Screen Space Foam Rendering

Nadir Akinci Alexander Dippel Gizem Akinci Matthias Teschner
nakinci,dippela,gakinci,teschner@informatik.uni-freiburg.de
University of Freiburg
Georges Koehler Allee 052
79110 Freiburg Germany

ABSTRACT

We present a method for the efficient rendering of large scale particle-based foam data in screen space using a GPU based rendering pipeline. Our approach employs a multi-pass rendering technique to imitate some of the effects that are commonly accomplished by using expensive ray-tracing based methods. We demonstrate through different scenarios that our pipeline is able to produce convincing foam renderings for large scale scenarios and it has a significant performance advantage compared to using ray-casting techniques for rendering such particle data.

Keywords

Rendering, Fluids, Foam, Particles

1 INTRODUCTION

Foam is a complex phenomenon whose behavior and appearance is challenging to simulate in computer graphics. When viewed from a close distance, foam is composed of many air bubbles sticking to each other. It can occur inside most fluids as a result of trapped air. One can observe milky white foam caused by dashing waves on seashores. For most semi-transparent materials, it is an interesting observation that, even though the underlying material may have a color, the foam usually looks whitish to the viewer. The reason for this behavior is that the foam is composed of thin films of fluid containing air. As the number of such thin films increase per unit volume, all incoming light is reflected without allowing any light to penetrate beneath it. This optical phenomenon makes the foam look brighter than the material itself, to the point that it looks almost white. This paper focuses on the efficient rendering of such white foam by approximating some important effects in screen space, that are otherwise time consuming to compute in a physically correct way. Our technique is specifically useful for complex large-scale scenarios, where large amount of foam data need to be rendered. In the remainder of this section, we first summarize the existing works about GPU accelerated rendering of fluid data (Sec. 1.1), foam simulation and rendering (Sec.1.2) and then highlight our contribution (Sec. 1.3).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1.1 GPU Rendering of Fluids

For non-interactive applications, fluid surfaces are generally visualized by triangulating the isosurface of the particle data (e.g. [ZB05, YT10, AIAT12]) and then rendering the resultant mesh using ray-tracing based techniques to produce convincing results. For real-time applications, the computational overhead of those approaches remains too high. Therefore, for the efficient GPU accelerated visualization of fluid surfaces, several methods have been proposed in the recent years, e.g., using screen space surface construction [MSD07, FAW10], height field techniques [CM10] and methods that are based on particle splatting [vdLGS09, BSW10]. Even though foam is actually composed of the molecules of the underlying fluid, its characteristic appearance requires it to be handled using different rendering approaches, which will be explained in the next section.

1.2 Foam Simulation and Rendering

In computer graphics, foam generation techniques are used to enhance the realism of existing fluid simulations. High quality foam simulation and rendering techniques are commonly encountered in movies [GLR$^+$06, BSK$^+$07] and in commercial fluid simulation and visualization packages [hyb11]. In those works, however, the underlying foam generation and rendering stages are usually described briefly. Although foam is composed of fluid and air mixture, some of the existing research also focus on generating foam particles, usually in a scale smaller than the fluid particles to be able to

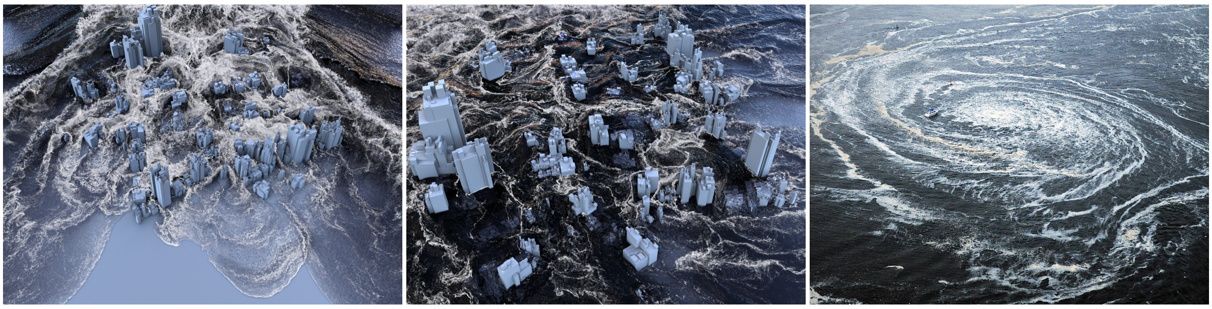


Figure 1: A flood scenario. Foam is rendered using our technique and composited with the rest of the scene (left and middle). Picture of real sea foam caused by a whirlpool (right) (©Reuters).

enhance the flow detail [TFK+03, GLR+06, LTKF08, MMS09, IAAT12].

For high quality foam renderings, ray-tracing methods are commonly preferred both for the fluid and the foam [GLR+06]. Although the fluid surface can be rendered efficiently using ray-tracing, non-homogenous phenomena such as foam require expensive volume rendering techniques. In [IAAT12], the authors employed a volume ray-casting method which accounts for absorption and emission of radiance but neglecting light scattering effects. In that method, each traced ray is sampled using equally spaced intervals; and according to the measured foam density at each sample point, the computed radiance is attenuated. The employed ray-casting approach, however, is time consuming to compute, especially for scenes with many millions of foam particles. The performance of volume ray-casting can be significantly improved by using the GPU-based method explained in [FAW10].

In [vdLGS09, BSW10], alternative to generating new particles, selected fluid particles are visualized as foam particles using GPU-based techniques for real-time applications. In [BSW10], Weber number thresholding is used to separate fluid and foam. Furthermore, the method also takes volumetric effects into account by rendering foam and fluid layers from back to front order. Therefore, it can visualize effects such as foam inside the fluid. Furthermore, based on the thickness of the foam, it generates foam color between two user defined colors. The approach, however, neglects information such as occlusion and irradiance from the environment when rendering foam, which limits its applicability to non-photorealistic real-time renderings.

There also exist methods for the modeling of larger scale foam effects by using air bubbles (e.g. see [KVG02, KLL+07, HLYK08, IBAT11, BDWR12]). In these works, air phase is either visualized by rendering spheres [KVG02, BDWR12], or by reconstructing the surface of the modeled air phase [KLL+07, HLYK08, IBAT11]. Since we are focusing on large scale scenarios, where the single air bubbles inside the foam are not clearly noticeable, such methods are beyond the scope of our paper.

1.3 Contribution

We present an efficient method for large scale foam rendering. In our approach, foam is rendered using a novel multi-pass rendering algorithm and finally composited with the pre-rendered images of the scene without foam. In comparison to volume ray-casting methods that compute only absorption and emission of radiance (e.g. [FAW10, IAAT12]), our approach is significantly faster as the foam particles are directly rendered. Furthermore, when compared to [BSW10], our pipeline takes the scene occlusion and lighting into account and therefore produces more convincing results that can be composited with realistic renderings. Results show that our new pipeline generates convincing large scale foam renderings (e.g. see Fig. 1) using modern GPU-based rendering architectures.

2 SCREEN SPACE FOAM RENDERING PIPELINE

As more air bubble layers implies more light scattering, we relate the foam thickness to the foam intensity as usually done in volume ray-casting. Later, we determine the regions on screen space which should receive, and therefore scatter less light using ambient occlusion and attenuate the foam intensity according to the occlusion factor. Afterwards, we approximate per-pixel foam irradiance to colorize the foam color according to the environment. Finally, the generated results are composited with the rest of the scene. We realized our approach using a seven pass rendering algorithm. The technical steps of our pipeline (illustrated in Fig. 2 and 3) can be summarized as:

- *PASS #1 and #2:* Storing eye space depth images of solid and fluid meshes in two textures, which are used to compute occlusion of foam fragments by those primitives in the later stages.
- *PASS #3:* Storing an eye space depth image of the foam particles in a texture, which is used in different parts of our pipeline. This pass also stores a search radius for each foam fragment, in whose range neighboring fragments are later considered for

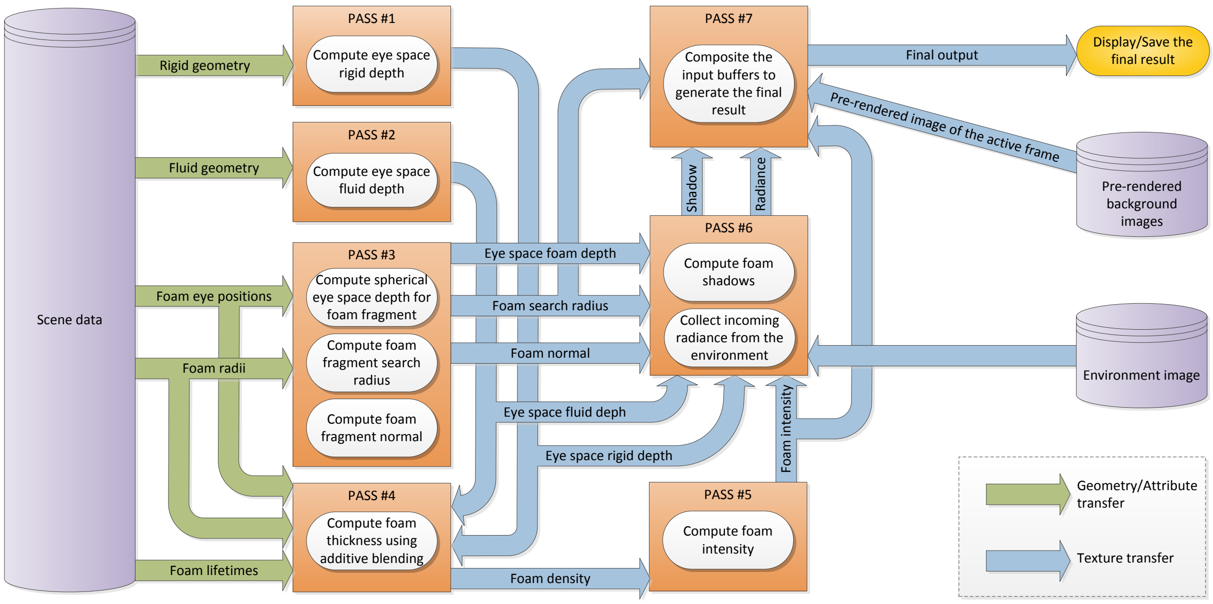
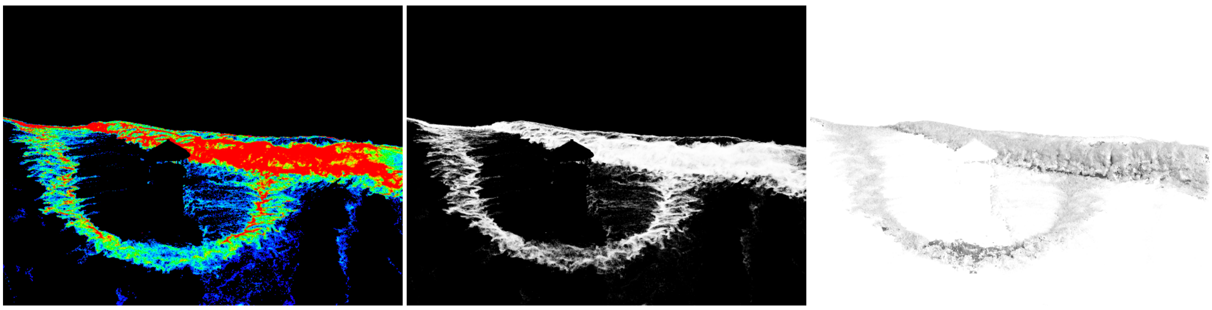
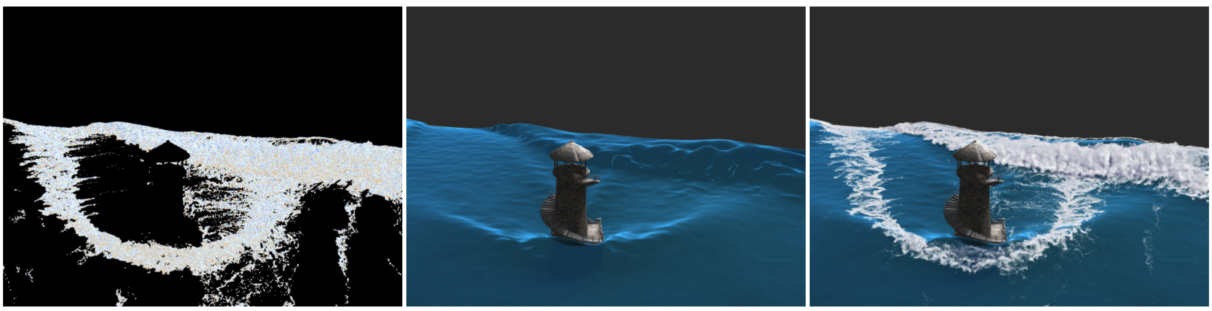


Figure 2: Diagram of our foam composition pipeline. Orange boxes denote the render passes and the arrows in between denote data flow and dependencies. For each frame, the render passes from #1 to #7 are executed. Each pass produces data explained in the enclosed rounded rectangles, which is then transferred through arrows to the subsequent passes. All of the generated textures have the same resolution as the final output.



(a) Foam thickness (red denotes thickest, blue denotes thinnest parts) (b) Foam intensity (c) Foam shadow (inverted)



(d) Foam irradiance (e) Pre-rendered image (f) Image composited with foam

Figure 3: Some of the intermediate textures from our foam composition pipeline (a-e) and the final composited result (f).

screen space ambient occlusion and final composition (Sec. 2.1). Additionally, this pass computes a normal for each foam fragment, which is used when approximating irradiance at the fragment location.

- *PASS #4:* Accumulating foam particles via additive blending to approximate per-pixel foam thickness. This pass also discards foam fragments that are occluded by solids and attenuates foam fragments that are inside of the fluid based on the fluid transparency (Sec. 2.2).
- *PASS #5:* Conversion of per-pixel foam thickness to per-pixel foam intensity (Sec. 2.3).
- *PASS #6:* Determination of foam fragments that should receive and scatter less light using screen space ambient occlusion (SSAO) and shadow generation for such regions (Sec. 2.4.1). This pass also approximates the irradiance at each foam fragment from an environment texture if the scene is illuminated using image based lighting (Sec. 2.4.2).
- *PASS #7:* Post processing of the foam and final composition with a pre-rendered image of the scene (Sec. 2.5).

Since the first step of the pipeline is relatively straightforward, we will focus on the remaining steps throughout this section. The following render passes are implemented using OpenGL Shading Language (GLSL).

2.1 Smoothed Depth and Search Radius Computation

We use point sprites instead of spheres for rendering foam particles. A regular point sprite has the same depth values for all of its fragments. However, to produce convincing results in the later steps of our pipeline, we modify the fragment depth values similar to [vdLGS09, BSW10], such that the spherical shapes of the particles are regained.

To create the initial depth information, foam particles with ids i and radii r_i in world space are rendered with depth testing and depth masking enabled. In [IAAT12], foam particles are separated to three different types, namely: spray, surface-foam and bubble particles. For bubble particles, we use half of r_i to make them less visible. Furthermore, particle radii are randomized as $r_i = \frac{r_i}{(i \bmod 5)+1}$ to make the particles look irregular between the scales $r_i/5$ and r_i.

The vertex shader computes eye space and projection space coordinates of the sprites and passes the resultant data to the fragment shader for further processing. In the fragment shader, the distance of the fragment position to the point sprite center is calculated using the sprite's texture coordinates to discard fragments that are outside of the circle. Afterwards, the flat depth values of the point sprite are transformed to spherical depth values. In this context, the first step is solving for the w coordinate of a unit sphere for the fragment's texture coordinates in uvw space as $w = \sqrt{1 - u^2 - v^2}$, where u and v denote texture coordinates of the fragment. Subsequently, the eye space z coordinate of the fragment is simply modified as

$$\mathbf{e}^{frag}_{foam_z} = \mathbf{e}^{frag}_{foam_z} + w \cdot r_i.$$

In contrast to [vdLGS09, BSW10], we do not apply filtering to the generated depth values since it would reduce the effect of ambient occlusion.

In the same render pass, the vertex shader also projects the search radius h_i for each particle as

$$h_i = \frac{r_i}{\tan\left(\frac{\alpha}{2}\right)\left|\mathbf{e}^{vert}_{foam_z}\right|},$$

where α is the field of view of the camera and $\mathbf{e}^{vert}_{foam_z}$ denotes z coordinate of the eye position of the point sprite (i.e., distance of the sprite to the camera). Afterwards, the search radius is passed to the fragment shader as h^{frag} to be written to a texture. The depth information and the search radius are essential when rendering the SSAO pass and when doing the final composition.

This pass also computes a world space normal for each fragment $\mathbf{n}_{frag}$ by transforming (u, v, w) using the transpose of the normal matrix, and stores the normals in a texture. Per fragment normals will be required when estimating irradiance in Sec. 2.4.2.

2.2 Thickness Estimation

Before estimating the intensity of foam at a given pixel position, we estimate the foam thickness for each pixel. In this step, foam particles are rendered again as point sprites with the spherical depth modification as in the previous render pass. Similar to [vdLGS09, BSW10], the foam fragments are blended additively to estimate thickness. Different from [vdLGS09, BSW10], however, depth buffer read and write is disabled as we do not require the frontmost particles to be visible.

As foam particles are separated to spray, surface-foam and bubble particles, we also employ this knowledge to render foam fragments differently by using a falloff function with different arguments, where the falloff is based on the fragment's distance to the particle center in texture coordinates. The falloff function f is defined as

$$f(x, b, n, m) = \begin{cases} \left[1 - \left(\frac{x}{b}\right)^n\right]^m & \frac{x}{b} \leq 1 \\ 0 & \text{otherwise} \end{cases}, \quad (1)$$

where x is the distance to the center, b is the maximum allowed distance, and $n \geq 0$ and $m \geq 0$ are exponents which determine the shape of the function (e.g.,

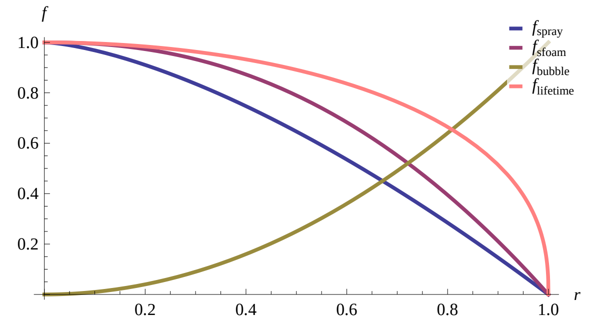


Figure 4: Different forms of the falloff function given in (1) that are used in our experiments .

$n = 1$ and $m = 1$ result in linear falloff). When rendering spray, surface-foam and bubble fragments, we used $f_{spray} = f(x, 1, 1.5, 1)$, $f_{sfoam} = f(x, 1, 2.25, 1)$ and $f_{bubble} = 1 - f(x, 1, 2, 1)$ respectively. These different falloff functions are illustrated in Fig. 4 and the corresponding intensity results are shown in Fig. 5. We preferred a larger overall intensity for surface foam particles to increase their visibility. Whereas, we preferred a comparatively smaller intensity value for the spray particles to make them relatively less visible. Furthermore, we used hollow circle like structures for the bubble particles to make their appearance more convincing under water.

In this step, the intensities of the foam particles are further modulated based on two additional factors. The first of these factors is the lifetime of the particle. For this purpose, we use $f_{lifetime} = f(l_i, 1, 2, 0.4)$, where $0 < l_i < 1$ denotes the normalized lifetime of a particle. Such a function allows a foam particle to remain visible for a sufficiently long time and fade smoothly near the end of its lifetime. Furthermore, when a particle lies in the back of the closest fluid surface (i.e. $0 < \mathbf{e}^{frag}_{fluid_z} < \mathbf{e}^{frag}_{foam_z}$, where $\mathbf{e}^{frag}_{fluid_z}$ is the eye space z coordinate of the fluid surface), we apply an additional falloff to its intensity, which is defined as

$$f_{att} = f(\mathbf{e}^{frag}_{foam_z} - \mathbf{e}^{frag}_{fluid_z}, \eta_{max}, \eta_n, \eta_m),$$

with the limiting distance η_{max}, where the foam fragment completely fades to invisible, and η_n and η_m are the exponents for shaping the attenuation curve.

At the end of this render pass, the final foam thickness values are stored in a texture (see Fig. 3a). In the next pass, the computed thickness values are processed and converted to normalized intensity values to lie between 0 and 1. For all subsequent passes, a screen-filling quad is rendered to further process the relevant information that are saved in the textures.

2.3 Intensity Estimation

As foam is composed of more bubble layers, it scatters more of the incoming light. We use this knowledge to relate the foam intensity proportional to foam

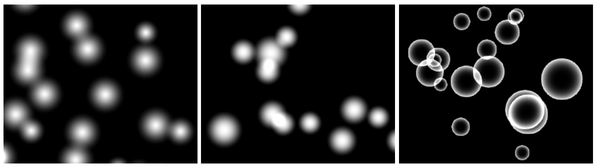


Figure 5: Intensity distributions of different types of foam particles, namely: spray particles (left), surface foam particles (middle) and air bubble particles (right).

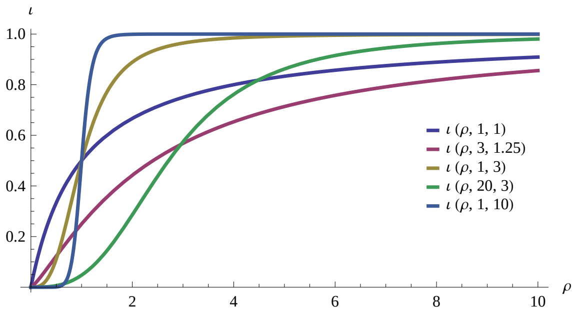


Figure 6: Different forms of the sigmoid function that can be applied to the accumulated foam densities. The function can be used to create different distributions as well. For instance, to reduce the intensities below some threshold, $\rho_{exp} \geq 2$, can be used. We use the $\iota(\rho, 3, 1.25)$ form in our experiments.

thickness. A texel from the previous render pass may have any value between $[0, \infty)$. In this render pass, we scale the values taken from that texture to the interval $[0, 1]$. However, scaling the values linearly to the target interval would make sparse areas invisible. We expect the foam to become completely opaque after some thickness threshold. Therefore, to increase the effective range of the thinner regions, to reduce the range of thicker regions and to normalize the intensities, we define the following sigmoid function ι to non-linearly scale a pixel thickness value ρ as

$$\iota(\rho, \rho_{mod}, \rho_{exp}) = \frac{\rho^{\rho_{exp}}}{\rho_{mod} + \rho^{\rho_{exp}}},$$

where $\rho_{mod} > 0$ and $\rho_{exp} > 0$ control how fast the function grows. Note that if $\rho > 0$ and $\rho_{exp} > 0$, $0 < \iota < 1$. ι is illustrated in Fig. 6 for different parameters. Furthermore, Fig. 7-top shows the effect of using different ρ_{mod} values.

At the end of this step, the normalized intensities are saved in a texture, which will be used in the following steps (see Fig. 3b).

2.4 Foam Radiance Estimation

Since foam is composed of many transparent layers of air bubbles, light can travel through it and then scatter. Until the current stage of our pipeline, we assume that foam scatters light uniformly, where the intensity

Figure 7: Application of different parameters for the setting presented in Fig. 1-middle. top-left: $\rho_{mod} = 1$; top-right: $\rho_{mod} = 5$; bottom-left: $AO_{ShScale}$ =0.1; bottom-right: $AO_{ShScale} = 2$.

of the light was only related to the foam thickness. In this section, we determine the regions which should receive, and therefore scatter less light using ambient occlusion (AO), and generate shadows for these regions (Sec. 2.4.1). Furthermore, the intensities that are computed in the previous section do not employ any knowledge about the actual illumination that comes from the scene. In this render pass, we will also use a very rough screen space approximation of the irradiance from the surrounding environment, which is used to colorize the foam fragments (Sec. 2.4.2).

This render pass again gets the textures that have been computed in the previous step as input and computes two additional textures, one for the shadow and another for the illumination of the foam (see Fig. 3).

2.4.1 Shadow Generation

As object space AO methods (e.g. [ZIK98, Bun05, RWS+06]) are very expensive to compute, especially for complex dynamical phenomena such as foam, we investigated SSAO techniques [TCM06, Mit07, SA07, RGS09, BS09, HL10]. Finally, we decided to build our SSAO approach upon the basic concept explained in [Mit07] because of its efficiency and simplicity. One important difference of our method in comparison to [Mit07] is that we apply multiple sample collection iterations to capture both small scale and large scale occlusions. Instead of increasing search radii, [HL10] used multiple depth maps with decreasing resolution to achieve the same effect. The search radii and total number of passes are controlled by three parameters: the initial search radius factor $AO_{InitSRFac}$, which is a factor for h^{frag} to capture small scale occlusions; the search radius increment factor $AO_{SRIncFac}$, which is another factor for h^{frag} to determine how much the search radius increases in each sample collection step; and finally $AO_{\#Passes}$, which limits the total number of SSAO passes. For each fragment, 3d samples are generated within the fragment search radius:

$$h^{frag}_{pass} = h^{frag}\left(AO_{InitSRFac} + AO_{SRIncFac} \cdot AO_{Pass}\right),$$

where AO_{pass} increases by 1 in each sample collection pass and $AO_{pass} \leq AO_{\#Passes}$. In our experiments we used: $AO_{InitSRFac} = 1$, $AO_{SRIncFac} = 7$ and $AO_{\#Passes} = 3$.

The total number of samples ν in each sample collection pass is controlled by a user defined sampling density parameter AO_{SDens} as

$$\nu \;=\; \mathrm{clamp}\left(\frac{3}{4}\pi {h^{screen}_{pass}}^3 AO_{SDens}, AO_{\#MinSamp}, AO_{\#MaxSamp}\right),$$

where h^{screen}_{pass} is the search radius projected to fragment coordinates. Since h^{frag} can be very small for distant fragments, a minimum value $AO_{\#MinSamp}$ is used for ν. An upper limit $AO_{\#MaxSamp}$ is also introduced to prevent too many samples from being generated for fragments that are very close to the viewer. In our experiments, we used $AO_{SDensity} = 0.5$, $AO_{\#MinSamp} = 16$ and $AO_{\#MaxSamp} = 512$. The samples are created inside a cube in the range $[-1,1]$ on all axes using the Halton sampling algorithm with a constant seed [Hal64], which produces low-discrepancy sequences. Subsequently, the samples are mapped to a sphere by simply neglecting the samples that lie outside of the sphere in the range $[-1,1]$.

Additionally, the occlusion contribution λ of a sample $\mathbf{s}$ depends on its distance to the fragment and we compute it using a quadratic falloff as

$$\lambda = (1 - |\mathbf{s}|)^2.$$

Furthermore, if a sample is occluded by a fragment with a distance larger than the user defined $AO_{MaxOcclDist}$, the sample does not contribute to the visibility of the fragment. This effect is necessary to prevent occlusion by distant fragments and is controlled using a quadratic falloff function as

$$\delta = \max\left[\left(1 - \frac{\left|\mathbf{e}^{frag}_{foam_z} - s_z\right|}{AO_{MaxOcclDist}}\right), 0\right]^2,$$

where $AO_{MaxOcclDist} = 5$ is used in our experiments. The sample $\mathbf{s}$ is used to look up the occlusion in eye space by other fragments (e.g. foam, fluid and solid fragments) in the scene. Based on the knowledge collected so far, the occlusion k of a sample is defined as

$$k = \begin{cases} 1 & \left[\left(s_z > \mathbf{e}^{frag}_{foam_z} \vee s_z > \mathbf{e}^{frag}_{fluid_z} \vee s_z > \mathbf{e}^{frag}_{rigid_z}\right) \wedge (0 < \delta < 1)\right] \\ 0 & \text{otherwise} \end{cases},$$

which basically states that; if a sample is occluded by any other fragment in the scene and if the occlusion distance is not larger than $AO_{MaxOcclDist}$, the sample is occluded.

Afterwards, we compute the occlusion factor ω of a fragment as

$$\omega = \frac{\sum_{AO_{pass}=1}^{AO_{\#Passes}} \left(\sum_{i=1}^{v} \lambda_i \cdot \delta_i \cdot k_i \cdot a_i\right)}{\sum_{AO_{pass}=1}^{AO_{\#Passes}} \left(\sum_{i=1}^{v} \lambda_i\right)},$$

where for the pass AO_{pass}, i iterates over all generated samples which are inside the render area, and a_i is the transparency of the sampled fragment, which is equivalent to ι_i for foam fragments. For rigid and fluid fragments, a_i is equivalent to the fragment's transparency. Additionally, if there are multiple overlapping transparent fragments at a sample position, a_i is computed by adding all of the transparency values.

Finally, so as to be more flexible about the appearance of the generated shadows, we compute the final shadow value ζ clamped into $[0,1]$ as

$$\zeta = \text{clamp}\left[(\omega \cdot AO_{ShScale})^{AO_{ShExp}} + AO_{ShOffset},\, 0,\, 1\right]$$

which is controlled by three self-explanatory user defined parameters. In the presented scenarios, we used: $AO_{ShScale} = 1$, $AO_{ShExp} = 1.5$ and $AO_{ShOffset} = -0.05$. The ambient occlusion step especially improves the regions that have similar intensities, which would look totally flat otherwise (e.g., see Fig. 8, top-middle). Furthermore, Fig. 7-bottom shows the effect of different $AO_{ShScale}$ values. The computed ζ values are written to a texture to be further used by the final composition step (see Fig. 3c).

2.4.2 Irradiance

If the scene is illuminated using image based lighting, we approximate the direct illumination of each foam fragment by looking up the environment map that has been used as the light source. Using the fragment normal $\mathbf{n}^{frag}$, we create a hemisphere around the normal and use the already generated samples from the SSAO step to create direction vectors $\mathbf{n}_i^{sample}$ that are used for looking up the intensity $\mathbf{P} = (r, g, b)$ at an environment map position. Finally, the irradiance that is coming from the environment to a fragment location is simply computed in a cosine weighted fashion as

$$\mathbf{I} = \left(\frac{\sum_{i=1}^{v} \mathbf{P} \cdot \left(\mathbf{n}_i^{sample} \cdot \mathbf{n}^{frag}\right)}{\sum_{i=1}^{v} i}\right),$$

where i iterates only over the samples that are generated for the first sample collection pass. The sole purpose of this step is to reflect the hue of the environment onto the foam fragments to make the foam not look too distinct from the rest of the scene. Finally, the computed $\mathbf{I}$ values are written to another texture to be used by the next and the final render pass (see Fig. 3d). The performance of this step can be improved by using an irradiance environment map and making color look-up once for every $\mathbf{n}^{frag}$.

2.5 Composition

In this render pass, the information that has been created in the previous steps and the pre-rendered images of the scene without foam are composited to generate a final image of the scene with foam (see Fig. 2).

Depending on the user defined $AO_{\#MaxSamples}$, the shadow and radiance values computed in the previous section can include high frequency noise. In order to alleviate this problem, we apply Gaussian blur with a filter radius of $\frac{3}{2}h_{pass}^{screen}$ to both textures to generate per-pixel $\zeta_{filtered}$ and $\mathbf{I}_{filtered}$ before doing the composition.

Afterwards, to compute a final shadow color ζ_{final} for a pixel, the filtered shadow values are modulated with a user defined color $\mathbf{C}_{ShadowColor}$ and clamped into $[0,1]$ as

$$\zeta_{final} = \text{clamp}\left[(\mathbf{c}_{white} - \mathbf{C}_{ShadowColor}) \odot \zeta_{filtered},\, 0,\, 1\right],$$

where $\mathbf{c}_{white} = (1, 1, 1)$, and $\odot$ denotes component-wise multiplication. We select $\mathbf{C}_{ShadowColor}$ similar to the visible color of the fluid that the foam is generated on, and it was chosen in our experiments as $(0, 0, 0.2)$ because of the dark blue appearance of the fluids in our renderings. Since ζ_{final} will be subtracted when doing the composition, the $\mathbf{C}_{ShadowColor}$ term is subtracted from white to invert it. From our experiences, colorizing shadows makes the foam blend better with the underlying fluid.

As foam is composed of many air-liquid interfaces, it has a very large scattering albedo which causes it to scatter most of the incoming light, but absorb only a small amount of it. Therefore, it is usually observed very bright. We control this phenomenon by linearly scaling the irradiance values $\mathbf{I}$ using a user defined parameter $C_{IrrScale}$, whose value depends on the desired foam brightness and the color range of the environment map used. Afterwards, we clamp the resulting color into the $[0,1]$ interval to compute

$$\mathbf{I}_{final} = \text{clamp}\left(C_{IrrScale} \cdot \mathbf{I}_{filtered},\, 0,\, 1\right).$$

Finally, the composited pixel color c is computed as

$$\mathbf{c} \;=\; (1 - \iota)\,\mathbf{c}_{bg} + \iota\left(\mathbf{I}_{final} - \zeta_{final}\right)$$

where $\mathbf{c}_{bg}$ is the color at the corresponding pixel position of the background image on which the foam is composited (see Fig. 3f).

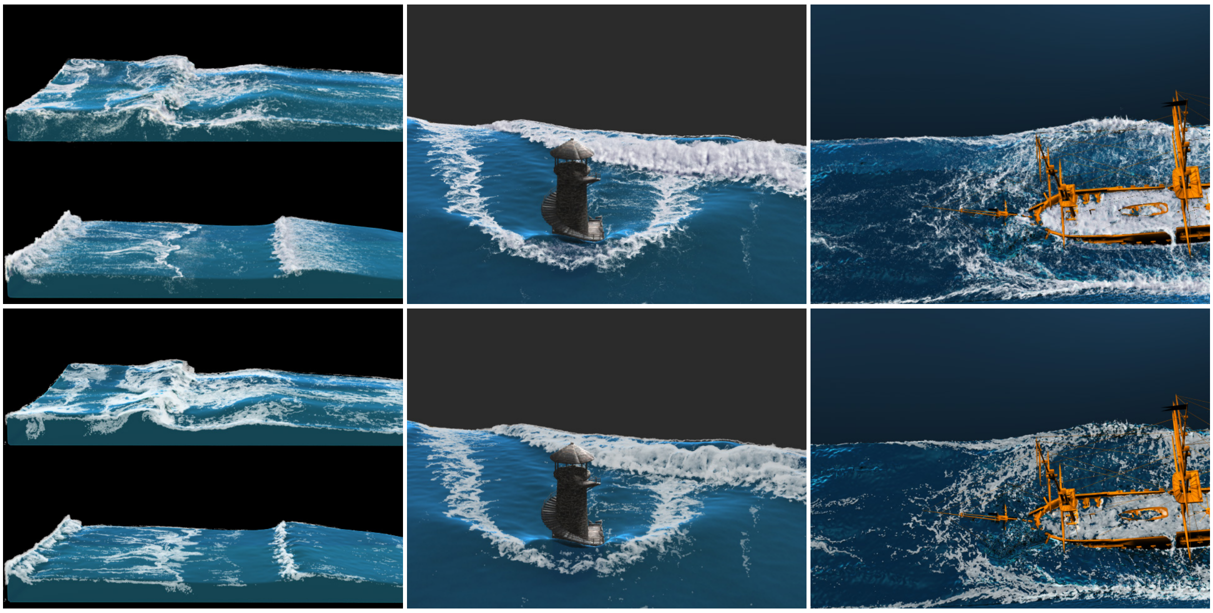


Figure 8: Comparisons of our method (top) to volume ray-casting that computes emission and absorption only (bottom). As our method approximates shadows in concave regions, the foam looks more volumetric and detailed. The scenes are named from left to right as: Wave, Lighthouse and Ship.

	# Foam particles	Resolution	Average foam rendering time per frame			
			Ray-casting [IAAT12]	Ray-casting [FAW10]	Ours (intensity only)	Ours (total)
Wave	up to 820K	800×600	2 min 10 s	235 ms	8 ms	52 ms
Ship	up to 9M	800×600	4 min 20 s	760 ms	16 ms	102 ms
Lighthouse	up to 15M	800×600	7 min 3 s	1 s	21 ms	150 ms
Flood	up to 29M	1280×960	16 min 19 s	1.7 s	33 ms	235 ms

Table 1: Performance analysis of the example scenes.

3 RESULTS

In this section, we demonstrate the versatility of our approach in different animation sequences. For all presented scenes, the underlying fluid has been simulated using the methods referred in [IAAT12], and the fluid surfaces have been reconstructed based on [SSP07, AIAT12, AAIT12]. The scenes were rendered using mental ray [NVI11] on an Intel Xeon X5690 CPU with 12 GB RAM, and the foam composition pipeline was implemented using GLSL and ran on an NVIDIA 480 GTX GPU with 1.5 GB RAM. The ray-casting code used in [IAAT12] was implemented as a mental ray shader and ran on the CPU, and an optimized version based on [FAW10] was implemented on the GPU. All scenes were illuminated using image based lighting with a clear sky environment map.

For all scenes, foam was simulated using [IAAT12] and the same foam data were used for the rendering comparisons to [IAAT12]. For the comparisons shown in Fig. 8, the ray-casting technique explained in [IAAT12] took 9 s to 20 min depending on the complexity of the frame, excluding the other scene geometry and loading of the foam data. Using the optimized volume ray-casting scheme, the computation time has been reduced down to 90 ms to 2.5 s. Using our pipeline, the foam rendering of a frame took 30 ms to 270 ms depending on the complexity of the foam in the scene being rendered, excluding the time spent for loading of the foam data from secondary storage to the GPU memory. The results produced by using a basic volume ray-casting scheme that only accounts for absorption and emission of radiance is similar to the results we achieve excluding the additional effects that are described in Sec. 2.4 (see also Fig. 3b). Excluding those additional effects, our pipeline took between 5 ms to 39 ms per frame. See Table 1 for additional information about each scene. As our pipeline also takes additional effects into account (i.e. ambient occlusion and irradiance estimation), our presented foam renderings look volumetric and blend with the rest of the scene (see Fig. 8). Note that in [IAAT12], the fluid surface has been constructed only for the fluid particles that have more than five neighbors. For our comparisons to [IAAT12], however, we used the whole fluid surface for our renderings to better estimate the SSAO of the foam by the fluid surface. Therefore, differences between the two fluid surfaces can be noticeable.

For all of our scenes, most of the rendering time has been spent on the foam radiance estimation pass (between 50-80%). Whereas, the computational overheads of the rest of the render passes were significantly lower.

4 DISCUSSION AND FUTURE WORK

Taking a closer look at sea foam from a distance less than a few meters, one may observe the underlying air bubbles at varying sizes which form the foam. Rendering of such scenarios is not handled by our approach. However, using an air bubble generation technique like [BDWR12] for such close-ups might be an interesting direction for future research.

For scenes where most of the light is coming from a specific direction at shallow angles (e.g. sunset scenarios), the currently employed SSAO based shadow generation technique can fail to capture the resultant potentially large shadows cast by distant objects. For such cases, an explicit shadow generation algorithm which can handle image based lighting such as the one explained in [CK09], or explicit shadow source selection as discussed in [Bjo04] can be employed. Since we assume that foam scatters most of the incident light randomly, we omitted Fresnel effect. However, it might be desirable to make the foam reflect the environment, when it is viewed from a shallow angle.

Our algorithm neglects many physical effects that could be otherwise simulated by using modern ray-tracing techniques. Those effects include; scattering of light inside the foam, influence of the foam on the appearance of the surrounding objects and vice versa. However, for large scale scenarios (e.g. as in Fig 1), those effects have less significance on the appearance of the foam, and our approximations can efficiently generate convincing results. However, for close-ups, the effects that we have omitted have more significance on the final outcome. For those cases, using a volume ray-casting method that simulates light scattering can definitely yield more convincing results (e.g. [RNGF03, GLR+06]).

Although we demonstrated our rendering scheme only for the particle data generated by the method explained in [IAAT12], we believe that our pipeline is mostly applicable to the rendering of other particle based foam simulation techniques.

5 CONCLUSION

We presented an efficient, screen-space foam rendering pipeline that can render large particle-based foam data sets on the GPU. Our approach uses a multi-pass rendering scheme, where different effects are added to the foam rendering incrementally, and the final foam rendering is composited with a pre-rendered image of the scene. The presented method can be used as an efficient alternative to ray-casting techniques for the rendering of large scale particle-based foam data.

6 REFERENCES

[AAIT12] G. Akinci, N. Akinci, M. Ihmsen, and M. Teschner. An efficient surface reconstruction pipeline for particle-based fluids. In *Workshop on Virtual Reality Interaction and Physical Simulation*, pages 61–68. The Eurographics Association, 2012.

[AIAT12] Gizem Akinci, Markus Ihmsen, Nadir Akinci, and Matthias Teschner. Parallel surface reconstruction for particle-based fluids. *Computer Graphics Forum*, 31:1797–1809, 2012.

[BDWR12] O. Busaryev, T.K. Dey, H. Wang, and Z. Ren. Animating bubble interactions in a liquid foam. *ACM Transactions on Graphics (TOG)*, 31(4):63, 2012.

[Bjo04] K. Bjorke. Image-based lighting, 2004.

[BS09] L. Bavoil and M. Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, page 45. ACM, 2009.

[BSK+07] R. Bredow, D. Schaub, D. Kramer, M. Hausman, D. Dimian, and R.S. Duguid. Surf's up: the making of an animated documentary. In *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2007 courses: San Diego, California*, volume 5, 2007.

[BSW10] F. Bagar, D. Scherzer, and M. Wimmer. A layered particle-based fluid model for real-time rendering of water. In *Computer Graphics Forum*, volume 29, pages 1383–1389. Wiley Online Library, 2010.

[Bun05] M. Bunnell. Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233, 2005.

[CK09] Mark Colbert and Jaroslav Krivanek. Real-time dynamic shadows for image-based lighting. *ShaderX 7 - Advanced Rendering Techniques*, page Section 4.3, 2009.

[CM10] N. Chentanez and M. Müller. Real-time simulation of large bodies of water with small scale details. In *Proc. of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 197–206, 2010.

[FAW10] R. Fraedrich, S. Auer, and R. Westermann. Efficient high-quality volume rendering of sph data. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1533–1540, 2010.

[GLR+06] W. Geiger, M. Leo, N. Rasmussen, F. Losasso, and R. Fedkiw. So real it'll make you wet. In *ACM SIGGRAPH*, 2006.

[Hal64] J.H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964.

[HL10] T.D. Hoang and K.L. Low. Multi-resolution screen-space ambient occlusion. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, pages 101–102. ACM, 2010.

[HLYK08] Jeong-Mo Hong, Ho-Young Lee, Jong-Chul Yoon, and Chang-Hun Kim. Bubbles alive. *ACM Trans. Graph.*, 27(3):48:1–48:4, August 2008.

[hyb11] Next Limit Technologies: Realflow 2012, Hybrido. White Paper. 2011.

[IAAT12] Markus Ihmsen, Nadir Akinci, Gizem Akinci, and Matthias Teschner. Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer*, pages 1–9, 2012. 10.1007/s00371-012-0697-9.

[IBAT11] M. Ihmsen, J. Bader, G. Akinci, and M. Teschner. Animation of air bubbles with sph. In *International Conference on Graphics Theory and Application*, pages 225–234, 2011.

[KLL+07] Byungmoon Kim, Yingjie Liu, Ignacio Llamas, Xiangmin Jiao, and Jarek Rossignac. Simulation of bubbles in foam with the volume control method. *ACM Trans. Graph.*, 26(3), July 2007.

[KVG02] Hendrik Kück, Christian Vogelgsang, and Günther Greiner. Simulation and rendering of liquid foams. In *In Proc. Graphics Interface '02*, pages 81–88, 2002.

[LTKF08] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-Way Coupled SPH and Particle Level Set Fluid Simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[Mit07] M. Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007.

[MMS09] V. Mihalef, D. Metaxas, and M. Sussman. Simulation of two-phase flow with sub-scale droplet and bubble effects. In *Computer Graphics Forum*, volume 28, pages 229–238. Wiley Online Library, 2009.

[MSD07] M. Müller, S. Schirm, and S. Duthaler. Screen Space Meshes. In *Proceedings of ACM SIGGRAPH / EUROGRAPHICS Symposium on Computer Animation (SCA)*, 2007.

[NVI11] NVIDIA ARC. mental ray 3.9 [software]. http://www.mentalimages.com/products/mental-ray/about-mental-ray.html, 2011.

[RGS09] T. Ritschel, T. Grosch, and H.P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82. ACM, 2009.

[RNGF03] N. Rasmussen, D.Q. Nguyen, W. Geiger, and R. Fedkiw. Smoke simulation for large scale phenomena. *ACM Transactions on Graphics (TOG)*, 22(3):703–707, 2003.

[RWS+06] Z. Ren, R. Wang, J. Snyder, K. Zhou, X. Liu, B. Sun, P.P. Sloan, H. Bao, Q. Peng, and B. Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 977–986. ACM, 2006.

[SA07] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 73–80. ACM, 2007.

[SSP07] B. Solenthaler, J. Schläfli, and R. Pajarola. A unified particle model for fluid-solid interactions. *Computer Animation and Virtual Worlds*, 18(1):69–82, 2007.

[TCM06] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1237–1244, 2006.

[TFK+03] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki. Realistic Animation of Fluid with Splash and Foam. *Computer Graphics Forum*, 22(3):391–400, 2003.

[vdLGS09] W.J. van der Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98. ACM, 2009.

[YT10] J. Yu and G. Turk. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 217–225. Eurographics Association, 2010.

[ZB05] Y. Zhu and R. Bridson. Animating sand as a fluid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 965–972, New York, NY, USA, 2005. ACM Press.

[ZIK98] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. *Rendering techniques*, 98:45–55, 1998.