Real-Time Screen-Space Liquid Rendering with Complex Refractions

T. Imai Y. Kanamori J. Mitani
University of Tsukuba
imai-t@npal.cs.tsukuba.ac.jp, {kanamori,mitani}@cs.tsukuba.ac.jp

Abstract

Particle-based liquid is often rendered only with single refraction in real-time applications, which deteriorates the reality of liquid. We present a screen-space method for rendering particle-based liquids with up to four refractions in real time. Our method separates liquid particles into splashes and aggregations (i.e., liquid bodies), and generates a pair of depth maps of front- and back-facing surfaces for each. We use the depth maps of splashes as they are, but smooth those of aggregations to reduce small bumps. For smoothing depth, we iteratively fit planes locally using moving least-squares, unlike previous filtering-based approaches that cause undesirable refractions around depth boundaries. By calculating refractions with physically-based light attenuation, our method can render liquids more realistically than previous methods.

Keywords: particle-based liquids, multiple refractions, real-time rendering, screen-space approach

1 Introduction

Particle-based liquid simulation [1, 2, 3] has been widely adopted in interactive applications. Nowadays even real-time liquid simulation with millions of particles is possible using massively parallel computation on multiple GPUs [4] or machine learning [5].

To render liquids realistically, accurate handling of refractions at liquid surfaces is visually crucial. In typical real-time applications, however, only single refraction at the front-facing surface is calculated to reduce the rendering cost, which cannot represent fascinating light interactions in liquid and thus deteriorates the reality of liquid. Accurate handling of refractions has been overlooked in most real-time rendering methods for particle-based liquids.

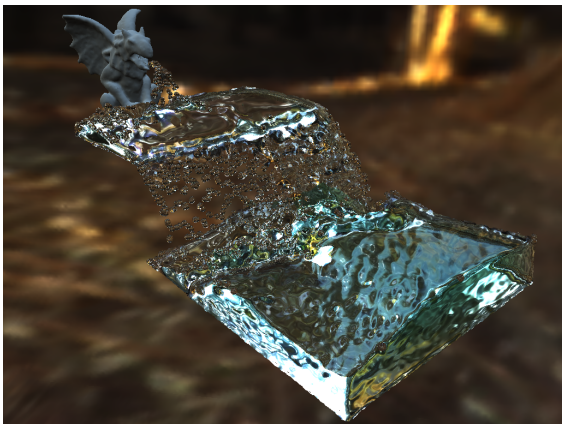


Figure 1: Our rendering result of liquid (40,000 particles, 67.6 fps at 1024×768) in consideration of up to four refractions and the *Beer-Lambert law*.

To the best of our knowledge, Imai et al.'s method [6] for rendering liquid is the only real-time method that can handle multiple refractions in the screen space. Their method calculates only two refractions per pixel because it only generates the nearest and farthest surfaces of liquid, and the iterative bilateral filtering used for depth smoothing is too costly for more than two depth layers. Here we point out that their

method has two problems that cause inaccurate refractions. First, if the viewing ray hits the front-facing surface of an isolated splash particle in front of liquid bodies, the refracted ray cannot find the back-facing surface of the splash, which is not recorded in depth maps. Second, because they employ filtering to smooth depth maps, the resultant depth maps unnaturally warp towards the viewpoint around depth boundaries. This second issue is also problematic in the previous filtering-based approaches [7, 8].

In this paper, we propose a real-time rendering method that addresses the problems above. Our method separates liquid particles into splash particles and aggregated particles, and generates a pair of depth maps of front- and back-facing surfaces for each, making up four depth maps in total. We use the depth maps of splashes as they are to keep their accurate shapes, but smooth those of aggregations to reduce small bumps on liquid surfaces. Instead of filtering to depth smoothing, our method applies an iterative plane fitting, which smoothes depth maps strongly while keeping slopes around depth boundaries.

Obviously more refractions with more depth layers would improve the realism of liquid, but we demonstrate that up to four refractions with four depth layers offers a good tradeoff between visual quality and speed. By additionally considering light attenuation based on the Beer-Lambert law, our method can render liquids more realistically than previous approaches, as shown in Figure 1.

2 Related Work

Liquid particles as metaballs. For rendering, particle-based liquid has been represented as blobs or metaballs. Liquid surfaces are then extracted as polygonal meshes using marching cubes, or directly rendered using ray casting/tracing. Kanamori et al. [9] rendered metaballs by ray-casting on the GPU only with single refraction. Gourmel et al. [10] accelerated ray tracing of metaballs using *Bounding Volume Hierarchy* (BVH). Although these approaches yield accurate surfaces, they are expensive to render liquid with a large number of particles in real time.

Scalar field discretization. An acceleration technique for metaball rendering is to accumulate the density field of metaballs in a uniform grid, and perform ray casting/tracing with the grid, as demonstrated by Fraedrich et al. [11] with single refraction. Rather than metaball's density field, Goswami et al. [12] employed ray-tracing with a distance field defined by particles. Problems with grid-based discretization are that coarse grids often miss small particles and cause temporal aliasing while dense grids require a large amount of memory to store the scalar fields. Sparse representation of dense grids would alleviate them, with additional performance overhead.

Screen-space approaches. An approximate yet fast way to generate liquid surfaces is to smooth a depth map generated by rendering particles as spheres or ellipsoids. Müller et al. [13] extracted liquid surfaces as screen-space polygon meshes from the depth map of particles, and then smoothed the meshes. Cords and Staadt [7] generated surfaces by smoothing depth maps using binomial filter. van der Laan et al. [14] also smoothed depth maps using *Mean Curvature Flow*, which minimizes mean curvature of liquid surface. They further calculated the pseudo thickness to imitate the thickness of liquid. Green [8] replaced Mean Curvature Flow with separated bilateral filtering to reduce the computational cost. Bagar et al. [15] introduced adaptive curvature flow as well as a layered model of liquids to handle foams. Macklin and Müller [1] demonstrated real-time rendering of particle-based liquids with ellipsoid splatting [16] followed by smoothing [14].

Screen-space methods are scalable w.r.t. the number of particles because only the step of initial depth generation depends on the number of particles, which can be accelerated by extracting near-surface particles and rendering them only [12]. However, the approaches above target single refraction at front-facing surfaces only. Our method also belongs to screen-space methods, but calculates up to four refractions.

Refraction handling. In the context of real-time rendering with polygon meshes, Wyman [17] showed that just two refractions

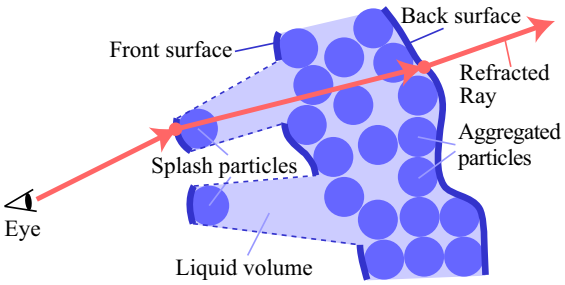


Figure 2: Problem 1. The refracted ray misses the back-facing surface of the splash. Also note that splash surfaces are flattened due to Problem 2.

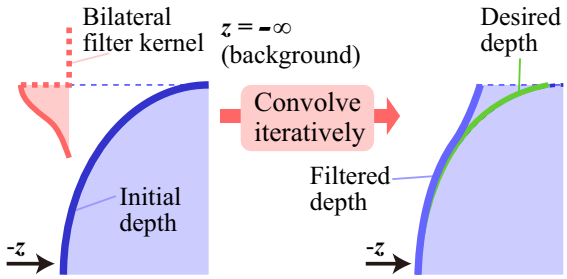


Figure 3: Problem 2. Filtered depth unnaturally warps forward compared to the desired depth. This warping causes undesirable refracted directions.

at front- and back-facing surfaces make results sufficiently photorealistic. His approach was further extended by calculating the ray-intersection at back-facing surfaces using the binary search to handle deformable objects [18] and by calculating total internal refraction [19].

Imai et al. [6] combined the above screen-space approaches for generating depth maps and calculating two refractions for the sake of liquid surfaces. Because their method is closely related to our work, we introduce it in the next section.

3 Two Refractions with Smoothed Depth Maps

Here we briefly review the algorithm by Imai et al. [6] for rendering liquid surfaces with two refractions in screen space, and then point out the two problems that cause inaccurate refractions, as introduced in Section 1.

3.1 Algorithm by Imai et al.

Their method consists of three major steps, namely, depth generation, depth smoothing, and calculation of refractions. For depth generation, their method renders particles as spheres, and records the depth values of front- and back-facing surfaces of each sphere simultaneously using dual depth peeling. It then smoothes both of the nearest and farthest depth maps by applying bilateral filter iteratively. Finally, it calculates ray-surface intersections using the secant method to calculate two refractions with the two depth maps.

3.2 Problems of Their Algorithm

Problem 1. Figure 2 illustrates their first problem. Their method renders isolated splash particles and aggregated particles into the same depth maps for nearest and farthest surfaces. If splash particles accidentally lie in front of aggregated particles, the splashes erroneously have elongated volumes along viewing directions, starting from the front-facing surfaces of the splashes and ending at the back-facing surfaces of aggregated particles. When viewing rays hit the front-facing surfaces of such splashes, the refracted rays cannot find the back-facing surfaces of the splashes, which yields undesirable refractions. From this fact, we conclude that refractions only with nearest and farthest surfaces do not suffice for liquid, unlike polygon meshes [17].

Problem 2. Figure 3 shows another problem. If we apply filtering to depth boundaries, valid depth values are available only on one side of the filter kernel and values on the other side (e.g., $z = -\infty$ for background) are ignored. Because the valid depth values are larger (i.e., closer to the viewpoint) than the value of the center pixel and the filter weights are summed up to one, the resultant depth also becomes larger than the original depth of the center pixel. Consequently, the smoothed depth maps unnaturally warp towards the viewpoint around depth boundaries, which is particularly noticeable in flattened look of splash particles. This warping perturbs surface normals, yielding undesirable refracted directions. As we explained, previous filtering-based approaches for depth smoothing [7, 8] share the same issue.

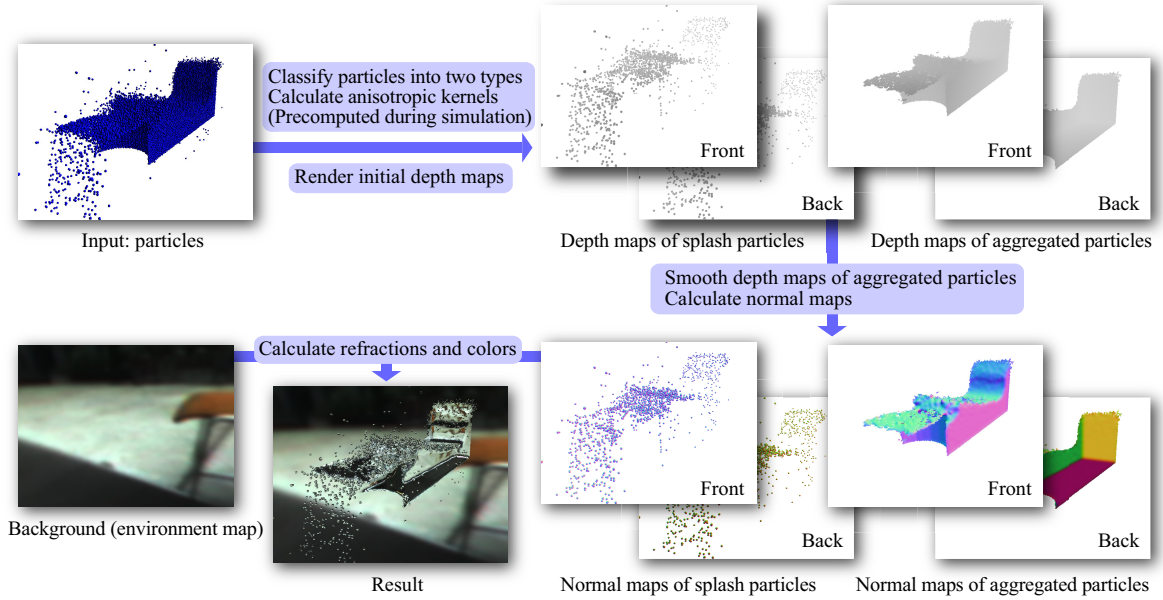


Figure 4: Overview of our method. Given liquid particles with classification (i.e., splash or aggregated particles) and anisotropic kernels for aggregated particles, initial depth maps for front- and back-facing surfaces are rendered. The depth maps of aggregated particles are smoothed by iterative plane fitting. Finally, refractions and output colors are calculated to generate a result image.

4 Our Method

4.1 Overview

We assume that classification of particles (i.e., splash or aggregated particles) based on the local density of neighboring particles as well as ellipsoidal shapes [16] of aggregated particles can be precomputed at negligible cost, on top of the nearest-neighbor searches during simulation [20]. The particles' positions and shapes (i.e., sphere radius or ellipsoidal shapes) are given as inputs.

We address the two problems described in Section 3.2 as follows. For Problem 1, we separate liquid particles into splash and aggregated particles, and generate depth maps of nearest and farthest surfaces for each of the two types of particles. A liquid particle is classified as splash if the local density is smaller than a threshold, or otherwise classified as aggregated. Splash particles are rendered as spheres whereas aggregated particles are rendered as ellipsoids [16] to flatten the liquid surfaces. For Problem 2, we do not smooth depth maps of splashes, but smooth those of aggregated particles via iterative local plane fitting, which does not cause depth warping around depth boundaries. After obtaining four depth maps, we seek for ray-surface intersections using the secant method to calculate refractions up to four times. To enhance the reality of liquid, we integrate the Beer-Lambert law to calculate more accurate light attenuation than the previous methods [14, 15].

Figure 4 summarizes the overview of our method. Below we explain our four major steps, i.e., initial depth generation, iterative local plane fitting, intersection tests with two types of particles, and calculation of colors with the Beer-Lambert law.

4.2 Initial Depth Generation

We obtain initial depth maps by rendering spheres for splash particles, and by rendering ellipsoids for aggregated particles. We use the optimized method of Kanamori et al. [9] for rendering spheres and Sigg et al.'s method [21] for rendering ellipsoids. Surface normals of spheres and ellipsoids are calculated analytically. The nearest and farthest depth maps of splashes are used as-is to keep the accurate shapes of splashes. On the other hand, those of aggregated

particles are smoothed by iterative local plane fitting, as explained in the next section.

In our implementation, each splash has the same radius r, while ellipsoid sizes of aggregated particles are varying but bounded by $1.2r$.

4.3 Iterative Local Plane Fitting for Depth Smoothing

We apply local plane fitting iteratively for smoothing depth maps of aggregated particles. Suppose we fit plane Π_i to pixel i based on pixel i and its neighbors $\mathcal{N}_i$. The neighbors $\mathcal{N}_i$ are sampled within a window of $s \times s$ pixels, where s is the window size. We first calculate the eye-space positions $\mathbf{p}_i$ for pixel i and $\mathbf{p}_j$ for its neighbors $j \in \mathcal{N}_i$ from their depth values z_i and z_j. Let the equation of plane Π_i be $\mathbf{n}_i^T \mathbf{x} = d_i$, where $\mathbf{n}_i$ is a unit normal (i.e., $\|\mathbf{n}_i\| = 1$) and d_i is the distance to the origin. We determine the plane Π_i by minimizing the following error:

$$E = \sum_{j \in \mathcal{N}_i} w_{ij} \left(\mathbf{n}_i^T \mathbf{p}_j - d_i\right)^2 \quad s.t. \ \|\mathbf{n}_i\| = 1, \quad (1)$$

$$w_{ij} = \exp\left(-(z_j - z_i)^2 / \sigma^2\right), \quad (2)$$

where σ is a user-specified parameter (we set $\sigma = 5.3r$). We calculate the surface normal $\mathbf{n}_i$ as a weighted average of normals:

$$\mathbf{n}_i = \frac{\sum_{j \in \mathcal{N}_i} w_{ij} \tilde{\mathbf{n}}_j}{\|\sum_{j \in \mathcal{N}_i} w_{ij} \tilde{\mathbf{n}}_j\|}, \quad (3)$$

where $\tilde{\mathbf{n}}_j$ are aggregated particles' normals before smoothing. Note that $\tilde{\mathbf{n}}_j$ are unchanged but $\mathbf{n}_i$ is changed in each iteration because the z values in Eq. (2) are updated accordingly. From condition $\frac{\partial E}{\partial d_i} = 0$, we obtain

$$d_i = \mathbf{n}_i^T \overline{\mathbf{p}}_i, \qquad \overline{\mathbf{p}}_i = \frac{\sum_{j \in \mathcal{N}_i} w_{ij} \mathbf{p}_j}{\sum_{j \in \mathcal{N}_i} w_{ij}}. \quad (4)$$

Weighted averaging of normals is a common technique in the literature of surface reconstruction from point clouds [22]. An important difference is that we use view-dependent weights, i.e., Eq. (2); see Section 5 for this choice of weighting functions. Given the unit direction vector $\mathbf{v} = (v_x, v_y, v_z)^T$ of a viewing ray passing through pixel i, the eye-space z coordinate of the ray-plane intersection is

$$z = \frac{d_i}{\mathbf{n}_i^T \mathbf{v}} v_z. \quad (5)$$

Note that the weighted averaged normals of Eq. (3) do not represent the normals of the smoothed surface accurately, and thus we recalculate surface normals after smoothing.

4.4 Intersection Tests with Two Types of Liquid Particles

For each viewing ray, we have to check if the ray hits splashes or liquid bodies, both of which are recorded in different pairs of depth maps. There are four patterns of orders in which a ray hits the two types of liquid, i.e., splashes and aggregations; splashes $\rightarrow$ splashes, splashes $\rightarrow$ aggregations, aggregations $\rightarrow$ splashes, or aggregations $\rightarrow$ aggregations. Note that we check whether incident rays hit the front-facing surfaces and whether exitant rays hit the back-facing surfaces, for each of splashes and aggregations. We check all the four patterns for each viewing ray. The percentage of each pattern is, for example, 34.2%/47.2%/6.7%/11.9% in Figure 7, and 18.0%/64.5%/14.8%/2.7% in Figure 8.

4.5 Output Color Calculation with Beer-Lambert Law

The output color $\mathbf{c}$ for each pixel is calculated recursively with k-th intersections ($k = 1, 2, 3$):

$$\mathbf{c} = F_1 \mathbf{b}(\mathbf{r}_1) + (1 - F_1) \mathbf{c}(\mathbf{t}_1), \quad (6)$$

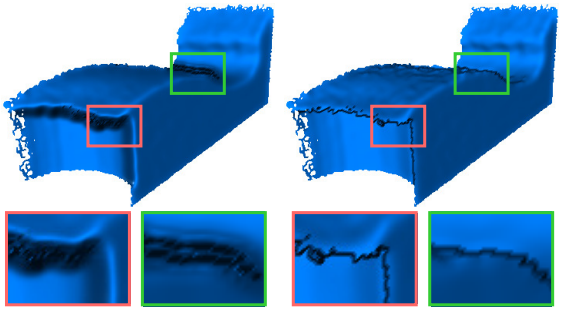
$$\mathbf{c}(\mathbf{t}_k) = F_{k+1} \mathbf{b}(\mathbf{r}_{k+1}) + (1 - F_{k+1}) \mathbf{c}(\mathbf{t}_{k+1}), \quad (7)$$

$$\mathbf{c}(\mathbf{t}_4) = \mathbf{b}(\mathbf{t}_4), \quad (8)$$

where $\mathbf{b}(\cdot)$ is the background color fetched from the specified direction, $\mathbf{r}_k$ the reflected direction, $\mathbf{c}(\mathbf{t}_k)$ the refraction color in the refracted direction $\mathbf{t}_k$, and F_k a Fresnel coefficient, for which we employ Shlick's approximation [23]. If a third intersection is not found, we set $\mathbf{c}(\mathbf{t}_2) = \mathbf{b}(\mathbf{t}_2)$.

Because the light path length in liquid is available, we can calculate the physically-based attenuation due to light absorption in liquid based on the Beer-Lambert law. Let c_b be the color of the light trasmitting in the liquid (either of r, g, or b channels). We calculate the color c_b' affected by the Beer-Lambert law as follows:

$$c_b' = c_b \exp(-a\, l\, c_a), \quad (9)$$



(a) With view-independent weights (b) With view-***dependent*** weights

Figure 5: Comparison of (a) view-independent and (b) view-dependent weights used in plane fitting. The latter preserves depth gaps better than the former.

where a is the absorption coefficient, l the light path length in the liquid, c_a the strength of the attenuation for each color channel, respectively. In our results, we experimentally set $a = 1.0$ to obtain good appearance.

5 Results

We implemented our method using C++ with OpenGL, GLUT, GLUI and GLEW. We wrote the shader codes using GLSL. The experiments were conducted on a PC equipped with an Intel Core i7-4770 3.40GHz CPU, 8GB RAM, and an NVIDIA GeForce GTX TITAN GPU.

The scenes used in the following results are "*wave*" with roughly 125,000 particles and "*steps*" scene with roughly 40,000 particles. The "*wave*" scene was pre-simulated while "*steps*" scene was simulated on the fly on the CPU. The reported computational times do not include the time for simulation. The times for calculating particle classification and anisotropic kernels are not included either, which we currently calculate on the CPU.

View-independent vs. dependent weights in plane fitting. Figure 5 compares opaque surfaces smoothed using two candidates of weighting functions in plane fitting; the left result was rendered with view-independent weights based on squared Euclid distance, i.e., $w_{ij} = \exp(-\|\mathbf{p}_i - \mathbf{p}_j\|^2/\sigma^2)$, while the right result with view-independent weights based on eye-space z values, i.e., Eq. (2). While the depth gaps are smoothed out with the view-independent weights, they are preserved with the view-dependent weights because the latter weights emphasize depth gaps more strongly.

Different parameter settings in plane fitting. Figure 6 shows a comparison with different window sizes and numbers of iterations in plane fitting. The resolution of each image is 640×480. The results show that our iterative plane fitting has a strong smoothing effect even with a few iterations. To balance the quality and speed, we chose 9×9 pixels as the window size. The numbers of iterations are two for "wave" scene in Figure 7(c)(f) and five for "steps" scene in Figure 8(c)(f).

Different numbers of refractions. Figures 7 and 8 compare the results with different numbers of refractions. (a)(b) were rendered with single and two refractions using the method by Imai et al. [6]. (d) is a reference image rendered using an offline ray tracer with surface meshes. Note that this reference image does not necessarily have the best quality, due to the low-resolution surface meshes reconstructed from the depth maps. (c)(f) were rendered using our method.

Figure 7(e) clarifies the problems in (b); due to Problem 1, the splashes seem as if they were fused to the liquid body behind them. Also due to Problem 2, the splashes seem as if they were flattened, yielding unnatural refracted colors. In Figure 8(e), (c1) and (d1) show our result looks comparable to that of ray tracing. However, from (c2) and (d2) in Figure 8(e), we can see our result is quite different from that of ray tracing. This is because we did not trace reflected rays recursively, which is a limitation of our method.

Timing comparison. We also compared the computational times of Imai et al.'s method and ours. As shown in the graphs of Figure 9, we can see the bottleneck of both methods is the depth smoothing step, and the overall performance of our method is better than their method; our method can smooth depth maps faster, to the comparable quality of their method, thanks to ellipsoidal kernels and iterative plane fitting. Note that our timings do not contain the times for cal-

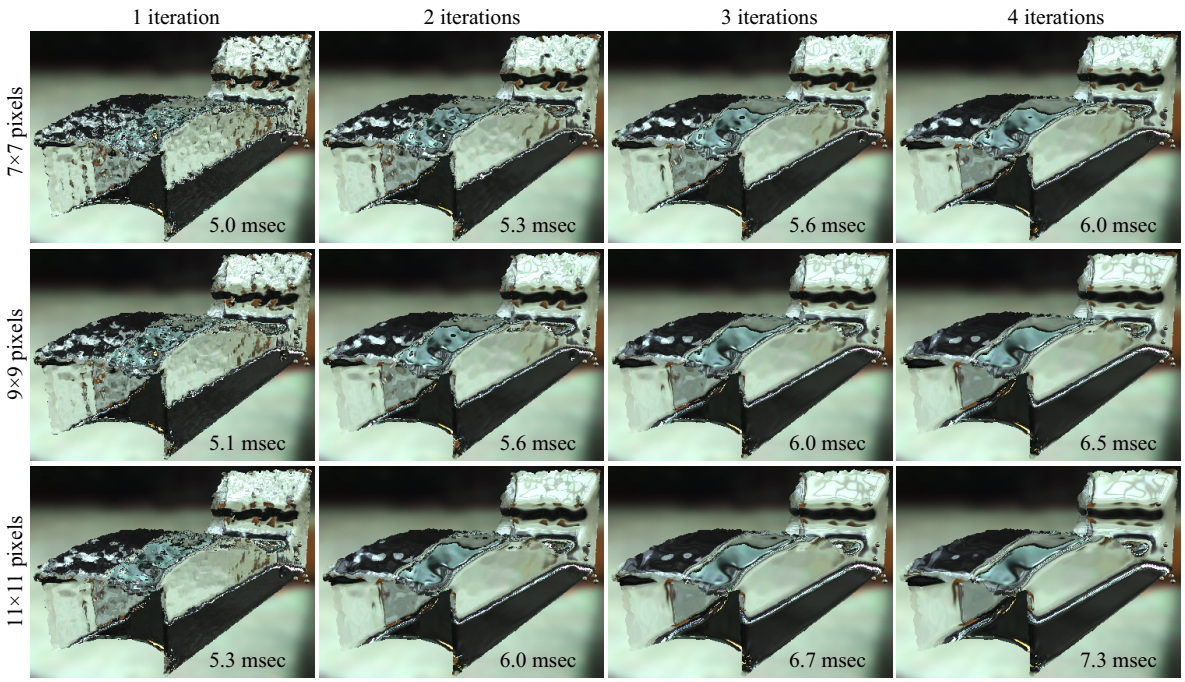


Figure 6: Comparison with different window sizes and numbers of iterations in plane fitting. Each image is trimmed to enlarge the liquid. The computational time is shown in each image.

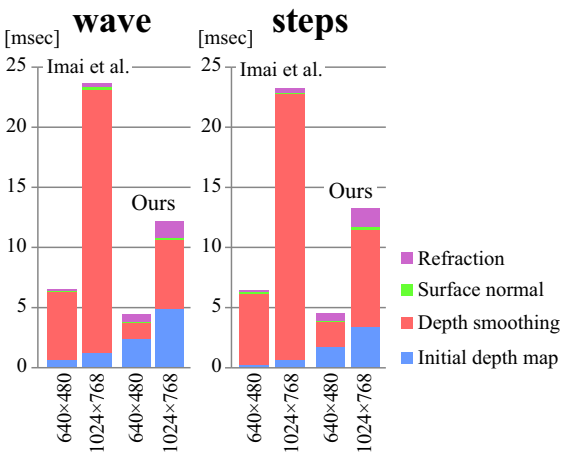


Figure 9: Graphs of computational times (msec) of Imai et al. [6] and ours in "wave" and "steps" scenes.

culating ellipsoidal kernels, and thus the performance gap would be shortened, depending on implementations.

Million particle results. Figure 10 shows our results with 1,056K particles, rendered at 1024×768 with five iterations of plane fitting. Thanks to our screen-space approach, the frame rates are still quite fast.

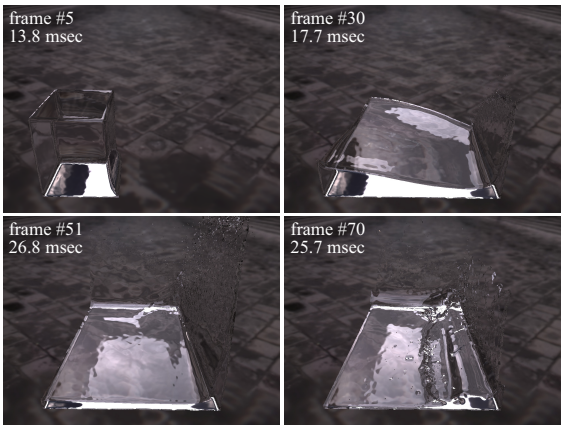
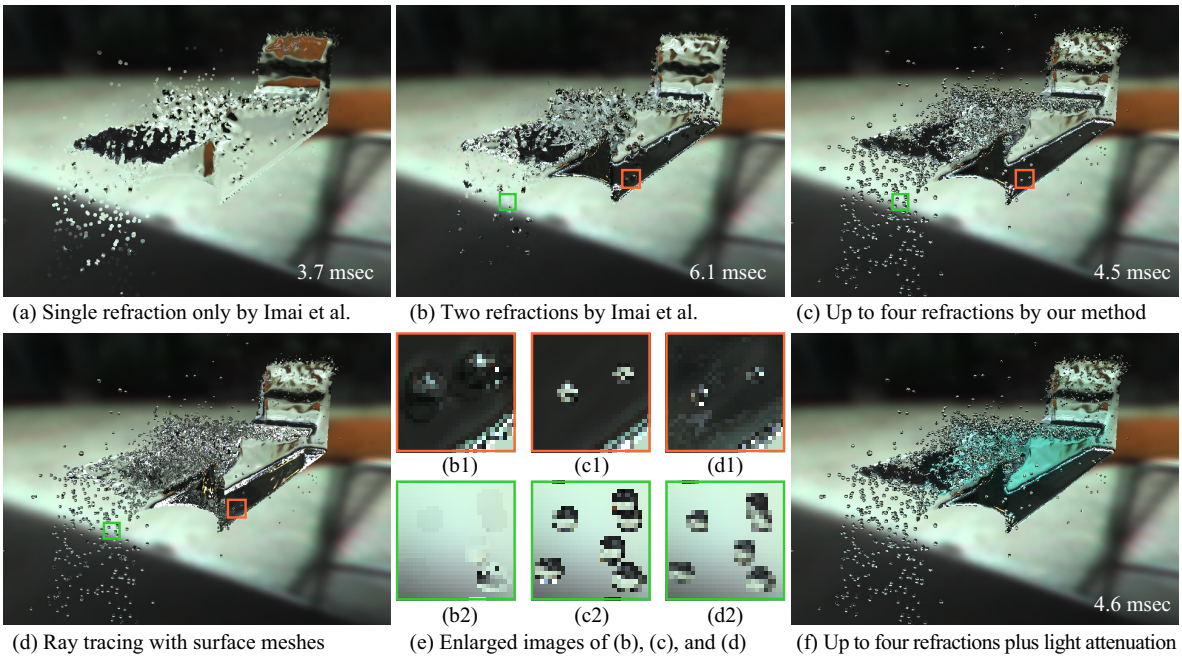


Figure 10: Our results with 1,056K particles, rendered at 1024×768.

Animation sequences. Figure 11 shows frames from animation sequences of "wave" and "steps" scenes rendered using our method with the Beer-Lambert law. Please also refer to the accompanying video for the original animations.

6 Conclusions and Future Work

In this paper, we have proposed a screen-space method for rendering particle-based liquids with



(a) Single refraction only by Imai et al. (b) Two refractions by Imai et al. (c) Up to four refractions by our method

(d) Ray tracing with surface meshes (e) Enlarged images of (b), (c), and (d) (f) Up to four refractions plus light attenuation

Figure 7: Comparison of "wave" scene with different number of refractions, rendered at 640×480.

up to four refractions in real time. We first pointed out the problems of the filtering-based method by Imai et al. [6] and their related methods [7, 8], which cause undesirable refractions. We solved the problems by separating liquid particles into splash and aggregated particles, and by performing iterative local plane fitting, instead of filtering-based depth smoothing. By calculating light attenuation based on the Beer-Lambert law, we could further enhance the reality of liquid.

For future work, we would like to accelerate our current bottleneck steps, namely, depth smoothing and initial depth map generation.

Acknowledgements

We would like to thank Dr. Barbara Solenthaler for helpful comments. We would also like to thank Dr. SoHyeon Jeong for proofreading our paper and providing the simulation data in Figure 10. Yoshihiro Kanamori is funded by JSPS Postdoctoral Fellowships for Research Abroad.

References

[1] Miles Macklin and Matthias Müller. Position based fluids. *ACM Trans. Graph.*, 32(4):104:1–104:12, July 2013.

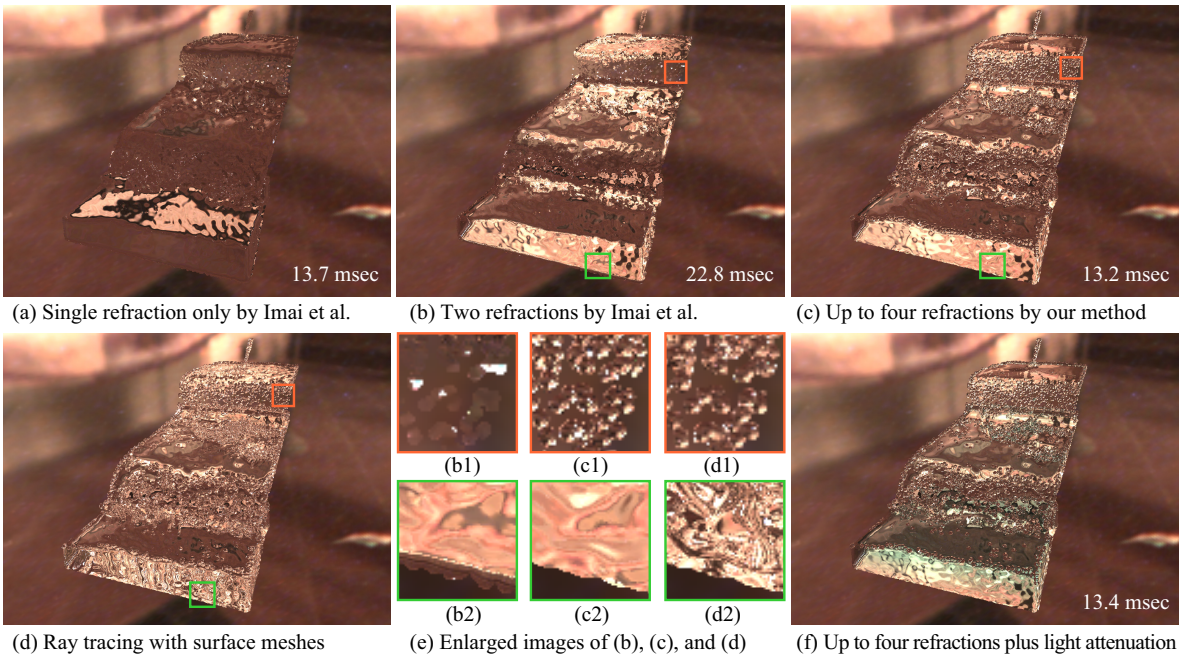
[2] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH Fluids in Computer Graphics. In Sylvain Lefebvre and Michela Spagnuolo, editors, *Eurographics 2014 - State of the Art Reports*. The Eurographics Association, 2014.

[3] Florian Reichl, Matthäus G. Chajdas, Jens Schneider, and Rüdiger Westermann. Interactive rendering of giga-particle fluid simulations. *Proceedings of High Performance Graphics 2014*, 2014.

[4] Takahiro Harada, Issei Masaie, Seiichi Koshizuka, and Yoichiro Kawaguchi. Massive particles: Particle-based simulations on multiple GPUs. In *ACM SIGGRAPH 2008 Talks*, pages 38:1–38:1, 2008.

[5] L'ubor Ladický, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. Data-driven fluid simulations using regression forests. *ACM Trans. Graph.*, 34(6):199:1–199:9, October 2015.

[6] Takuya Imai, Yoshihiro Kanamori, Yukio Fukui, and Jun Mitani. Real-time screen-



(a) Single refraction only by Imai et al. (b) Two refractions by Imai et al. (c) Up to four refractions by our method

(d) Ray tracing with surface meshes (e) Enlarged images of (b), (c), and (d) (f) Up to four refractions plus light attenuation

Figure 8: Comparison of "steps" scene with different number of refractions, rendered at 1024×768.

space liquid rendering with two-sided refractions. In *Proceedings of NICOGRAPH International 2014*, pages 71–76, 2014.

[7] Hilko Cords and Oliver Staadt. Instant liquids. In *Poster proceedings of ACM Siggraph/Eurographics symposium on computer animation*, 2008.

[8] Simon Green. Screen space fluid rendering for games. GDC 2010: Game Developers Conference 2010.

[9] Yoshihiro Kanamori, Zoltan Szego, and Tomoyuki Nishita. GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum*, 27(2):351–360, 2008.

[10] Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loïc Barthe, and Pierre Poulin. Fitted BVH for fast raytracing of metaballs. *Computer Graphics Forum*, 29(2):281–288, 2010.

[11] Roland Fraedrich, Stefan Auer, and Rüdiger Westermann. Efficient high-quality volume rendering of SPH data. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2010)*, 16(6):1533–1540, 2010.

[12] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 55–64, 2010.

[13] Matthias Müller, Simon Schirm, and Stephan Duthaler. Screen space meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 9–15, 2007.

[14] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pages 91–98, 2009.

[15] Florian Bagar, Daniel Scherzer, and Michael Wimmer. A layered particle-based fluid model for real-time rendering of water. *Computer Graphics Forum (Proceedings EGSR 2010)*, 29(4):1383–1389, June 2010.

[16] Jihun Yu and Greg Turk. Reconstructing surfaces of particle-based fluids using

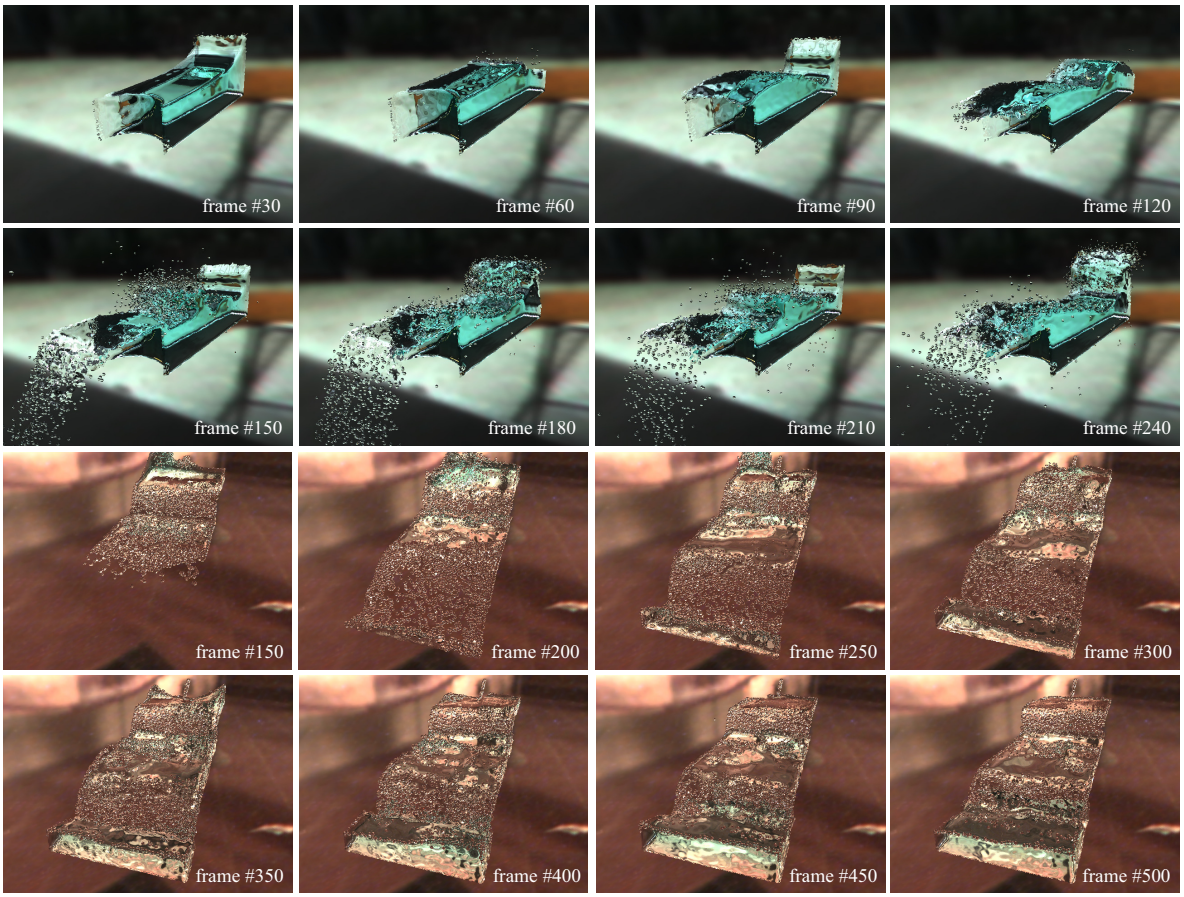


Figure 11: Animation sequences of “wave” (top) and “steps” (bottom) scenes.

anisotropic kernels. *ACM Trans. Graph.*, 32(1):5:1–5:12, 2013.

[17] Chris Wyman. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053, 2005.

[18] Manuel M. Oliveira and Maicon Brauwers. Real-time refraction through deformable objects. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 89–96, 2007.

[19] Scott T Davis and Chris Wyman. Interactive refractions with total internal reflection. In *Proceedings of Graphics Interface 2007*, pages 185–190, 2007.

[20] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, 2003.

[21] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. GPU-based ray-casting of quadratic surfaces. In *Proceedings of the 3rd Eurographics/IEEE VGTC conference on Point-Based Graphics*, pages 59–65, 2006.

[22] Marc Alexa, Markus H. Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. Point-based computer graphics. In *SIGGRAPH 2004, Course Notes*, page 7, 2004.

[23] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.