UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Narrow-Band Screen-Space Fluid Rendering using Layered Neighborhood Method

Felipe Orlandi de Oliveira

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Felipe Orlandi de Oliveira

Narrow-Band Screen-Space Fluid Rendering using Layered Neighborhood Method

Dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Afonso Paiva Neto

USP – São Carlos
December 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

O48n

Oliveira, Felipe Orlandi
Narrow-Band Screen-Space Fluid Rendering using Layered Neighborhood Method / Felipe Orlandi Oliveira; orientador Afonso Paiva Neto. -- São Carlos, 2021.
82 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2021.

1. screen-space fluid rendering. 2. real-time rendering. 3. particle based simulation. I. Neto, Afonso Paiva, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Felipe Orlandi de Oliveira

Renderização de Fluidos em Espaço de Tela usando Superfície Visível usando método de Vizinhança em Camadas

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Afonso Paiva Neto

USP – São Carlos
Dezembro de 2021

ACKNOWLEDGEMENTS

I have received a great deal of assistance and support throughout development of this work. I would like to thank my supervisor, Prof. Dr. Afonso Paiva Neto, who helped and guided me until the end. I also thank my family for always supporting me in doing this research and, finally, my friends for distracting me when I was too tired to continue working with fluids.

RESUMO

OLIVEIRA, F. O. **Renderização de Fluidos em Espaço de Tela usando Superfície Visível usando método de Vizinhança em Camadas**. 2021. 82 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Este trabalho apresenta uma nova abordagem para o rendering de fluidos em screen-space. Construímos sobre formulações já existentes e, fazendo uso da superfície livre do fluido, conseguimos um aumento notável de performance. Introduzimos uma simples abordagem para extração de *narrow-band* - *Layered Neighborhood* - que é compatível com o pipeline de rendering em screen-space e não introduz complexidade para o simulador de fluidos, juntos, extração e técnica de rendering, conseguem aumentar a performance do processo de desenho sem perda de qualidade.

Palavras-chave: Rendering de fluidos em screen-space, Rendering em tempo real, Simulação baseada em partículas..

ABSTRACT

OLIVEIRA, F. O. **Narrow-Band Screen-Space Fluid Rendering using Layered Neighborhood Method**. 2021. 82 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

This works introduces a new approach for screen-space fluid rendering. We built on top of classic methods and, with the use of fluid narrow-band, we heavily increase performance of those methods. We present a simple formulation for a narrow-band extraction - the *Layered Neighborhood Method* - that is compatible with the screen-space rendering pipeline and does not introduce complexity into the fluid solver. Together, extraction and rendering technique, are able to provide state-of-the-art quality rendering and achieve better performance than classic methods alone.

Keywords: Screen-space fluid rendering, Real-time rendering, Particle based simulation..

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER

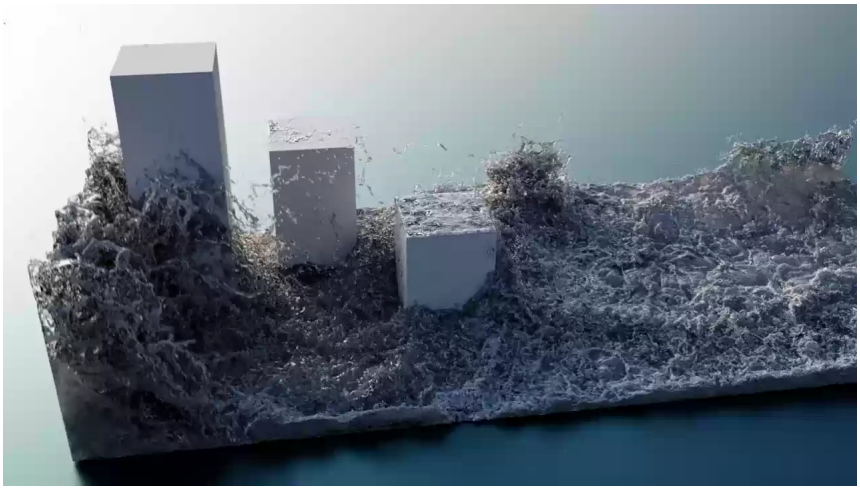
1

INTRODUCTION

In graphics, particles play a vital role in representing free-surface flow simulations, frequently used to track Liquid interface in Lagragian methods such as Smoothed Particle Hydrodynamics (SPH), (IHMSEN *et al.*, 2014), hybrid approaches such as the Fluid Implicit Particle (FLIP), (ZHU; BRIDSON, 2005) and more recently with the use of the Material Point Method (MPM). Although these methods have been successfully introduced in real-time environments, due to modern GPU's architecture and parallel potential, high quality rendering and animation of fluid in real-time remains a challenge. Traditionally the rendering of liquids consists of two steps: first, splatting the level-set function defined by the particles to a regular grid. Second, the liquid surface is reconstructed through a polygonization algorithm such as Marching Cubes, where the final mesh represents the isosurface. However this entire process is computationally expensive since the surface smoothness and coherence between frames requires a high resolution grid. Because of this most of the mesh based rendering is restrict to offline-renderers, systems wihtout time constraints that are allowed to generate high quality results, Figure 1 shows a image generated with Blender and the FLIP Fluids project where high quality rendering is obtained.

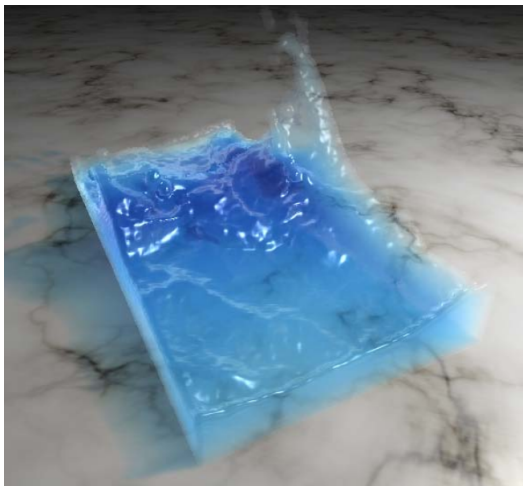
An alternative is to compute the surface directly from the particles in screen-space, without generating mesh. The screen-space technique consists in computing fluid surface directly from the geometric primitives (spheres or ellipsoids) representations of the particle system, extracting rendering properties such as depth and surface normal and applying a lighting formulation to the result. Because this process is done entirely on screen-space, it only generates the frontmost surface along the vieweing direction, although different methods have been introduced for capturing environmental properties, (SHAH; KONTTINEN; PATTANAIK, 2007), allowing for a much wider range of effects. Figure 2 shows an classic example of the screen-space method applied to a fluid simulation, (GREEN, 2010), combining the screen-space rendering of the surface with secondary effects

Figure 1 – Photorealistic FLIP simulation.



Source: Blender Flip Fluids.

Figure 2 – Screen-space fluid with secondary effect.



Source: (GREEN, 2010).

(shadows). Lighting, however, is constantly evolving as modern architectures allow for physically based algorithms to bring better results through improvements on ray tracing capabilities of the GPU and more advanced solution, (ANDERSSON *et al.*, 2019) and with alternative estimators, (BITTERLI *et al.*, 2020), however, physically based rendering has not yet been integrated with the screen-space method. The lack of a *real* geometry (mesh) makes it difficult that photorealistic algorithms, such as Path Tracing, can be correctly computed, forcing that most implementations of the screen-space method use some variation of the classic Blinn-Phong model.

In this work we present a novel and practical screen-space fluid rendering for particle-based methods that works with both GPU and CPU solvers, adding minimal changes to the pipeline proposed by Simon Green. Our method performs particle filtering only in a narrow-band around the boundary particles to provide smooth liquid surface. To build a reliable narrow-band for surface reconstruction we present a simple yet efficient approach to narrow-band extraction that does not introduce complexity to particle-based solvers. We show the effectiveness of our method against both boundary extraction and screen-space solutions in several examples, and we build on top of them so that their performance can be increased. Our method also decreases the amount of storage required to maintain simulation as we can achieve the same quality result using only narrow-band particles. In the next chapter we revisit the screen-space method and show its basic pipeline and present a few different commonly used strategies for surface reconstruction, in Chapter 3 we present several algorithms for boundary extraction and show why they cannot be used alone to reduce scene complexity for the screen-space method as well as introduce our approach to narrow-band extraction, we also present several different comparison between this approach and classic methods commonly found in literature and, finnaly, in Chapter 4 we inspect the problems introduced to the screen-space by using the narrow-band only and present our solution to it.

CHAPTER

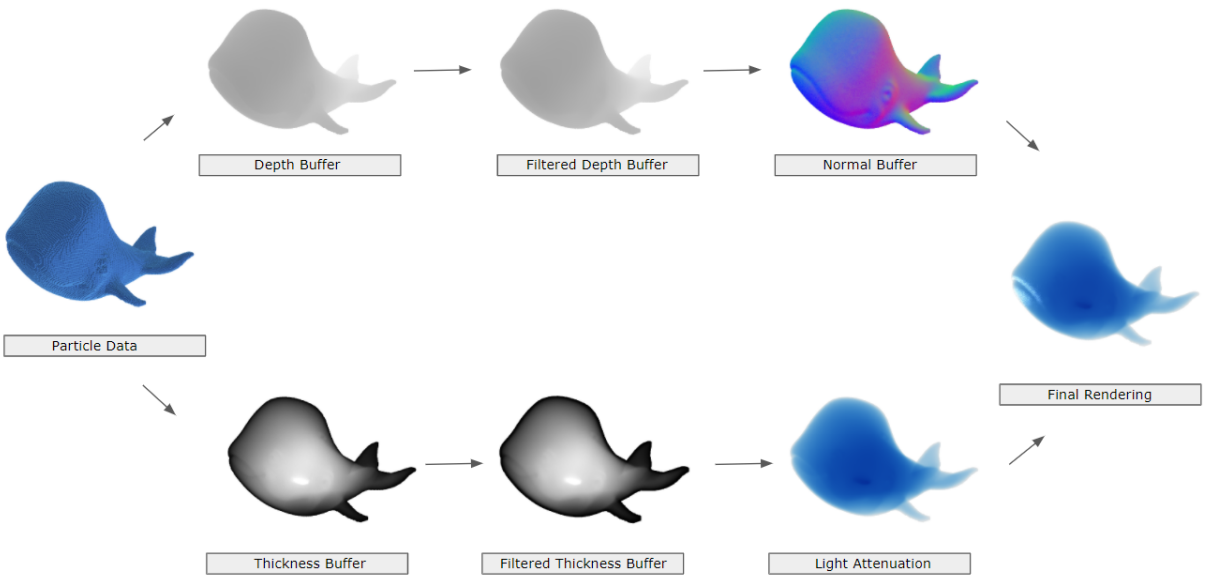
2

SCREEN-SPACE FLUID RENDERING IN A NUTSHELL

As we have mentioned in Chapter 1, the screen-space method is a fast and easy approach to fluid rendering that does not rely on mesh generation. In this chapter we present the original method as developed in (GREEN, 2010) and a few variations that were published along the years. Several methods have been developed targeting surface quality, in here we will not discuss all of them but will show a few alternatives to the original method. The screen-space pipeline is heavily based on image processing and beautifully simplifies the geometry handling of the rendering process, but at its core it rely on particle projection and particle splatting, both steps are affected by our work and will be discussed in the following chapters. The method also is already heavily studied in several different works, because of this we will not show the complete mathematical derivation of each step as most of the steps are already well discussed in the Computer Graphics community and/or are simple techniques vastly used and considered standard. In this chapter we'll focus on introducing the original method and present already developed strategies that have been proven to be effective when integrating to the original pipeline.

The main idea behind the screen-space method is to use particle projection as a way to estimate the fluid surface. By getting the particles distance to the viewer one can estimate a surface by smoothing this *depth distribution*, from the smoothed depth it is possible to reconstruct properties such as positions and normals and perform a complete geometric rendering pipeline on the input buffers that hold the depth values. The smoothing of the depth values creates a *visual* perception of the fluid and eliminates particles from the image, Figure 3 shows the complete pipeline of the method. Note that in a real application this pipeline would probably be much bigger as light systems can require more information about geometry depending on the quality required. Effects such as dynamic shadows, scene reflection/refraction and caustics, for example, introduce its

Figure 3 – Screen-Space pipeline.



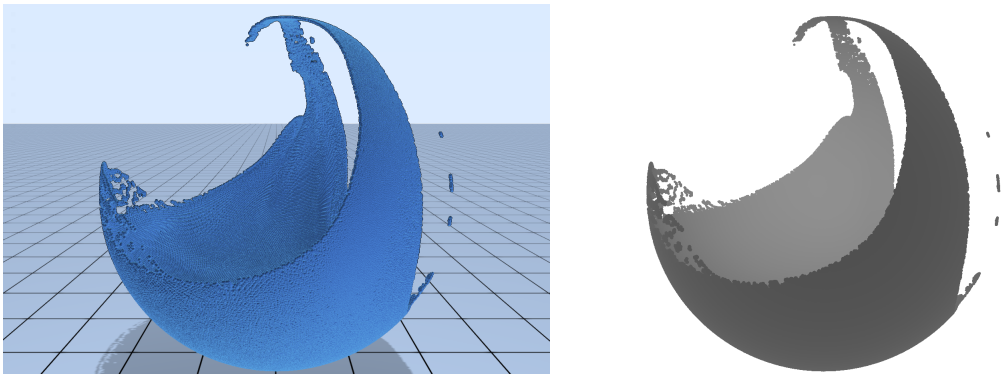
Source: Elaborated by the author.

own challenges and complexity in the middle of this pipeline, Figure 3 is what we call the *minimal* steps required to create a representation of the fluid, in the rest of this work we will only consider the steps showed in Figure 3 and, even if some of the images later on will have more effects applied to them, is what we will refer as the *original pipeline/method*.

In the graphics context, the first stage (depth generation) is solved in a single execution where each particle in the fluid simulation is rendered as a point sprite and the resulting quad is transformed into a sphere by discarding pixels outside the [0,1] range. Figure 4 shows the point sprite render of all particles in a fluid simulation (left). Note that even if Figure 4 shows solid particles, it is not necessary to completely render them to a *Color Buffer* as, at this point, we are only interested in the depth value of each particle with regards to the rendering viewpoint, one can simply dump the results of depth computation into a *Depth Buffer* for future steps of the pipeline. A typical *Depth Buffer* is showed on the right of Figure 4. Because of the way the rendering API used (OpenGL) defines distances for each fragment, we see darker values for closer geometry and lighter values for distant objects. Depth values are generated by projecting every fragment of the point sprite on top of a sphere, projecting the sphere into *Clip Space* and then clamping the projection depth to the interval [0,1].

Once the *Depth Buffer* is generated we have a simple estimate of the fluid surface where we could generate normal vectors for lighting purposes, however in order to obtain a better (*smooth*) surface we need a filter that can remove the high variation that can

Figure 4 – Point sprite render of a fluid simulation (left) and the resulting depth buffer (right).



Source: Elaborated by the author.

happen depending on particle to viewpoint distance, particle radius and other factors that influence the overall image quality of the rendering. In the original pipeline the author suggests the use of a filter based on two weights:

1. Pixel distance: When pixels that are close present too much variation in depth values, the final image gives the perception that the fluid is composed by *multiple layers* and in turn have less quality;

2. Particle depth variation: Against Item 1 however, one must be careful to not lose all depth perception and turn the fluid flat, so if depth values change *too much* for neighboring pixels we wish to not completely remove the difference but instead to *connect* them giving the sensation of transition in the fluid surface.

A two dimensional filter that can be applied that considers the previous items is given bellow. For every pixel the distance to its neighbors r is accumulated through the the weight w_1 and the change in depth is captured by the the difference d and accumulated through w_2, k_1 and k_2 are constant values that help shape the way the filtering is performed.

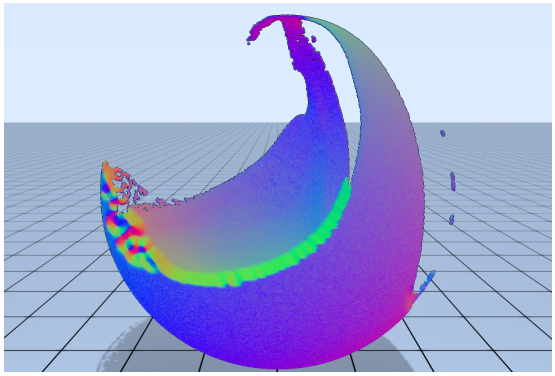
$$w_1 = k_1 e^{-r^2}$$
$$w_2 = k_2 e^{-d^2}$$

Results of this formulation can be seen in Figure 5, note how this is a very subtle filter that does not equally change the way the surface is perceived. With the final Depth Buffer, after filtering, we can reconstruct the normal vectors required for lightning, this process can be achieved in several ways but in the original work a simple central difference method is applied generating the *Normal Buffer*, results can be seen in Figure 6 where colors direct relate to the direction of each normal.

Figure 5 – Filtered Depth Buffer.

Source: Elaborated by the author.

Figure 6 – Normal vectors computed from final *Depth Buffer*.



Source: Elaborated by the author.

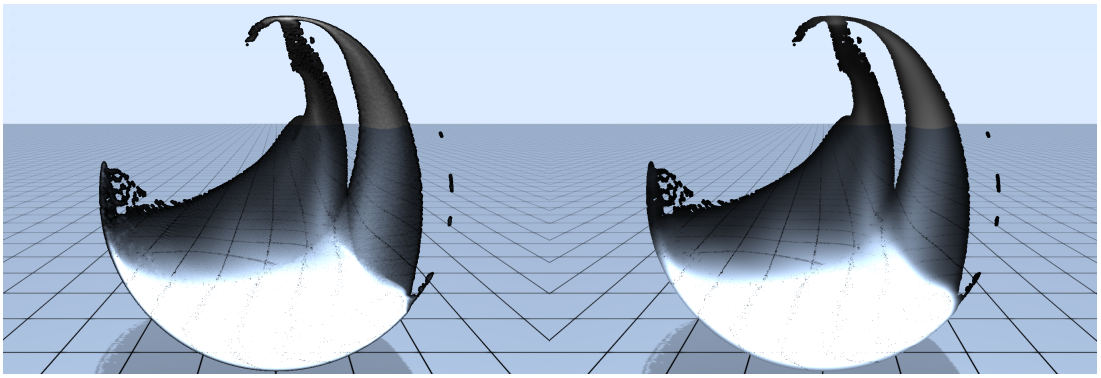
Before going into the lighting step we need to introduce an element that can represent volume in the final image, this will drastically enhance the quality obtained so far and helps reducing the flat look the method generates, this volume can then be rendered applying some absorption relation, in the original work it is used the *Lambert-Beer* relation:

$$T = e^{-\tau} \quad \text{with} \quad \tau = \sum_i \sigma_i \int_0^l \eta_i(z) dz \,, \tag{2.1}$$

where T is the transmittance of the material, τ the optical depth that is related to the amount i of samples which have attenuation σ_i and number density η_i.

The Equation (2.1) is not easy to solve, in order to simplify its solution the method assumes there exists only one sample and that its density is constant across the path

Figure 7 – Alpha blending without filter (left), with filter (right).



Source: Elaborated by the author.

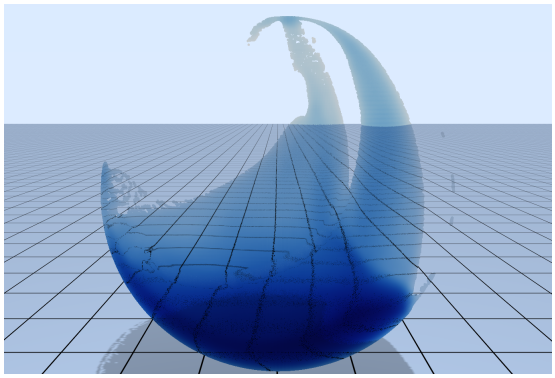
length l, we can then write that:

$$\begin{aligned} T &= e^{-\sum_i \sigma_i \int_0^l \eta_i(z)dz} \\ &= e^{-\sigma \int_0^l \eta(z)dz} \\ &= e^{-\sigma\eta l} = e^{-kl}, \end{aligned}$$

where k becomes a constant factor for controlling the appearance of the fluid. This simplification depends however on one unknown value, the path length l which represents the distance light "travels" inside the sample. The screen-space method smartly estimates this value by rendering all particles again but this time performing alpha blending on the output buffer, this will generate different values for regions of fluid that are made of a different particles amount, we use this accumulated contribution as a value that represents how much a ray of light would need to travel through a specific pixel to get out of the fluid. Because of the nature of the rendering (particles as spheres) the output of the blending operation will again need to be smoothed so that we can generate volume across all regions of the surface, Figure 7 shows both results, the original alpha blending (left) and the alpha blending after filtering (right), the output of this estimation is often referred as the *Thickness Buffer*.

Now that we have an estimation of the light absorption and have normal vectors for each point in the fluid surface we can make a composition pass that performs lighting. As research improves lighting schemes, the overall quality of this step can be improved, however in the original work a simple *Blinn-Phong* method is applied, the final result can be seen in Figure 8. From this stage different types of composition can be performed, we could potentially integrate the scene around the fluid using reflection and refraction effects, caustics also contribute to the final quality of the image, *noise* can also be used to disturb the normal vectors so that lightning is not so regular and many other approaches. To summarize, the screen-space method is a very well crafted method to render particle based

Figure 8 – Final composition of the screen-space pipeline.



Source: Elaborated by the author.

fluids without the need to reconstruct surface using expensive algorithms, its pipeline consist of the following steps:

1. Render all particles to a *Depth Buffer*;
2. Smooth the *Depth Buffer* by using a two weight bilateral filter;
3. Reconstruct the normal vectors from the smoothed *Depth Buffer* generating the *Normal Buffer*;
4. Render all particles again performing alpha blending to get a volume estimation, the *Thickness Buffer*;
5. Filter the *Thickness Buffer* to make sure the volume is smooth;
6. Compose the *Normal Buffer* and the *Thickness Buffer* using the *Lambert-Beer* relation and any lighting model to obtain the final image.

Most of the pipeline presented so far is solved entirely in screen-space (where the name comes from) with exception of the particle projection, during depth map construction, and particle splatting, during thickness generation. However the most expensive part of the pipeline (in most situations) is depth map smoothing for the surface generation step. This happens because in order to correctly smooth the *Depth Buffer* we must iterate and apply a 2D filter several times, sometimes one can get good results with a single step of the filter presented before, but in order to have a high quality surface this must be avoided. In order to really achieve interactive frame rates (60+) we must avoid using a high iteration count. In the next sections we discuss a few different approaches developed that in a way attempt to increase quality of the original work and provide different views of the fluid rendering.

2.1 Curvature Flow

The method we are going to discuss was presented in (LAAN; GREEN; SAINZ, 2009), which we'll refer as *Curvature Flow* for simplicity. It is an interesting method that increases the overall quality of the surface generated by the *Depth Buffer* smoothing we have discussed in the initial part of this chapter. The main idea for this approach is that instead of using the Bilateral Filter, initially proposed, we can view the problem of smoothing the *Depth Buffer* as a problem of minimizing the *Curvature* of the fluid surface. This process can be done if one evolves the fluid surface along its normal direction with a speed that depends on the magnitude (and sign) of the mean curvature of the surface, we define that the curvature H is a function of the depth z over time:

$$H = \frac{\partial z}{\partial t}$$

However it also known that the *mean curvature* is defined as the divergence of the unit normal of a surface:

$$2H = \nabla \cdot \hat{n} \tag{2.2}$$

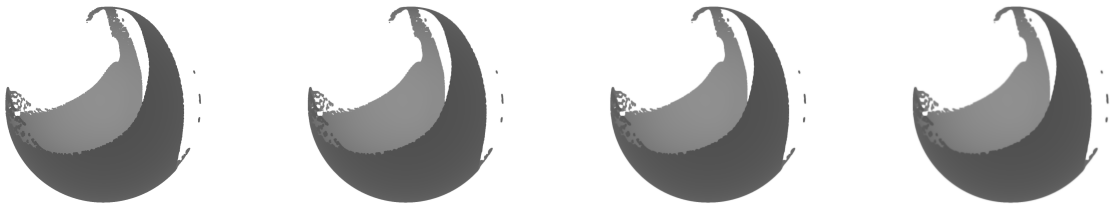
In this particular application the normal vector at each pixel can be obtained by inverting the projection transformation of the values of the *Depth Buffer* and performing the cross product in *view space*, it is possible to show that if V_x and V_y are the dimensions of the viewport, and F_x and F_y are the focal length in the x and y coordinates, respectively, than the normal is given by:

$$\hat{n} = \frac{(-C_y \frac{\partial z}{\partial x}, -C_x \frac{\partial z}{\partial y}, C_x C_y z)^T}{\sqrt{D}}$$
$$D = C_y^2 \Big(\frac{\partial z}{\partial x}\Big)^2 + C_x^2 \Big(\frac{\partial x}{\partial y}\Big)^2 + C_x^2 C_y^2 z^2$$
$$C_x = \frac{2}{V_x F_x}, C_y = \frac{2}{V_y F_y}$$

Full derivation of the previous relations is quite extensive and can be viewed in its full on the original paper. This set of equations look difficult to solve, however they are not. The values of V_x and V_y are immediately obtained from the rendering system as they are encoded in the viewport being used, F_x and F_y (when using OpenGL) can be leveraged directly from properties of the projection matrix and can easily be obtained if one recall that, usually, in OpenGL the projection matrix is given by:

$$P = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Figure 9 – Different steps of the Curvature Flow iteration. From left to right: 1, 30, 60 and 120 iterations, respectively.



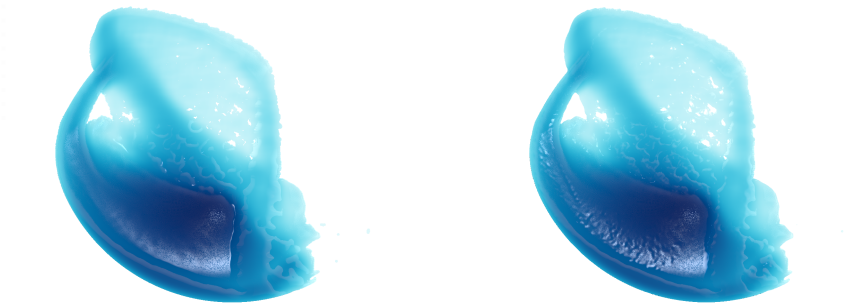
Source: Elaborated by the author.

where the focal length F_x is the described by e and F_y by e/a, one can refer to (SELLERS; WRIGHT; HAEMEL, 2015) for a comprehensive analysis of the usual OpenGL rendering process. For the differential values of z we can perform central differences. This means we can return to Equation (2.2) and compute the value of the mean curvature:

$$
\begin{aligned}
2H &= \frac{\partial \hat{n}_x}{\partial x} + \frac{\partial \hat{n}_y}{\partial y} = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}} \\
E_x &= \frac{1}{2}\frac{\partial z}{\partial x}\frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2} D \\
E_y &= \frac{1}{2}\frac{\partial z}{\partial y}\frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2} D
\end{aligned}
$$

This makes the computation of a **single** value of the mean curvature not very difficult to compute in a *fragment* pass. However the method does not rely on a single value of mean curvature but in a **evolution** scheme of the depth values obtained, this means that for correct implementation of this approach one needs to iterate through this process multiple times, recompute the mean curvature and accumulate them to a final Depth Buffer before usage. Results can be seen in Figure 9 where we show different steps of the iteration. Even if the surface does look better and it in fact generates a better final image (Figure 10), this approach comes with a high cost. Depending on particle configuration the number of iterations can be as high as 120 and the screen-space pipeline then starts to spend all its time on Depth Buffer smoothing, this makes that even if the Bilateral Filter does not generates such a good surface, it heavily outperforms the Curvature Flow method in modern hardware and is therefore a better approach when seeking high performance. The calculation steps required for all the iterations force the pipeline to constantly perform GPU/CPU synchronizations as it is not possible to completely solve a single pixel without memory barriers, the implementation is also more complex than the original method making this approach not so used.

Figure 10 – Comparison between Bilateral Filter (left) and Curvature Flow (right).



Source: Elaborated by the author.

2.2 Plane Fitting

Another approach for surface reconstruct for screen-space, with a lower iteration count, is by using the *Plane Fitting* method, (IMAI; KANAMORI; MITANI, 2016). This method attempts to fit a plane Π_i for each pixel i using properties of the neighborhood of i. Consider the pixel i with position p_i in eye space, Plane Fitting will define for this pixel a plane Π_i with equation $d_i = \mathbf{n}_i^T \mathbf{x}$, where $\mathbf{n}_i$ is the normal vector at Π_i and d_i the distance from the origin of the system. We wish to compute the distance z_i that a ray, going from this view point into the middle of pixel i, would need to travel in order to hit Π_i. For each pixel i we define the plane Π_i by minimizing the following error:

$$E = \sum_j w_{ij} \left(\mathbf{n_i^T} p_j - d_i \right)^2 \quad \text{with} \quad w_{ij} = e^{-(z_j - z_i)^2/\sigma^2} \tag{2.3}$$

where σ is a user defined parameter. The iteration process should stop when $\partial E / \partial d_i$ becomes close to zero. In order to solve Equation (2.3) we must define the normal vector at each pixel i during depth smoothing. The original authors suggest that for normal estimation we use:

$$\mathbf{n}_i = \frac{\sum_j w_{ij} \tilde{\mathbf{n}}_j}{|\sum_j w_{ij} \tilde{\mathbf{n}}_j|} \tag{2.4}$$

where $\tilde{\mathbf{n}}_j$ is the real normal value for the particle that lies on pixel j. The sum in Equation (2.4) is very common in both the screen-space pipeline and in fluid simulation where a value is determined by taking a weighted average of neighboring values. Once the normal vector at i is computed we can create a first estimation of the distance d_i using:

$$d_i = \mathbf{n_i^T} \bar{p}_i \quad \text{with} \quad \bar{p}_i = \frac{\sum_j w_{ij} \mathbf{p}_j}{\sum_j w_{ij}} \tag{2.5}$$

and finally the output z_i value is given by performing ray-plane intersection with the estimated plane:

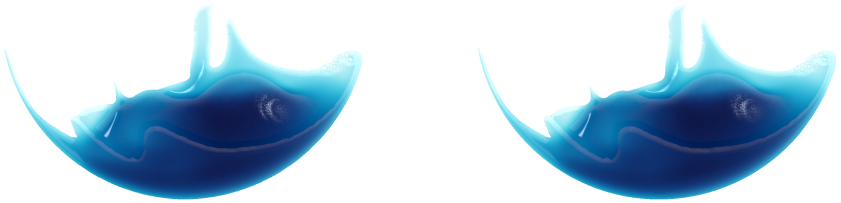
$$z_i = \frac{d_i}{\mathbf{n}_i^T \mathbf{v}} v_z \tag{2.6}$$

where $\mathbf{v}$ is the direction of the ray in eye-space. With the previous equations in mind, a single step of the Plane Fitting method for the pixel i is performed in the following steps. Note how even if there are multiple sums over neighborhood required they can all be computed in a single step.

1. Compute the viewing direction $\mathbf{v}$ in eye-space for pixel i;
2. For each neighboring pixel j do
   - Compute and accumulate the product $w_{ij}\tilde{\mathbf{n}}_j$;
   - Compute and accumulate the product $w_{ij}\mathbf{p}_j$;
3. Compute $\mathbf{n_i}$ using Equation (2.4);
4. Compute $\bar{p}_i$ and d_i using the equations given in Relation (2.5);
5. Output z_i using Equation (2.6).

The output distance z_i computed in the last step is provided again to the depth smoothing solution for the next step, making this process iterative. But in contrast to the Curvature Flow method showed earlier, the Plane Fitting method does not need too many iterations. Usually it starts to give a good approximation with 3 or 4 iterations, note however that the Curvature Flow method only relies on the direct neighbors of the center pixel i while the Plane Fitting (and other methods mentioned here) need to sample the depth buffer multiple times in a window of resolution sometimes large (15). Figure 11 shows a comparison of the Curvature Flow method (left) and the Plane Fitting (right), note how they look very close and yet when rendering this specific frame, the Plane Fitting method heavily outperformed Curvature Flow, where Curvature Flow took 25ms Plane Fitting performed in 10ms.

Figure 11 – Comparison between Curvature Flow (left) and Plane Fitting (right).



Source: Elaborated by the author.

2.3 Narrow Range Filter

The *Narrow Range Filter* method, just like the Curvature Flow approach, introduces a different scheme for depth value smoothing in order to improve surface and overall final image quality. This method was first introduced in (TRUONG; YUKSEL, 2018) and is, today, one of the best choices when applying the screen-space pipeline, just like other approaches it also iterates a few times over the depth values in order to smooth the surface but usually around a low number (between 1 and 3). The pixels z_i (output depth) values are computed using the same scheme as before:

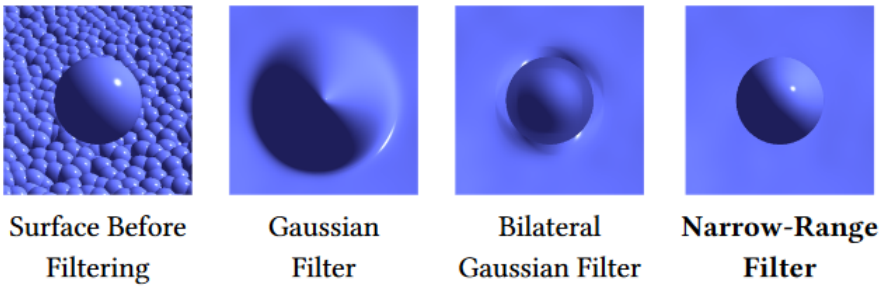
$$z_i = \frac{\sum_j \omega_{ij} f(z_i, z_j)}{\sum_j \omega_{ij}}$$

where ω_{ij} is the weight related to the filtering operation and f is the *clamping function* given by

$$f(z_i, z_j) = \begin{cases} z_j & \text{if } z_j \geq z_i - \delta \\ z_i - \mu & \text{otherwise} \end{cases}$$

with μ and δ user-defined parameters satisfying $\mu \leq \delta$. The main idea of the clamping function f is that a given pixel is not allowed to differ too much during filtering, if $z_i - z_j$ becomes too big, i.e.: bigger than δ, a clamped z value is used instead: $z_i - \mu$. This is performed in order to prevent blending of depth values for particles that are too far apart, the clamping also works by creating smoothed edges between the particles and not ignoring their presence, Figure 12, taken from the original work, shows a comparison of the different methods and the different edges each one generates.

Figure 12 – Different edges generate by smoothing functions.



Source: (TRUONG; YUKSEL, 2018).

In order to prevent particles from the foreground to distort background particles during the filtering operation the method also makes a small change to the way the gaussian weights ω_{ij} are calculated:

$$\omega_{ij} = \begin{cases} 0 & \text{if } z_j > z_i + \delta \\ G(p_i, p_j, \sigma_i), & \text{otherwise} \end{cases}$$

with p_i and p_j being the pixel coordinates in 2D and G the standard *Gaussian* function and σ_i the standard deviation parameter. The weights ω_{ij} however do not completely work depending on viewing direction, pixels that are occluded could suffer filtering only by part of the filter kernel because of the term $z_j > z_i + \delta$. To avoid this problem the author adds a small term for *bias correction* where pixels p_k that are on the opposing side of pixel p_j also be considered during the clamp operation, the weight ω_{ij} can be rewritten as:

$$\omega_{ij} = \begin{cases} 0 & \text{if } z_j > z_i + \delta \text{ or } z_k > z_i + \delta \\ G(p_i, p_j, \sigma_i), & \text{otherwise} \end{cases}$$

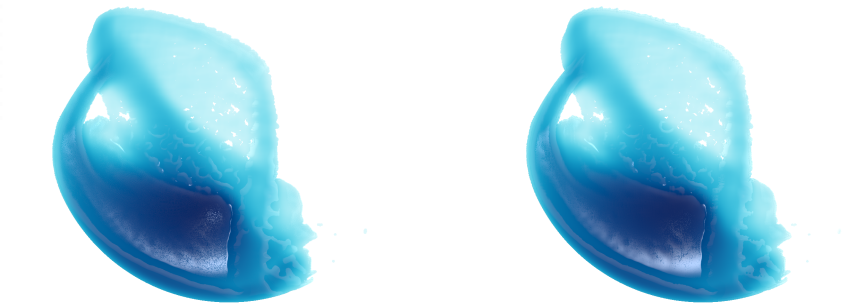
Finally the screen-space filter kernel size at pixel i is computed as $3\sigma_i$, where

$$\sigma_i = \left\lceil \frac{H\sigma}{2|z_i|tan(\alpha/2)} \right\rceil$$

and H is the the vertical resolution being used, σ a fixed filter size and α the camera's field of view.

The previous formulation provides a simple and fast way to generate a smooth surface for the fluid in screen-space, however one thing that becomes troublesome with the method is defining the right value for δ. Too small δ values might cause the filter to mistakenly classify depth discontinuities in several regions where they do not exist, on the other hand a large δ might filter together depth discontinuity and loose the quality of

Figure 13 – Comparison of the original screen-space Bilateral Filter(left) and the Narrow Range Filter (right).



Source: Elaborated by the author.

the filtering solution. Because of this the author also introduces a *dynamic δ adjustment* where the previous weight condition is replaced, again, with

$$z_i + \delta_{high} \geq z_j \geq z_i - \delta_{low}$$

where at the beginning we have $\delta_{low} = \delta_{high} = \delta$ and as the filter accepts pixels j the values get automatically updated according to:

$$\delta_{low} = \max(\delta_{low}, z_i - z_j + \delta)$$
$$\delta_{high} = \max(\delta_{high}, z_j - z_i + \delta)$$

Figure 13, shows a comparison of the original method proposed in (GREEN, 2010) and the Narrow Range Filter approach. Note how even if both images look very close, the surface generated through the Narrow Range Filter generates a smoother final rendering and can be generated in the same amount of iterations, contrary to the Curvature Flow method.

As we have mentioned earlier, the screen-space method does not impose any condition on lighting and type of depth smoothing filter that must be used, in this work all following images generated using the screen-space pipeline use the Narrow Range filter presented before. The choice of filter can heavily impact the performance of the screen-space pipeline, for example: using the Curvature Flow filter with a high iteration count will make the entire pipeline be slower than using a low iteration filter such as the Narrow-Range. The method we will present for performance - the Layered Neighboorhood Method (LNM), heavily increases performance but if the filtering method becomes too expensive then even with our improvements, rendering will be compromised. In the next chapter we introduce several boundary extraction methods and show the problem they introduce in the screen-space pipeline as well as the LNM formulation, our solution.

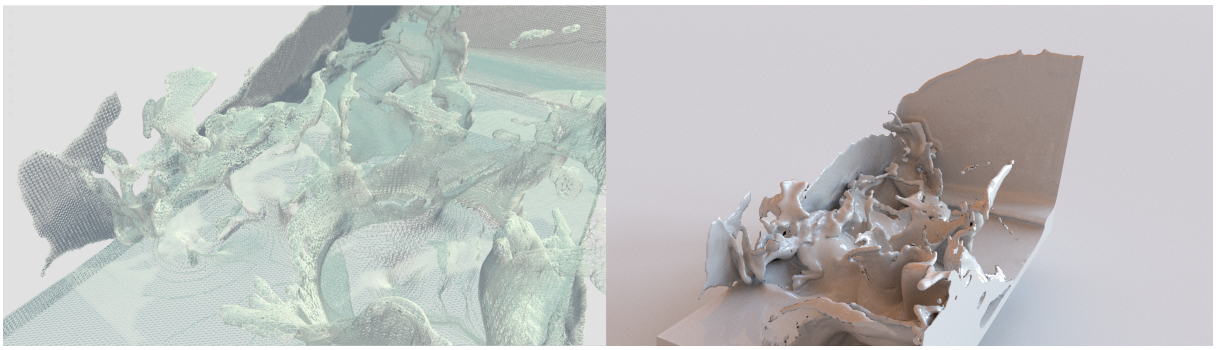
CHAPTER

3

BOUNDARY METHODS AND THE SCREEN-SPACE

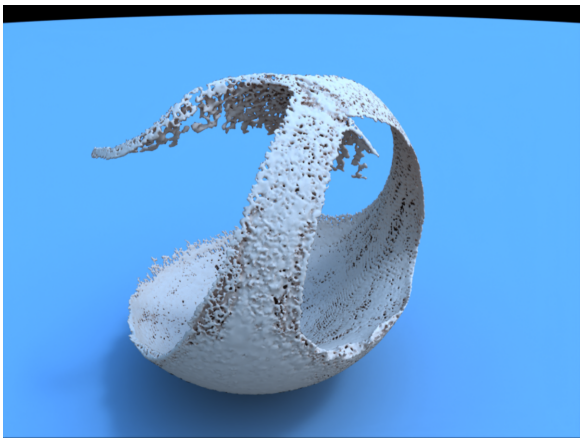
When rendering fluids there exists a variety of choices in the way surface reconstruction can be computed. As we mentioned in the previous chapter the screen-space framework allows for various methods for surface reconstruction under different depth filtering solutions. Mesh based methods can leverage more robust algorithms such as Marching Cubes, (LORENSEN; CLINE, 1987), with many different signed field distance (SDF) formulations, and Poisson Surface Reconstruction, (KAZHDAN; BOLITHO; HOPPE, 2006). Figure 14 shows the result of executing the Marching Cubes algorithm after a frame of fluid simulation (left) and the *Path Tracing* result (right), Figure 15 shows the final rendering after Poisson Surface Reconstruction. Both methods allow for complete execution with the use of the narrow band particles only, Poisson only requires that normal vectors be provided and Marching Cubes only requires that information about the interior fluid results in negative values for the SDF during execution. We wish to develop a method that allow for similar approaches for the screen-space pipeline so that we can increase overall performance of the final application allowing for faster rendering and less memory requirements. On the following sections we present in detail commonly used methods for boundary extraction. We present the theory and implementation behind each approach and show why, when alone, they cannot be use to generate a particle set representation of the narrow band that satisfies the requirements for surface reconstruction using screen-space depth filtering solutions. This is a important part because our solution can work alongside them refining the boundary and it is usefull to understand how each of these methods work. At the end of this chapter we provide a complete solution that is fast, scalable and can be easily implemented in any particle-based fluid simulators.

Figure 14 – Fluid surface generate using the Marching Cube algorithm, left, and a realistic result, right, obtained using the Path Tracing algorithm.



Source: Elaborated by the author.

Figure 15 – Path Tracing result of a fluid frame reconstructed using Poisson Surface Reconstruction.



Source: Elaborated by the author.

3.1 Boundary Extraction

Fluid boundary extraction is a large topic and there exists a lot of different approaches that can be used. Most methods rely on the *Smoothed Particle Hydrodynamics* (*SPH*) formulation and take advantage of one or more geometric properties, so we begin our discussion of boundary extraction by recalling the basics of the *SPH* approach to fluid simulation.

When solving fluid simulation, using the *SPH* method, properties such as density and viscosity are computed by considering a neighborhood of particles around a central particle. Take Figure 16 for example, we are trying to solve fluid dynamics on the *ith*

particle (red), to do so we take the neighborhood around particle i and compute *field values* for it. Let $\mathbf{x}_k$ be the position vector of the *k-th* particle and j a particle in the neighborhood of particle i, we construct the *Smoothing Function*, also known as the *SPH kernel*, as the function W that can *spread* values of this field into neighboring particles. To achieve this property we impose a few conditions into the kernel W. We say that the kernel is a *symmetric* function, if $\mathbf{r} = \mathbf{x}_j - \mathbf{x}_i$ than $W(\mathbf{r}) = W(\|\mathbf{r}\|)$.

The kernel function is also said to be *normalized*, i.e.: it satisfies Equation (3.1). This function usually contains its peak at the origin where $\mathbf{r} = 0$ and than decays monotonically to 0 as we get further away, increasing $\mathbf{r}$. The distance $\|\mathbf{r}\|$ from where the kernel becomes zero is usually referred as the *kernel radius*. With larger kernel radius, more particles perceive contribution from particle i and, on the other hand, with smaller kernel radius the neighborhood of i becomes smaller.

$$\oint W(\mathbf{r})\, d\mathbf{r} = 1 \tag{3.1}$$

By utilizing the previous definition of *kernel* function the *SPH* than states that for a field Φ at particle i, with position $\mathbf{x}_i$, the field value $\Phi(\mathbf{x}_i)$ is given by

$$\Phi(\mathbf{x}_i) = \sum_j m_j \frac{\Phi_j}{\rho_j} W(\mathbf{x}_j - \mathbf{x}_i) \tag{3.2}$$

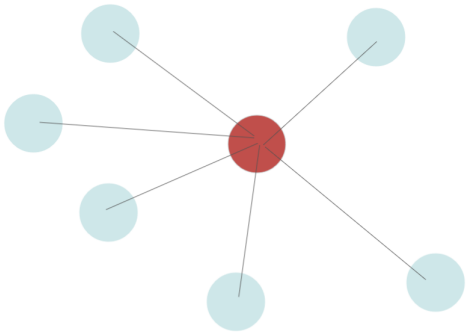
where m_j, Φ_j and ρ_j are mass, field value and density at particle j, respectively. Equation (3.2) states that in order to compute the value of Φ for particle i is sufficient to take all particles j around i and accumulate their contribution weighted by the *kernel*. This type of computation, based on neighboring search, is very common in fluid simulation and is present in all methods presented here. It is the cornerstone of particle-based solvers and as far as implementation goes it usually has the format of Algorithm 1, where *GetNeighbors* is a function that can obtain the neighbors of particle index i and *ComputePairContribution* is a function that can compute the internal value of the sum in Equation (3.2).

Algorithm 1 – Basic neighboring process

```
1: for i ← 0 to NumberOfParticles do
2:     initialize Φ_i
3:     B ←GetNeighbors(i)
4:     for particle j ∈ B do
5:         Φ_i = Φ_i+ComputePairContribution(i, j)
6:     end for
7:     process Φ_i
8: end for
```

For fluid particle boundary extraction the previous formulation is all that is required, but a complete discussion on fluid dynamics using SPH is available in (KIM, 2016). Next we discuss several boundary extraction strategies.

Figure 16 – Particle neighborhood



Source: Elaborated by the author.

3.1.1 Color Field Method

One of the simplest methods for getting the boundary of a particle-based simulation is to consider the following function $\mathcal{C}$, known as the *Color Field*.

$$\mathcal{C}(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} \text{ is inside the fluid;} \\ 0, & \text{otherwise.} \end{cases}$$

In the *SPH* representation the value of $\mathcal{C}$ is always 1 for every particle and 0 where no particles can be found. Based on Equation (3.2) we can write for every position $\mathbf{x}$ that

$$\mathcal{C}(\mathbf{x}) = \sum_j m_j \frac{\mathcal{C}(\mathbf{x}_j)}{\rho_j} W(\mathbf{x}_j - \mathbf{x})$$

however any point in space that we can perform sampling ($\mathbf{x}_j$) will always have $\mathcal{C}(\mathbf{x}_j) = 1$ by definition, so we can rewrite the previous relation as

$$\mathcal{C}(\mathbf{x}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{x}_j - \mathbf{x})$$

but even if we can solve $\mathcal{C}(\mathbf{x})$ for any $\mathbf{x}$, the value is not of much interest on itself since we already know it will be 1 inside the fluid and 0 outside. A much more interesting value can be found if one is to observe the function $\mathcal{S}(\mathbf{x}) = \nabla\mathcal{C}(\mathbf{x})$, where at each position $\mathbf{x}$ it assigns the gradient of the color field $\mathcal{C}(\mathbf{x})$. Because of the nature of the gradient operator, we expect that the values of $\mathcal{S}$ to be nearly 0 everywhere except where changes happen in $\mathcal{C}$ but the only possible way $\mathcal{C}$ can change is by going in or out of fluid, meaning we can obtain the boundary by analysing places where $\mathcal{S} \neq 0$. Applying the *SPH* formulation to $\mathcal{S}(\mathbf{x})$ we can write that

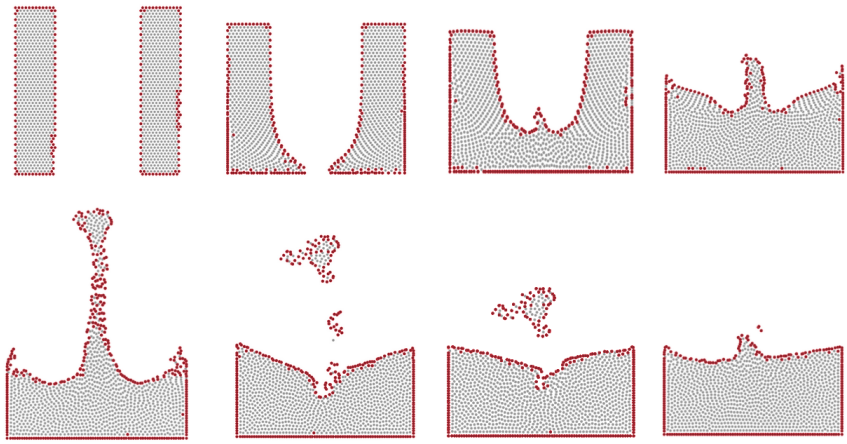
$$\mathcal{S}(\mathbf{x}) = \nabla\mathcal{C}(\mathbf{x}) = \sum_j m_j \frac{1}{\rho_j} \nabla W(\mathbf{x}_j - \mathbf{x}) \tag{3.3}$$

where ∇W is the gradient of the chosen *kernel* W and we say $\mathcal{S}$ is the *free surface of the fluid*. Implementation of Equation (3.3) is done by using the Algorithm 2 where one would, for every particle, compute the value of $\mathcal{S}$, accumulating the internal part of the sum in Equation (3.3), and if the value obtained is large the particle is then classified as a boundary particle.

Algorithm 2 – Color Field boundary classification

```
1:  procedure ColorField(P,threshold)        ▷ Boundary S of the particle set P
2:      S ← {}
3:      for particle p_i ∈ P do
4:          S_i ← 0
5:          B ←GetNeighbors(p_i)
6:          for particle p_j ∈ B do
7:              S_i ← S_i + (m_j/ρ_j) ∇W(x_j − x_i)        ▷ Sum of Equation (3.3)
8:          end for
9:          if ||S_i|| ≥ threshold then        ▷ Check if S_i ≠ 0 based on threshold
10:             S ← S ∪ {p_i}
11:         end if
12:     end for
13:     return S
14: end procedure
```

Figure 17 – Boundary detected using the *Color Field* method in 2D

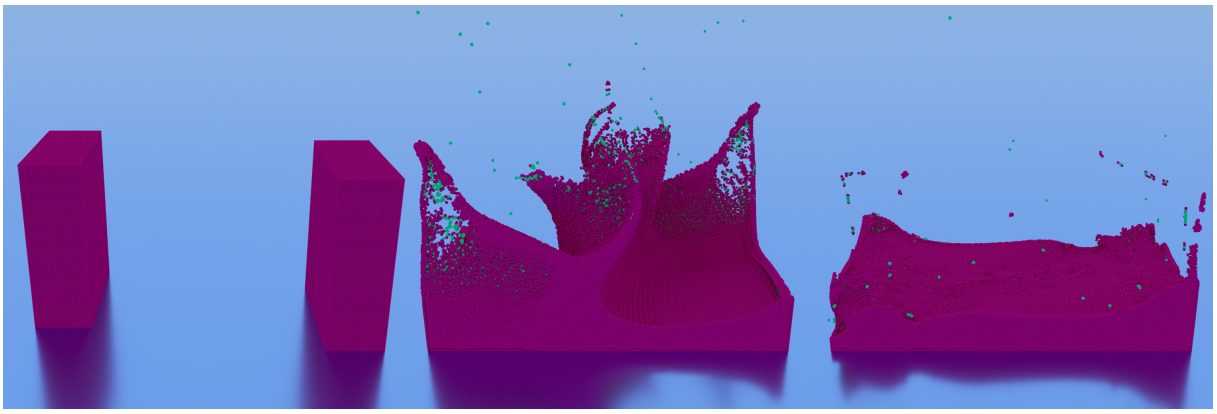


Source: Elaborated by the author.

The *Color Field* method provides a fast way to obtain boundary particles and solvers that implement *SPH* should easily integrate this scheme of computations, Figure 17 shows several frames of a 2D fluid simulation of the *Double Dam Break* scene. Boundary particles (red in Figure 17) and interior particle (green) are correctly computed and the result is somewhat accurate. This method has a few advantages over other boundary methods, as we mentioned earlier it is fast, probably the fastest one, it only relies on sampling the particle neighborhood once for every particle and in a single pass can mark particles as in

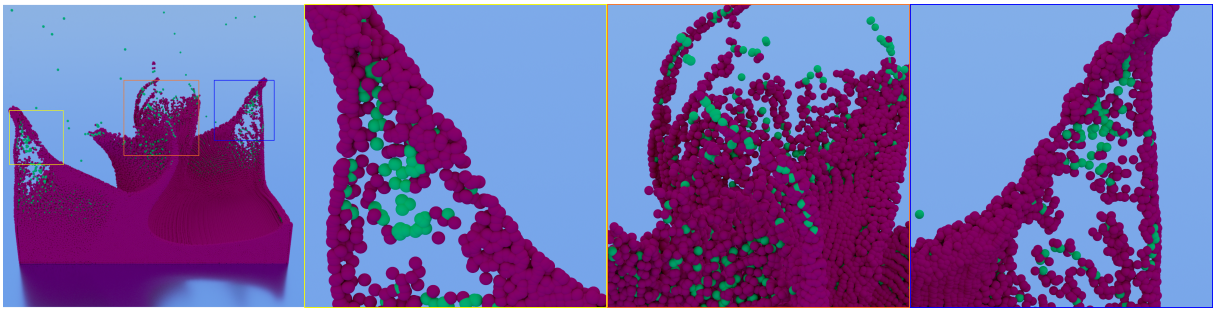
boundary or not. The method extends trivially to 3D because all computations remain the same and no operator requires change, this simplicity makes it very appealing to use. Figure 18 display the same configuration with the *Double Dam Break* but this time in a 3D environment. Again, particles in red are classified as boundary particles and particles in green are interior to the fluid body. While the methods is simple and very fast it is also unreliable. The gradient that is analysed is not enough to reliably detect boundary particles as the particles at the boundary suffer from insufficient neighborhood (discussed in Section 3.1.3). Figure 19 displays one frame of this simulation with splash regions highlighted. These regions are essential for a consistent redering of the fluid simulation making this method unsuitable for the screen-space method. We can also see from the 2D case that sometimes the particles selected as boundary are not actually the proper ones (Figure 17, third frame from top right). Some methods build on top of the *Color Field* idea with additional information for a better boundary extraction, (HE *et al.*, 2012), which we'll discuss in the next section.

Figure 18 – Boundary detected using the *Color Field* method in 3D



Source: Elaborated by the author.

Figure 19 – Double Dam Break with splash regions highlighted.



Source: Elaborated by the author.

3.1.2 Asymmetry Value Method

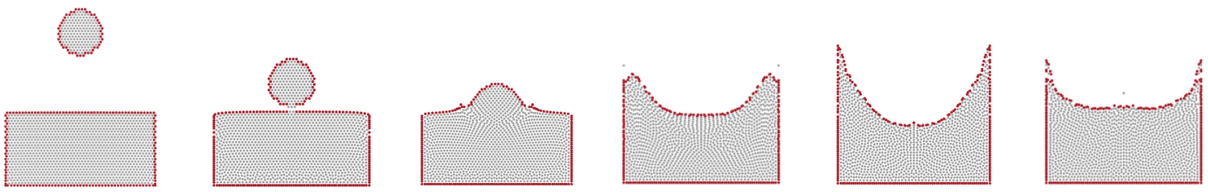
This method was developed in (HE *et al.*, 2012) and is part of a bigger process where the authors are describing a new method for fluid simulation with focus on fluid-solid interactions. However, the approach developed for boundary extraction is interesting and performs very well and since our focus is to find a boundary method that can fit the screen-space framework, it is a valid approach. The method does not actually have a name, but it depends on a value the authors call *Asymmetry Value* that is present in all particles in the fluid simulation, so we'll refer to this method as the *Asymmetry Value Method*.

In *Staggered Meshless Solid-Fluid Coupling*, the authors identified that there exists a relation between particles that are at the boundary of the fluid body and their distribution, and so created the notion of *asymmetry of a particle*. In the context of the boundary extraction, this value represents how *spread* one particle is from its neighbors and that particles that are *too isolated* must be at the boundary. The asymmetry value $\mathcal{A}_i$ of particle i of the fluid is determined by computing the distance from particle i to the *kernel weighted* position of the center of all neighbor particles j to i, as in Equation (3.4).

$$\mathcal{A}_i = \left|\mathbf{x}_i - \sum_j \mathbf{x}_j W(\mathbf{x}_j - \mathbf{x}_i)\right| \tag{3.4}$$

Equation (3.4) is, as mentioned before, computing *how far* the particle i is from the center of the neighbors of i, weighted by the kernel function. The classification for boundary detection turns into an analysis of the values of $\mathcal{A}_i$ and defining how far is actually far enough. In the original paper the authors propose that the boundary should be defined by all i particles that satisfy $\mathcal{A}_i > 0.3h$, with h being the kernel radius. We can use the basic idea of the Algorithm 2 to compute $\mathcal{A}_i$ values, the result can be seen in Figure 20, a 2D version of the *Water Drop* scene, where, again, particles in red were classified as boundary particles and green ones as internal to the fluid body.

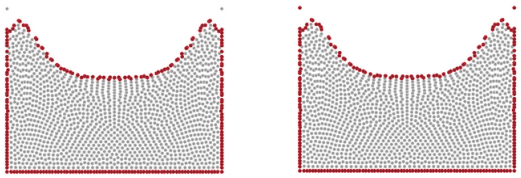
Figure 20 – Boundary detected using the *Asymmetry* method in 2D



Source: Elaborated by the author.

Figure 20 shows that the *Asymmetry* method also suffer from the same issue from the *Color Field* method where sometimes isolated regions are not correctly captured. To minimize this problem the authors of the original method propose that the fluid boundary be composed by not only particles that satisfy the condition $\mathcal{A}_i > 0.3h$ but also by

Figure 21 – Comparing frames without ρ_i filtering, left, and with ρ_i filtering, right.

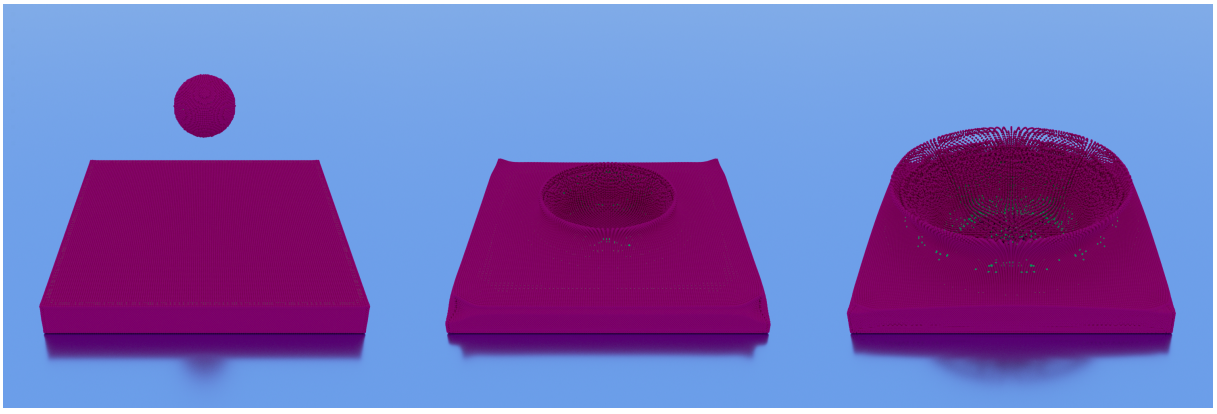


Source: Elaborated by the author.

particles that satisfy $\rho_i < 0.7\bar{\rho}$, where ρ_i is the density of particle i and $\bar{\rho}$ is the fluid reference density. The density test can locate particles that are isolated and would not be discovered by their $\mathcal{A}_i$ value because they do not have enough neighbors. Combined they can successfully locate the fluid free surface, Figure 21 compares the two approaches: without ρ_i filtering, left, and with ρ_i filtering, right.

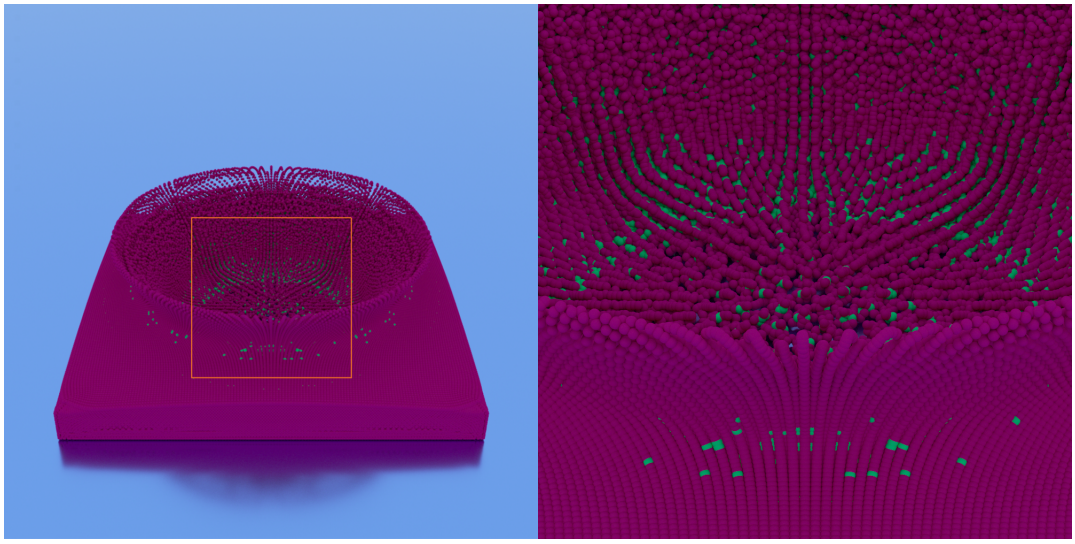
The *Asymmetry Value* approach, just like the *Color Field* method, also extends trivially to 3D. Because there are no geometric transformations that require change from 2D to 3D the implementation on the 3D environment is the same. Figure 22 shows the result of running this method on a 3D version of the *Water Drop* scene with 739,522 particles. By looking at Figure 22 and Figure 23 we can see immediately that even if the *Asymmetry Value Method* looks correct in the 2D version, when particles start to become too separated the boundary classification starts to become inconsistent and, for our goal, it is not suitable.

Figure 22 – Boundary detected using the *Asymmetry* method for the 3D Water Drop scene.



Source: Elaborated by the author.

Figure 23 – Highlight of frame of the Water Drop scene with the Asymmetry boundary.



Source: Elaborated by the author.

3.1.3 Randles - Doring Method

The method we'll introduce next comes from a very interesting work entitled *Smoothed Particle Hydrodynamics: Some recent improvements and applications*, (RANDLES; LIBERSKY, 1996). The method has strong mathematical background and the final implementation actually depends on another work, a PhD Thesis from Mathieus Doring, (DORING, 2005). We will not cover the full derivation of the boundary classification, however we will introduce the basics of the method and how both works contribute to the final version.

If we look at the previous sections, where we introduced the *Color Field Method* and the *Asymmetry Value Method*, we noticed that both methods have issues at isolated regions of the fluid. This problem rises from the fact that at those locations we cannot correctly perform *kernel* sums (Equation (3.3) and (3.4)) because of deficiencies that happens in this region. According to Equation (3.2) the density at a position $\mathbf{x}$ inside the fluid is given by

$$\rho(\mathbf{x}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{x}_j - \mathbf{x})$$

$$\rho(\mathbf{x}) = \sum_j m_j W(\mathbf{x}_j - \mathbf{x}) \tag{3.5}$$

as we can see the density directly depends on the ability to sample particle neighbors j, this can create a problem when analysing fluid discontinuities which leads to the problems of the first two methods presented. In (RANDLES; LIBERSKY, 1996) the authors propose that we split the neighborhood of a particle i at the boundary into three categories. Let $\mathcal{B}_i$ be the neighborhood of a particle i, we create the subsets:

- $\mathcal{I}_i$ that contains interior particles that are neighbors of i;
- $\mathcal{S}_i$ that contains boundary particles (including i) that are neighbors of i;
- $\mathcal{E}_i$ that contains the missing exterior particles, which would be neighbors of i.

It is easy to see that, given the previous sets of particles, if j is a neighbor of i then $j \in \mathcal{B}_i$ with $\mathcal{B}_i = \mathcal{I}_i \cup \mathcal{S}_i \cup \mathcal{E}_i$. Applying Equation (3.5) with the *SPH* formulation we have that the density ρ_i of a particle is $\rho_i = \sum_{j \in \mathcal{B}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i)$, using the previous sets we rewrite this relation as:

$$\rho_i = \sum_{j \in \mathcal{I}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i) + \sum_{j \in \mathcal{S}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i) + \sum_{j \in \mathcal{E}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i) \tag{3.6}$$

From the *Color Field* section (3.1.1) we known that for any particle, with the previous formulation, the *unitary* relation is valid:

$$\mathcal{C}(\mathbf{x}_i) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{x}_j - \mathbf{x}_i)$$

$$1 = \sum_{j \in \mathcal{I}_i} \frac{m_j}{\rho_j} W(\mathbf{x}_j - \mathbf{x}_i) + \sum_{j \in \mathcal{S}_i} \frac{m_j}{\rho_j} W(\mathbf{x}_j - \mathbf{x}_i) + \sum_{j \in \mathcal{E}_i} \frac{m_j}{\rho_j} W(\mathbf{x}_j - \mathbf{x}_i) \tag{3.7}$$

By multiplying Equation (3.7) by ρ_i, assuming that particles that are from the boundary $\mathcal{S}_i$ have the same density $\rho_j = \rho_i$ and combining with Equation (3.6) we arrive that for every particle i at the boundary, the density is given by:

$$\rho_i = \Big(\sum_{j \in \mathcal{I}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i) \Big) \Big/ \Big(\sum_{j \in \mathcal{I}_i} m_j W(\mathbf{x}_j - \mathbf{x}_i)/\rho_j \Big) \tag{3.8}$$

Equation (3.8) is a type of normalization scheme that correctly computes the density of a particle at the boundary. In (RANDLES; LIBERSKY, 1996) it is shown that a similar approach can be used if one wishes to construct the divergent of a linear stress tensor σ, $\nabla \cdot \sigma$, using the *SPH* formulation. The following second rank tensor $\mathcal{T}$ arises when computing $\nabla \cdot \sigma$:

$$\mathcal{T} = \left(- \sum_{j \in \mathcal{B}_i} m_j (\mathbf{x}_j - \mathbf{x}_i) \otimes \nabla W(\mathbf{x}_j - \mathbf{x}_i)/\rho_j \right)^{-1} \tag{3.9}$$

(MARRONE *et al.*, 2009) presents a boundary computation strategy that levarages Equation (3.9), it uses the previous definition of the tensor $\mathcal{T}$ and is the one we will use to guide the implementation. The formulation however is different from the previous methods we have shown, this approach does not *directly* computes the boundary particles but instead is able to get regions of fluid that *are not part of the boundary*. Let μ be a constant

value that filters different regions of the fluid and $\mathcal{T}$ the tensor previously described, we say that a particle i does *not* belong to the free surface if the minimum eigenvalues of $\mathcal{T}$, λ, is greater than μ, i.e.: $\lambda > \mu$ is a condition of exclusion for every particle.

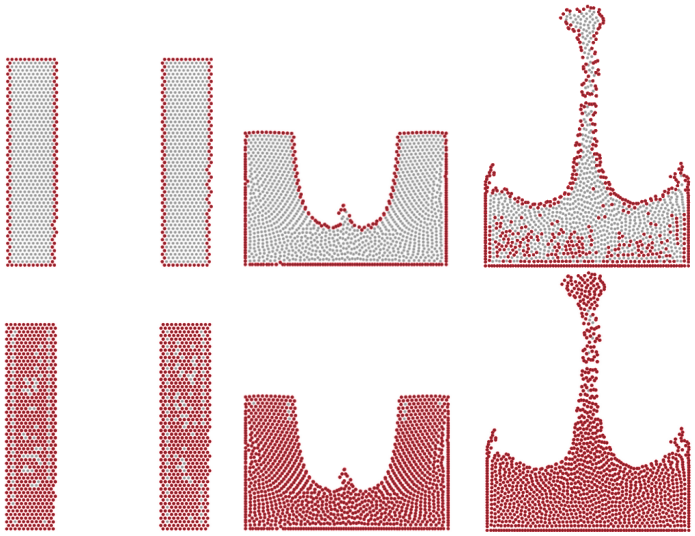
Implementation of this scheme requires that for every particle i the matrix given by $\mathcal{T}$ and its eigenvalues be computed. Note also that it is required that we invert a matrix for every particle in the fluid simulation, this is a very expensive operation depending on the scale of the simulation. As for the λ value, (MARRONE *et al.*, 2009) suggests that the threshold value, μ, to detect boundary particles be $\lambda < \mu = 0.75$, we have found however that the value μ is very difficult to adjust and depending on the scale of the simulation can become quite troublesome to find. The (3D) implementation is (again) based on Algorithm 1 where for every particle i we first initialize the matrix $\mathcal{T}_i^{-1}$ to the zero matrix and by looking to every neighboring particle j to i solve equation (3.9) with the following steps:

- Compute the distance and direction to neighboring particle j;
- Compute the gradient G of the *SPH Kernel* multiplied by $mass/density$ of particle i using the distance and direction obtained previously;
- Accumulate the matrix $G^T G$ to $\mathcal{T}_i^{-1}$;
- Once the neighborhood of i is processed, invert $\mathcal{T}_i^{-1}$ to obtain $\mathcal{T}_i$;
- Generate the characteristic equation of $\mathcal{T}_i$, given by $-\lambda^3 + Tr(\mathcal{T}_i)\lambda^2 - 0.5(Tr(\mathcal{T}_i)^2 - Tr(\mathcal{T}_i^2))\lambda + Det(\mathcal{T}_i) = 0$, where $Tr(\mathcal{T}_i)$ is the trace of $\mathcal{T}_i$ and $Det(\mathcal{T}_i)$ is the determinant of $\mathcal{T}_i$;
- Compute the eingenvalues of $\mathcal{T}_i$ by applying some algorithm that can solve the 3rd degree characteristic equation obtained in the previous step, we use the *Cardano Tartaglia* method;
- Get the minimum eingenvalue, λ, of the previous step and compare it with the chosen μ, if $\lambda < \mu$ we say particle i is part of the boundary.

This method is very expensive to be executed, a naive implementation can easily require 3 to 4 hours to generate the boundary of a 1 million particle simulation. The value $\mu = 0.75$ needs to be manually adjusted, Figures 24 shows comparison of the 2D *Double Dam Break* scene for $\mu = 0.4$ and $\mu = 0.75$ in different frames. Unfortunately the method heavily relies on the threshold value, μ, and choosing an incorrect value completely disrupts the resulting boundary. Because of this problem, the method struggles to consistently generate a 3D boundary. Extensions exists to this approach (MARRONE *et al.*, 2009) where the boundary scheme uses a second stage of processing where a *scan test* is performed from the particles detected by the $\lambda < \mu$ test. However because of the

complexity of the method and the difficult in picking correct μ values, it becomes very hard to make this solution works for the screen space pipeline. In the next session we introduce a geometric method that achieves better results than this method.

Figure 24 – Comparison between $\mu = 0.40$ (top) and $\mu = 0.75$ (bottom)



Source: Elaborated by the author.

3.1.4 Sandim's Method

The last method we'll discuss before going into the screen-space framework is *Sandim's Method*. This is a pure geometric solution to the boundary extraction problem presented in *Boundary Detection in Particle-based Fluids*, (SANDIM *et al.*, 2016). Sandim's approach is very interesting, it relies on the fact that most simulators have access to a uniform grid where particles can be distributed and neighbors can easily be found. The goal is to detect particles that compose the boundary by looking from the outside of the fluid to the inside and inspecting the regions that make the interface. In order to achieve this, it builds on top of the *Hidden Point Removal* (HPR) operator introduced in (KATZ; TAL; BASRI, 2007).

The HPR operator is an operator that can find the visible points of a surface $\mathcal{S}$ with regards to a viewpoint $\mathcal{C}$, in the words of the author: Given a set of points $P = \{p_i | 1 \leq i \leq n\} \subset \mathbf{R}^D$, which is considered a sampling of a continuous surface $\mathcal{S}$, and a viewpoint (camera position) $\mathcal{C}$, our goal is to determine $\forall p_i \in P$ whether p_i is visible from $\mathcal{C}$. The HPR operator works with two steps:

1. Inversion: Given P and $\mathcal{C}$ we transform the coordinates of P so that we *invert* their relative position, i.e.: points p_i that are close to $\mathcal{C}$ are mapped away from $\mathcal{C}$ and distant points are mapped closer to $\mathcal{C}$, we shall refer to this new set of points as $\mathcal{Q}$;

2. Convex Hull: We compute the convex hull of the set $\mathcal{Q}$, this will serve as guide to index which points in the set P are actually visible from $\mathcal{C}$.

After computation of the convex hull we get the set of points in $\mathcal{Q}$ that when *inverted back* result in the set of visible points from $\mathcal{C}$, I, however we do not actually need to perform a second inversion, simply keeping track of the index of each point can already give the set of visible points. To perform the first step (inversion) one needs to define a function that can map points p_i along the ray from $\mathcal{C}$ to p_i. In the original paper the author suggests two methods one called *Spherical Flipping* and one called *Exponential Flipping*, because *Sandim's Method* is introduced with *Exponential Flipping*, we will stick with it. In *Exponential Flipping*, we build the set $\mathcal{Q}$ by taking each point p_i from P and computing its position p_i' using Equation (3.10), where $\gamma > 1$ is a parameter.

$$p_i' = \frac{p_i}{\|p_i\|^\gamma} \tag{3.10}$$

Once the inversion process is done, we can use any convex hull generation algorithm for step 2, our implementation uses the *Jarvi's March* for 2D and *Quick Hull* for 3D, (ROURKE, 1998). With the convex hull built we simply inspect all index i from its points and find the visible points p_i from P with regards to $\mathcal{C}$, the HPR algorithm can be seen in Algorithm 3:

Algorithm 3 – Basic HPR

```
initialize Q, I
for point p_i ∈ P do
    p_i' = (p_i − C)/(‖p_i − C‖^γ)
    add p_i' to Q
end for
Hull = ConvexHull(Q)
for index j ∈ Hull do
    add p_j to I
end for
```

With the previous definition of the HPR operator we can now look at Sandim's strategy to boundary extraction. The problem with the HPR, when trying to detect the fluid free surface, is how can we chose the viewpoint $\mathcal{C}$ such that we can obtain the *complete* boundary? Sandim solves this problem by taking a regular grid, with cells of length 2ρ, in the fluid and placing viewpoints on every cell that is empty and neighbor to a cell that has particles, with this formulation we can take the neighbors of the viewpoint cell and apply HPR locally, this will give a fraction of boundary with respect to this viewpoint, the union of all local solutions will therefore be the complete boundary of the fluid. The complete mathematical derivation and proofs can be seen in the original work, in here we

are only interested in the algorithm and the quality of the boundary generated, with that in mind the steps for *Sandim's method* are:

- Distribute all particles in the regular grid;
- Place a viewpoint in the center of every empty cell that is neighbor to a cell that has particles;
- For every cell that has a viewpoint, get its neighbor particles in a chosen radius (4ρ in original work) and apply the HPR operator with $\gamma = 1.3$.

Because the method cannot locate holes inside the fluid, as it only looks for empty cells, the authors add one more rule to the viewpoint generation process: if a cell is labeled as full (has particles in it) it looks to its neighbors and check if it is possible to find a ball of radius ρ, in case it is, it generates a viewpoint at the center of the ball. This consideration leads to the following formulation of the position V_i of viewpoints inside the fluid

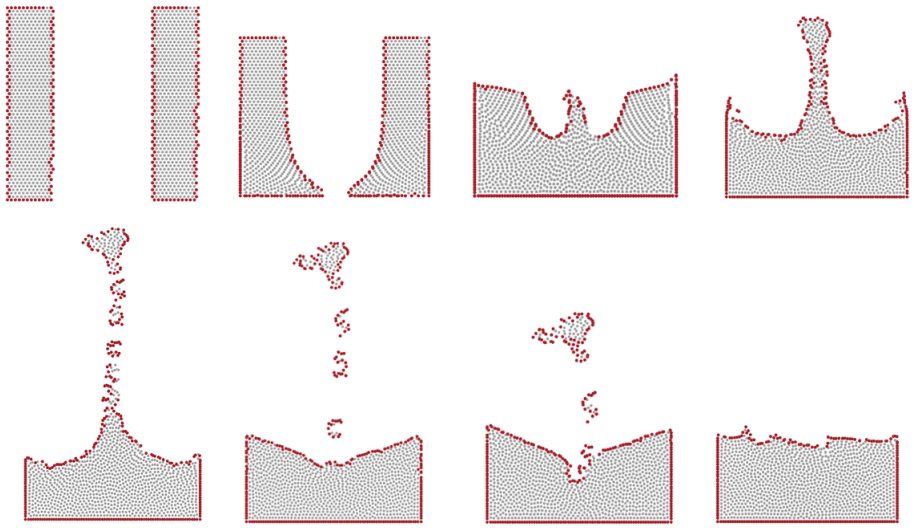
$$V_i = \begin{cases} p_i + \rho \frac{\delta_i}{\|\delta_i\|} & \text{if } \|\delta_i\| \neq 0 \\ p_i & \text{otherwise} \end{cases} \text{ with } \delta_i = p_i - \frac{1}{|N_i|} \sum_{j \in N_i} p_j$$

where j indexes the particles in the neighborhood N_i of radius 2ρ from particle p_i.

Figure 25 shows a few results of the previous algorithm on the 2D *Double Dam Break* scene, notice how when the particle form a thin layer (bottom left in Figure 25) the method struggles to find the boundary, depending on the usage this might be a serious problem. Figure 26 shows a 3D version of the *Double Dam Break* but with four blocks of fluid, notice how the method is more precise than the previous ones, but we can still perceive the inner particles (green ones). Figure 27 shows splash regions of a single frame of the simulation, again showing issues on the surface of the fluid.

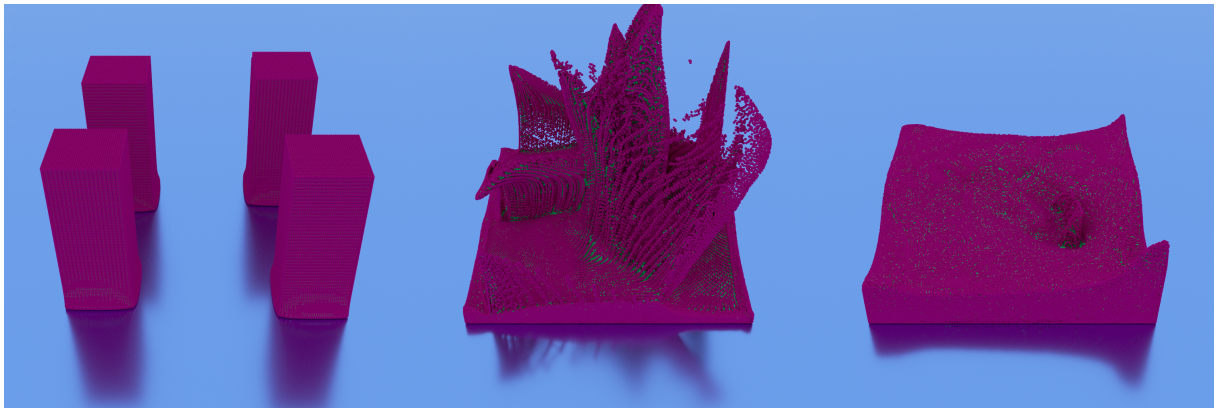
Even if the method is better than previous approaches, as it generates a more consistent boundary, it does have its flaws. The boundary generated, as previous methods, does not prevent a application to be able to sample interior particles when viewing from the outside, it is also very difficult to implement efficiently in a heavily parallel architecture (GPU) as convex hull computation is not simple to be solved and it causes a strong impact in its memory model. Convex hulls also can become unsolvable if particles overlap, requiring therefore a more precise control from the simulator. It does however outperforms the other approaches on a pure CPU execution, for large simulations, as it avoid unnecessary computations in the interior of the fluid.

Figure 25 – *Double Dam Break* scene in 2D



Source: Elaborated by the author.

Figure 26 – *Sandim's Method* on a 3D scene

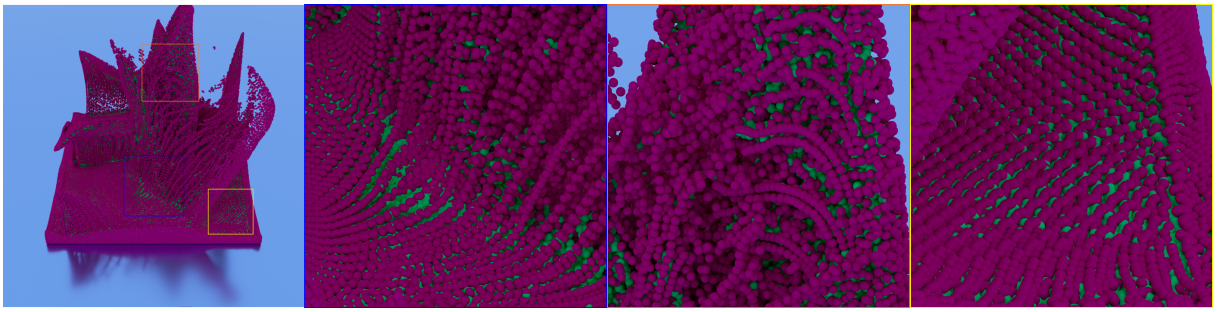


Source: Elaborated by the author.

3.2 Narrow-Band for Screen-Space

In the beginning of this chapter we mentioned that most boundary extraction methods cannot be used as a complete narrow-band with the screen-space pipeline and, in the previous sections, we shown that even if some of the methods do produce good results, they either suffer from thin layers that can form (Color Field) or they introduce strong impact on the simulation requirements (Sandim's Method). These problems come from the fact that the boundary particles rely on the kernel radius as a parameter to define what "visibility" means and define boundary as a layer from where interactions between particles stops. Computer graphics on the other hand, need that the approximation to the narrow-band be robust enough to make sure that no visibility and sampling issues happen

Figure 27 – Highlight of splash regions with *Sandim's Method*



Source: Elaborated by the author.

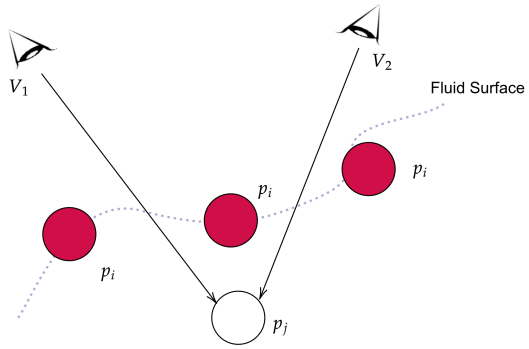
inside the geometry and, as result, these methods cannot reliably be used alone for the screen-space method. Contrary to what happens in mesh methods, when one discards interior particles, as computed by the methods presented so far in this chapter, severe errors can be perceived. The screen-space approach needs that *part* of the interior particles be present or it will generate incorrect results. Figure 28 shows a simple diagram of the problem displayed in Figure 29. We can see that the screen-space method, when applied to the boundary particles, fails on two aspects:

1. Surface Reconstruction and;
2. Volume Estimation.

Surface Reconstruction problems arises from the fact that boundary extraction methods that do not the consider rendering radius ignore gaps that might exist between two boundary particles. In this situation, it is possible to construct a ray from an interior particle, p_j, and the outside of the fluid without intersecting any other particle in the neighborhood of p_j. If this happens and an application attempts to render the scene from a view point, V_k, that is aligned with such ray, then sampling on that point fails and a *hole* is perceived in the fluid surface, V_1 and V_2 (Figure 28) are examples of such points. The methods presented so far will classify particles p_i as boundary and particles similar to p_j as interior. The results of such a classification can be seen on Figure 30, where small black spots appear in the fluid surface. One might be tempted to increase the radius of the surface reconstruction algorithm being used, this will in fact hide the problem depending on particle configuration, however it comes with a high cost. Increasing the radius of the reconstruction filter often makes the fluid looks more *blobby* and, in the screen-space part of the pipeline, it is the most expensive part of the rendering. Different approach can be seen in recent work, (XIAO; ZHANG; YANG, 2018), where the authors solve this problem by Ray Casting through the closest surface of the fluid. Their approach solves the *hole* problem but, because they do not go *through* the fluid, they do not estimate the

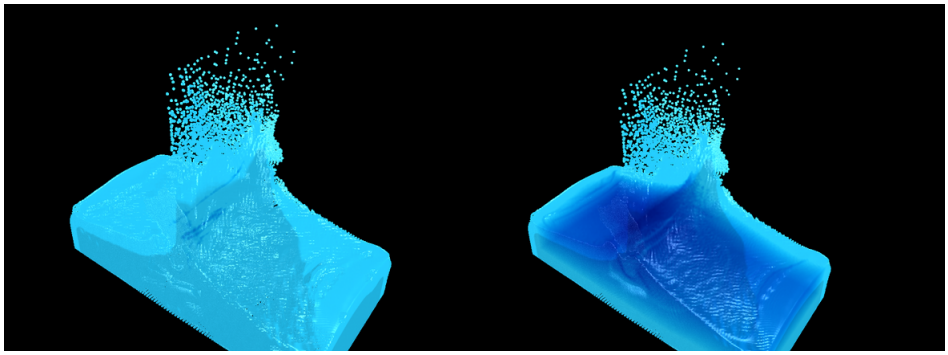
volume and the final image is unable to capture fluid depth. The screen-space pipeline can also suffer during computation of secondary effects, such as shadows or caustics, with a poor narrow-band detection routine. These effects rely on some formulation of a visibility estimation using rays that are sent from light sources. When the fluid surface contains holes, rays can travel without ever interacting with the fluid and generate incorrect results. Figure 31 shows an image where shadows are not correctly rendered because light rays, when crossing the fluid surface, are able to hit the floor without ever intersecting the reconstructed surface.

Figure 28 – Missing particle during rendering causes a gap to appear.



Source: Elaborated by the author.

Figure 29 – Screen-Space with poor narrow-band, left, all particles, right.

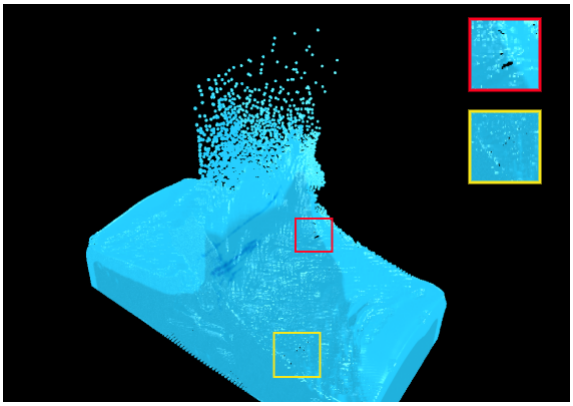


Source: Elaborated by the author.

We will now present our solution to the missing particle problem that is based on boundary detection. We chose this approach because it is natural to most particle-based simulators and, as we will show, the requirements are already solved during simulation step and as such, do not require any special treatment. This approach also works with other algorithms that attempt to perform secondary effects such as Shadow Mapping and Caustics Mapping, (SHAH; KONTTINEN; PATTANAIK, 2007), as we are able to

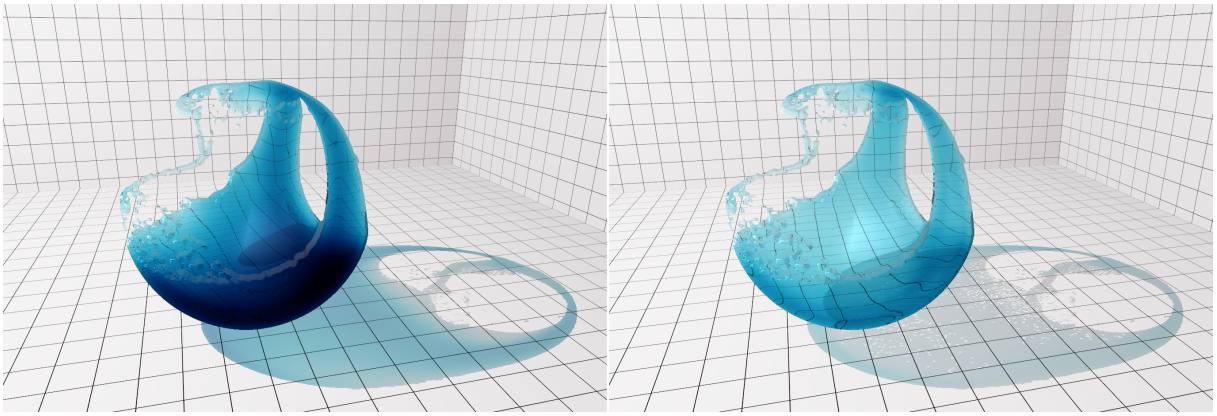
generate a complete surface no matter what viewpoint is being used and so is not bound to a specific camera location. Our solution also works well for different applications that can use real-time simulation data or previously generated, reducing the amount of particles the CPU and GPU need to share and the amount of input/output operations required. The volume estimation problem will be solved in the next chapter, where we will return to the graphical part of the screen-space pipeline.

Figure 30 – Black spots on fluid surface.



Source: Elaborated by the author.

Figure 31 – Comparison between the complete rendering with all particles (left) and the poor narrow-band only (right), giving incorrect shadows.



Source: Elaborated by the author.

3.2.1 Layered Neighborhood Method

In order to solve the problems described in the previous section, we wish to develop a method that can, *most of the times*, classify particles p_j (Figure 28) as in the narrow-band (NB). Our focus for this method is to achieve the following:

1. The method should be fast: Performance is our main goal. We want to be able to perceive faster loading and rendering of particle-based fluids so we need that the narrow-band extraction method does not become a time consuming task that slows down simulation. For real-time applications we aim for a performance under 20ms;

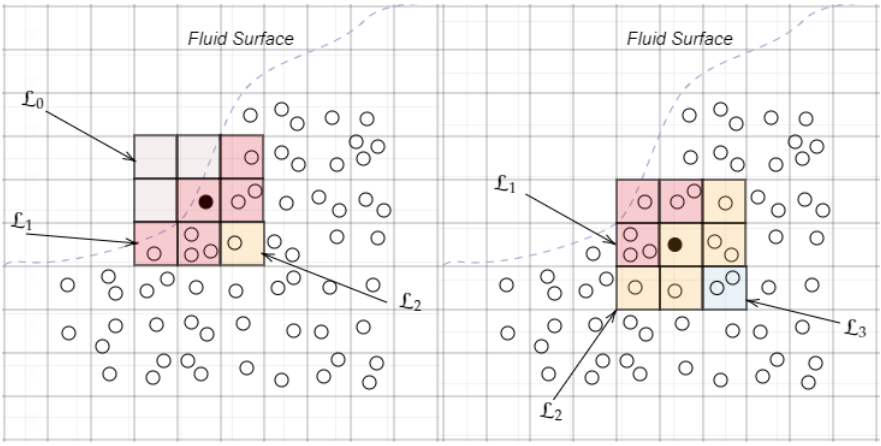
2. The method should, *most of the times*, cover any possible holes that would cause failure during sampling: A perfect solution perhaps can be achieve with a geometric method based on (HAQUE; DILTS, 2007), where sphere intersection is computed, however this method is extremely expensive, so we accept false positives inside the fluid while maintaining consistency on the outside;

3. The method should be simple: We want our method to be an auxiliary tool for developers using particle-based simulations that can be simply *inserted* into their simulators without having to re-design how they compute the simulation steps or what properties need to be tracked. Simplicity at its core.

With the previous items in mind, we design a simple algorithm that most of the times prefers to label particles as in the NB instead of interior, this guarantees that when unsure about visibility problems it includes the particles to the NB in order to be safe. We explore the fact that most particle-based simulations perform their neighbor querying on a regular grid composed of voxels that hold, at minimum, the index i of every particle p whose position would hash to it, and we write that if $\mathcal{H}$ is this hashing function then $\mathcal{H}(p) = \mathcal{V}_j$ for some voxel $\mathcal{V}_j$.

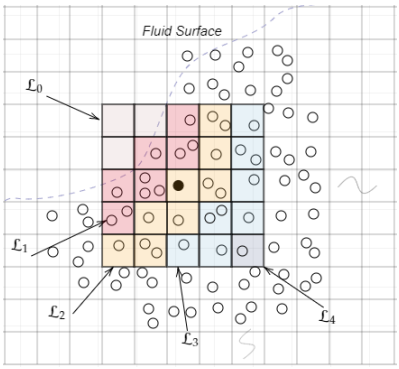
Our method looks at fluids by layering different regions (as we will show) and *peeks* the fluid from the outside to the inside. This approach enables us to avoid the processing of every single particle and increase overall performance, to do that we first must classify different regions of the fluid using some criteria. We begin by defining a classifier for voxels in the domain as follow: a voxel $\mathcal{V}_i$ is said to be of class $\mathcal{L}_0$ if it has no particles inside of it and $\mathcal{L}_k$, $k > 0$, if it is possible to find a k-far-neighbor voxel $\mathcal{V}_j$ that is $\mathcal{L}_0$, we call this neighborhood $\mathcal{N}_{k\mathbf{x}k}$. This has the effect of classifying how far a given voxel $\mathcal{V}_i$ is from an empty voxel. Figure 32 shows this classification on a simple 2D particle distribution, where each voxel (cell in 2D) is classified using the previous definition, the center voxel from each neighborhood is marked by a dark particle. On Figure 32 we say that we classified the $\mathcal{N}_{1\mathbf{x}1}$ neighborhood of the center voxel, meaning we only took 1 voxel in every direction from the center voxel. If we were to look at a $\mathcal{N}_{2\mathbf{x}2}$ - 2 voxels in every direction, we could

Figure 32 – Classification result from two different center voxels.



Source: Elaborated by the author.

Figure 33 – Classification result in a $\mathcal{N}_{2x2}$ neighborhood.



Source: Elaborated by the author.

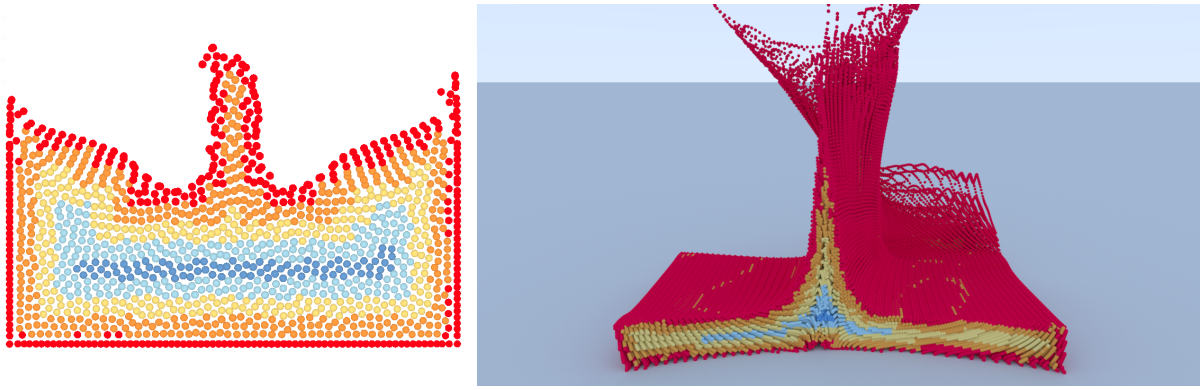
get the results from Figure 33, were we omit the interior particles. Overall we can write that for every voxel $\mathcal{V}_i$, that has particles in it, its class $\mathcal{L}_s$ is determined by looking at its $\mathcal{N}_{1\mathbf{x}1}$ neighborhood and inspecting class values $\mathcal{L}_{s_n}$ obtained, and is given by:

$$s = 1 + \min_{n \in \mathcal{N}_{1\mathbf{x}1}} s_n \tag{3.11}$$

A 2D (and 3D) voxel classification can be seem in Figure 34 where colors direct relate to the distance from class $\mathcal{L}_0$ voxels, for all tests we use voxel side length $d = 4r$, where r is the particle radius. Note how this scheme separates particles by layers which we'll use in our NB model.

We can now write our version (definition) of the narrow-band constructed around the boundary particles of the fluid. Let $\mathcal{F}_s$ be a binary classifier for any particle p_i such that $\mathcal{H}(p_i) = \mathcal{V}_j$ for some voxel $\mathcal{V}_j$ that is $\mathcal{L}_s$ with $s > 0$. $\mathcal{L}$ a function that, given a voxel

Figure 34 – Classification result in a 2D and 3D fluid simulation. Each different color represents a different layer and how far a particle from this layer is from $\mathcal{L}_0$ voxels, where red particles are the closest and blue ones the furthest.



Source: Elaborated by the author.

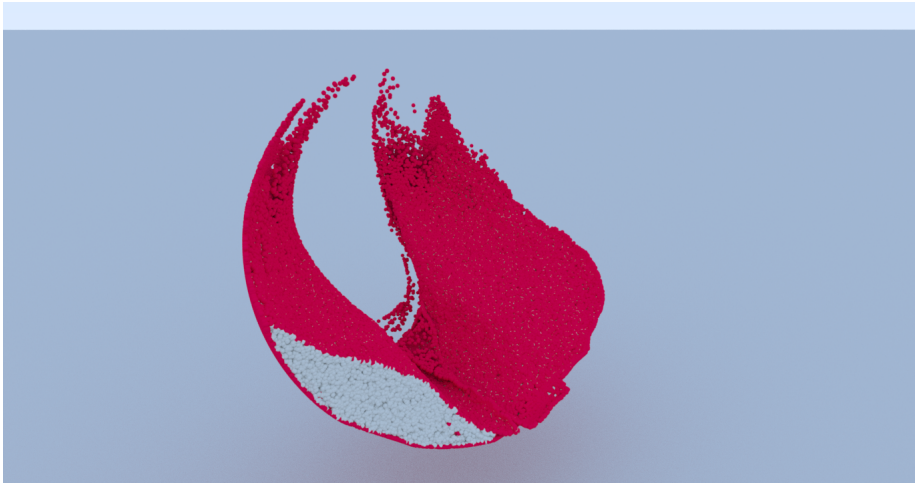
$\mathcal{V}_j$, can retrieve its class $\mathcal{L}_s$ and $\mathcal{H}$ the grid hashing function mentioned before, we define that the narrow-band of the fluid $\mathcal{S}$, made of particles p_i, is given by:

$$\mathcal{S} = \{p_i : \mathcal{F}_s(p_i) = 1 \text{ with } \mathcal{L}(\mathcal{H}(p_i)) = \mathcal{L}_s\} \tag{3.12}$$

The idea behind Equation (3.12) is that we can look at the problem of searching for the narrow-band of the fluid as an inspection, from the outside to the inside, that continues until we reach a point (layer) where no function $\mathcal{F}_s$ can ever give 1. We say that $\mathcal{F}_s$ is a visibility test that either accepts particles inside the voxel $\mathcal{V}_j$ or rejects it. This construction allows us to define how precise we wish to make the NB and how expensive we want the process to be. Each layer made of $\mathcal{L}_i$ voxels we add to the process creates its own NB $\mathcal{S}_i$ (also referred as $\mathcal{L}_i$-boundary), we write the complete NB $\mathcal{S} = \bigcup_i \mathcal{S}_i$, we say that each $\mathcal{S}_i$ is a *refinement* of the previous boundary set $\mathcal{S}_{i-1}$. A complete NB $\mathcal{S}$ is then located by inspecting all voxels and generating all $\mathcal{S}_i$ boundaries. This process, of course, is very expensive and would not allow for a good enough approach that could be easily integrated in different applications, we shall now relax this formulation so it can be more easily solved.

We first start by inspecting the narrow-band one would achieve if our solutions was perfect, i.e.: we inspect all particles in the fluid and check for sphere intersection. This approach is similar to the method presented in (HAQUE; DILTS, 2007) and generates a very accurate narrow-band, Figure 35 shows a single frame of such method. Even if it is the ideal result, such approach is very expensive, Figure 35, for example, took around 1 minute of computation in the GPU, for our purposes, this is not a time we can afford. However it brings a interesting result, Table 1 shows the location of the narrow-band particles per layer for Figure 35. Notice how more than 99% of the narrow-band is located in the first two layers of the fluid, made by classes $\mathcal{L}_1$ and $\mathcal{L}_2$. Figure 36 and Table 2 shows another example on the *Quadruple Dam Break* scene where, once again, more than 99% of

Figure 35 – Narrow-band classification using geometry intersection in a single frame of the Fluid in Ball scene.



Source: Elaborated by the author.

the boundary is located in first two layers. In fact, we were unable to locate a frame where more than 2% of the boundary could be found in a layer generated by classes $\mathcal{L}_s$ where $s > 2$. This result was checked against 60K frames, from several different simulations, and in all of them it showed to be correct, so we take it to be a good approximation to the behaviour of the boundary distribution per layer.

Layer	**Nº Narrow-Band Particles**	**Nº Interior Particles**	**Total Particles**
$\mathcal{L}_1$	106,540	144,474	251,014
$\mathcal{L}_2$	2,116	202,509	204,625
$\mathcal{L}_3$	59	64,252	64,311
$\mathcal{L}_4$	3	3,242	3,245
$\mathcal{L}_5$	0	0	0

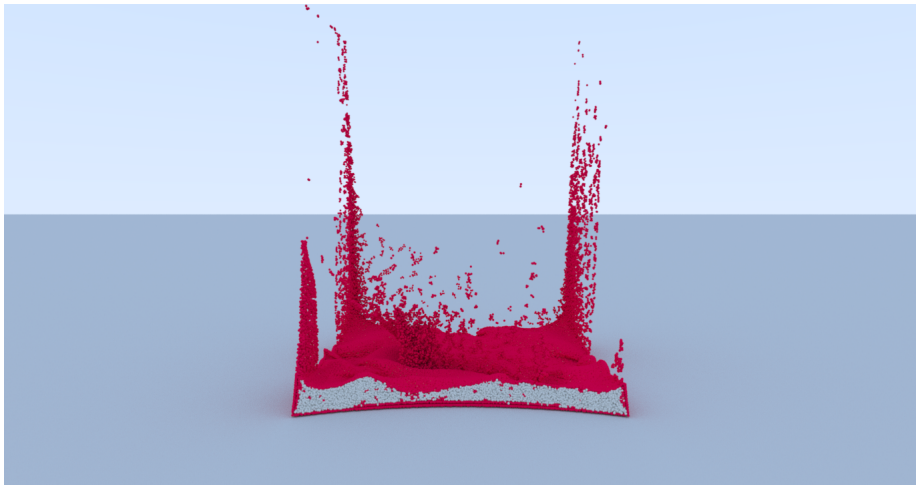
Table 1 – NB particles per layer using the geometric solution for Figure 35.

Layer	**Nº Narrow-Band Particles**	**Nº Interior Particles**	**Total Particles**
$\mathcal{L}_1$	160,767	36,127	196,894
$\mathcal{L}_2$	56,173	138,911	195,084
$\mathcal{L}_3$	966	53,787	54,753
$\mathcal{L}_4$	2	223	225
$\mathcal{L}_5$	0	0	0

Table 2 – NB particles per layer using the geometric solution for Figure 36.

With this result we come to our first simplification. We only consider calculation of fluid regions where voxels have class $\mathcal{L}_s$ such that $s = 1$ or $s = 2$ and we write that the fluid narrow-band $\mathcal{S}$ is approximately the union of $\mathcal{S}_1$ and $\mathcal{S}_2$, i.e.: $\mathcal{S} \approx \mathcal{S}_1 \cup \mathcal{S}_2$.

Figure 36 – Narrow-band classification using geometry intersection in a single frame of the Quadruple Dam Break scene.



Source: Elaborated by the author.

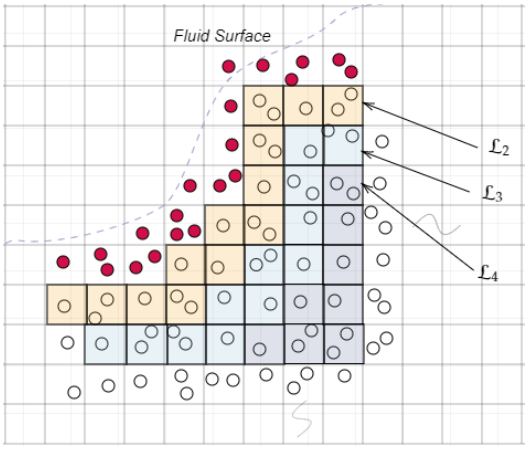
The previous statement, $\mathcal{S} \approx \mathcal{S}_1 \cup \mathcal{S}_2$, is very important as it heavily reduces complexity of the fluid narrow-band computation. We can now start to design our algorithm for solving this computation in the following steps:

1. Find all voxels that are $\mathcal{L}_0$;
2. Get the $\mathcal{N}_{\mathbf{1x1}}$ neighbors of $\mathcal{L}_0$, voxels that have particle inside are $\mathcal{L}_1$ by definition;
3. For every $\mathcal{L}_1$ voxel check if there is need to process its neighbor $\mathcal{L}_2$, if it does, select these voxels;
4. Process every particle in $\mathcal{L}_1$ and $\mathcal{L}_2$ accordingly to chosen $\mathcal{F}_1$ and $\mathcal{F}_2$, mark particles that result in 1 as narrow-band particles.

Because classification of $\mathcal{L}_1$ is the easiest to perform this means we can easily have access to most of the narrow-band particles the perfect sphere intersection would find. $\mathcal{S}_1$ however is the layer where computation is also more expensive, since most of the boundary is located there we would need to apply $\mathcal{F}_1$ (which we did not yet define) everywhere. As it turns out the best approach to have a competitive result in real-time and get most of the narrow-band particles, even if does introduce errors in the final result, is to *not apply any visibility test at all*. This gives the trivial classifier $\mathcal{F}_1$ to be applied for the generation of the narrow-band layer $\mathcal{S}_1$:

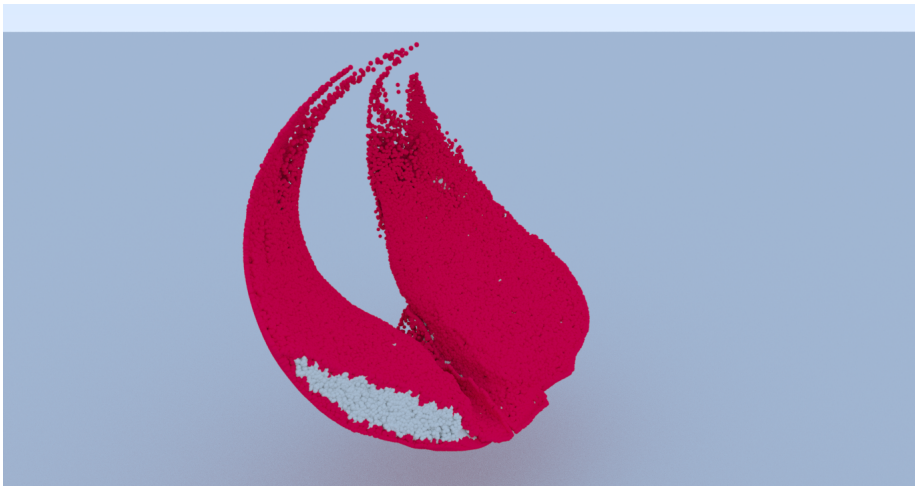
$$\mathcal{F}_1(p_i) = 1 \quad \text{if} \quad \mathcal{L}(\mathcal{H}(p_i)) = \mathcal{L}_1. \tag{3.13}$$

Figure 37 – $\mathcal{L}_1$-boundary and interior particles in 2D.



Source: Elaborated by the author.

Figure 38 – $\mathcal{L}_1$-boundary and interior particles in 3D.



Source: Elaborated by the author.

One might argue that applying the relation in Equation (3.13) will incorrectly mark particles. We find however that this is actually a good approach as it generates a thicker narrow-band that helps the screen-space pipeline during rendering. Figure 37 shows a simple scheme of the classification so far, particles p_i in red satisfied $\mathcal{F}_1(p_i) = 1$ and so they are automatically classified as narrow-band particles. Figure 38 shows the result from the previous classification, note how indeed the NB "looks" almost complete with a ticker layer. The problem however is that a significant part of the correct boundary is located in the $\mathcal{S}_2$ set and since we have not processed anything in there we would have several errors if we wished to render the resulting fluid without mesh generation.

For the refinement step, computation of $\mathcal{S}_2$, several different approaches can be applied depending on applications requirements. For $\mathcal{L}_2$-boundary we define the binary classifier $\mathcal{F}_2$ as a refinement of the previously defined $\mathcal{F}_1$ in order to prevent incorrect surface rendering. We want to create a function $\mathcal{F}_2$ that is able to correctly fill in gaps generated by the first step of the classification (Figure 28). Because we already took a thicker layer in $\mathcal{S}_1$ we do not actually need to process all parts of the $\mathcal{S}_2$ but instead only the regions where the potential gap might exist. We define such regions as regions where the volume of particles located in the $\mathcal{L}_1$ neighboring voxel is not enough to completely block the entire view of the voxel. For our formulation we define the minimal amount of particles n_1 in every $\mathcal{L}_1$ voxel to prevent $\mathcal{L}_2$ processing to be given by:

$$n_1 = (d/h)^D \tag{3.14}$$

where d is the length of the voxel, h the diameter of the particle and D the dimension of the simulation. The value n_1 reduces processing and limits the computation to voxels that have a high chance that a viewing ray that crosses the interfaces of the given $\mathcal{L}_1$ voxel will reach the neighboring $\mathcal{L}_2$. We can now define what it means to select the $\mathcal{L}_2$ voxels:

1. Find all voxels that are $\mathcal{L}_0$;

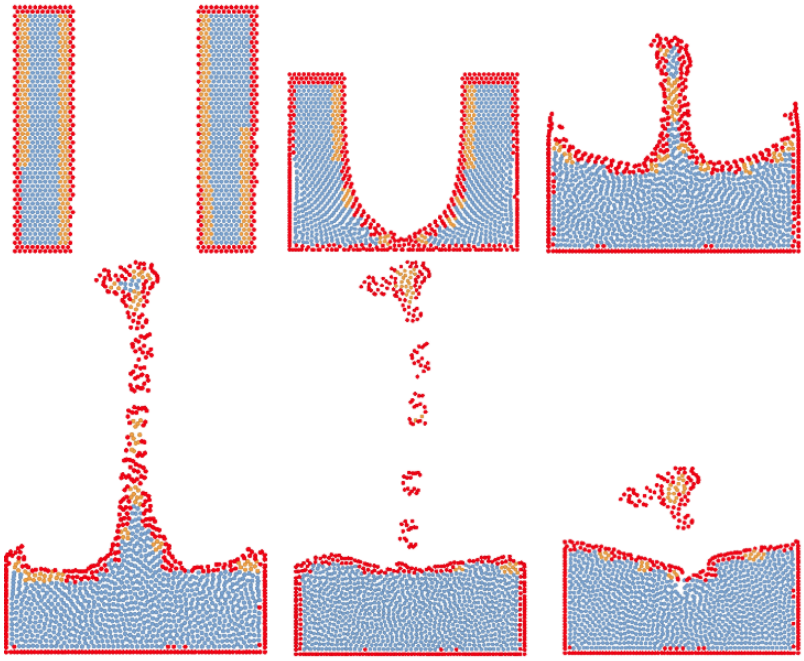
2. Get the $\mathcal{N}_{\mathbf{1x1}}$ neighbors of $\mathcal{L}_0$, voxels that have particle inside are $\mathcal{L}_1$ by definition and therefore **all particles inside are narrow-band particles and should be added to** $\mathcal{S}_1$;

3. **For every** $\mathcal{L}_1$ **voxel check if the number of particles** n_i **inside of it satisfy** $n_i \leq n_1$**, if it does, the neighboring** $\mathcal{L}_2$ **needs to be processed, otherwise it can be safely ignored**;

4. Process every particle p_i in **selected** $\mathcal{L}_2$ **accordingly to chosen** $\mathcal{F}_2$**, add particles that result in** $\mathcal{F}_2(p_i) = 1$ **to** $\mathcal{S}_2$.

The previous selection of voxels $\mathcal{L}_2$ using $n_i \leq n_1$, once again, heavily reduces the amount of processing required for generation of $\mathcal{S}_2$, Figure 39 shows a sequence of frames from the 2D version of the Double Dam Break scene where on average, 7.34% of $\mathcal{L}_2$ voxels required processing, note however that in our tests around 20% (average) of $\mathcal{L}_2$ voxels actually require a $\mathcal{F}_2$ pass. Which brings us to the final step of the method, construction of the $\mathcal{S}_2$ set. Function $\mathcal{F}_2$ can be constructed in different ways depending on how precise we wish the computation to be. In this work we have experimented with 3 different approaches which we discuss next, after some implementation considerations.

Figure 39 – 2D Double Dam Break scene. Particles in red are from the $\mathcal{S}_1$ layer, orange particles are inside voxels $\mathcal{L}_2$ that require processing and blue particles are interior to the fluid body.



Source: Elaborated by the author.

3.2.1.1 Implementation

In order to implement the methods mentioned in the previous section we split computation in two different steps. The first one locates the first layer of the set $\mathcal{S}_1$ and second builds on top of this result to build the second layer $\mathcal{S}_2$. Algorithm 4 shows an overview.

Algorithm 4 – Narrow-band of particle set $\mathcal{P}$ distributed over the domain $\mathcal{D}$

```
1: procedure NARROW-BAND(P, D, d, h, D)  ▷ Computes the NB S of the particles P
2:     n_1 ← (d/h)^D                                          ▷ Equation 3.2.1
3:     S ← FindL1(P, D, D)                                    ▷ Algorithm 5
4:     S ← S ∪ FindL2(P, D, n_1)                              ▷ Algorithm 6
5:     return S
6: end procedure
```

Locating voxels $\mathcal{L}_1$ is performed simultaneously to computation of the layer $\mathcal{S}_1$ as they can be viewed as the same operation. In our implementation we do not explicity mark voxels that are $\mathcal{L}_2$, while this can be performed to guide the original formulation in Equation (3.12), it is not necessary for the approximation $\mathcal{S} \approx \mathcal{S}_1 \cup \mathcal{S}_2$, however we do need to mark the voxels that are $\mathcal{L}_1$. The $\mathcal{S}_1$ generation can be seen in Algorithm 5, it is straightforward. It begins by computing the number of neighbors it is expected for each

voxel (threshold), this value is used to check if a voxel lies on the edges of the domain, in case it does, and it has particles inside, it can be safely assumed to be $\mathcal{L}_1$. For all other voxels we simply loop through their neighbors and check if there is any voxel $\mathcal{L}_0$ which by our definition defines the current voxel $\mathcal{V}_i$ to be $\mathcal{L}_1$. In case a voxel $\mathcal{V}_i$ is defined to be $\mathcal{L}_1$, we mark it, add its particles to the layer $\mathcal{S}_1$, and go to the next voxel inside the domain.

Algorithm 5 – Compute $\mathcal{S}_1$ for a given particle distribution $\mathcal{P}$ over the domain $\mathcal{D}$

```
1:  procedure FindL1(P, D, D)
2:      S_1 ← {}                                   ▷ Initialize S_1 to the empty set
3:      threshold ← 3^D                  ▷ Maximum expected neighbors for a given voxel
4:      for voxel V_i ∈ D do
5:          s ← NumberOfNeighboringVoxels(V_i)     ▷ Number of voxels neighbors to V_i
6:          n ← NumberOfParticles(V_i)             ▷ Number of particles in voxel V_i
7:          if n > 0 and s ≠ threshold then        ▷ Handle edge cases over N_1x1
8:              S_1 ← S_1∪ GetParticles(V_i)       ▷ V_i is L_1
9:              Mark V_i as L_1                    ▷ H(V_i) = 1
10:         else if n > 0 then
11:             for voxel V_j ∈ NeighborsOf(V_i) do   ▷ For every neighbor in N_1x1
12:                 if NumberOfParticles(V_j) == 0 then
13:                     S_1 ← S_1∪ GetParticles(V_i)   ▷ V_j is L_0 so V_i is L_1
14:                     Mark V_i as L_1 and break      ▷ H(V_i) = 1
15:                 end if
16:             end for
17:         end if
18:     end for
19:     return S_1
20: end procedure
```

Once all voxels $\mathcal{L}_1$ are defined we can perform computation of $\mathcal{L}_2$ with the call to the routine FindL2, Algorithm 6. Again, we loop through voxels inside the domain looking for unclassified voxels that have particle inside, for these voxels $\mathcal{V}_i$, we inspect the neighborhood looking for previously computed $\mathcal{L}_1$ voxels. If they satisfy condition on Equation 3.2.1 we process its particles based on $\mathcal{F}_2$, otherwise we continue the search. The return of $\mathcal{F}_2$ is transformed into a set and added to the layer $\mathcal{S}_2$.

The methods presented in Algorithms 5 and 6 are easy to be made parallel as porting to GPU is also straightforward. We replace the boundary set with a simple boolean indicator for every particle that marks if it pertences to the boundary or not. Whenever we conclude that a voxel $\mathcal{V}_i$ is $\mathcal{L}_1$ or $\mathcal{L}_2$ we set this indicator accordingly to the result of executing $\mathcal{F}_2$ over the particle. If the function $\mathcal{F}_2$ is complex it is wise to instead solve it in a separated call to avoid reducing potential parallelism because voxels can have different execution flows. In these situations we add a parallel queue for enqueuing particles present in voxels $\mathcal{L}_2$ and launch a new GPU call to solve $\mathcal{F}_2$ on all elements of the queue. The previous algorithms are a simple way to implement the method that should be easy to

Algorithm 6 – Compute $\mathcal{S}_2$ for a given particle distribution $\mathcal{P}$ over the domain $\mathcal{D}$

```
1: procedure FindL2(P, D, n_1)
2:     S_2 ← {}                                  ▷ Initialize S_2 to the empty set
3:     for voxel V_i ∈ D do
4:         n ← NumberOfParticles(V_i)            ▷ Number of particles in voxel V_i
5:         if H(V_i) ≠ 1 and n > 0 then          ▷ Check if voxel was not yet handled
6:             for voxel V_j ∈ NeighborsOf(V_i) do   ▷ For every neighbor in N_1x1
7:                 n_j ← NumberOfParticles(V_j)
8:                 if H(V_j) == 1 and n_j ≤ n_1 then   ▷ Filter the neighboring L_1
9:                     Q ← F_2(GetParticles(V_i))      ▷ Apply F_2 over all particles in V_i
10:                    S_2 ← S_2 ∪ {Q} and break       ▷ Mount the resulting set
11:                end if
12:            end for
13:        end if
14:    end for
15:    return S_2
16: end procedure
```

add to any fluid simulator. There is however a faster way to classify the particles if the grid component supports query for $\mathcal{N}_{\mathbf{2x2}}$ neighbors. It is possible to perform all tasks in a single call with the use of Algorithm 7, presented at the end of this chapter. In the following sections we show a few of the many possibles $\mathcal{F}_2$ we consider to be appropriate for processing $\mathcal{L}_2$ voxels, based on application requirements.

3.2.1.2 *Visibility tests*

Function $\mathcal{F}_2$ plays an important role in the procedure of refining the $\mathcal{L}_1-$boundary and assuring no holes are visibles. Because we do not impose any conditions in the formulation of $\mathcal{F}_2$ applications are free to pick any method they wish. For example, it is possible to simply apply sphere intersection in all particles from $\mathcal{L}_2$. However our goal is to have a very fast method that correctly renders with the Screen-Space pipeline and is *pratically* invisible during simulation. Even if some of the methods presented earlier could be applied, we find that the best solution is to simply re-apply the $\mathcal{F}_1$ formulation, i.e.:

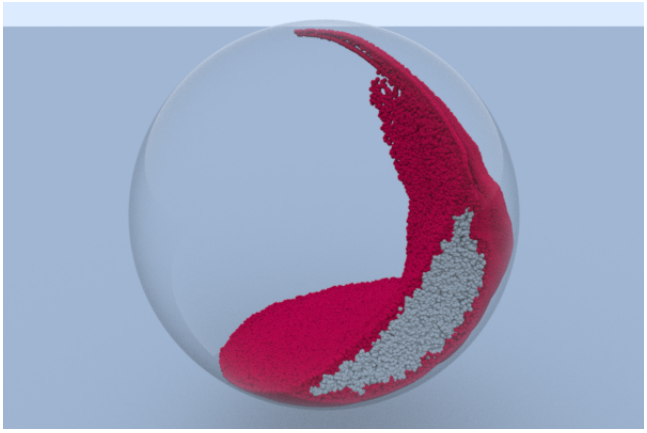
$$\mathcal{F}_2(p_i) = \begin{cases} 1 & \text{if} \quad \mathcal{L}(\mathcal{H}(p_i)) = \mathcal{L}_2 \\ 0 & \text{otherwise.} \end{cases} \tag{3.15}$$

The methods presented simply do not perform well enough to be completely applied during simulation. For example, the Sphere Intersection method can take more time than several steps of simulation and Sandim's method needs carefull design to even be solvable in GPU[†]. Because we are already reducing computation (a lot) by applying the $n_i \leq n_1$ test on the $\mathcal{L}_2$ voxels, accepting all particles inside of it does not introduce too many particles in the

[†] Convex hull computation requires special attention on the scale of fluid simulation as the geometry of particles is very small and is heavily affected by float point precision.

narrow-band and is the fastest we can achieve. With this approach we were able to generate narrow-bands for simulations as high as 1.2 million particles in 4ms (depending on the particle distribution), Figure 40 shows a example of a simulation where the narrow-band of a simulation made by 523,195 particle is computed in 3ms. This approach also reduces a lot of complexity during implementation as $\mathcal{F}_2$ (Algorithm 7) becomes trivial and the extraction routine can be enterily solved in a single pass through voxels without ever have to inspect particles. Figure 41 shows several frames of the fluid in ball simulation, (KIM, 2016). Note how interior particles are not visible, Figure 42 shows a cut comparison of the exact method (Sphere Intersection) top, and our approach bottom.

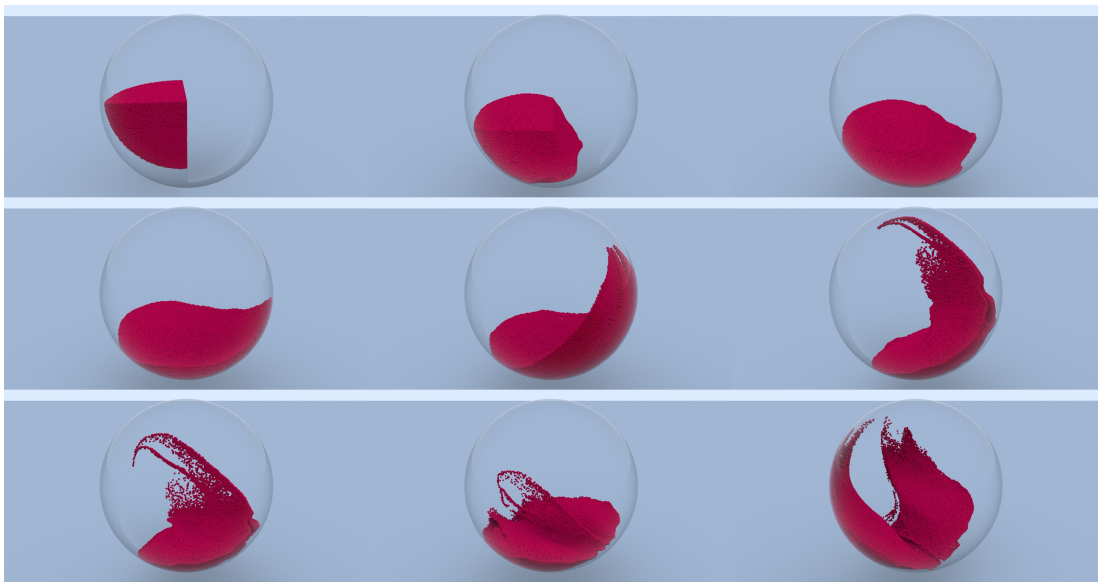
Figure 40 – 3D Narrow-band generated in 3ms.



Source: Elaborated by the author.

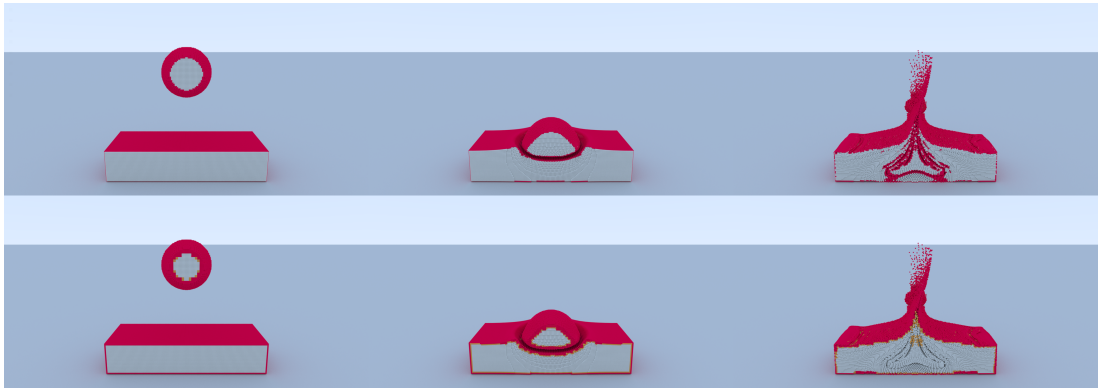
Picking the entire layer of $\mathcal{L}_2$ voxels to insert into the boundary can be however a very bold move. If we wish to compute how much of the $\mathcal{L}_2$ layer should be part of the narrow-band we could develope an intuition (like before) applying Sphere Intersection only on $\mathcal{L}_2$. Figure 43 shows the amount of particles that we should include to the NB, from $\mathcal{L}_2$, based on the "perfect" visibility test (Sphere Intersect). It is easy to see that there is a huge difference in the ideal set $\mathcal{S}_2$ and the one used before. Sphere Intersection shows that only around 2k (8%) of all particles in $\mathcal{L}_2$ acutally belong to the narrow-band, but we added around 40k particles. This difference however comes at a heavy cost, running Sphere Intersection on $\mathcal{L}_2$ still requires approximately 4 seconds of execution. A much better result than the original 16 seconds, that might be acceptable for some applications, but still huge when comparing to the performance of a single PCISPH step for this simulation (approximately 2 second). We can attempt to improve the quality of $\mathcal{L}_2$ by making use of the Asymmetry Value method over $\mathcal{L}_2$. In this new formulation we could write that a valid function $\mathcal{F}_2$ is:

Figure 41 – Our method applied to several frames of the Fluid in Ball scene.



Source: Elaborated by the author.

Figure 42 – Comparison between exact method by Sphere Intersection, top, and our approach, bottom.



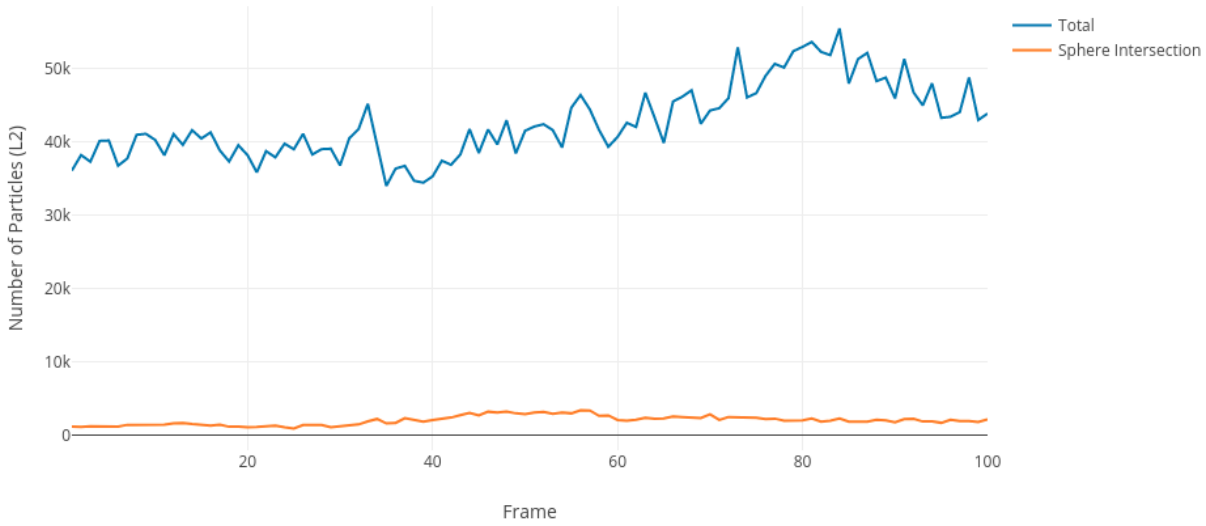
Source: Elaborated by the author.

$$\mathcal{F}_2(p_i) = \begin{cases} 1 & \text{if } \mathcal{A}_i > 0.1h \text{ and } \mathcal{L}(\mathcal{H}(p_i)) = \mathcal{L}_2 \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathcal{A}_i = \left| \mathbf{x}_i - \sum_{j \in \mathcal{N}_{1\mathbf{x}1}} \mathbf{x}_j W(\mathbf{x}_j - \mathbf{x}_i) \right|$$

if we, again inspect the curves for this solution we can see a better result. Figure 44 shows a comparison again of the total particles in $\mathcal{L}_2$, the amount of particles added by the Sphere Intersection method and the particles added by the Asymmetry approach. The result is much better even if it is still present a quite large difference from the ideal $\mathcal{S}_2$.

Figure 43 – Comparison of the amount of particles present in $\mathcal{L}_2$ and the amount of particles that are picked by Sphere Intersection in the first 100 frames of the Sphere in Ball scene.



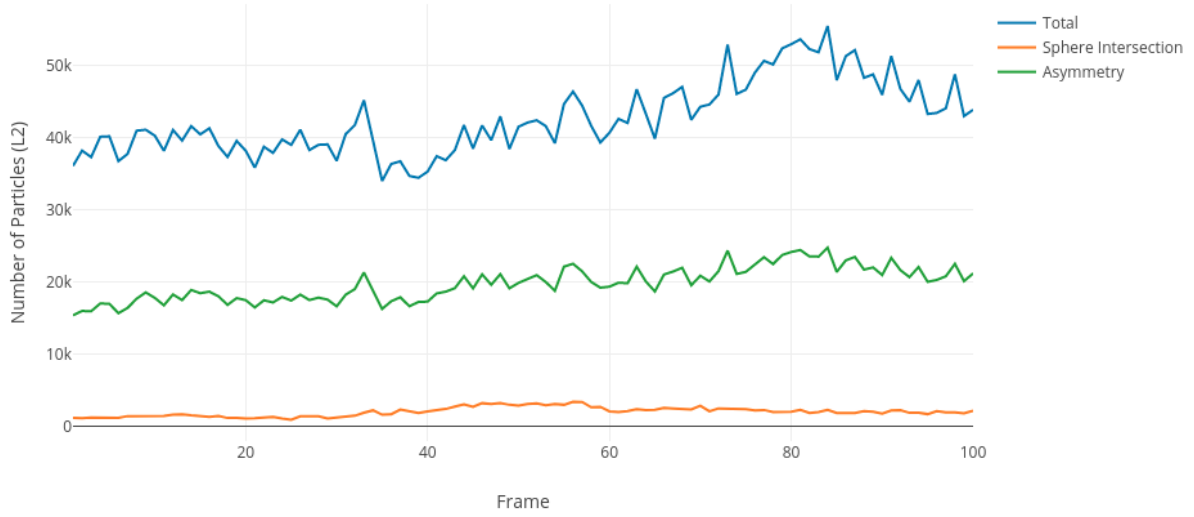
Source: Elaborated by the author.

The Asymmetry method also does not present failures in the sampling because all errors introduced in its solution are already resolved by the $\mathcal{S}_1$ set. The Asymmetry plot shows that around 18k particles are added to the boundary set, which is still a lot considering only 2k are required. However it is now able to solve the entire boundary computation in under 500ms. The other approaches presented in this chapter are not fit for $\mathcal{S}_2$ computation, both Color Field and Sandim's method simply propagate errors (holes) into interior layers and do not guarantee a complete narrow-band and Sandim's has a strong impact on GPU implementations.

We find that, for the purposes of having a real-time method for narrow-band computation, only the first approach is valid, i.e.: $\mathcal{F}_2(p_i) = 1$. Other approaches, while possible, introduce a heavy weight into the solver itself and are dependent on application requirements. Finally, we present a comparison of the result different methods give for the screen-space pipeline in Figure 45 and dynamic shadows in Figure 46[†]. From all approaches displayed ours is the only that can maintain surface details across all parts of the fluid. The Color Field method, even if it at first glance does not display black spots in its surface, it is missing part of the particles of the splash region (yellow square) because of miss-classification on low density areas, while both Sandim's approach and the Asymmetry method fail to maintain consistency and prevent failure during surface reconstruction. Note how our approach keeps correct shadows while the Color Field does not, showing that, even if it is difficult to visually detect failures, algorithms that require a complete surface will struggle if the narrow-band is not correct. Our method not only

[†] Volume is computed by the method described in Chapter 4.

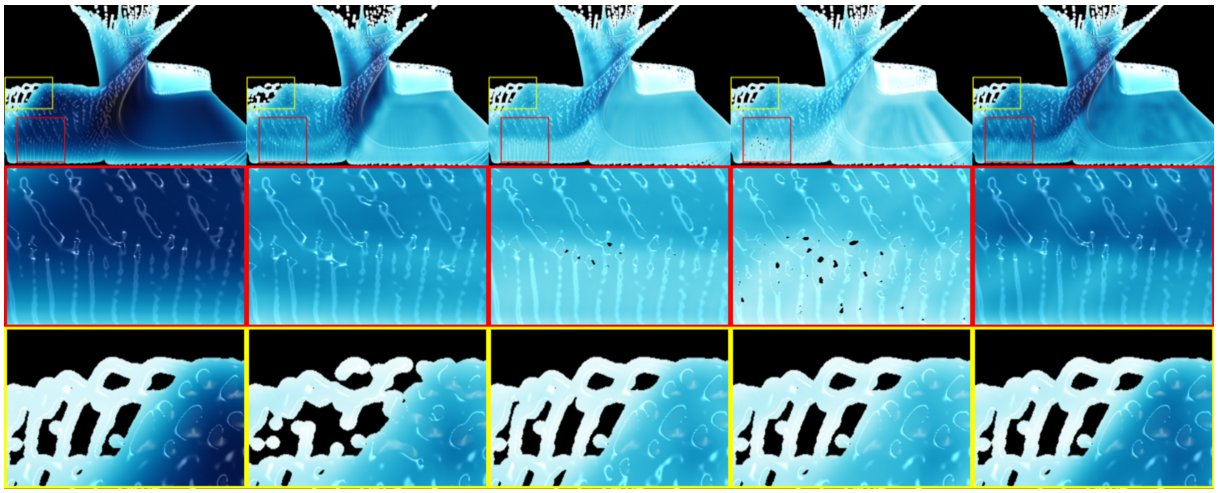
Figure 44 – Comparison of the amount of particles present in $\mathcal{L}_2$ and the amount of particles that are picked by Sphere Intersection and the Asymmetry Value method in the first 100 frames of the Sphere in Ball scene.



Source: Elaborated by the author.

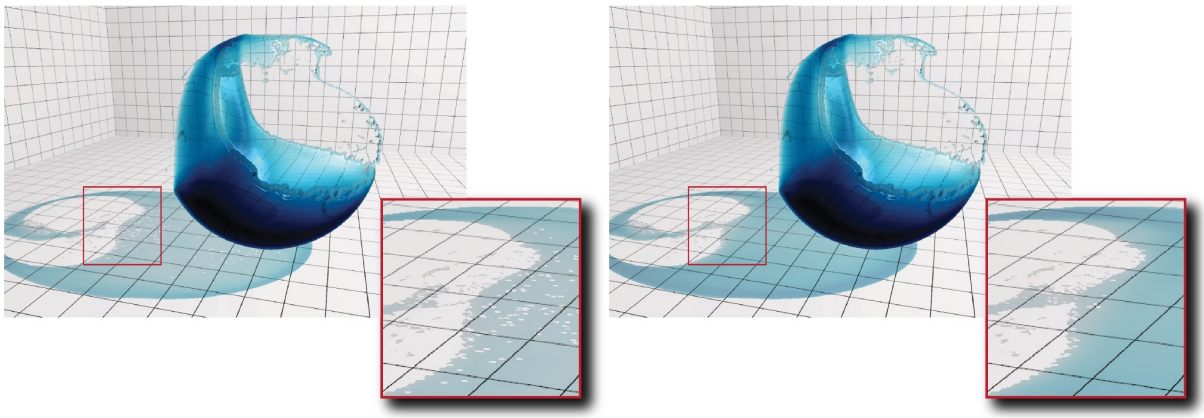
heavily outperforms these approaches, it is simpler and achieves a better result during rendering. While, according to Figure 45, all methods failed volume computation, our method will restore it, which is the topic of the next chapter. From now on all images rendered using narrow-band and screen-space are rendered with the simple formulation of $\mathcal{F}_2(p_i) = 1$. While there is no differences in image quality with the Sphere Intersection, Asymmetry method and picking all $\mathcal{L}_2$, we choose to increase the boundary extraction routine performance over having less particles to be rendered. The screen-space method is already very efficient and the differences in performance from using these approaches does not equally translates into final framerate so in this work we prefer to optimize for the fluid solver instead.

Figure 45 – Screen-Space comparison of different boundary extraction approaches. From left to right: Reference frame, Color Field method, Asymmetry method, Sandim and Ours.



Source: Elaborated by the author.

Figure 46 – Screen-Space comparison of the generated shadow for Color Field (left) and LNM (right).



Source: Elaborated by the author.

Algorithm 7 – Compute $\mathcal{S}$ for a given particle distribution $\mathcal{P}$ over the domain $\mathcal{D}$ in one pass

procedure NARROW-BAND($\mathcal{P}, \mathcal{D}, d, h, D$)
  $\mathcal{S} \leftarrow \{\}$ ▷ Initialize $\mathcal{S}$ to the empty set
  $n_1 \leftarrow (d/h)^D$ ▷ Equation 3.2.1
  threshold $\leftarrow 3^D$ ▷ $\mathcal{N}_{\mathbf{1x1}}$ neighborhood size for a given voxel
  for voxel $\mathcal{V}_i \in \mathcal{D}$ **do**
    $T_e \leftarrow \{\}$, $T_c \leftarrow \{\}$ ▷ Initialize utility sets for empty and small voxels
    $s \leftarrow$ NumberOfNeighboringVoxels($\mathcal{V}_i$) ▷ Number of neighbors to $\mathcal{V}_i$ in $\mathcal{N}_{\mathbf{1x1}}$
    $n \leftarrow$ NumberOfParticles($\mathcal{V}_i$) ▷ Number of particles in voxel $\mathcal{V}_i$
    if $n > 0$ and $s \neq$ threshold **then** ▷ Handle edge cases over $\mathcal{N}_{\mathbf{1x1}}$
      $\mathcal{S} \leftarrow \mathcal{S} \cup$ GetParticles($\mathcal{V}_i$) ▷ $\mathcal{V}_i$ is $\mathcal{L}_1$
    else if $n > 0$ **then**
      solved $\leftarrow$ false ▷ Flag for looking for $\mathcal{L}_2$
      for voxel $\mathcal{V}_j \in$ NeighborsOf($\mathcal{V}_i$) **do** ▷ For every neighbor in $\mathcal{N}_{\mathbf{2x2}}$
        $n_j \leftarrow$ NumberOfParticles($\mathcal{V}_j$)
        $c_j \leftarrow$ IsPairN1x1($\mathcal{V}_i, \mathcal{V}_j$) ▷ Check if $\mathcal{V}_j$ and $\mathcal{V}_i$ are within $\mathcal{N}_{\mathbf{1x1}}$ range
        if $n_j == 0$ **then** ▷ Empty voxel
          if $c_j ==$ true **then**
            solved $\leftarrow$ true
            $\mathcal{S} \leftarrow \mathcal{S} \cup$ GetParticles($\mathcal{V}_i$) and break ▷ $\mathcal{V}_i$ is $\mathcal{L}_1$
          end if
          push $\mathcal{V}_j$ into T_e ▷ $\mathcal{V}_j$ is an empty voxel outside $\mathcal{N}_{\mathbf{1x1}}$
        else if $c_j ==$ true and $n_j \leq n_1$ **then**
          $s \leftarrow$ NumberOfNeighboringVoxels($\mathcal{V}_j$)
          if $s \neq$ threshold **then**
            solved $\leftarrow$ true
            $Q \leftarrow \mathcal{F}_2$(GetParticles($\mathcal{V}_i$)) ▷ Apply $\mathcal{F}_2$ over $\mathcal{V}_i$
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{Q\}$ and break ▷ $\mathcal{V}_i$ is $\mathcal{L}_2$
          end if
          push $\mathcal{V}_j$ into T_c ▷ $\mathcal{V}_j$ might allow $\mathcal{V}_i$ to be $\mathcal{L}_2$
        end if
      end for
      if solved $\neq$ true **then**
        for pair $(\mathcal{V}_e, \mathcal{V}_c) \in \{T_e, T_c\}$ **do**
          if IsPairN1x1($\mathcal{V}_e, \mathcal{V}_c$) **then** ▷ Check if voxels are within $\mathcal{N}_{\mathbf{1x1}}$
            $Q \leftarrow \mathcal{F}_2$(GetParticles($\mathcal{V}_i$)) ▷ Apply $\mathcal{F}_2$ over $\mathcal{V}_i$
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{Q\}$ and break ▷ $\mathcal{V}_i$ is $\mathcal{L}_2$
          end if
        end for
      end if
    end if
  end for
  return $\mathcal{S}$
end procedure

CHAPTER

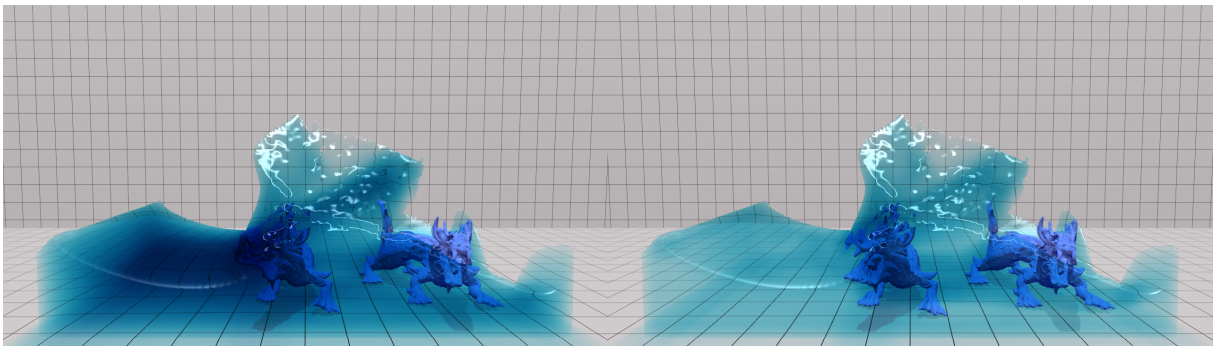
4

VOLUME RESTORATION

In the previous chapter we have discussed different approaches for boundary extraction and proposed a simple search method which heavily samples the outermost particles of the fluid in order to generate a better surface for the screen-space pipeline. This surface is built by using $\mathcal{L}_1$ and part of the $\mathcal{L}_2$ class Voxels, reducing the amount of particles missing during sampling, avoiding possible sampling errors that can happen and increasing performance by simplifying the scene. However one key aspect of the screen-space technique, that is highly desirable, is lost: the ability to obtain a volume representation. Recall that the original pipeline, proposed by Simon Green, leverages the fact that it is possible to perform splatting on all particles and obtain a representation that is fit to compute an approximation of the Lambert-Beer law that can be used to represent volume (Chapter 2). Fluid volume contribution is very important to allow perception of different fluid regions, in Figure 47, for example, we display a single frame of a simulation where a water block splashes against 2 dragons. The image on the left is rendered using the complete simulation and the image on the right the narrow-band only. Note how depth changes the perception of the underwater dragon surface. Volume also contribute to depth perception for different fluid regions, Figure 48 shows a *Double Dam Break* like scene with three cylinders as obstacles rendered in a cartoon style, where we can notice that more particles are concentrated in the center of the fluid allowing for a custom stylistic final image.

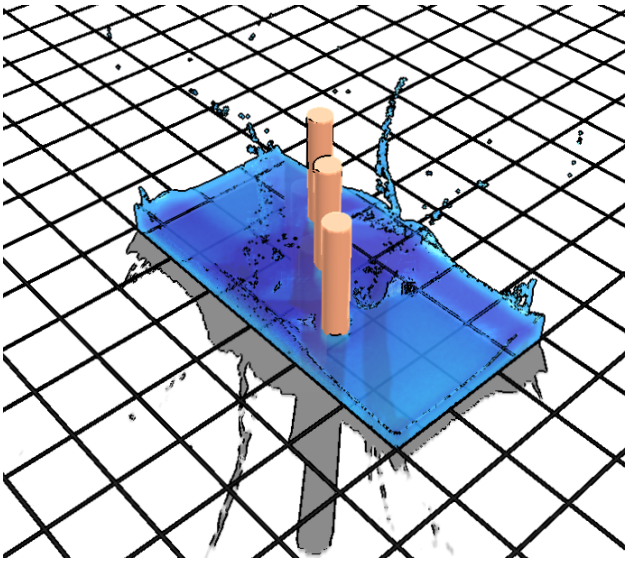
In order to get a volume estimation one would need to somehow transverse the fluid, from the viewpoint, and compute "how much fluid" is perceived. The usual way to do this would be by shooting a ray from a viewpoint V through the center of the pixels in the image plane and, from the found particle p, march through the fluid and compute the distance required to exit it. This distance is then used to estimate how much attenuation a ray of light would perceive when performing the *backward* path to the viewpoint V, a reference can be seen in Figure 49.

Figure 47 – Comparison between boundary only fluid (right) and the complete one (left).



Source: Elaborated by the author.

Figure 48 – Cartoon like rendering of a fluid using the screen-space approach.



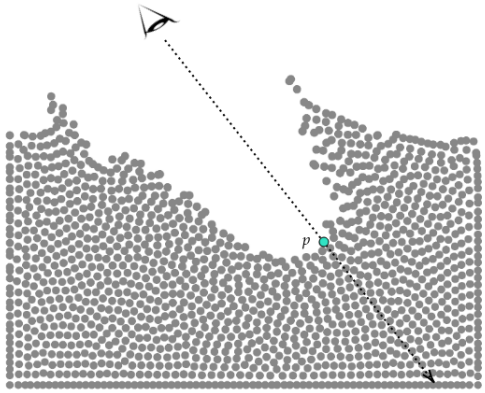
Source: Elaborated by the author.

A simpler approach can be constructed by accumulating some property that can represent the amount of particles that were discovered before exiting the fluid, and we can write that the volume estimator $V_p(\vec{d})$ from a particle with position p on a direction $\vec{d}$ is

$$V_p(\vec{d}) = \sum_t P(p + t\vec{d}) \tag{4.1}$$

where P is some positive per particle property that can be compute at position $p + t\vec{d}$, with t a positive parameter. The screen-space method smartly solves the previous equation by rendering all particles again without depth testing and performing alpha blending. This procedure has the ability to automatically accumulate whatever is written to the output color buffer without having to perform any Ray Tracing/Ray Marching operation. However

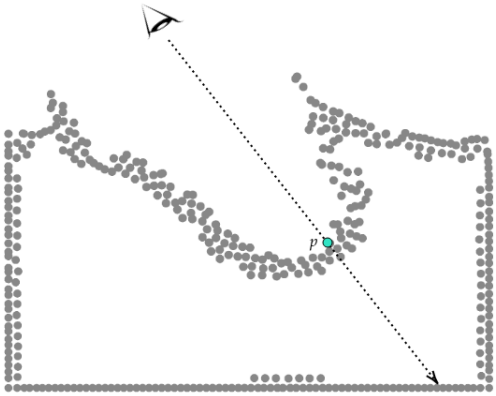
Figure 49 – Marching through the fluid.



Source: Elaborated by the author.

when we removed the interior particles from the fluid, using a narrow-band extraction method, we created the situation in Figure 50. It is no longer possible to transverse the fluid and compute Equation (4.1) as the accumulation will no longer give a correct representation of the particle distribution inside the fluid and thus giving results like the Figure 47 (right), where the fluid surface does not show any volume.

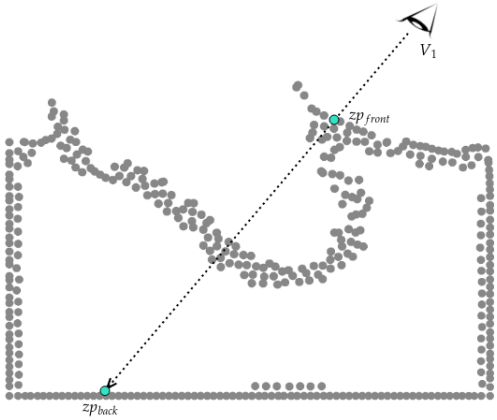
Figure 50 – Marching through the fluid with the narrow-band only.



Source: Elaborated by the author.

One might be tempted to perform a two pass render scheme where the *front* and *back* part of the fluid are rendered computing the z values, zp_{front} for the closest point and zp_{back} for the furthest, and then perform a estimation of volume by computing the relation $V_p = zp_{back} - zp_{front}$. This however assumes particles in the interior of the fluid are regularly distributed and will discard any complex particle configuration where the given ray from V (mentioned earlier) intersects multiple *front* and *back* segments in splash regions, Figure 51 shows such a situation where the difference $zp_{back} - zp_{front}$ is not a valid

Figure 51 – Attempting to compute $zp_{back} - zp_{front}$ gives a incorrect estimator for volume.



Source: Elaborated by the author.

approximation of volume when considering the viewpoint V_1. If we wish to re-introduce the volume property of the screen-space pipeline we will need to fill the fluid with *something* that can be used to estimate Equation (4.1). In this work we will, once again, rely on the formulation of the fluid solver in the same way we did in Chapter 3 and make use of the regular grid that is available for neighbor querying. Instead of splatting the particles from the fluid we will splat the *grid itself*. Usually the number of voxels present in the fluid volume is significantly smaller than the number of particles, in our tests we have seen that even for a high resolution grid where voxels have side $l = 4r$, with r being the particle radius, voxel rendering heavily outperforms particle rendering as they can be seen as a *blob* that represents a set of particles and can be optimized for memory consumption. Table 3 show a comparison between the number of particle and the maximum number of voxels (containing particles) for a few simulations presented before and Figure 52 shows this *blob* being rendered as particles, intensity varies with the amount of particles inside each voxel. Note that even if Figure 52 shows spherical voxels we actually render voxels as full quads.

Scene	**Nº Particles**	**Nº Voxels**	**%**
Rotating Sphere	523,195	127,449	24.3
Double Dam Break	405,360	258,720	63.8
Water Drop	739,522	218,988	29.6

Table 3 – Comparison between the number of particles and the maximum number of voxels (that have at least one particle inside of it) for different scenes using a high resolution grid. Their relation can be seem in the final column.

Rendering the *Voxel Blob* allows us to solve Equation (4.1) and therefore get a volume estimation, we introduce volume by weighting the splat by the amount of particles

Figure 52 – *Voxel Blob*. Each voxel is rendered as spherical point sprite, color varies with the amount of particles inside, darker areas represent less particles.

Source: Elaborated by the author.

present inside the voxel during the computation. Given a voxel with δ particles and a pixel inside the quad at position $\mathbf{x}$ we compute the contribution for volume estimation through alpha-blending to be:

$$\mathbf{V}_{\mathbf{x}} = \frac{\beta\delta}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{d_{\mathbf{x}}}{\sigma})^2} \tag{4.2}$$

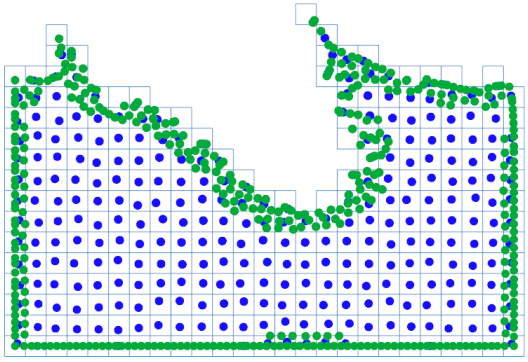
where β is a user defined parameter for controlling the intensity of the operation and $d_{\mathbf{x}}$ the distance from position $\mathbf{x}$ to the origin of the quad. However splatting through Equation 4.2 will allow artifacts to occur because our grid is uniform, and therefore voxels with the same amount of particles δ will give the same composition in the final buffer. We need to *follow* the particle distribution whenever particle distribution is not uniform, so instead of rendering the grid uniformly, we render the voxels as full quads with center matching the centroid of the particles inside of it., i.e.:

$$C_v = \frac{1}{\delta}\sum_i p_i \tag{4.3}$$

where δ is the amount of particles inside the voxel and p_i the position of particle i. Once again we can see that Equation 4.3 is very simple and can be computed as we are computing narrow-band extraction routines, again helping increase performance of our solution. Figure 53 shows a simple diagram of the rendering process, green particles represent the narrow-band extracted with our method while blue points are voxel's centroid.

Applying the centroid formulation helps improves the accuracy of the final image and removes artifacts that could happen at the boundary of the fluid because of the lack of particles in those voxels[†].

Figure 53 – Simple diagram of the rendering procedure, green particles represent the NB and blue points the voxel centroid being rendered.

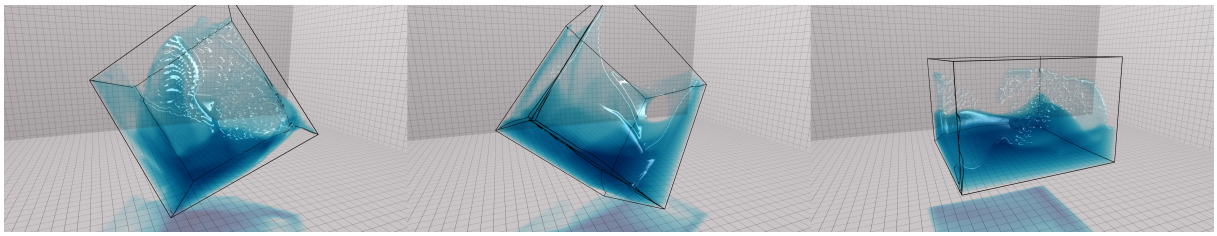


Source: Elaborated by the author.

The formulation we presented can be integrated with any filtering solution presented in Chapter 2 without any issues. Implementing the previous relations is also straightforward, a simple modification of the rendering pipeline is performed to use the voxels instead of particles and the shaders are slightly modified to account for Equation (4.2). Figures 54, 55, 56 and 57 shows different simulations rendered using the filters described in Chapter 2 with our narrow-band formulation. All rendering procedures were implemented using C++ and OpenGL. Narrow-Band extraction and simulations procedures however were implemented in CUDA. For these experiments we generated images with full HD resolution (1920×1080), all filters run with a size of 7×7 and 4 iterations but curvature flow runs with 30 iterations. Figure 58 shows the boxplots of the computational times for different strategies of screen-space fluid rendering. As can be seen, our narrow-band approach improves the performance considerably regardless of filter's choice, which demonstrates our method's efficiency. Overall for filters that require less iterations we find that our narrow band solution achieves a speedup of $\approx 2\times$, while still maintaining high quality rendering. Finally, Figure 59 shows different fps (frames per second) value for a single frame of the Happy Whale simulation (Figure 57) for all discussed filters. Note again that even in the worst case (curvature flow filter) our method still gives an excellent increase in performance ($\approx$ 15 fps) and in the best case (narrow range filter) it almost doubles the performance with up to 50 fps.

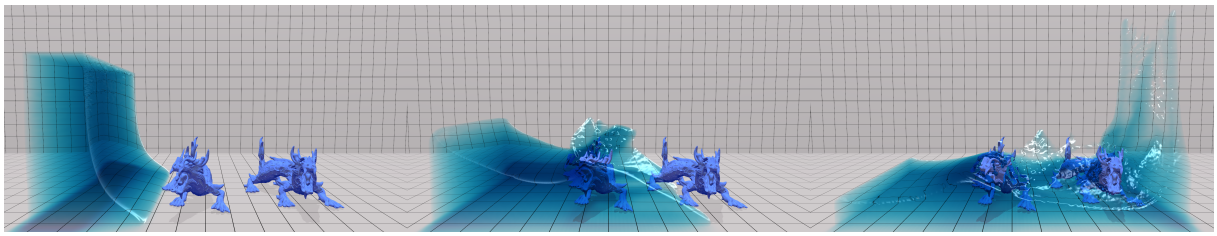
[†] Particles can group at a position that does not follow the center of the voxel and so rendering the center of the voxel would shift the volume.

Figure 54 – Rotating box simulation rendered using Plane Fitting with our NB method.



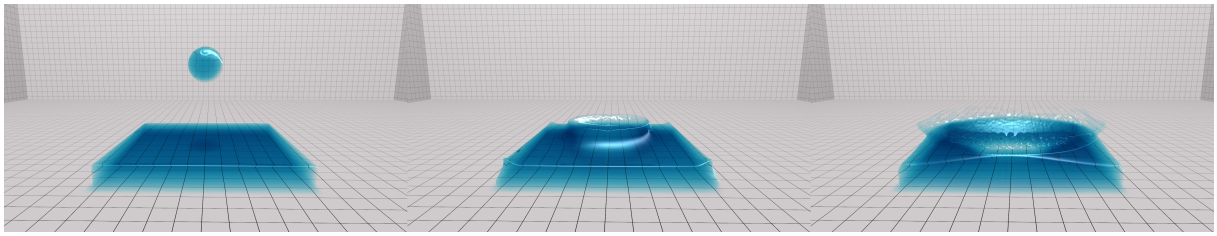
Source: Elaborated by the author.

Figure 55 – Dam break with obstacles rendered using Bilateral Filter with our NB solution.



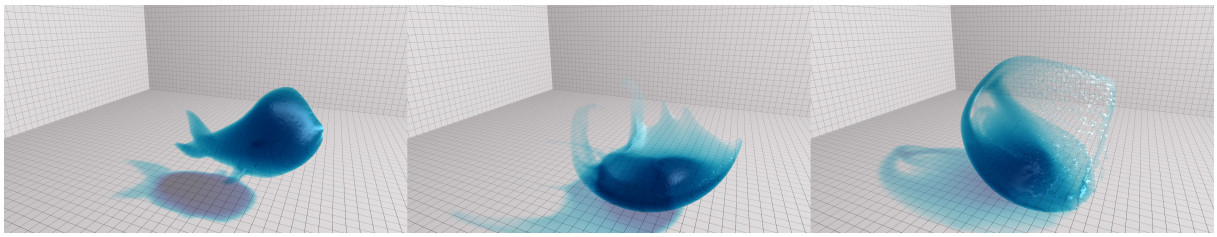
Source: Elaborated by the author.

Figure 56 – Water drop scene rendered using Curvature Flow with our NB solution.



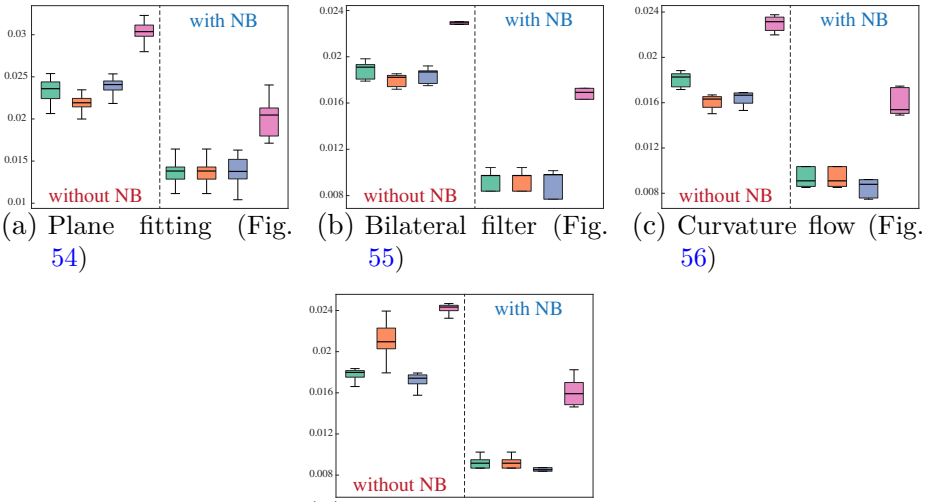
Source: Elaborated by the author.

Figure 57 – Happy Whale scene rendered using Narrow Range with our NB solution.



Source: Elaborated by the author.

Figure 58 – Boxplots of the computational times (in milliseconds) for screen-space rendering with different filters: bilateral filter (■), narrow-range (■), plane fitting filter (■) and curvature flow filter (■).

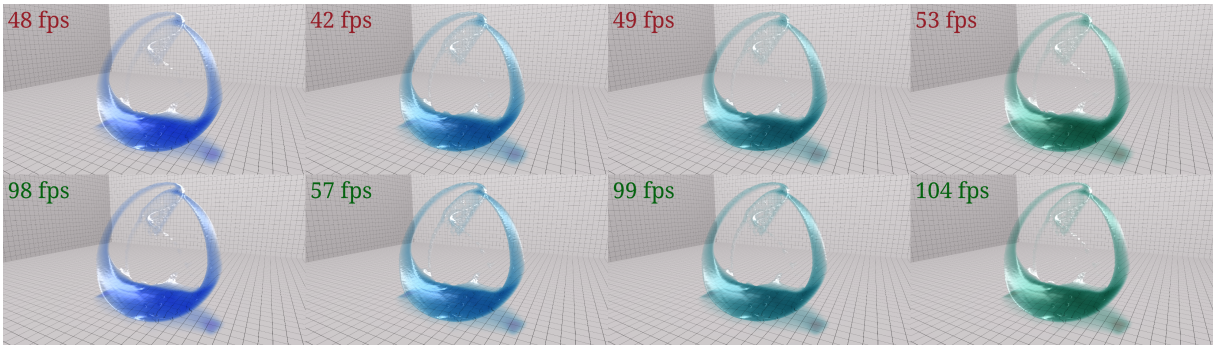


(a) Plane fitting (Fig. 54) (b) Bilateral filter (Fig. 55) (c) Curvature flow (Fig. 56)

(d) Narrow range filter (Fig. 57)

Source: Elaborated by the author.

Figure 59 – Comparison of the complete rendering of all filters presented in Chapter 2 (top) and with our formulation using the NB only (bottom). From left to right: Bilateral filter, Curvature Flow, Plane Fitting and Narrow Range filter.



Source: Elaborated by the author.

CHAPTER

5

CONCLUSION

Considerable work was done to assure that the LNM consistently helps different solutions to the Screen Space fluid rendering technique. We have shown, and compared, several examples of both NB solutions and rendering options, and discussed why using boundary alone is not enough to correctly render under the Screen Space conditions. The LNM also heavily increased performance of even slower, more robust, filters such as the Curvature Flow, helping, in some cases, achieve 60 frames per second for large simulations. It has a simple algorithm for computation that can easily be integrated with any particle-based simulator and, during rendering, it only changes the computation of volume, which would not be possible using the boundary particles. However even if the LNM successfully integrates with different filters for the Screen Space method, it still needs improvement during its volume estimation. Because it renders voxels as macro-particles that hold δ particles inside of it, it generates a higher value of volume than the original pipeline would. This happens because all pixels inside the voxel quad will be considered to be part of the macro-particle and the value computed in Equation 4.2 will be slightly higher than one would expect, this forces the introduction of a compensation factor β to bring the distribution down. A better approach perhaps can leverage the fact that the values of $\mathbf{V_x}$ in Equation 4.2 do not need to specifically decay with a gaussian like curve but instead rely on some formulation that can assure that at any pixel $\mathbf{x}$, inside the voxel, its representation of the amount of particles δ, $\delta_\mathbf{x}$, is such that $\int \delta_\mathbf{x} = \delta$ for all pixels. Our method for boundary extraction also is limited to bounded computational domains due to the grid it uses to classify voxels. For simulations that have a very large domain this grid might start to become prohibitively expensive and a more robust solution such as dynamical hierarchical sparse grids (WU *et al.*, 2018) might be interesting. However our implementation currently does not support such structure. Even with these consideration we feel that the LNM strategy for faster Screen Space fluid rendering is a valuable contribution to anyone looking for ways to obtain better performance in fluid rendering while still maintaning high quality.

BIBLIOGRAPHY

ANDERSSON, P.; NILSSON, J.; SALVI, M.; SPJUT, J.; AKENINE-MÖLLER, T. Temporally dense ray tracing. In: **High Performance Graphics**. [S.l.: s.n.], 2019. Citation on page 20.

BITTERLI, B.; WYMAN, C.; PHARR, M.; SHIRLEY, P.; LEFOHN, A.; JAROSZ, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 4, Jul. 2020. ISSN 0730-0301. Available: <https://doi.org/10.1145/3386569.3392481>. Citation on page 20.

DORING, M. **Développement d'une méthode SPH pour les applications à surface libre en hydrodynamique**. 1 vol. (III-159 f.) p. Phd Thesis (PhD Thesis), 2005. Thèse de doctorat dirigée par Delhommeau, GérardAllessandrini, Bertrand et Ferrand, Pierre Dynamique des fluides et des transferts Nantes 2005. Available: <http://www.theses.fr/2005NANT2116>. Citation on page 45.

GREEN, S. Screen space fluid rendering for games. **SIGGRAPH**, 2010. Citations on pages 19, 20, 23, and 35.

HAQUE, A.; DILTS, G. A. Three-dimensional boundary detection for particle methods. **Journal of Computational Physics**, v. 226, n. 2, p. 1710 – 1730, 2007. ISSN 0021-9991. Citations on pages 55 and 57.

HE, X.; LIU, N.; WANG, G.; ZHANG, F.; LI, S.; SHAO, S.; WANG, H. Staggered meshless solid-fluid coupling. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 6, Nov. 2012. ISSN 0730-0301. Available: <https://doi.org/10.1145/2366145.2366168>. Citations on pages 42 and 43.

IHMSEN, M.; ORTHMANN, J.; SOLENTHALER, B.; KOLB, A.; TESCHNER, M. SPH Fluids in Computer Graphics. In: LEFEBVRE, S.; SPAGNUOLO, M. (Ed.). **Eurographics 2014 - State of the Art Reports**. [S.l.]: The Eurographics Association, 2014. ISSN 1017-4656. Citation on page 19.

IMAI, T.; KANAMORI, Y.; MITANI, J. Real-time screen-space liquid rendering with complex refractions. **Comput. Animat. Virtual Worlds**, John Wiley and Sons Ltd., GBR, v. 27, n. 3–4, p. 425–434, May 2016. ISSN 1546-4261. Available: <https://doi.org/10.1002/cav.1707>. Citation on page 31.

KATZ, S.; TAL, A.; BASRI, R. Direct visibility of point sets. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 3, p. 24–es, Jul. 2007. ISSN 0730-0301. Citation on page 48.

KAZHDAN, M.; BOLITHO, M.; HOPPE, H. Poisson surface reconstruction. In: **Proceedings of the Fourth Eurographics Symposium on Geometry Processing**. Goslar, DEU: Eurographics Association, 2006. (SGP '06), p. 61–70. ISBN 3905673363. Citation on page 37.

KIM, D. **Fluid Engine Development**. [S.l.]: CRC Press, 2016. 300 p. ISBN 9781498719926. Citations on pages 39 and 65.

LAAN, W. J. van der; GREEN, S.; SAINZ, M. Screen space fluid rendering with curvature flow. In: **Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games**. New York, NY, USA: Association for Computing Machinery, 2009. (I3D '09), p. 91–98. ISBN 9781605584294. Citation on page 29.

LORENSEN, W.; CLINE, H. Marching cubes: A high resolution 3d surface construction algorithm. **ACM SIGGRAPH Computer Graphics**, v. 21, p. 163–, 08 1987. Citation on page 37.

MARRONE, S.; COLAGROSSI, A.; TOUZé, D. L.; GRAZIANI, G. A fast algorithm for free-surface particles detection in 2d and 3d sph methods. **Proc. 4th SPHERIC Int. Workshop**, 01 2009. Citations on pages 46 and 47.

RANDLES, P. W.; LIBERSKY, L. D. Smoothed particle hydrodynamics: Some recent improvements and applications. 1996. Citations on pages 45 and 46.

ROURKE, J. **Computational Geometry in C**. [S.l.]: Cambridge University Press, 1998. 392 p. ISBN 9780521649766. Citation on page 49.

SANDIM, M.; CEDRIM, D.; NONATO, G. L.; PAGLIOSA, P.; PAIVA, A. Boundary detection in particle-based fluids. 2016. Citation on page 48.

SELLERS, G.; WRIGHT, S. R.; HAEMEL, N. **OpenGL Superbible: Comprehensive Tutorial and Reference**. [S.l.]: Addison-Wesley Professional, 2015. 880 p. ISBN 9780672337475. Citation on page 30.

SHAH, M.; KONTTINEN, J.; PATTANAIK, S. Caustics mapping: An image-space technique for real-time caustics. **IEEE transactions on visualization and computer graphics**, v. 13, p. 272–80, 03 2007. Citations on pages 19 and 53.

TRUONG, N.; YUKSEL, C. A narrow-range filter for screen-space fluid rendering. **Proc. ACM Comput. Graph. Interact. Tech.**, The Association for Computers in Mathematics and Science Teaching, USA, v. 1, n. 1, Jul. 2018. Citations on pages 33 and 34.

WU, K.; TRUONG, N.; YUKSEL, C.; HOETZLEIN, R. Fast fluid simulations with sparse volumes on the gpu. **Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018)**, v. 37, n. 2, p. 157–167, 2018. Citation on page 79.

XIAO, X.; ZHANG, S.; YANG, X. Fast, high-quality rendering of liquids generated using large scale sph simulation. **Journal of Computer Graphics Techniques (JCGT)**, v. 7, n. 1, p. 17–39, March 2018. ISSN 2331-7418. Available: <http://jcgt.org/published/0007/01/02/>. Citation on page 52.

ZHU, Y.; BRIDSON, R. Animating sand as a fluid. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 24, n. 3, p. 965–972, Jul. 2005. ISSN 0730-0301. Available: <https://doi.org/10.1145/1073204.1073298>. Citation on page 19.


ICMC
USP
SÃO CARLOS