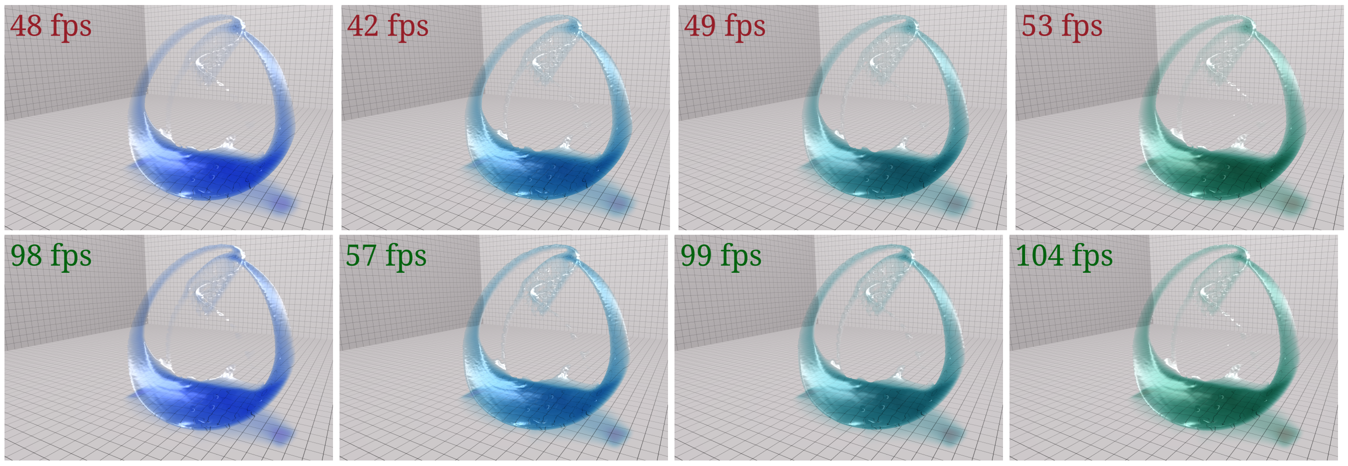


(d) Dam break (Fig. 9)

Figure 10: Boxplots of the computational times (in milliseconds) for screen-space rendering with different filters: bilateral filter (🟩), narrow-range filter (🟧), plane fitting filter (🟦), and curvature flow filter (🟪).

Grid resolution. The particle resolution h defines the grid resolution, i.e., the resolution of $\mathcal{G}_h$ increases when the value of h decreases and vice-versa. Figure 14 shows how the grid resolution impacts on the quality and performance of our NB rendering. We test the dam breaking simulation (Figure 9) with different voxel sizes, as we expected, the performance improves when we decrease the grid resolution. However, the rendering quality deteriorates in coarse grids due to the poor thickness approximation, as can be verified by the error defined by the color difference[‡] between the NB rendering and the reference image without NB and also by its *mean squared error* (*MSE*). The value of $2h$ is the best choice of the voxel size regarding the tradeoff between efficiency and rendering quality.

[‡] We use the function `imcolordiff` available in MATLAB R2020b.



(a) Bilaterl filter [Gre10] (b) Curvature flow filter [LGS09] (c) Plane fitting filter [IKM16] (d) Narrow-range filter [TY18]

Figure 11: Comparison of screen-space rendering methods in a single frame of the simulation illustrated by Figure 1 and their performance in fps: regular approaches (*top*) and with our NB (*bottom*).

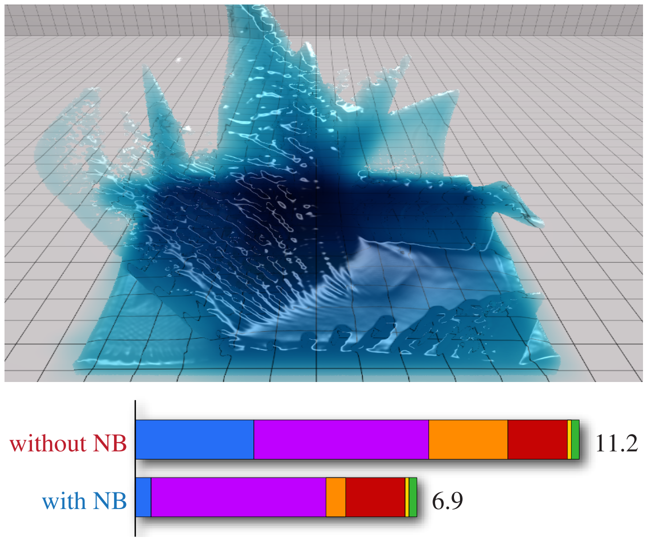


Figure 12: Liquid splashing rendered using our NB screen-space method with narrow-range filter (*top*) and the computational timing (in milliseconds) of each rendering pipeline stage for this frame (*bottom*): depth buffer (🟦), filtered depth buffer (🟪), thickness buffer (🟧), filtered thickness buffer (🟥), normal buffer (🟨), light attenuation and rendering (🟩).

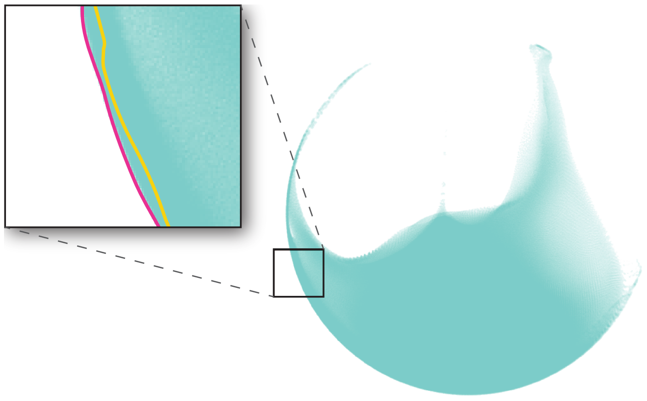


Figure 13: Thickness $T_{\mathcal{G}}$ on voxels: reference thickness $T_{\mathcal{P}}$ computed over the particles $\mathcal{P}$ (*right*) and a zoom-in region (*top-left*) showing the volume silhouettes provided by $T_{\mathcal{G}}$ using particle centroids (pink) and voxel centers (yellow).

Dynamic shadow generation. Beyond the liquid surface, the poor representation of the NB can lead the screen-space rendering pipeline to some shortcomings in the computation of secondary physical effects, such as shadows. The shadow casting relies on some formulation based on a visibility estimation using rays sent from light sources. When the liquid surface contains holes, a light ray can travel without ever interacting with the fluid generating spurious results, as illustrated by Figure 15a. While Figure 15b shows that our NB provided by LNM is resilient to holes formation on the surface affording correct shadows.

Limitations. Although our NB method yields liquid rendering at high frame rates and low memory footprint for large SPH-based simulations, it inherits some limitations typical to prior screen-space rendering techniques, such as the over-smooth appearance of the liquid surface and visual artifacts near discontinuities.

Future work. Our current GPU implementation is limited to bounded computational domains due to the full grid used by LNM. It opens some opportunities for replacing our data structure with dynamic hierarchical sparse grid representations using GVDB Voxels [WTYH18]. Another direction of future research is to incorporate some physical effects neglected by our pipeline, such as foam and caustics.

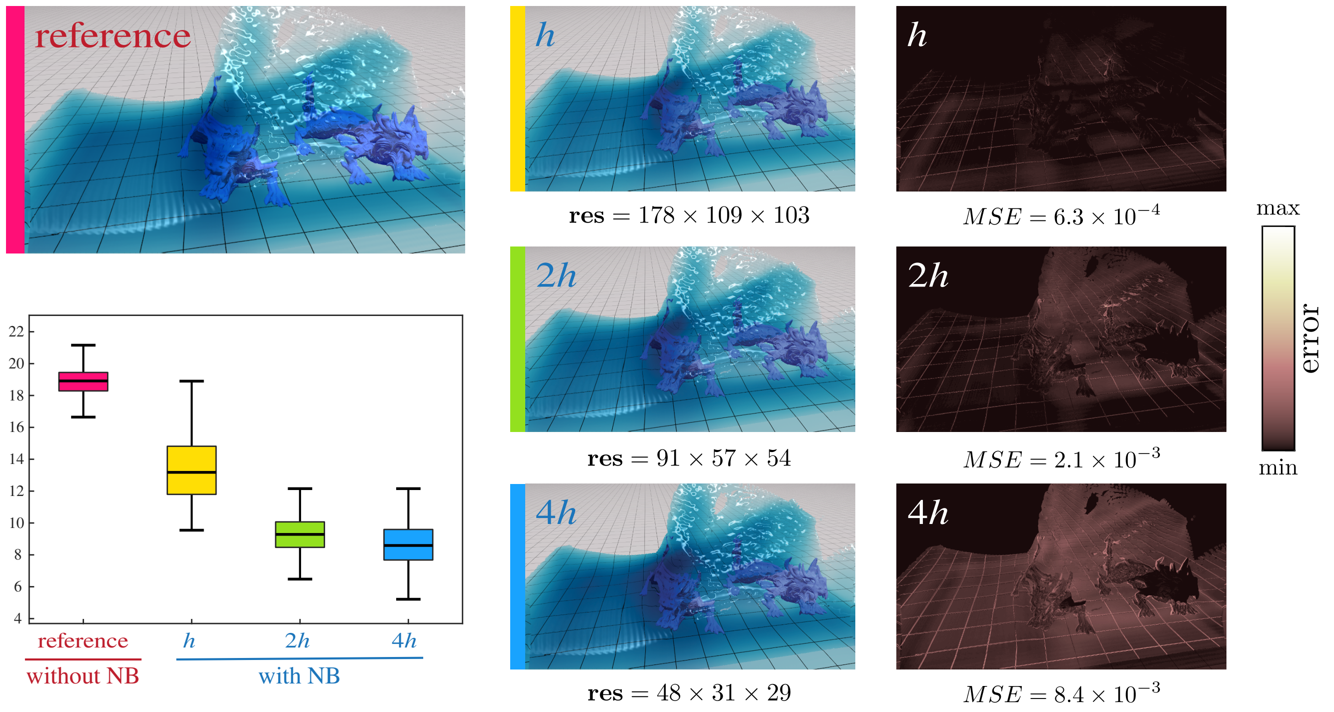
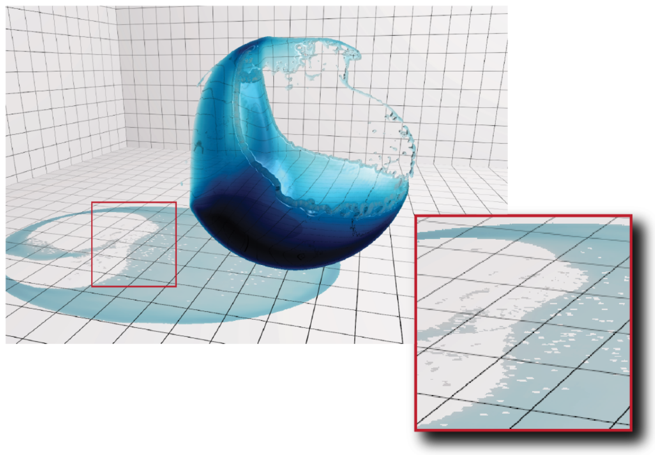
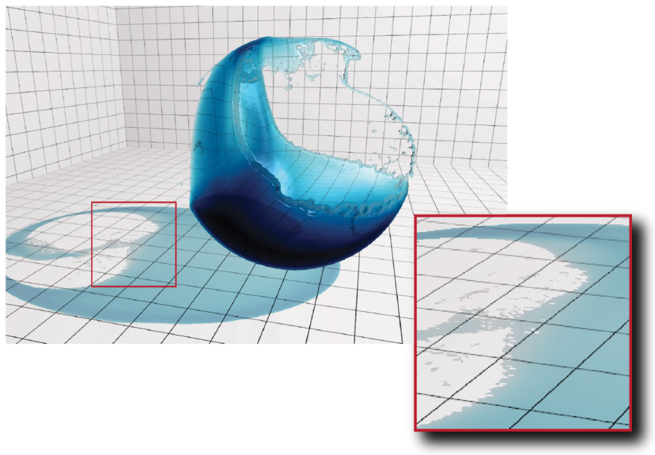


Figure 14: Analysis of the grid resolution: NB screen-space renderings using the narrow-range filter (*middle*), boxplots of the computational times (in milliseconds) for different voxel sizes (*bottom-left*), and the error images (*right*) between our NB renderings and the reference image without NB (*top-left*).



(a) NB with Müller et al. [MCG03].



(b) NB with our LNM.

Figure 15: Shadow generation using different NB screen-space fluid renderings. (a) Muller's NB produces incorrect shadows due to the presence of holes on the poor reconstructed liquid surface. (b) Our NB represents the shadow accurately due to the excellent light occlusion by the well-reconstructed surface yielded by LNM.

7. Conclusion

We introduced a simple and novel narrow-band screen-space liquid rendering pipeline for SPH fluids on CPU and GPU architectures. The presented method performs particle filtering only in an narrow-band around the boundary particles to provide a high-quality liquid surface. We also proposed an efficient boundary method suited to screen-space rendering called Layered Neighborhood Method. Our method speeds up and reduces memory footprint of the regular screen-space fluid rendering methods regardless of the choice of the depth buffer filtering scheme, as attested by the set of experiments and comparisons carried out in the paper.

Acknowledgements

We want to thank the anonymous reviewers for their valuable suggestions. This study was financed in part by the National Council for Scientific and Technological Development – Brazil (CNPq) under grant 309226/2020-1, and the São Paulo Research Foundation (FAPESP) under grant 2019/23215-9. The computational resources provided by the Center for Mathematical Sciences Applied to Industry (CeMEAI), also funded by FAPESP (grant 2013/07375-0).

References

[AIAT12] AKINCI, G., IHMSEN, M., AKINCI, N., and TESCHNER, M. "Parallel Surface Reconstruction for Particle-Based Fluids". *Comput. Graph. Forum* 31.6 (2012), 1797–1809. DOI: 10.1111/j.1467-8659.2012.02096.x 2.

[APKG07] ADAMS, BART, PAULY, MARK, KEISER, RICHARD, and GUIBAS, LEONIDAS J. "Adaptively Sampled Particle Fluids". *ACM Trans. Graph.* 26.3 (2007). DOI: 10.1145/1276377.1276437 2.

[BSS*18] BIEDERT, T., SOHNS, J.-T., SCHRÖDER, S., et al. "Direct Raytracing of Particle-Based Fluid Surfaces using Anisotropic Kernels". *Symposium on Parallel Graphics and Visualization (EGPGV '18)*. 2018, 1–12 2.

[BSW10] BAGAR, F., SCHERZER, D., and WIMMER, M. "A Layered Particle-Based Fluid Model for Real-Time Rendering of Water". *Eurographics Conference on Rendering (EGSR '10)*. 2010, 1383–1389. DOI: 10.1111/j.1467-8659.2010.01734.x 2.

[Cha04] CHAMBOLLE, A. "An Algorithm for Total Variation Minimization and Applications". *J. Math. Imaging Vis.* 20.1–2 (2004), 89–97. DOI: 10.1023/B:JMIV.0000011325.36760.1e 2.

[CS09] CORDS, H. and STAADT, O. G. "Interactive Screen-Space Surface Rendering of Dynamic Particle Clouds". *Journal of Graphics, GPU, and Game Tools* 14.3 (2009), 1–19. DOI: 10.1080/2151237X.2009.10129282 2.

[CZZ21] CHEN, QIAORUI, ZHANG, SHUAI, and ZHENG, YAO. "Parallel realistic visualization of particle-based fluid". *Comput. Animat. Virt. W.* 32.3–4 (2021), e2019. DOI: https://doi.org/10.1002/cav.2019 2.

[FAW10] FRAEDRICH, R., AUER, S., and WESTERMANN, R. "Efficient High-Quality Volume Rendering of SPH Data". *IEEE Trans. Vis. Comput. Graph.* 16.6 (2010), 1533–1540. DOI: 10.1109/TVCG.2010.148 2.

[Gre10] GREEN, S. *Screen Space Fluid Rendering for Games*. http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf. Game Developers Conference. 2010 2, 3, 7, 9.

[GSSP10] GOSWAMI, P., SCHLEGEL, P., SOLENTHALER, B., and PAJAROLA, R. "Interactive SPH Simulation and Rendering on the GPU". *Symposium on Computer Animation (SCA '10)*. 2010, 55–64 2.

[HLW*12] HE, X., LIU, N., WANG, G., et al. "Staggered Meshless Solid-fluid Coupling". *ACM Trans. Graph.* 31.6 (2012), 149:1–149:12. DOI: 10.1145/2366145.2366168 4, 6.

[HZGJ19] HU, Y., ZHANG, X., GAO, M., and JIANG, C. "On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects". *ACM SIGGRAPH 2019 Courses*. 2019. DOI: 10.1145/3305366.3328075 1.

[IKM16] IMAI, T., KANAMORI, Y., and MITANI, J. "Real-Time Screen-Space Liquid Rendering with Complex Refractions". *Comput. Animat. Virtual Worlds* 27.3–4 (2016), 425–434. DOI: 10.1002/cav.1707 2, 3, 7, 9.

[IOS*14] IHMSEN, M., ORTHMANN, J., SOLENTHALER, B., et al. "SPH Fluids in Computer Graphics". *Eurographics 2014 - State of the Art Reports*. 2014, 21–42. DOI: 10.2312/egst.20141034 1, 4.

[KH13] KAZHDAN, M. and HOPPE, H. "Screened Poisson Surface Reconstruction". *ACM Trans. Graph.* 32.3 (2013), 29:1–29:13. DOI: 10.1145/2487228.2487237 2.

[LC87] LORENSEN, W. E. and CLINE, H. E. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". *SIGGRAPH '87*. 1987, 163–169. DOI: 10.1145/37402.37422 1.

[LGS09] Van der LAAN, W. J., GREEN, S., and SAINZ, M. "Screen Space Fluid Rendering with Curvature Flow". *Symposium on Interactive 3D Graphics and Games (I3D '09)*. 2009, 91–98. DOI: 10.1145/1507149.1507164 2, 3, 7, 9.

[MCG03] MÜLLER, M., CHARYPAR, D., and GROSS, M. "Particle-based Fluid Simulation for Interactive Applications". *Symposium on Computer Animation (SCA '03)*. 2003, 154–159 2, 4, 6, 10.

[MM13] MACKLIN, M. and MÜLLER, M. "Position Based Fluids". *ACM Trans. Graph.* 32.4 (2013). DOI: 10.1145/2461912.2461984 1.

[MSD07] MÜLLER, M., SCHIRM, S., and DUTHALER, S. "Screen Space Meshes". *Symposium on Computer Animation (SCA '07)*. 2007, 9–15 2.

[NA17] NETO, L. S. R. and APOLINÁRIO JR., A. L. "Real-Time Screen Space Cartoon Water Rendering with the Iterative Separated Bilateral Filter". *Journal on Interactive Systems* 8.1 (2017). DOI: 10.5753/jis.2017.672 2.

[RCSW14] REICHL, F., CHAJDAS, M. G., SCHNEIDER, J., and WESTERMANN, R. "Interactive Rendering of Giga-Particle Fluid Simulations". *Symposium on High Performance Graphics (HPG '14)*. 2014, 105–116 2.

[SCN*16] SANDIM, M., CEDRIM, D., NONATO, L. G., et al. "Boundary Detection in Particle-based Fluids". *Comput. Graph. Forum* 35.2 (2016), 215–224. DOI: 10.1111/cgf.12824 2, 4–6.

[SP09] SOLENTHALER, B. and PAJAROLA, R. "Predictive-Corrective Incompressible SPH". *ACM Trans. Graph.* 28.3 (2009). DOI: 10.1145/1531326.1531346 7.

[SPd20] SANDIM, MARCOS, PAIVA, AFONSO, and DE FIGUEIREDO, LUIZ HENRIQUE. "Simple and reliable boundary detection for meshfree particle methods using interval analysis". *J. Comput. Phys.* 420 (2020), 109702. DOI: https://doi.org/10.1016/j.jcp.2020.109702 3.

[SSP07] SOLENTHALER, B., SCHLÄFLI, J., and PAJAROLA, R. "A Unified Particle Model for Fluid–Solid Interactions". *Comput. Animat. Virt. W.* 18.1 (2007), 69–82. DOI: 10.1002/cav.162 2.

[TY18] TRUONG, N. and YUKSEL, C. "A Narrow-Range Filter for Screen-Space Fluid Rendering". *ACM Comput. Graph. Interact. Tech.* 1.1 (2018). DOI: 10.1145/3203201 1, 2, 7, 9.

[WTYH18] WU, KUI, TRUONG, NGHIA, YUKSEL, CEM, and HOETZLEIN, RAMA. "Fast Fluid Simulations with Sparse Volumes on the GPU". *Comput. Graph. Forum* 37.2 (2018), 157–167. DOI: https://doi.org/10.1111/cgf.13350 9.

[XZY18] XIAO, X., ZHANG, S., and YANG, X. "Fast, High-Quality Rendering of Liquids Generated using Large Scale SPH Simulation". *Journal of Computer Graphics Techniques (JCGT)* 7.1 (2018), 17–39 2.

[YG20] YANG, W. and GAO, C. "A Completely Parallel Surface Reconstruction Method for Particle-based Fluids". *Vis. Comput.* 36 (2020), 2313–2325. DOI: 10.1007/s00371-020-01898-2 2, 6.

[YT13] YU, J. and TURK, G. "Reconstructing Surfaces of Particle-based Fluids using Anisotropic Kernels". *ACM Trans. Graph.* 32.1 (2013), 5:1–5:12. DOI: 10.1145/2421636.2421641 2.

[ZB05] ZHU, Y. and BRIDSON, R. "Animating Sand as a Fluid". *ACM Trans. Graph.* 24.3 (2005), 965–972. DOI: 10.1145/1073204.1073298 1, 2.

[ZD15] ZIRR, T. and DACHSBACHER, C. "Memory-Efficient on-the-Fly Voxelization of Particle Data". *Symposium on Parallel Graphics and Visualization (PGV '15)*. 2015, 11–18 2.