Real-Time Screen Space Rendering of Cartoon Water

Liordino dos S. Rocha Neto, Filipe Deó Guimarães, Antonio L. Apolinário Jr. and Vinícius M. Mello
Department of Computer Science
Federal University of Bahia
Salvador, Brazil

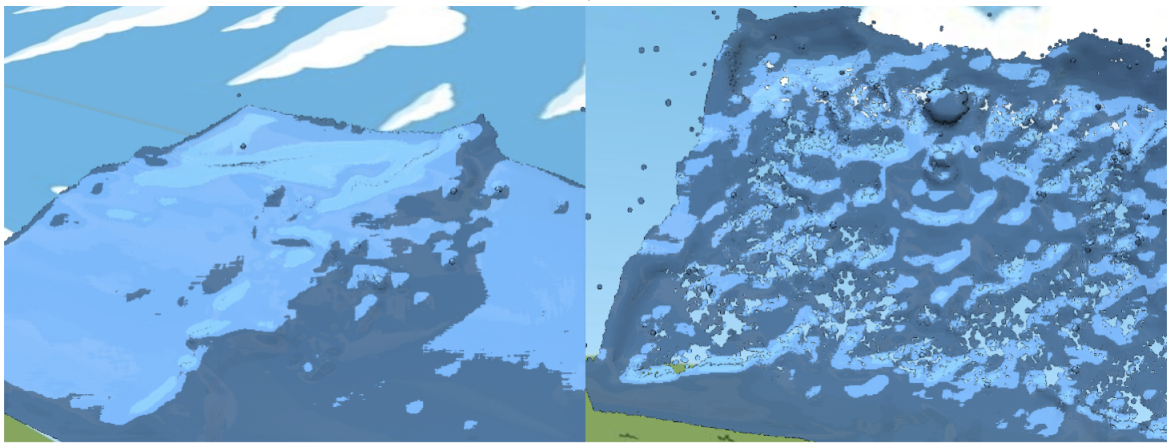


Fig. 1: Examples of the results achieved with our method for two different scenes.

***Abstract*—A non-photorealistic rendering style is constantly chosen in games to emphasize the fantasy of the story. In this scenario, the presence of natural elements such as water is common. Simulation and rendering of water in 3D worlds still presents some technical challenges though. This paper describes an approach to render cartoon style water in real time environments. The method takes a Smoothed Particles Hydrodynamics (SPH) fluid simulation as input and obtains its surface using a point splatting technique, employing a dynamic cartoon style shading on its visualization. It also takes into account refraction and reflection, and use a bilateral filter to smooth the fluids surface and prevent a jelly-like appearance in the final rendering. An empirical analysis were made in order to determine a speed/quality trade-off. All steps of this solution were implemented directly on the GPU, producing cartoon water animation with a great number of particles at a frame rates compatible with real-time applications such as games.**

Keywords: water rendering, non-photorealistic rendering, cartoon shading, smoothed particle hydrodynamics, gpu programming, screen space rendering

I. Introduction

Stylized games using non-photorealistic rendering (NPR) are becoming increasingly popular. The use of NPR rendering as a mean to emphasize the fantasy aspects of the story, or gameplay, has been proven effective [1]. In this scenario, natural elements, such as water, sand or even sunlight, are essential to give the game a coherent aspect.

Particle based simulations are used as an important tool to provide realistic visual effects, like water, explosions and smoke in applications such as games and movies [2]. Particularly in the game industry, it's possible to notice the use of simulated water in 2D games like "Disney's Where's My Water"[1] as a significant gameplay element [3]. The same can't be told about the majority of 3D games, where it's commonly used as a static decoration without much interaction. But, as rigid body physics emerges as an important aspect of games, water simulation could provide interesting gameplay experiences thanks to its complex behaviour. Its use still feels very natural, as water is an well known element for everyone since its childhood early stages [3]. Within this context, due to recent advances in computer graphics technology, manually created animation, as in cartoon animation, is increasingly being replaced by its computer-generated counterpart [4].

In the context of liquid simulation on interactive applications, methods like Smoothed Particle Hydrodynamics (SPH) are preferred to grid based representations[5]. This occurs due to the fact that on this kind of simulation the fluids can flow in the entire scene without the need to define a finite grid, which would be costly in terms of memory and computation [5]. SPH fluid simulations are commonly resolved by modelling surface tensions forces and using the Navier-Stokes equation of conservation of momentum to derive the fluids viscosity and pressure force fields [6]. This method is

[1] http://disney.go.com/wheresmywater/

also convenient to integrate on interactive environments and physics systems, since particles can interact with the rest of the scene geometry just like any other rigid body objects. Its drawback is a more complex surface extraction for rendering [5]. This extraction can be made using poligonization methods like Marching Cubes, which requires the reconstruction of the fluid surface in world space [5], or in screen space. The former is usually computation and memory intensive, making it not suitable for real-time use in games, while on the latter the surface reconstruction occurs in screen space, becoming a more affordable approach to achieve a real-time environment experience, and preventing the usual grid artifacts of Marching Cubes methods [5]. After extracting the fluid surface the next step is to render it as a water like representation.

In this paper we present a real-time non-photorealistic (NPR) water rendering method that works on top of a SPH fluid simulation, achieving real-time performance using a screen space surface extraction and smoothing method, while taking reflection and refraction into account. The paper is structured as follows: section II lists the related works, highlighting their contributions to our method, which is described in section III. The fourth section shows our results, and after that the paper is concluded in section V.

II. Related Work

The rendering of liquid surfaces has been an active field of research even before the adoption of water simulation methods on it's pipeline, with methods that simulate waves using techniques like bump mapping [7]. The use of SPH to simulate fluids with free surfaces was introduced by [6]. This work uses surface splatting and marching cubes poligonization techniques to obtain the fluid surface, showing visual comparisons between both. According to the authors, the point splatting technique achieves plausible results, while the rendering with marching cubes algorithm is more convincing, but with a performance drawback. The work described on [8] presents another technique for volume rendering of SPH data which can render iso-surfaces. It's a poligonization technique that resamples the particles on a view-dependent grid, leaving out of sight particles left aside the resampling process and thus avoiding to process unnecessary data. It can achieve a rate of 4.5 frames per second rendering a iso-surface of a set of 2.2 million particles. Although these are interesting results, a screen space approach is sufficient to meet our method's expected outcome, while being simpler to implement. Another poligonization method is presented on [9]. This work proposes a parallel implementation of the Marching Cubes method that works both on CPUs and GPUs and that achieves a performance gain by only considering grid nodes in a narrow band around the fluids surface. It results on smooth surfaces, but is designed for off-line rendering.

While the previous methods work in object space (except for [6], which compares both approaches), in [5] a photorealistic splatting-based approach is presented. Its surface extraction occurs in screen-space, hence it's not based on poligonization, and consists of several steps performed directly on the graphics hardware, using fragment shaders and intermediate render targets. It achieves real-time performance and uses a curvature flow smoothing algorithm to prevent a jelly-like appearance on the fluids surface. The work presented in [10] is also a photorealistic approach and extends [5] introducing an adaptative curvature flow, which accounts for perspective, and a physically based foam rendering method, using a layered particle set. Another work based on [5] is showed in [11], where the curvature flow smoothing is replaced by a two-step bilateral filter. Our surface extraction approach is based on the methods presented in [5] and [11], and unlike the one presented in [10] don't take foam rendering into consideration since it would diverge of our non-photorealistic goal.

Some contributions like [12] and [13] help to address the matter of rendering the fluid as a cartoon like water representation, but they lack the level of detail that a traditional cartoon artist achieves, as they treat water as an opaque object, ignoring its optical characteristics. A more recent contribution [4] address this, taking into account optical features like transparency, reflection and refraction, but it was developed as a Maya plugin, making it unsuitable for real-time environments. [12] take as input a liquid surface obtained from a physically based liquid simulation system to render a cartoonesque representation of water inspired in animations such as Futurama and The Little Mermaid. [13] presents a template-based approach, classifying water in different types, like flowing water and water jets, and designing shapes that can be grouped according to some rules. As mentioned earlier, the work described on [4] is designed for off-line rendering, and like the one described on [12], uses a surface from a three-dimensional physically based fluid simulation as input, but combines several shading steps to compose a NPR water that take into account its optical features, while adding special lines to represent the flowing motions of water and bold outlines to the objects that interact with it. They also introduced an automatic control of reflection and refraction.

Our work is based on the methods described on [5], [11] and [4]. Like [5] and [11], we assume that a SPH fluid simulation has already been carried out, and use its data as input to our rendering method, which in our case is limited to the positions of its particles in any order. This way, we combine different aspects of these methods with our own research to create a new one, adapting each approach to our specific needs. Thus, we were able of designing a new method, capable of rendering non-photorealistic water in real time while considering its optical characteristics.

III. NPR Water

The NPR method here presented can be broken down on the following steps: from the particles position, surface depth and thickness are extracted (explained in sections III-A and III-C). Then, the surface depth is smoothed (section III-B), and after that, a composing pass is performed, combining the smoothed surface depth with a texture of the scene (without the fluid) into its final rendering, using the extracted thickness to create the transparency effect on the cartoon water shader, based on the one described in [4] (section III-D).

A. Surface Extraction

The surface extraction process used in our method is based on [5]. Its premise requires the fluid's front-most surface to be determined according to the camera's viewpoint. To achieve this the SPH particles are rendered as spheres, and through

depth test its closest values for each pixel are obtained and stored in a texture using a depth replacement technique in the fragment shader. The particles are rendered as spheres using screen oriented quads, or simply point sprites, by discarding the pixels outside the circle inscribed in the quad. The point sprite size is computed relative to the viewers distance to the fluid surface, making it bigger as the camera approaches the fluid. This way we keep the particles close to each other, preventing the occurrence of holes in the final rendering. The surface normals are calculated from the depth values while rendering. This provides a fluid surface representation from the viewer's point of view while avoiding any kind of complex geometry.

B. Surface Smoothing

Since the surface depth is obtained from particles rendered as spheres, it needs to be smoothed to prevent a jelly-like appearance in the final rendering. A Gaussian blur filter could be used, but the fluid's silhouette edges would be lost in the process, with the particles getting blended into background surfaces [11]. The solution traditionally applied in literature is the use of a Bilateral Filter [14]. Below we describe the Bilateral Filter and an approximation of its implementation, which divides the filter in two passes, a horizontal and vertical one, for the sake of performance.

The Bilateral Filter employs a regular Gaussian filter with a spatial kernel f, but unlike the standard Gaussian, the weight of a pixel inside this kernel depends also on a function g in the intensity domain [15]. This function decreases the weight of pixels with large intensity differences, and essentially preserve the edges while still smoothing the surface by means of a combination of nearby depth values. Therefore, the output value of the bilateral filter for a pixel s can be defined as stated by [15]:

$$J_s = \frac{1}{k} \sum_{p \in \Omega} f(p - s) g(Ip - Is) Ip \tag{1}$$

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

$$k_s = f(p - s) g(Ip - Is) \tag{2}$$

In practice, the bilateral filter combines two Gaussian filters, one in the spatial domain and another on the intensity domain, making the value of an output pixel s influenced mainly by pixels that are close both spatially and in intensity [15].

Since it iterates through both the width and the height of the spatial kernel at the same time, the Bilateral Filter can become expensive as the kernel size grows. Although it's not strictly separable due to it's intensity dependency, it's possible to implement the Bilateral Filter in a separable way and still satisfy the noise reduction and edge preservation requirements [16], producing a aproximation of the Bilateral Filter while creating some artifacts [11]. This is done by applying a one-dimensional filter to the first dimension of the image, and filtering the intermediate result in the subsequent dimensions [16]. This way the computational complexity of the Bilateral Filter becames $O(p)$, making it faster than the full kernel approach, which uses a two-dimensional kernel that makes its complexity $O(p^2)$ (where p is the number of pixels in the image) [17]. Both the Bilateral Filter and it's separeted version were applied in this projet and will be further analysed in section IV.

To achieve a higher performance we managed to balance the smoothing process according to the distance between the camera and the fluid. Namely, as the camera gets closer to the fluid we increase the size of the spatial kernel f. Analogously, as the camera gets farther from the fluid, we decrease the value of f. This simple variation results in a great visual boost while reducing the overall cost, by reducing the use of unnecessary large spatial kernels when the fluid is far enough not to have a significant visual difference between a large kernel or a smaller one. The visual improvement is given by increasing the the kernel size as the camera gets closer to the fluid and that results in a smoother fluid surface, while also increasing the performance cost. Since getting a fluid close-up isn't as common as looking at the fluid from mid range or farther, this shouldn't have any major negative effects on the overall performance.

C. Thickness

It's expected that an object underwater 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 [5], 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 the 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 [11].

D. Water Rendering

Once the fluid surface is available the next step is to render it applying the cartoon water effect. A proper representation of water should have optical features, and so our method take reflection and refraction into account, making use of a illumination model based on [5]. The final illumination of the Cartoon Water Shader I_{wcs} is obtained as follows:

$$I_{wcs} = a + bF_r \tag{3}$$

where a is the refracted fluid color, b is the reflected scene color and F_r is the reflection factor. The reflection effect, represented by b, is simplified by taking into consideration a cubemap texture, which is sampled using a reflection vector computed through the surface normal and the view vector. Naturally, this could be replaced by a dynamic generated skybox or by a multi object composed scene.

Since our purpose is to render a cartoon like water, we adapted the final shading equation and didn't use a Fresnel approximation as [5] employs it, as it would introduce a realism factor in our method. Instead, we implemented a simple interpolation function, based on [4], that determines when to apply reflection or refraction according to the viewer's

position. 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 approach isn't appropriate, though. A common approach in traditional 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 [4], although the artist usually doesn't 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 make use of the aforementioned interpolation:

$$F_r = \begin{cases} K_{Rmax} & \text{if } x < \cos s_{max} \\ K_{Rmin} & \text{if } x > \cos s_{min} \\ f(\bar{n} \cdot \bar{V}, K_{Rmax}, K_{Rmin}) & \text{Otherwise} \end{cases} \quad (4)$$

$$F_t = \begin{cases} K_{Tmax} & \text{if } x < \cos s_{max} \\ K_{Tmin} & \text{if } x > \cos s_{min} \\ f(\bar{n} \cdot \bar{V}, K_{Tmax}, K_{Tmin}) & \text{Otherwise} \end{cases} \quad (5)$$

where F_r and F_t are respectively the computed reflection and transparency factors and both $f(\bar{n} \cdot \bar{V}, K_{Rmax}, K_{Rmin})$ and $f(\bar{n} \cdot \bar{V}, K_{Tmax}, K_{Tmin})$ are cubic polynomial interpolation functions. The vectors $\bar{n}$ and $\bar{V}$ represent the normal direction and the viewpoint direction, respectively. K_{Rmax} and K_{Rmin} and the maximum and minimum reflection, while K_{Tmax} and K_{Tmin} are the maximum and minimum transparency, respectively. To understand the interpolation process it's necessary to define a θ_c, the angle where the critical change between reflection and refraction occurs, and θ_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. A graphical explanation of a similar interpolation is presented by [4].

To create the cartoon style we use a technique described in [18], 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:

$$Q_{cos} = (\lfloor \cos(L \cdot n) \rfloor) \frac{1}{l} \quad (6)$$

where L is the direction of light source, n is the surface normal and l 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 [4], we extend this approach dividing these levels in 3 regions: bright, medium and dark. This is done by defining two thresholds that we call T_{bc} and T_{dc} for the bright and medium values, respectively, as shown in equation 8. The goal here is to turn the intensity that are over T_{bc} brighter, and the ones that are under T_{dc} darker, while keeping the medium values unchanged. This is done by multiplying the bright and dark values by two factors: F_{bc}, which must be greater than 1, for the bright values, and F_{dc}, less than 1, for the dark values. By doing this we can ultimately limit the visual aspect of the fluid in three color levels, which prevents the water surface to have a blobby appearance, while keeping the cartoon style with simplified color regions. The quantized diffuse intensity is then obtained as follows:

$$I_d = \begin{cases} Q_{cos} F_{dc} & \text{if } Q_{cos} > T_{dc} \\ Q_{cos} F_{bc} & \text{if } Q_{cos} < T_{bc} \\ Q_{cos} & \text{Otherwise} \end{cases} \quad (7)$$

Based on the process described in III-C 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

$$a = lerp(I_d C_f, C_s, e^{-T(x,y)} F_r) \quad (8)$$

where C_f is the untreated fluid color that when multiplied by the quantized diffuse term I_d generates the cartoonified color and C_s is the scene refraction based on the thickness previously extracted. We use an exponential fall-off, $e^{-T(x,y)}$, in order to make a vary in an interesting way with the thickness [5]. At this point the previously described refraction factor, F_r, is used to further restrain the fluids transparency.

To achieve the transparency effect, first the scene without the fluid is rendered to a background texture $S(x, y)$, 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:

$$C_s = S(x + \beta n_x, y + \beta n_y) \quad (9)$$

This sample also employs a β factor, which increases linearly with the thickness:

$$\beta = T(x, y)\lambda \quad (10)$$

where λ is a constant that determines how much of the background is refracted, and depends on the type of fluid. Its intention is to make the color scene distortion stronger as the thickness value increases.

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

All presented results were obtained on a machine with an Intel Core i7 4770 processor, which have a 3.4 GHz clock, 16 GB DDR3 RAM and a NVIDIA GeForce GTX 680 in 1024 x 768 resolution. This video card have a 2048MB GDDR5 memory and 1536 CUDA cores. As far as the development environment goes, we utilized Visual Studio 2010, CUDA, OpenGL and GLSL in Windows 7 to develop our project, which was integrated with Fluids v3 [19], a real-time SPH fluid simulator. The datasets used to run all tests were 2 scenes provided by the Fluids v3 simulator: Large Ocean Waves, and Dual Wave Pool. The Large Ocean Waves dataset is composed of 262144 particles and a bounding box of dimensions 400x200x400 (width, height, depth) in world

coordinates, while the Dual Wave Pool scene comprises 65536 particles in a 200x100x200 bounding box. Figures 2 and 3 shows images of the Large Ocean Waves and Dual Wave Pool scenes respectively, each one with the fluid involved by its bounding box, textured with an ambient image.

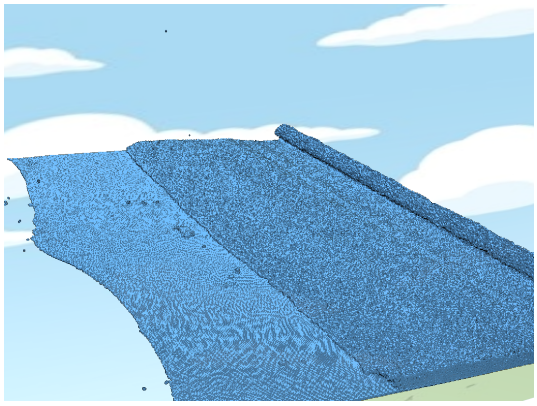
Performance values for both scenes are shown in table I. We compare the smoothing with the full kernel and separated versions of the bilateral filter with different kernel sizes. With these values it's possible to notice that the kernel size has a stronger influence in the full kernel than in the separate one. This can be explained by the quadratic behaviour of the full kernel filter, in contrast with the separated one, which have a linear performance, as explained in section III-B. It's noteworthy that the values shown in I include the simulation cost.

TABLE I: Performance comparison (in Frames Per Second) of our method with different settings

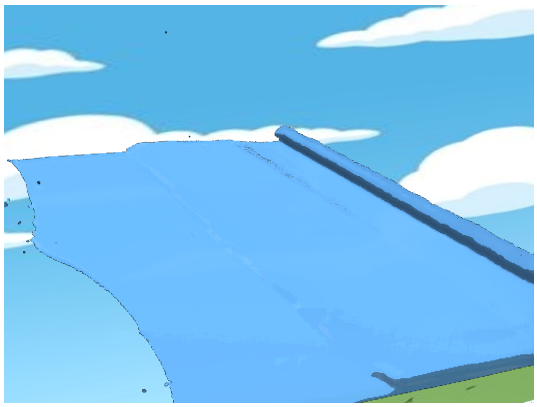
Large Ocean Waves	Kernel Size	FPS
Full Kernel	10	40
	20	30
	30	17
Separated	10	42
	20	38
	30	37
Dual Wave Pool		
Full Kernel	10	86
	20	47
	30	22
Separated	10	95
	20	84
	30	78

Figure 2 shows the Large Ocean Waves scene rendered without depth smoothing and with smoothing using the full kernel and separated versions of the bilateral filter. The same rendering methods are shown in figure 3, but for the Dual Wave Pool scene instead. There are clearly visible bumps on the fluids surface that show the non-smoothed rendering of the scenes, which are smoothed out by both versions of the bilateral filter. Although this process makes the fluid more natural, it's still possible to notice some bumps with a close-up on the fluid surface as seen in figure 5a. Figures 5b and 5c shows how the fluid surface become smoother as the kernel size grows. Since the fluid color is simplified to achieve the cartoon look, these bumps make the color change often through the surface, creating artifacts that becomes more visible as the camera approaches the liquid. Also we can clearly see a high occurrence of artifacts created by the separated bilateral filter on the fluid borders. These artifacts, which appears as a consequence of the axis-aligned nature of the separated filter, ends up masked by the shading effects of a photorealistic rendering like [11], but in a non-photorealistic scenario they are visible, making the use of the separated version of the bilateral filter infeasible to reach our goals.

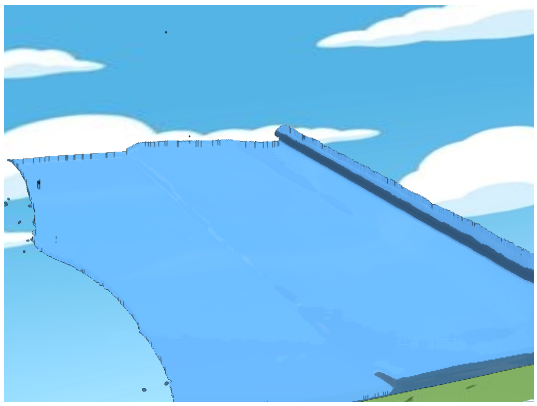
Figures 4 and 5 shows the visual results of the final rendering using the full kernel version of the bilateral filter as its kernel size grows. Figure 4 shows the results from a distant view while 5 does the same for a close-up, both with the Large Ocean Waves scene. Here we can clearly see how the fluids surface get smoother as the kernel size of the filter grows. The close-up figures shows that while a smoother surface can be obtained with a larger kernel it is not sufficient to eliminate the blobby aspect of the fluid if the viewer is close to it, living room for improvements in this aspect.



(a) Without smoothing

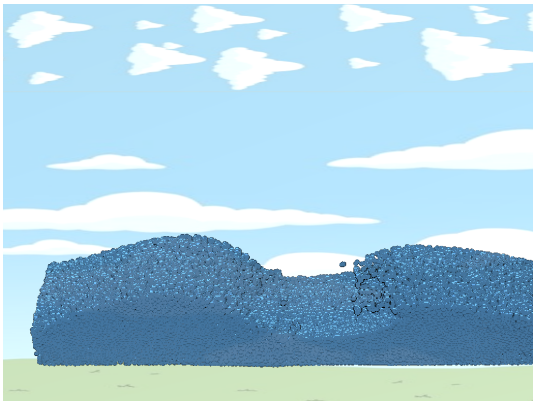


(b) Full kernel bilateral filter

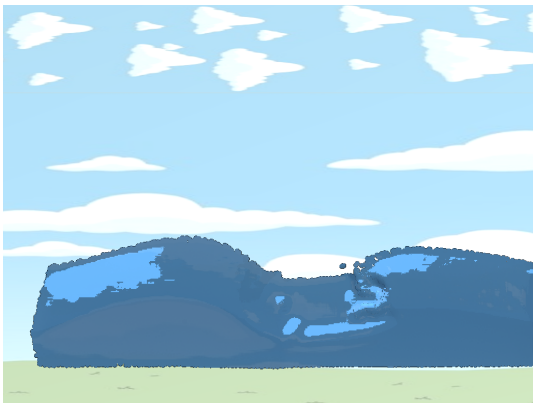


(c) Separable bilateral filter

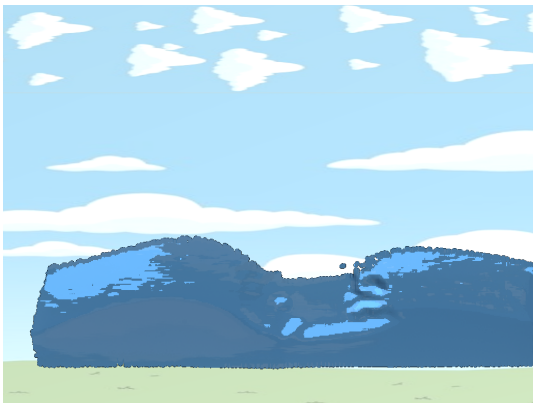
Fig. 2: The final rendering of the Large Ocean simulation with 262144 particles.



(a) Without smoothing

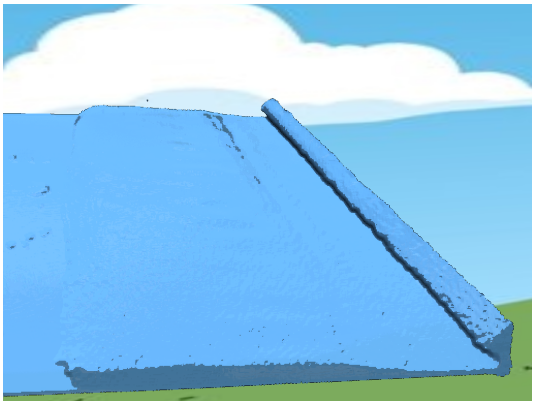


(b) Full kernel bilateral filter

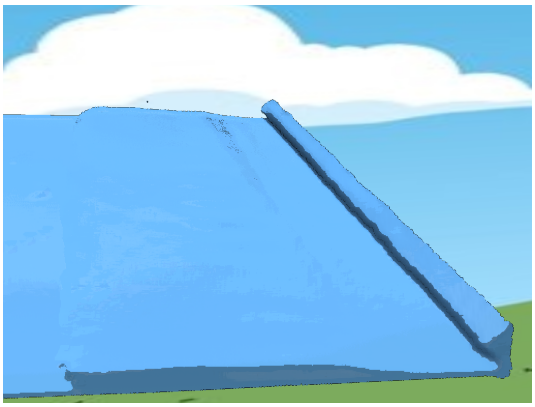


(c) Two-step bilateral filter

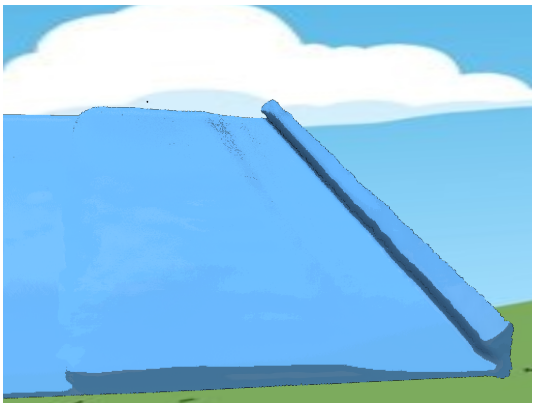
Fig. 3: The final rendering of the Dual Wave Pool simulation with 65536 particles.



(a) Kernel size 10

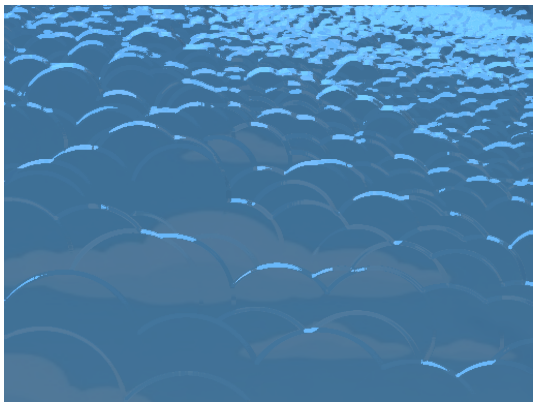


(b) Kernel size 20

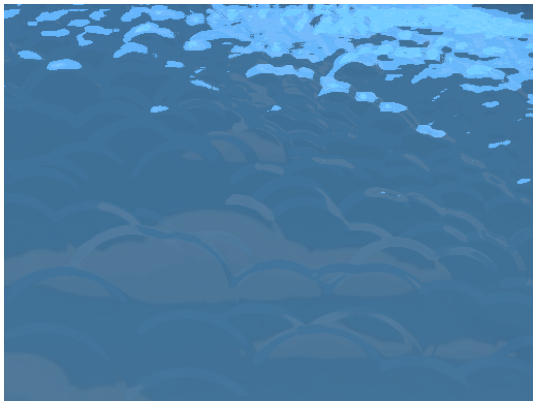


(c) Kernel size 30

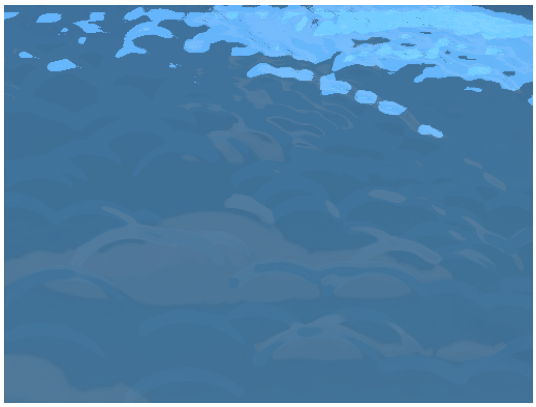
Fig. 4: The final rendering of the Large Ocean simulation with different kernel sizes.



(a) Kernel size 10

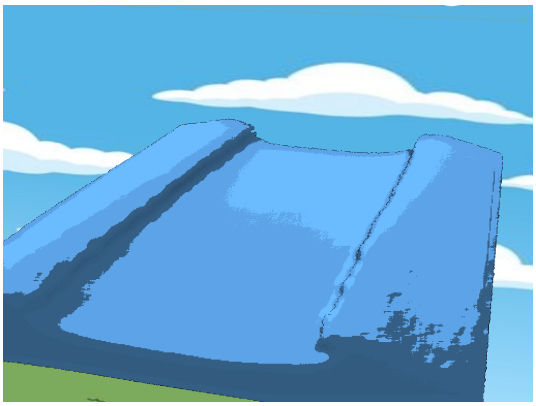


(b) Kernel size 20

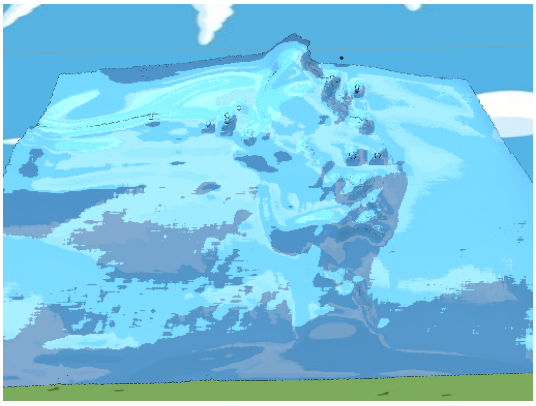


(c) Kernel size 30

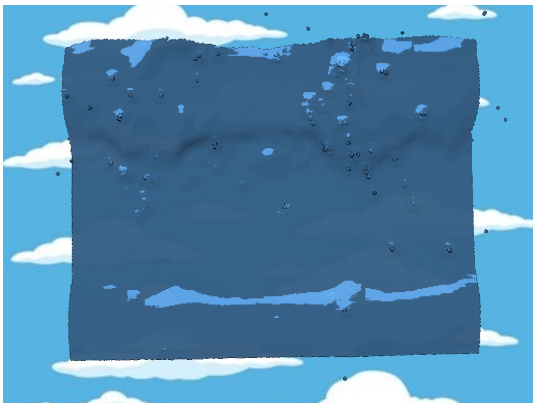
Fig. 5: Close-up of the Large Ocean Waves scene with different kernel sizes.



(a) Cartoon fluid without optical effects



(b) Cartoon fluid with reflection



(c) Cartoon fluid with refraction

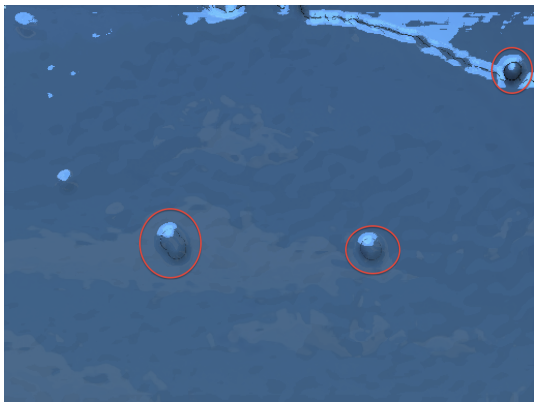
Fig. 6: Fluid's optical components.

Figure 6 shows 3 different renders: the figure 6a shows the fluid without any optical effect, only with the opaque cartoon color, figure 6b shows the fluid with the reflection effect, without refraction, and the figure 6c shows the fluid only with the refraction effect. Here it's possible to notice how these effects helps to convey some realism to the otherwise opaque fluid surface without harming the non-photorealistic look.

Finally, figure 7 shows the behaviour of stray particles. The figure 7a shows how the presence of a stray particle impacts the final rendering when it is close to the fluid surface in the camera Z axis and the figure 7b shows the impact when it is far from the fluid. In both images the relevant particles are highlighted by red circles. Here we can notice that when a stray particle is far enough from the fluids surface there is no influence of it in the smoothing process, and its edges are

preserved, but as they approaches the surface this influence grows, generating some artifacts around the particle until it finally enters the fluid again.

In order to improve the reader's visualization and evaluation of our results, a video was made available [2].



(a) Close to the fluid surface (Z axis)

(b) Far from the fluid surface (Z axis)

Fig. 7: Stray particles behaviour.

V. Conclusion and Future Work

In this project we were able to achieve a real time visualization of NPR water based on a SPH fluid simulation by using literature established techniques in new contexts, adapting them to meet our goals. The method here proposed should be able to be seamlessly integrated with any particle simulation ready game engine to obtain the results presented in the previous section.

Regarding future works, the Bilateral Filter used in this work left room for optimization in both visual and performance aspects and an adaptive curvature flow technique like the one applied in [10] could be used to replace it, but a further analysis would be required to define the best approach. The render method could also be benefited by a separated treatment of stray particles while obtaining the surface depth [5], since this kind of particle don't form a surface. We believe that such a feature might improve the Bilateral Filter results, considering it would allow a higher smooth level on its intensity domain while creating fewer artifacts. Additionally, our method could benefit from using lower resolution textures to render intermediate steps, sacrificing some quality to improve it's performance.

Finally, some cartoon rendering methods, not focused on fluids, employs the incorporation of bold outlines on rendered objects. An evaluation of the effect of such methods on a cartoon fluid might aggregate some new visual effects to this work.

[2] http://bit.ly/CartoonWaterShader

Acknowledgment

Although we ended up choosing a screen space approach, we thank Stefan Auer for sending us code-snippets of the surface extraction described in [8]. We'd also like to thank Mark Harris, from NVIDIA, for his suggestions and support on the beginning of our project, and Torsten Späte, who gently provided the code for the GLSL shaders used on his Fluid Sandbox project[3], which is based on [11]. Finally, we'd like to thank Rama Hoetzlein, author of the Fluids v3 project [19], for his support to our project since it's conception, and FAPESB, for financing the machines used in the development and tests of this work.

[3] http://bit.ly/FluidSimulationDemoEvolvesToFluidSandbox

References

[1] M. McGuire, "An introduction to stylized rendering in games," in *SIGGRAPH 2010 Course Notes*, 2010.

[2] S. Green, "Volumetric particle shadows," *Nvidia Developer Zone http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/smokeParticles/doc/smokeParticles.pdf*, 2008.

[3] T. Kellomäki, "Water simulation methods for games: a comparison," in *Proceeding of the 16th International Academic MindTrek Conference*. ACM, 2012, pp. 10–14.

[4] M. You, J. Park, B. Choi, and J. Noh, "Cartoon animation style rendering of water," in *Advances in Visual Computing*. Springer, 2009, pp. 67–78.

[5] 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*. ACM, 2009, pp. 91–98.

[6] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 2003, pp. 154–159.

[7] C. T. Pozzer, "Uso de técnicas de síntese de imagem aplicadas a um ambiente de representação de superfícies líquidas estáticas e dinâmicas," Ph.D. dissertation, MSc Thesis, Computer Science Department, ITA, 2000.

[8] R. Fraedrich, S. Auer, and R. Westermann, "Efficient high-quality volume rendering of sph data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, no. 6, pp. 1533–1540, 2010.

[9] G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner, "Parallel surface reconstruction for particle-based fluids," in *Computer Graphics Forum*, vol. 31, no. 6. Wiley Online Library, 2012, pp. 1797–1809.

[10] F. Bagar, D. Scherzer, and M. Wimmer, "A layered particle-based fluid model for real-time rendering of water," in *Computer Graphics Forum*, vol. 29, no. 4. Wiley Online Library, 2010, pp. 1383–1389.

[11] S. Green, "Screen space fluid rendering for games," in *Proceedings for the Game Developers Conference*, 2010.

[12] A. M. Eden, A. W. Bargteil, T. G. Goktekin, S. B. Eisinger, and J. F. O'Brien, "A method for cartoon-style rendering of liquid animations," in *Proceedings of Graphics Interface 2007*. ACM, 2007, pp. 51–55.

[13] J. Yu, X. Jiang, H. Chen, and C. Yao, "Real-time cartoon water animation," *Computer Animation and Virtual Worlds*, vol. 18, no. 4-5, pp. 405–414, 2007.

[14] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on*. IEEE, 1998, pp. 839–846.

[15] F. Durand and J. Dorsey, "Fast bilateral filtering for the display of high-dynamic-range images," 2002.

[16] T. Q. Pham and L. J. Vliet, "Separable bilateral filtering for fast video preprocessing," in *In IEEE Internat. Conf. on Multimedia & Expo, CD14*. IEEE, 2005, pp. 1–4.

[17] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "A gentle introduction to bilateral filtering and its applications," in *ACM SIGGRAPH 2007 courses*. ACM, 2007, p. 1.

[18] D. Wolff, *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, Jul. 2011.

[19] R. C. Hoetzlein, "Fluids v.3 - a large-scale, open source fluid simulator," Released under Z-lib license, 2012.