Real-Time Screen Space Cartoon Water Rendering
with the Iterative Separated Bilateral Filter

Liordino dos S. Rocha Neto, Antonio L. Apolinário Jr

Department of Computer Science
Federal University of Bahia
Salvador, Brazil

Email: liordino.neto@ufba.br, antonio.apolinario@ufba.br

Figure 1. Example of the results achieved with our method for a waterfall scene. Notice the presence of optical effects, foam and droplets, as well as the absence of striped artifacts common to the separated bilateral filter smoothing results.

Abstract—We present the improvements and new results of our method for rendering particle-based liquid simulations that runs in real-time and includes an adjustable performance/quality trade-off. Our approach smooths the fluid surface with an iterative version of the separated bilateral filter, and introduces a new method for generating foam and droplets that is appropriate for non-photorealistic styles. The entire method occurs in screen space, which avoids the usual artifacts of polygonization techniques. All the steps are implemented directly in the graphics hardware. Improvements include a new method to generate foam and droplets, visual enhancements in the optical effects generated by the interaction between water and light, and tests in an environment with support to collision of water particles with rigid bodies. Performance and visual analysis that includes comparisons with previous methods shows the applicability of our approach.

Keywords—non-photorealistic rendering; screen space rendering; smoothed particle hydrodynamics simulation;

I. INTRODUCTION

Rendering physically simulated fluids has been a topic of much research nowadays, thanks to the great demand for realistic simulations in computer graphics [1]. Although this is true for environments like movies, it represents a greater challenge in interactive real-time applications such as games, where the dynamic behavior of fluids can be a desirable gameplay element [2]. Three basic steps must be performed to get a graphical representation from physical simulations like these: simulating the fluid, extracting a renderable representation of it and finally performing the rendering itself.

There are two main broad categories that fluid simulations can be divided into: eulerian (grid-based) and Lagrangian (particle-based) [3]. While grid-based approaches have the advantage of a high-quality surface straightforward extraction, they are usually more costly than particle-based ones in both memory and computation while also resulting in limited surface detail when the grid have a low resolution, what makes the latter preferred in real-time interactive environments [4]. Other grid-based works address the limited surface detail in low resolution grids [47] or aims to simulate only the fluids surface, bypassing the simulation of it’s entire volume [46], but they still don’t achieve real-time performance, and are therefore beyond the scope of our current work.

Several approaches have been proposed recently to extract and render smooth surfaces from particle-based fluids, but the majority of these methods aim only at photorealistic renditions [4] [5] [6]. The surface extraction and smoothing can be done either through object-space polygonization methods like Marching Cubes or screen-space approaches. Due to the computation and memory intensive behavior of the former, screen-space methods are preferred in real-time environments like games [4]. Methods to render non-photorealistic water were recently developed, but they are
either not meant for real-time applications [7] [8] or don’t use fluid simulation [9].

The interaction of simulated liquids with rigid bodies is another factor that brings new challenges on how to handle it from a visual point of view, generating effects like foam and spray. Although techniques to do this specifically for cartoon water were developed recently, they either rely on pre-loaded textures and procedural methods instead of on the simulation itself [11] [12] or leave room for visual improvement [38].

Our method’s rendering style is inspired by modern cartoon animations like One Piece1 (Fig. 2(a)) and Avatar: The Last Airbender2 (Fig. 2(b)), and further enhances the visual style observed in these examples with optical effects. The rendering process uses as input a set of particles that represents the fluid. The Smoothed Particle Hydrodynamics (SPH) method is used to simulate the behavior of the fluid. Historically, it is first used in astrophysical simulations [42] [43], and later adapted for computer graphics [44] [20] and interactive applications [13]. SPH method is based on kernel functions defined on each particle, that can be interpolated to reconstruct scalar and vector fields, such as pressure and velocity. The readers who are interested in more details about the fluid simulation using the SPH method, we recommend reading the survey [48] or these two books [49], [50]. We present an approach to render a cartoon-style fluid simulation that extends our previous work [38] with the following contributions:

- A simpler and improved technique to generate foam and spray suitable for non-photorealistic rendering;
- Visual improvements on the effects of the interaction between light and water;
- Tests in an environment that approximates a commercial game, providing an optimized simulation and collision with rigid bodies;
- Visual and performance comparisons with previous surface smoothing methods, like the Curvature Flow [4].

This paper is structured as follows: section II presents related works, highlighting their contribution to our method, which section III describes. Section IV explains the conducted experiments and discuss their results. Finally, section V presents our conclusions, along with suggestions for future works.

II. RELATED WORK

SPH as means to simulate water in computer animation was first introduced by Müller et al. [13], where it was successfully used in interactive real-time applications. Improvements over the initial approach were developed, first being fully implemented in CPU [14] [15], and later in GPU [16] [40]. More recent works successfully achieves the simulation of highly viscous fluids by manipulating an SPH particle-set velocity field, but not in real-time [45]. Our focus on this work is on rendering SPH-based water simulations with a traditional cartoon style, and it is assumed that an SPH simulation is already carried out. The following subsections list works related to ours in different categories.

A. Surface Extraction and Smoothing

Müller’s work, while introducing the use of SPH for real-time fluid simulations, also successfully performed its surface extraction with both marching cubes [18] and surface splatting (also called point splatting) [19] methods. While marching cubes extracts a simulation isosurface, the point splatting goal is to render surfaces from point clouds without any connectivity. Van der Laan et al. [4] proposes a point splatting based screen space technique that renders particles as screen oriented point sprites through several steps performed in the graphics hardware, reducing geometric computations. The same framework was used in our previous work [38] to render cartoon water.

Many fluid rendering methods employee smoothing algorithms to prevent a bloopy appearance when rendering surfaces extracted from particle-based simulations. An iterative curvature flow [20] is used in Van der Laan’s work [4]. This technique consists of repeatedly shifting a surface along its normal vector depending on its mean curvature, and was also used in several more recent works to render particle based or hybrid fluid simulations [39] [40]. An adaptive version of this filter was used in the work by Bagar et al. [6], varying the number of iterations to produce a consistent fluid surface independent of the viewer distance. Green’s work [21] shows the same workflow, but instead of curvature flow it uses a separable bilateral filter [22] to smooth the fluid surface. It greatly improves performance over the full kernel bilateral filter [23], but generates striped artifacts over the smoothed surface. Prior shading steps hide these artifacts in photorealistic renderings [21], but that’s not enough if an NPR style is chosen [37] [38]. Gastal and Oliveira [24] shows an approach to make these artifacts virtually unnoticeable by iteratively applying a separated filter reducing it’s kernel at each iteration. We applied this idea to the separated bilateral filter in our previous work [38], successfully removing striped artifacts with a low performance impact.

B. NPR Rendering

Both Selle et al. [25] and McGuire and Fein [26] present methods to create cartoon style renderings with information gathered from particle-based fluid simulations, being the later in real-time. Unfortunately these approaches are used to render smoke, what makes them not ideal for our purposes, since smoke does not have a free surface that makes its interface with air, like liquids [7]. Winemoller introduced the XDoG operator [27], an extended difference of Gaussians

---

1©Toei Animation, http://www.toei-anim.co.jp/tv/onep/
2©Nickelodeon Animation Studios, http://www.nick.co.uk/shows/avatar/
filter that produces smooth edges and stylized images. Other approaches for image stylization are the anisotropic Kuwahara [28] and coherence-enhancing filters [29]. Images can also be stylized through bilateral filter’s iterative application [30]. Although these filters can successfully create NPR style renderings, they show visual results that differ from our traditional cartoon approach.

Contributions by Eden et al. [7] and Yu et al. [9] propose cartoon water rendering techniques, but they ignore water’s optical characteristics. You et al. [8] takes into account optical properties like transparency, reflection and refraction, but don’t achieve real-time performance. They used a surface generated by a physically based fluid simulation as input, combining several shading steps to compose its result. An automatic control of reflection and refraction is also introduced. We applied it to our previous method [38], which we extend here.

C. Spray and Foam Generation

Foam rendering has been the object of study of some recent works. Yingst et al. method [11] emulates sea foam dissipation with a precomputed dither array, on a method suitable for real-time environments, but produces pixelation as the camera approaches the liquid surface. Liao et al. [12] presents another method, consisting of procedural modeling of water caustics and foam employing a texture generated with Voronoi diagrams. Both methods don’t need a physical simulation. Zhang et al. work [31] uses the Weber number to simulate foam and sprays generated by particle-based fluid motion. This method calculates each particle Weber number and compare it to critical thresholds that divide them into water, transition and foam. Each group is them rendered in a different way to compose the result, also achieving real-time performance. Bagar’s work [6] also rely on Weber number to generate particle-based foam. Our previous work [38] separates particles with thresholds over their densities, but the transition between these states is not smooth and cause flickering. In this extension we make use of a simple interpolation to prevent flickering and blend foam and droplet with water, further smoothing this transition.

III. TECHNIQUE OVERVIEW

We build our technique on top of the cartoon water rendering presented in our previous work [38]. We assume that an SPH simulation is already carried out, and use it’s particles positions as input. Fig. 3 shows a step by step overview of the technique. The process starts by using a point splatting technique to get the frontmost surface depth of both water and foam/depth (section III-A) and also the thickness (section III-B), storing each one in a separated offscreen buffer. Next, we perform a smoothing step on the fluid depth map to prevent a blobby aspect on its surface. Previous works [21] [37] [38] used three versions of the bilateral filter for smoothing the depth map: full kernel, separated, and iterative separated with a reduced kernel for each iteration. In this work we used the third bilateral filter version, which removes the artifacts created by it’s non-iterative separated version while having a low performance impact (section III-C). Finally, we merge these intermediate steps with a texture of the scene without the fluid to generate the final composition (section III-D). A simple stencil operation is executed between the fluid and background original depth maps to perform the occlusion of the scene over the water.

A. Surface Depth Extraction

The first step that must be done to render the water is to determine it’s frontmost surface as seem from the viewpoint of the camera. To achieve this, we render the SPH particles as spheres using screen oriented quads (point sprites), by
Fig. 3. Overview of the intermediate steps of our method. From the SPH particles cloud we obtain three textures: water’s depth map, droplet/foam depth map, and thickness. The water depth map is smoothed with the iterative separated bilateral filter to prevent a blobby aspect. Finally, these intermediate textures are blended with the background scene texture to generate the final compositing. The depth maps were enhanced digitally for better visualization.
discarding the pixels outside the circle inscribed in the quad. Its closest values for each pixel are obtained through depth test and stored in a texture using a depth replacement technique in the fragment shader. The surface normals are calculated from the depth values while rendering.

This step generates a grayscale texture containing all the particles, but a threshold value over their density \( T_d \) separates water from foam and droplet. This way, particles that have a density higher than the \( T_d \) become water, while the ones with a density below \( T_d \) become foam or droplets. To keep the particles close to each other and prevent the occurrence of undesirable holes in the final water rendering, each particle size is computed using the distance between the viewer and the fluid surface. Foam and droplet particles also have their size affected by a simple interpolation over 0 and \( T_d \) and have their radius further reduced. By the end of this process, the water depth texture contains all particles, but while all the water particles have the same size, foam and droplet particle sizes vary with their density. The size interpolation over the particle densities have also the effect of treating stray particles as droplets, rendering them with smaller sizes or removing them from the scene and thus avoiding artifacts caused by their presence in the smoothing process. We repeat this process to generate another texture that contains only foam and droplet particles. In this second texture they are rendered with the same size, but the same linear interpolation is used to vary their alpha value with the density. The combination of these two textures make the transition between particle states smoother, avoiding the sudden state changes that cause flickering.

B. Fluid Thickness

An object underwater should become less visible as the amount of fluid that is in front of it grows. To simulate this behaviour we need to compute the amount of fluid between the viewer and the nearest opaque object underwater for each pixel. Like van der Laan [4], we call this the thickness attribute, and use it to attenuate the color and the transparency of the fluid.

Here the rendering process is similar to the one described in Section III-A, but instead of the depth value the fragment shader outputs the thickness of the particle at that position. This value is computed by rendering a 2D Gaussian distribution over each point sprite, turning on additive blending to accumulate the amount of fluid at each position [21].

C. Fluid Surface Smoothing

After obtaining a fluid surface from particle positions, a smoothing process is needed to prevent a blobby-like appearance in the final rendering. A simple Gaussian blur is not suitable for this task, since it would blur the fluid’s borders, blending the particles with background surfaces [21]. An alternative to the Gaussian blur is the bilateral filter [23], which employs a regular Gaussian filter with a spatial kernel \( f \) and a function \( g \) on the intensity domain to determine a pixel’s weight, combining two Gaussian filters, one in the spatial domain and another in intensity domain. This process makes the value of a certain output pixel \( s \) more influenced by pixels close to it in both domains [33]. Its output is:

\[
J_s = \frac{1}{K} \sum_{p \in \Omega} f(p-s)g( Ip - Is ) Ip,
\]

where \( p \) is a pixel on the image \( \Omega \), \( Ip \) and \( Is \) are the values of pixels \( p \) and \( s \) on the intensity domain, and \( k \) is a normalization term:

\[
k_s = f(p-s)g( Ip - Is ).
\]

Bilateral filter iterates through both width and height of its spatial kernel at the same time, making it expensive as the kernel size grows. For this reason, a faster approximation version of this filter that still satisfies the noise reduction and edge preservation requirements can be used for the sake of performance. This version first applies a one-dimensional filter to one dimension of the image, and then filters this intermediate result in the subsequent dimension [22]. This way the computational complexity of the separated Bilateral Filter becomes \( O(p) \), faster than the \( O(p^2) \) full kernel approach, where \( p \) is the number of pixels in the image [34]. Although this complexity reduction is obtained at the expense of the generation of some artifacts, this strategy was applied successfully to the photo-realistic water rendering method presented in [21], since its shading steps mask these artifacts. In a cartoon style rendition, though, they are not enough to hide these artifacts [37] [38].

Gastal and Oliveira [24] shows that these striped artifacts come from the fact that the filtering of a 2D signal using a 1D transform is not a separable operation. This occurs because with a single iteration of a 1D filtering (a horizontal pass followed by a vertical pass, or vice-versa) some pixels that belong to the same region, affected by the filters kernel, may end not being combined. To solve this problem, we implemented a simpler version of their technique, relying on two key observations: every 1D step removes artifacts introduced by the previous, what makes the stripes present only along the last filtered dimension, and that their length is proportional to the dimensions of the kernel used on the last pass. This way a sequence of vertical and horizontal passes are interleaved, reducing its kernel dimensions in half at every new iteration and progressively reducing the extension of the striped artifacts, making them virtually unnoticeable, while causing little performance impact. Our experiments (section IV) further elaborates on the algorithms results.

D. Final Compositing

With all the intermediary textures available, they are finally combined to render the cartoon water with foam and droplets. The water color \( I_{\text{wcs}} \) is obtained as follows:
\[ I_{wcs} = \text{mix}(a, b + c, K_{mix}), \]

where the mix operator represents a linear interpolation between the refracted fluid color \( a \) an the sum of the the refracted and reflected scene colors \((b + c)\), while \( K_{mix} \) is a blend factor between \( a \) and \( b + c \), that determines how much of the optical effects will be seen in the final rendering.

In a realistic driven rendering process it would be ideal to show reflection and refraction simultaneously, as it happens with actual fluids. In a cartoon driven process this is not appropriate, though. A common approach in traditional hand drawn animation is to depict either reflection or refraction separately, as artists tend to emphasize refraction when the angle between the camera’s view vector and the fluid’s normal is small and emphasize reflection otherwise, although the artist usually does not actually calculate the angle, but define this intuitively. In an interactive environment, such as a game, an unwanted flickering would appear if the transition between reflection and refraction was too sharp. To avoid that kind of negative effect and create a smooth transition between both effects we use a simple interpolation based on the work of You et al. [8]:

\[
F_r = \begin{cases} 
K_{Rmax} & \text{if } x < \cos(s_{max}) \\
K_{Rmin} & \text{if } x > \cos(s_{min}) \\
f(\pi \cdot \vec{V}, K_{Rmax}, K_{Rmin}) & \text{Otherwise}
\end{cases}
\]

(4)

\[
F_t = \begin{cases} 
K_{Tmax} & \text{if } x < \cos(s_{max}) \\
K_{Tmin} & \text{if } x > \cos(s_{min}) \\
f(\pi \cdot \vec{V}, K_{Tmax}, K_{Tmin}) & \text{Otherwise},
\end{cases}
\]

(5)

where \( F_r \) and \( F_t \) are respectively the computed reflection and transparency factors and both \( f(\pi \cdot \vec{V}, K_{Rmax}, K_{Rmin}) \) and \( f(\pi \cdot \vec{V}, K_{Tmax}, K_{Tmin}) \) are cubic polynomial interpolation functions. The vectors \( \pi \) and \( \vec{V} \) represent the normal direction and the viewpoint direction, respectively. \( K_{Rmax} \) and \( K_{Rmin} \) are the maximum and minimum reflection, while \( K_{Tmax} \) and \( K_{Tmin} \) are the maximum and minimum transparency (or refraction), respectively. To understand the interpolation process it is necessary to define the angles \( \theta_c \), where the critical change between reflection and refraction occurs, and \( \theta_s \), the interpolation interval. Finally, \( x \) is the cosine resulting from the dot product between the normal and viewpoint direction. The interpolation occurs when \( x \) is between \( \cos(s_{min}) \) and \( \cos(s_{max}) \), and when its value is outside these limits only one of the effects is shown.

To create the cartoon style we use a technique described in [41], which consists in a modification of the shading model that creates large blocks of the same color with sharp transitions between them. This technique discards the specular component, and apply a quantization in the cosine of the angle between the normal and the light source, creating a fixed number of levels. This quantized cosine value \( Q_{\cos} \), which is normally used in the diffuse term, is obtained:

\[ Q_{\cos} = \left(\frac{(\cos(L \cdot n))}{T}\right), \]

where \( L \) is the direction of light source, \( n \) is the surface normal and \( T \) is the number of levels. We could also change this to use the ceiling instead of the floor of the dot product between \( L \) and \( n \), resulting in a slightly brighter output.

Based on the process described in the section III-B the value of the thickness \( T(x, y) \) is used to control the blending of the fluid’s refracted color so that the background color is more attenuated as the fluid gets thicker. The fluid color \( a \) is then defined by

\[ a = I_d C_f, \]

(7)

where \( C_f \) is the untreated fluid color that when multiplied by the quantized diffuse term \( I_d \) generates the cartoonified color.

To achieve the transparency effect \( b \), first the scene without the fluid is rendered to a background texture \( S_b(x, y) \), which is in turn perturbed based on the surface normal \( n \) to convey the illusion of refracting the background, composing the scene color as follows:

\[ b = S_b(x + n_x, y + n_y)e^{-T(x,y)}F_t. \]

(8)

We use an exponential fall-off, \( e^{-T(x,y)} \), in order to make a vary in an interesting way with the thickness \( \delta \), and the previously described refraction factor, \( F_t \), to further restrain the fluids transparency.

The reflection effect, represented by \( c \), is computed through the sampling of a cubemap texture \( S_c(x, y) \), and have it’s value restrained by the reflection factor \( F_r \):

\[ c = S_c(x, y)F_r. \]

(9)

Naturally, this could be replaced by a dynamic generated skybox or by a multi object composed scene.

At this point all acquired values are put together following the equation 3 and \( I_{wcs} \) is generated for each fragment.

IV. RESULTS AND DISCUSSION

To evaluate the technique a series of experiments were conducted using a machine with a 3.4 GHz Intel Core i7 4770 processor, 16 GB DDR3 RAM and an NVIDIA GeForce GTX 680 in 1024 x 768 resolution. This video card has a 2048MB GDDR5 memory and 1536 CUDA cores. Rendering was developed with OpenGL and GLSL, while NVidia’s PhysX SDK 3 was used to perform the fluid simulation. For the tests we used a scene containing

\[ \text{www.geforce.com/hardware/technology/physx} \]
37500 fluid SPH particles interacting with a heightmap. All parameter values used in the experiments were obtained empirically. Fig. 4 displays the visual results with different parameter values for the smoothing process. It shows that a small value of spatial $\sigma$ leave a blobby aspect that gradually decreases with the increase of this parameter. Increasing the value of domain $\sigma$ too much increases the influence of pixels that have distant depth values, causing the fluid to blend with the background. Fig. 5 shows the variation of results for the separation of water foam and droplet particles. In this case, the higher the value of the density threshold $T_d$ more particles will turn into foam and droplets. If extremely high values are set to $T_d$ more particles start to assume the droplet state and are also removed from the scene.

Table I provides performance values showing the average rendering time per frame of different methods in the same scene. We compare the iterative version of the separated bilateral filter with its single iteration separated version and the curvature flow method proposed by van der Laan [4]. The results shown were obtained with the same parameters (kernel dimensions of 15x15, domain $\sigma$ equals 0.01 and spatial $\sigma = 10.0$) for all versions of the bilateral filter. Here we can also see the performance degradation caused while increasing the number of iterations on the iterative filters. A visual comparison of the filters is shown on Fig. 6. The visual impact of increasing the number of iterations on the iterative separated bilateral filter and the curvature flow are shown in Fig. 7 and Fig. 8, respectively.

Increasing the number of iterations of the separated bilateral filter shows a small performance degradation, that can be explained by its linear behavior, as we can see in Fig. 7. The curvature flow method needs 50 times more iterations to achieve the same visual results of the iterative separated bilateral filter (Fig. 8), what explains the performance degradation observed on Table I. Another key observation that explains the much higher performance degradation on the curvature flow filter is that every new iteration of the separated bilateral filter have a kernel with half of the size of the previous one, what makes the former faster. The curvature flow filter iterations always have the same parameters and thus the same average running time.

To assess the visual quality of the iterative method, it is compared with the results obtained with its full kernel and separated versions, and also with the curvature flow, although the performance of the full kernel filter and the artifacts generated by its single iteration separated version precludes their use in practical situations. Fig. 6(a) shows our scene without depth smoothing, while Fig. 6(b)-(g) shows the same dataset rendered with different smoothing methods: full kernel, single and 3 iterations separated bilateral filter and 220 iterations curvature flow, respectively. Striped artifacts generated by the single iteration separated bilateral filter can be clearly seen in Fig. 6(c), while three iterations are applied to get the results shown in Fig. 6(d) where these
Fig. 5. Results for different foam and droplet thresholds: (a) with $T_d = 0.5$ almost no foam and droplets are generated, (b) $T_d = 2.0$ creates a good amount of foam and droplets, and (c) $T_d = 7.5$ generates foam and droplets where it should be water.

artifacts becomes virtually unnoticeable. The curvature flow filter repeated 220 times shows similar visual results when compared with the three iterations separated bilateral filter, but with a slower performance, as shown in Table I.

Table I

<table>
<thead>
<tr>
<th>Filter</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Smoothing</td>
<td>2.85</td>
</tr>
<tr>
<td>Curvature Flow, 10 it.</td>
<td>3.10</td>
</tr>
<tr>
<td>Curvature Flow, 110 it.</td>
<td>22.05</td>
</tr>
<tr>
<td>Curvature Flow, 220 it.</td>
<td>43.65</td>
</tr>
<tr>
<td>Separated Bilateral Filter, 1 it.</td>
<td>3.25</td>
</tr>
<tr>
<td>Separated Bilateral Filter, 3 it.</td>
<td>3.60</td>
</tr>
<tr>
<td>Separated Bilateral Filter, 4 it.</td>
<td>3.64</td>
</tr>
</tbody>
</table>

Fig. 7 shows how the striped artifacts generated by the separated version of the bilateral filter are gradually removed if the iterative filter is applied. Two iterations are applied to obtain the results shown in Fig. 7(a), that shows smaller artifacts when compared with the single iteration algorithm (Fig. 6(b)). If 3 iterations are applied (Fig. 7(b)) these artifacts become virtually unnoticeable, what suggests that the application of a fourth iteration (Fig. 7(c)) is not necessary. The same analysis is made for the curvature flow filter on Fig. 8(a)-(c), showing that the higher the number of iterations, more smoothed the fluid’s surface will be.

A close-up of the water surface is shown on Fig. 10(a)-(b), rendered with the full kernel and three iteration bilateral filter methods respectively, with a kernel size of 15x15. Fig. 10(c) shows the same scene when smoothed with 220 iterations of the curvature flow filter. Fig. 9(a)-(c) shows how the surface becomes smoother as the kernel size grows when applying the three iterations separated bilateral filter.

Fig. 12 shows how the blobby appearance of this element is seen if the viewer is close to the fluid’s surface. Unlike our previous work though, the linear interpolation used to set the size and transparency of foam particles prevents the occurrence of flickering in the foam and droplet generation, making the water look more natural. The alpha blending process made between foam and water particles also have the effect of giving foam particles more than one color, what makes its visuals looks more like what’s seen in traditional cartoons. An accompanying video shows animation results for both cartoon and photorealistic styles of rendering.

By replacing the interpolation function from which reflection and refraction are obtained with a Fresnel effect our method can also generate a photorealistic rendering, as shown on Fig. 11(a)-(c).

V. CONCLUSION AND FUTURE WORKS

In this paper we presented a method renders a visualization based on a SPH fluid simulation that achieves real-time performance while smoothly generating foam and
Fig. 6. The waterfall scene rendered with different methods: (a) Without smoothing, (b) Full kernel bilateral filter, (c) Single iteration separated bilateral filter, (d) the same filter with 3 iterations, and (e) curvature flow filter with 220 iterations. The same parameters where used for all the bilateral filter methods: spatial kernel dimensions of 15x15, domain $\sigma = 0.01$, spatial $\sigma = 10.0$, $T_d = 2$. 

[Picture of waterfall scene rendered with different methods]
Fig. 7. Results for different number of iterations of the separated bilateral filter: (a) 2 iterations, (b) 3 iterations and (c) 4 iterations. The following parameters were used for all tests: spatial kernel dimensions of 15x15, domain $\sigma = 0.01$, spatial $\sigma = 10.0$, $T_d = 2$.

Fig. 8. Results for different number of iterations of the curvature flow filter: (a) 10 iterations, (b) 110 iterations and (c) 220 iterations.

Fig. 9. The same scene smoothed with the iterative separated bilateral filter and different dimensions for the kernel: (a) 10x10, (b) 20x20 and (c) 30x30. Domain $\sigma = 0.01$, spatial $\sigma = 10.0$, $T_d = 2$.

Fig. 10. A close-up on the fluid surface rendered with different methods: (a) Full Kernel Bilateral Filter, (b) Iterative Separated Bilateral Filter with 3 iterations and (c) Curvature Flow with 220 iterations. Kernel size of 15x15, domain $\sigma = 0.01$, spatial $\sigma = 10.0$, $T_d = 2$ for (a) and (b).

Spray effects based on [38]. Our approach presents a simpler technique to generate foam and spray that is suitable for non-photorealistic rendering and is performed in a smaller number of shading steps while improving visual results. We
Fig. 11. Example of a photorealistic rendering obtained with our method. Here the optical effects interpolation was replaced by a Fresnel effect, and the color quantization was removed.

Fig. 12. A close-up on the generated foam rendered with the Iterative Separated Bilateral Filter with 3 iterations and kernel size of 15x15, domain $\sigma = 0.01$, spatial $\sigma = 10.0$, $T_d = 2$ for (a) and (b).

The iterative separated bilateral filter algorithm successfully removes striped artifacts and have a low performance impact when compared with its single iteration version. It is also capable of generating fluids with surfaces smoother than the ones obtained with its full kernel version. We also tested our approach against the widely used Curvature Flow method [4], obtaining considerably superior visual and performance results. A reduction of the blobby aspect of the fluid’s surface even when the viewer is close to it is also obtained with this filter. The proposed method should be able to be integrated in any game engine that employs particle simulations to get the results here presented.

Although our method efficiently separate water from foam and droplets while producing a smooth transition between these states, foam and droplet particles still look more blobby than water particles when they are close to the viewer. Also, the fluid’s visual obtained with our method is not kept consistent while the viewer approaches its surface. We leave for future works the development of a technique to generate smoother foam and an adaptive smoothing process that varies its level with the distance between the surface and the viewer, keeping its visual consistent with this distance, as implemented in the work of Bagar et al. [6].

ACKNOWLEDGMENT

The authors would like to thank Cheng-Tso Lin for providing the code of his real-time fluid simulation project, which is based on [40] and [4], including the Curvature Flow shaders used in our comparison tests, and FAPESB for financing the machines used in the development and tests of this work.

REFERENCES


4https://otaku690.appspot.com/


