

TDK-dolgozat

Kiovics Dániel
2024.



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Editing fluid simulations using differentiable screen-space rendering

Scientific Students' Association Report

Written by:

Dániel Kiovics

1st year master's student

Supervisor:

dr. Márton Vaitkus,
senior lecturer

2024.

Contents

Absztrakt	i
Abstract	ii
1 Introduction	1
2 Related work	3
2.1 Fluid simulation	3
2.1.1 Navier-Stokes equations	3
2.1.2 Major fluid simulation approaches	4
2.1.3 Hybrid simulation method discretization	4
2.1.4 Specific hybrid simulation techniques	5
2.2 Fluid surface visualization	6
2.3 Differentiable rendering	7
2.4 Fluid state reconstruction	8
3 Visualization	9
3.1 Rendering particles	9
3.2 Fluid surface	10
3.2.1 Depth filtering	10
3.2.2 Rendering the fluid surface	12
3.2.3 Transparency	12
3.2.4 Spray and surface noise	13
4 Fluid state reconstruction	14
4.1 The error function	14
4.2 Gradient computation	14
4.2.1 Stochastic gradient estimation	15
4.2.2 Computing the per-pixel parameter lists	15
4.2.3 Gradient smoothing	17

4.3	Optimization	18
4.3.1	Adam	18
4.3.2	Density control	19
4.3.3	Geometry based approach	19
4.3.4	Physics based approach	20
5	Implementation details	22
5.1	Architecture	22
5.2	Render engine	23
5.3	Fluid simulation engine	23
5.4	Visualization	23
6	Evaluation	25
6.1	Gradient estimation correctness	25
6.1.1	Evaluation based on gradient visualization	25
6.1.2	Error function value	26
6.2	Fluid state reconstruction	27
6.2.1	Depth vs. RGB based error function	27
6.2.2	Optimizing using particle vs. surface based visualization	28
6.2.3	Geometry vs. physics based reconstruction	28
6.2.4	Simple drawings as reference images	29
6.3	Simulation from the reconstructed fluid state	30
6.4	Performance	30
6.5	Comparative analysis	32
6.6	Limitations	32
7	Conclusion	34
7.1	Future work	34
	References	35
	Appendix A	37

Absztrakt

A valós (vagy közel valós) idejű folyadékszimulációk tudományterülete igen sokat fejlődött az utóbbi időben a nagy mennyiségű, párhuzamos számításra specializálódott hardveres és algoritmikus megoldásoknak köszönhetően. Ezek a szimulációs eljárások a megfelelő megjelenítési megoldásokkal párosítva nagyban hozzá tudnak járulni egy jelentős felhasználói élmény növekedéshez a szórakoztatóiparban.

A fejlesztés során tervezői/fejlesztői oldalról gyakran felmerülhet igény a szimuláció testre szabására és finomhangolására. Általánosságban elmondható, hogy ezt a szimulációs paraméterek variálásával és a kiindulási állapot manuális változtatásával szokták elérni, ami meglehetősen időigényes. Minderre egy potenciális megoldás lehetne a folyadék képtérbeli szerkesztésének lehetősége, mely alapján automatikusan lehetne módosítani a háttérben rejlő geometriát. Ehhez hasonló módszerek kaptak némi figyelmet az utóbbi években, viszont ezek nagyrészt csak szilárd geometriákkal foglalkoztak.

A TDK dolgozatomban a differenciálható renderelés alkalmazását fogom bemutatni részecske alapú folyadékszimulációs eljárások esetében. A folyadékok képtérben való szerkesztésének lehetőségét értékelem ki az imént említett párosítás esetében, a nemrég publikált sztochasztikus gradiens becslési algoritmus [7] különböző verzióinak és egy sztochasztikus gradiens optimalizáló algoritmus kombinálásával.

Algoritmikus és gyakorlati szempontból is bemutatom egy APIC [11] (affine particle-in-cell) szimulációs motor és egy képtérbeli folyadékfelület megjelenítő eljárás [20] létrehozását. Továbbá a részecske alapú raszterizációs megjelenítő és a szimulációs motor részecske paraméterek szerinti differenciálhatóvá tételének lehetőségeit is részletezem.

Abstract

The field of real-time (or near real-time) fluid simulation has experienced a rapid development with the advent of high-performance parallel computing and efficient, modern algorithms. Fluid simulations can greatly enhance the immersive experience for the consumer when combined with a suitable rendering technique.

From the artist’s/developer’s side the need for more precise control over the fluid’s behavior might arise during development. Generally speaking, this is usually solved by tweaking the simulation parameters and the initial state of the fluid, which tends to be cumbersome. A potentially easier approach would be to edit the fluid state in screen space and modify the underlying geometry accordingly. Similar techniques have seen rapid development in recent years, under the name of differentiable or inverse rendering, but so far such methods were mostly limited to solid objects.

In this article we demonstrate the use of differential rendering for a particle-based fluid simulator. Using this combination, we explore the possibility of editing the fluid state and behavior from screen space. We test multiple versions of a recently published stochastic gradient estimation algorithm [7] and demonstrate the results when combined with a stochastic gradient based optimizer.

We discuss the algorithmic and practical implementation of an APIC [11] (affine particle-in-cell) hybrid fluid simulation engine and a screen space fluid surface renderer [20]. Furthermore, we also go into detail about differentiating a particle-based fluid rasterizer (and potentially the fluid simulation engine itself) with respect to the particle parameters.

Chapter 1

Introduction

Fluid simulation has been used for a wide range of problems since its emergence in the 1950's and 60's [5][6]. Since then different methods have been developed for use in a variety of fields, including aerodynamics, weather forecasting, and computer graphics. While most of these methods initially were exclusively used for scientific computations and modelling, in the recent decades they have appeared in a lot of other fields.

One of the best examples is the entertainment industry, where the likes of Pixar and Disney have pioneered the use of fluid simulations to simulate different substances in their movies [12]. Nowadays, with the advent of cheap, accessible hardware capable of high performance parallel computation, running these simulations has become a viable option for the end-user on their own computing devices. This has started a revolution in the gaming industry, and resulted in the appearance of realistic looking fluids in games as well as in consumer grade animation software like Blender.

When animators design such scenes containing fluid simulations, they usually want some sort of control over the behavior of the fluid. Generally speaking, this is achieved by fine-tuning the initial simulation parameters – however this approach is not very efficient and consumes a significant amount of time. A better solution would be to enable defining and editing the fluid state from screen space – this way the artists would be able to make small changes to the way the fluid looks in the viewport, or even make the fluid have a certain shape just by specifying a reference image.

In this paper I present the inner workings and implementation details of a hybrid fluid simulator [19, 11]. I also demonstrate the use of screen space fluid surface rendering [20] in case of such a hybrid fluid simulator. I explore different possibilities of achieving the aforementioned screen space fluid state editing capabilities using a recently published gradient estimation algorithm [7] and a gradient optimizer. My main contributions to this field of research are the following:

- I have successfully created and implemented a novel screen space fluid state editing technique.
- I have adapted the previously mentioned gradient estimation algorithm for use with screen space fluid surface rendering.
- I have demonstrated how most parts of my method can be parallelized on the GPU (Graphics Processing Unit).
- I have explored the different possible variations of my method and their capabilities.

The following parts of the paper are structured into 6 chapters. In the Related work chapter I briefly present the relevant scientific literature in the field, including some overview of the methods used in the following chapters. The Visualization and Fluid state reconstruction chapters contain detailed description about my modified version of screen space fluid surface rendering and my fluid state editing technique, respectively. Implementation details, including the description of my fluid state rendering and optimization framework can be found in the Implementation chapter. In the Evaluation chapter I test the correctness of my fluid state reconstruction method, evaluate its performance for different inputs using a wide range of possible configurations and finally I discuss some of the limitations. I summarize the achieved results and write about possible future work in the Conclusion chapter.

Chapter 2

Related work

In this chapter I present the relevant research results and write a small recap about them.

2.1 Fluid simulation

There are lots of different fluid simulation approaches that have been developed over the years. It would be impossible to go through all of them, but there is a lot of great material about these subjects. The one I have been relying on the most while learning about fluid simulations is Robert Bridson's excellent Fluid Simulation book [4]. Another helpful source for getting familiar with this subject in a more practical way is Matthias Müller's Ten Minute Physics blog and videos¹.

2.1.1 Navier-Stokes equations

When modelling incompressible fluids (i.e. fluids in the classical sense and also gases when they are not compressed), the so called Navier-Stokes equations should hold throughout the fluid:

$$\frac{d\mathbf{u}}{dt} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla p + \mathbf{g} + \nu \nabla \cdot \nabla \mathbf{u} \text{ and} \quad (2.1)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (2.2)$$

where $\mathbf{u}$ is the velocity vector of the fluid at a given point, ρ is the density constant of the fluid, p is the pressure, $\mathbf{g}$ is the gravity (or any other acceleration that affects the fluid in a uniform way) and ν is the viscosity of the fluid – all units are SI.

Without going into much detail, the first equation is called the *momentum equation*, which describes how an infinitely small volume (particle) of the fluid behaves. It can be broken down into $\mathbf{a}_p = \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}_3$, where:

- $\mathbf{a}_p = \frac{d\mathbf{u}}{dt} + \mathbf{u} \cdot \nabla \mathbf{u}$ is the net acceleration of a particle at a given location, where the first term is the actual change in velocity at a given point (see subsection 2.1.2), and the second term is correcting for how much of that velocity change is caused by just the fact that another particle with a different velocity moves to this location,

¹<https://matthias-research.github.io/pages/tenMinutePhysics/index.html>

- $\mathbf{a}_1 = -\frac{1}{\rho}\nabla p$ is the acceleration caused by change in pressure at the given point,
- $\mathbf{a}_2 = \mathbf{g}$ is the uniform acceleration caused by (for example) gravity,
- $\mathbf{a}_3 = \nu\nabla \cdot \nabla \mathbf{u}$ is the acceleration (or deceleration) caused by the velocity difference in space and the friction (viscosity) in between particles.

The second Navier-Stokes equation 2.2 is simpler: it basically states that the velocity vector integral on a fluid volume's surface is zero – in other words the fluid cannot compress or expand, this is the *incompressibility equation*. Using these two equations it is possible to derive the fluid simulation algorithm.

2.1.2 Major fluid simulation approaches

There are two major fluid simulation viewpoints, which can be distinguished by the way they represent the fluid:

- **Lagrangian viewpoint:** the fluid is represented using particles. Each particle has mass, velocity and a very small volume.
- **Eulerian viewpoint:** the fluid is represented by a velocity field.

While the former method is more intuitive for the description of the particle velocities as they move through space according to the *momentum equation* 2.1, it is easier to make sure that the fluid satisfies the *incompressibility equation* 2.2 using the latter viewpoint. This is the reason why simulation methods that use a *hybrid* viewpoint are generally simpler to implement.

2.1.3 Hybrid simulation method discretization

The hybrid fluid simulation approach combines the Lagrangian and Eulerian viewpoints in order to perform each calculation on the viewpoint that is more fitting. In this case the fluid state is tracked at all time using particles (that have positions and velocities) and a velocity field. While mathematically speaking both the particles and the velocity field grid cells must be infinitely small, in reality it is necessary to define a discrete version of both in order to make this algorithm computable. The discretization of particles is fairly straight forward – a particle size must be defined as well as a finite particle count.

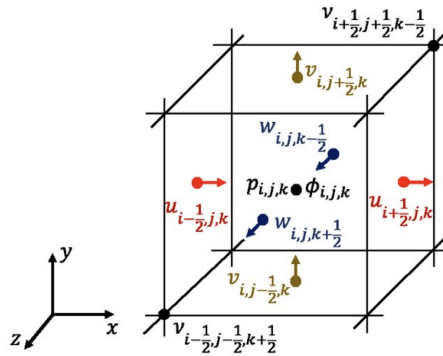


Figure 2.1: The cell structure of the MAC (marker-in-cell) grid

While selecting a fitting structure for the velocity field's grid, one must consider that the selected structure should make the computation of the calculations performed on the grid quicker and easier. A good choice of structure is the so called MAC (marker-in-cell) grid. This grid has a couple of special properties that can be seen on figure 2.1. Firstly, the velocity vectors are stored as individual components, each at the center of the corresponding cell face; secondly, the pressure p is stored in the middle of the cell – this combination of properties makes computing the incompressibility step a lot easier, as the pressure differential at the center of each face can be precisely computed.

The fluid particles should behave (as much as possible) in accordance to the Navier-Stokes equations. The simulation is divided into simulation iterations, each of which has a time duration of Δt , thus the equations must hold true in each iteration. Without going into too much detail about how the step decomposition is done exactly, this means that a simulation iteration consists of the following steps:

1. Advection – in this step the particles are moved according to the elapsed time and their velocity.
2. P2G (particle to grid) transfer – during this step the particle velocities are interpolated onto the velocity grid using some sort of 3D interpolation function.
3. Adding the uniform acceleration to the velocity field in accordance with the amount of time that has passed.
4. Solving the incompressibility criteria – the second Navier-Stokes equation must hold true at the end of this step for each cell. This can be done by modifying the pressure value at the center of each cell until the velocity field (influenced by the pressure values) does not satisfy the equation. Basically, this means finding the solution of a linear equation system, where each cell's pressure value is a variable².
5. G2P (grid to particle) transfer – basically the inverse of the second step, the velocities are transferred from the updated velocity field to the particles using interpolation.

I must point out that this is a very general description that does not handle a lot of small but important edge cases, like for example empty grid cells, solid obstacles, boundaries and much more.

2.1.4 Specific hybrid simulation techniques

All hybrid simulation algorithms follow the scheme described in the previous subsection. However, significantly different results can be achieved by modifying how exactly are the P2G and G2P steps executed. The original hybrid simulation papers [5][6] have used a method known as PIC (particle in cell) method. This method does a basic interpolation to and from the grid, just as described in the previous subsection. While this approach results in a very stable fluid simulation method, it inherently introduces a lot of smoothing (a certain kind of internal friction) into the simulation, which makes the fluid very viscous. This problem can be seen on figure 2.2.

Brackbill et al. tried to solve this problem using the FLIP (fluid implicit particle) method [3], which modifies the G2P transfer step by just updating each particle's original velocity by the change of the interpolated velocity instead of overwriting it. This method

²This can be done by a linear equation system solving algorithm, like the Gauss-Seidel or the MICCG (Modified Incomplete Cholesky Conjugate Gradient) algorithm.

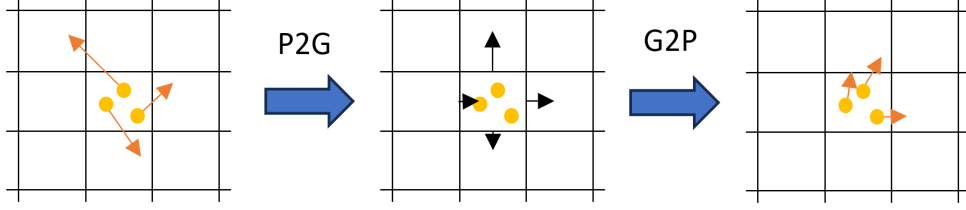


Figure 2.2: The smoothing introduced by a single iteration of PIC

did remove the excessive smoothing, but it also introduced significant noise into the particle velocities.

The noise problem was mostly also solved by *Jiang et al.* in their APIC (affine particle in cell) paper [11]. They have achieved this by keeping the PIC approach, while also introducing an extra matrix for each particle that stores the relationship between the velocities around it in the grid. This way, during the G2P step the differences between the grid velocities are not smoothed out (and lost) from being transferred into the single velocity vector of the given particle. Thus during the P2G step these differences can be restored and so the smoothing is almost completely eliminated without introducing any noise. The comparison can be seen in 2D on figure 2.3 – the difference is more apparent in 2D than in 3D.

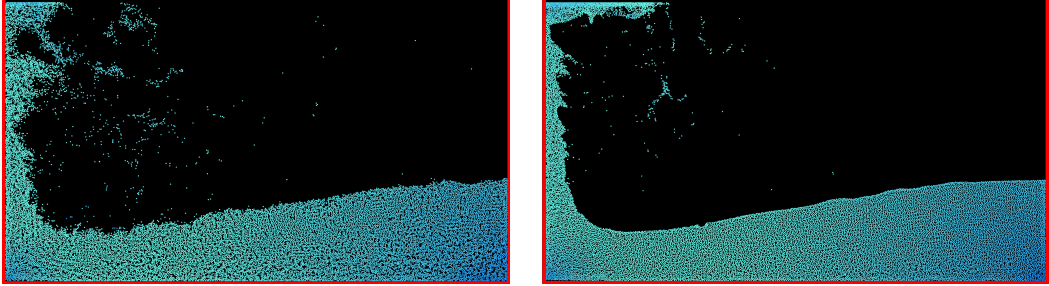


Figure 2.3: The noisiness of the FLIP method compared to APIC

2.2 Fluid surface visualization

There are two major approaches to the visualization of fluid simulation data: mesh based approaches and screen space methods. *Incahuanaco and Paiva* present an excellent overview comparison of these in their paper [10].

Mesh based methods usually rely on creating a level set function from the fluid data. Both in case of Lagrangian and hybrid simulation methods this can be achieved by interpolating the particle densities onto a grid – this grid can then function as the level set. Generally, the fluid surface mesh itself is generated using some version of marching cubes or tetrahedrons. These methods tend to suffer from having a blobby appearance, although work has been done by *Bhattacharya et al.* to mitigate this issue [2] by minimizing the thin-plate energy of the surface.

Screen space methods render the fluid data (particles) directly to the image buffer and do the fluid surface generation as a form of post processing rather than generating a mesh first. These tend to create more visually appealing fluid surfaces, but have a tendency to suffer of at least some distortions of the original particle data. However, they are still

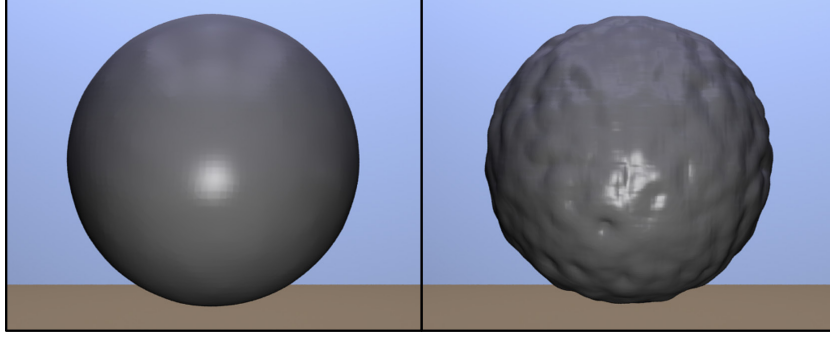


Figure 2.4: The method proposed by *Bhattacharya et al.* vs. a more traditional one

preferable for (near) real time applications, as the mesh based methods can get a huge performance penalty if the fluid spreads a lot in space and so the level set grid becomes very large. One such method has been presented by *van der Laan et al.* in their paper [20]. I base my fluid surface visualization on their work with a couple of added modifications. A more detailed description of the method can be found in chapter 3.

2.3 Differentiable rendering

Consider a scene of some sort with the parameters that are necessary for rendering it. This can be interpreted as a function $f : \theta \rightarrow \mathbf{I}$, where θ is a vector consisting of the scene parameters, $\mathbf{I}$ is the rendered image, and f is the rendering process itself. Now if one would want to determine what scene parameters would result in a certain reference image, than it is necessary to first define an error function $e(\mathbf{I})$, which tells how much difference there is between the reference image and the rendered image $\mathbf{I}$.

Clearly the goal is to find such scene parameters that would minimize the value of $e(f(\theta))$. In order to compute these, the gradient of function e and f is required – the former can be easily computed, as it is usually a simple numeric function. Computing the gradient of the latter is a lot harder and is still a field of active research called *differentiable rendering*. Most of the currently known methods rely on highly specialized rendering procedures using which some sort of gradient backpropagation can be performed – in a similar fashion as it is done when training neural networks.

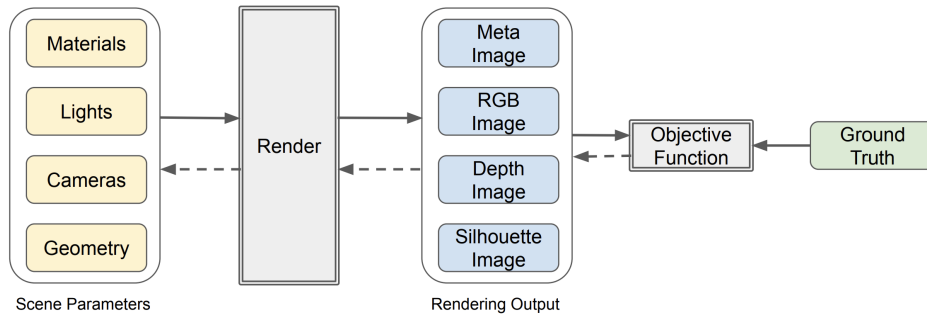


Figure 2.5: A basic overview of the rendering function f and its gradient

There are a lot of different viable approaches for achieving this, a great survey about these has been published by *Kato et al.* [13]. A recently published method [7] demonstrates

a different approach to this problem – in a lot of cases it is simpler to implement and does not require the previously mentioned, highly specialized rendering machinery. It is based on the idea of estimating the gradient in a stochastic way. In chapter 4 I write more about the theory behind this method, and also present how I have used it in order to achieve my initial goals.

2.4 Fluid state reconstruction

The reconstruction of fluids from images or videos have been researched in a large number of articles. Most of these apply the classical methods used in computer vision to reconstruct the fluid surface [17], but recently inverse/differentiable rendering techniques have come to the forefront [16]. In these works, the simulation is often purely Lagrangian or Eulerian, and the rendering uses mesh extraction or ray tracing/marching. In contrast, my work focuses on hybrid simulation and screen-space rendering.

A closely related approach to the one presented in this paper is the recently published *3D Gaussian Splatting* (3DGS) [14], that reconstructs a Gaussian point cloud, rendered via a specialized rasterization technique. The 3DGS literature is growing very rapidly, but its applications currently appear to be limited to solid scenes and objects. One exception is *Gaussian Splashing* [8] which integrates Gaussian splats with particle-based fluid simulation, but fluids are not actually reconstructed, only simulated along with reconstructed solids.

There is a large and diverse literature about the interactive animation and editing of fluids – for example *Pan et al.* [18] use keyframes and forces to influence the fluid’s behavior, but it is not based on inverse rendering.

Chapter 3

Visualization

The efficient rendering of large quantities of particles is a well known problem in the field of graphics and fluid simulation. There are countless solutions, ranging from methods fine-tuned for games while running on a single GPU to methods that use enormous GPU clusters to process and display particle systems consisting of billions of particles. In this chapter I demonstrate a real-time method mostly developed by me for rendering fifty to one hundred thousand particles – while retaining easy differentiability, as required for gradient-based optimization.

3.1 Rendering particles

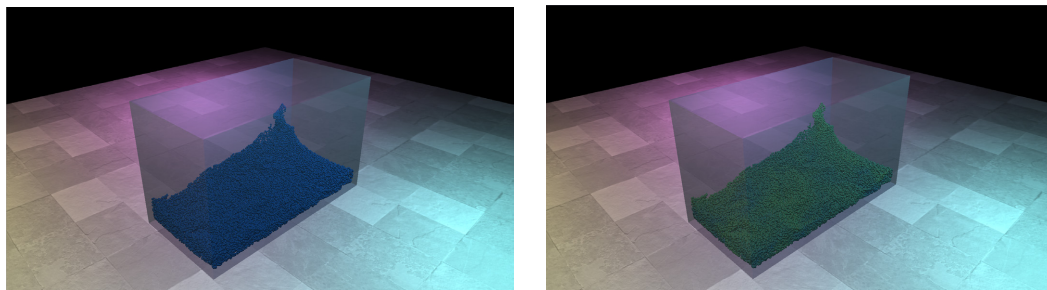


Figure 3.1: Particles rendered using a single color (to the left) and color interpolation (to the right)

A generic approach for visualizing particle based fluid simulators is to render the particles themselves. Using this technique one can achieve a fluid-like appearance that is modestly convincing, given that the average particle density per-pixel is high enough in the viewport. Furthermore, experience shows that even better visual quality is achieved when using a particle coloring scheme as the one shown on figure 3.1. This scheme assigns colors to particles in such a way that there is a smooth color transition between particles in space. One of the simpler solutions is to interpolate between $\mathbf{c}_1$ and $\mathbf{c}_2$ colors based on each particle’s $\mathbf{v}$ velocity – obviously the velocity value’s magnitude must be corrected according to the simulation settings using v_n :

$$k = \text{clamp}(\sqrt{|\mathbf{v}| \cdot v_n}, 0, 1) \quad \mathbf{c} = \mathbf{c}_1 \cdot (1 - k) + \mathbf{c}_2 \cdot k \quad (3.1)$$

In order to achieve a more realistic in-game appearance for the simulated fluid, it is rendered in a semi-transparent box on top of a well-lit floor. Lighting is done according to the well known Phong-Blinn model.

Rendering the particles themselves can be achieved by uploading a sphere mesh to the GPU and then using an instanced draw command – this is the naive approach. In this case only the per-instance position buffer must be updated before every frame, which improves performance. A more sophisticated approach is to render simple quads in place of complex sphere meshes, and use these as billboards that always face the camera [20]. In the pixel shader the exact pixel position on the sphere can be calculated in world space coordinates using texture coordinates, as well as the surface normal vector. This approach not only makes the particle edges smoother as can be seen on figure 3.2, but also further improves rendering performance.

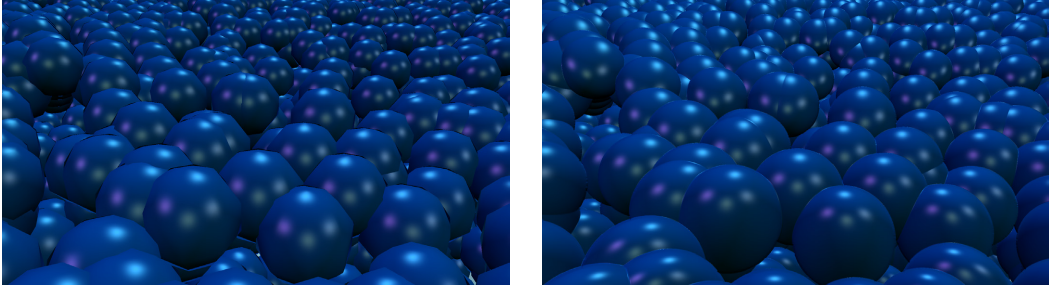


Figure 3.2: Particles rendered the mesh approach (to the left) and the billboard approach (to the right)

3.2 Fluid surface

Generally, fluid surface rendering techniques can be categorized into two large groups: mesh based and screen space. While the former can usually better represent the surface, it is also harder to implement. In this section I present my adopted version of the screen space fluid surface rendering technique [20] with some modifications. The high level flowchart overview for the computations of a single frame can be seen on figure 3.3.

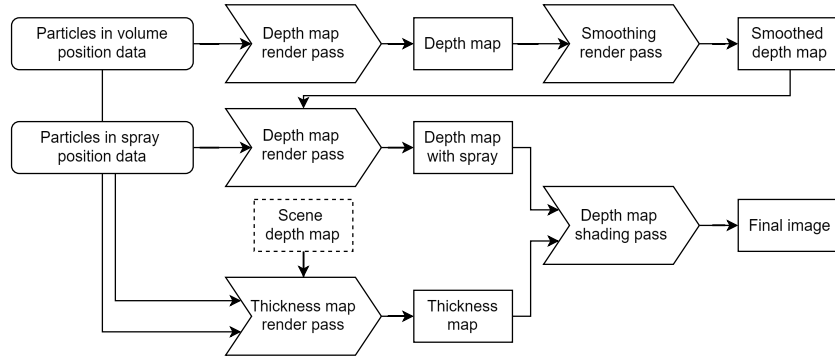


Figure 3.3: High level flowchart of the computations for a single frame

3.2.1 Depth filtering

At the start of each frame, all of the particle data is moved from the CPU to an SSBO (shader storage buffer object) on the GPU. Using the previously mentioned particle bill-

board rendering technique, I can compute the depth value in each pixel (the color data is not needed for now). By applying some sort of a smoothing filter to the resulting depth map, we can get a smoothed depth map, from which it is possible to reconstruct a smoothed fluid surface.

One such smoothing filter can be a generic Gaussian blur. In this case, a Gaussian smoothing kernel is applied to each depth map pixel according to the following equation:

$$d(x_0, y_0) = \frac{\sum_{(x,y)}^{kernel(x_0,y_0)} (e^{-\frac{x^2+y^2}{2\sigma^2}} \cdot depth(x, y))}{\sum_{(x,y)}^{kernel(x_0,y_0)} e^{-\frac{x^2+y^2}{2\sigma^2}}}, \quad (3.2)$$

where σ is the kernel width in pixels. Computing this is not feasible for larger kernels, because the runtime is $O(\sigma^2)$ – but we can use the separability property of the kernel in order to split the computation into two render passes: one for computing the one dimensional Gaussian blur in the x direction and one for computing final values by summing the x direction values in the y direction. This results in $O(\sigma)$ runtime, which is preferable.

The disadvantage of the Gaussian blur is that it tends to blend patches of fluid in the foreground into patches of fluid in the background. I mostly solved this issue by using bilateral Gaussian blur [1], which preserves the high frequency components (edges) of the image. The disadvantage of this approach is that the filter mathematically speaking is not separable, which in strict terms means that the runtime is $O(\sigma^2)$. But in practice a separable implementation results in artifacts that are only minimally perceptible. The blur’s equation is the following:

$$d(x_0, y_0) = \frac{\sum_{(x,y)}^{kernel(x_0,y_0)} (e^{-\frac{x^2+y^2}{2\sigma^2} - (depth(x_0,y_0) - depth(x,y))^2} \cdot depth(x, y))}{\sum_{(x,y)}^{kernel(x_0,y_0)} e^{-\frac{x^2+y^2}{2\sigma^2} - (depth(x_0,y_0) - depth(x,y))^2}} \quad (3.3)$$

It is worth pointing out that in both cases pixels with a depth value of 1 (i.e. pixels that do not have a fluid particle in them) must be ignored during the summation process, because these are invalid depth values. Without ignoring these, strange behavior would occur on the edges of the fluid and near small gaps between particles.

Another important detail is that the smoothing kernel must be of constant size in world coordinates, which means that its size varies in screen space. This is necessary because otherwise the smoothness of the fluid surface would depend on its distance from the camera. The smoothing kernel size is approximated for each pixel by first computing the camera space position (which for this purpose is equivalent to the world space position) from the depth value, then adding an offset equal to the desired smoothing kernel size, finally transforming it back to screen space and computing the resulting offset there. A pixel’s camera position can be determined from its depth value and texture coordinates according the following equation:

$$\mathbf{p}_{\text{ndc}} = (2x_0 - 1, 2y_0 - 1, 2d_0 - 1) \quad \mathbf{p}' = \mathbf{P}^{-1}\mathbf{p}_{\text{ndc}} \quad \mathbf{p} = \mathbf{p}'_{\text{xyz}}/\mathbf{p}'_{\text{w}}, \quad (3.4)$$

where at first the $\mathbf{p}_{\text{ndc}}$ device coordinates are calculated, and then from these the world space position $\mathbf{p}$ is determined using the inverse projection matrix $\mathbf{P}^{-1}$.

3.2.2 Rendering the fluid surface

According to the previously mentioned procedure, the world position of the fluid surface patch in each pixel can be computed. The world space surface tangents that correspond to the processed pixel can be determined by offsetting its position in the x and y directions and computing the world space position of these – then the surface tangents are $\mathbf{dx} = \mathbf{p}_x - \mathbf{p}$ and $\mathbf{dy} = \mathbf{p}_y - \mathbf{p}$. From this the surface normal can be determined by calculating the cross product of these vectors and normalizing the result. This way the fluid surface position and normal in each pixel can be determined, so it is possible to render a basic fluid surface from the smoothed depth map, as can be seen on figure 3.4.

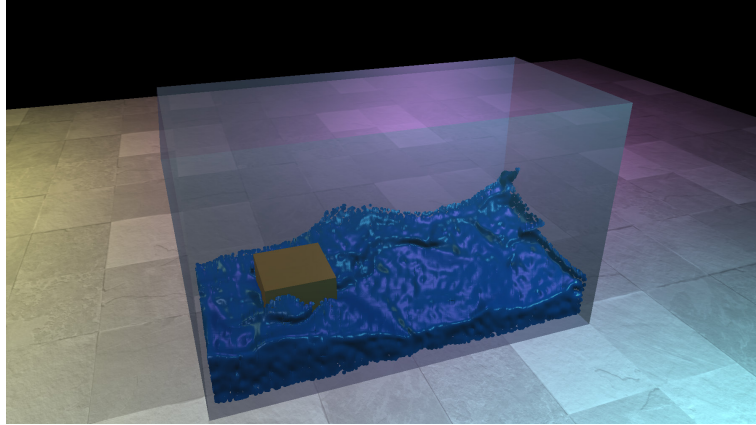


Figure 3.4: The shaded fluid surface splashing around a cube

3.2.3 Transparency

In case of transparent fluids, the fluid thickness (i.e. distance that the light ray has to travel through the fluid) directly determines how much light is able pass through the fluid without interruption – the rest is scattered or absorbed by the fluid particles. This is an inverse exponential correlation:

$$L(s) = L_0 \cdot e^{-\mu s}, \quad (3.5)$$

where L_0 is the initial light intensity and $L(s)$ is the intensity after traveling an s distance through the fluid.

In order to compute how much fluid is in between any given pixel and the first solid object in that direction, firstly I compute the "fluid thickness" texture, which contains the distances the light has to travel in case of each pixel before hitting a solid object. The computation consists of the following steps:

1. All particles are rendered in a similar fashion as described in section 3.1, with two major differences: while depth testing is enabled. there is nothing written to the depth buffer during the render pass, and the pixel shader's output is the given particle's thickness at that location, which gets additively blended to the render target texture value.
2. The resulting fluid thickness map (texture) is smoothed using the same technique described in section 3.2.1.

I achieve the desired fluid transparency by incorporating this thickness map into my fluid surface rendering method. Using additive alpha-blending, I am able to compute the color of each fluid surface pixel simply by setting a correct alpha value according to equation 3.5, where s is the value in the thickness map at the given position. From a strict physics perspective this is not correct, because the fluid color’s intensity is not necessarily the complement of the background solid object’s intensity, this approach still leads to a visually appealing result.

3.2.4 Spray and surface noise

In order to achieve a more appealing fluid effect, I demonstrate two further techniques using which I am able to improve the visual quality. The fluid surface generation technique presented in section 3.2.1 does not handle small individual particles well – as it tends to blend them into the background. One possible solution that I will refer to as *spray* is to render them separately from the rest of the particles (which form the fluid surface). This can be achieved by including an average particle density value for each particle in the input buffer, which can be extracted from the G2P step (section 2.1.3) of the simulation engine. The only necessary modification in the render pipeline to achieve this is to exclude particles with low density from the initial depth buffer computation and depth buffer smoothing, and render their depth values to the smoothed depth map according to figure 3.3 – this way the rest of the render tasks need no modification.

The resulting fluid surface tends to be unnaturally smooth because of too much depth smoothing. This can be mitigated by introducing some noise to the surface normals in order to create small waves on the fluid. This is achieved by using a 3D noise function that is computable on the GPU without the need for costly texture lookups [9], specifically the *Classic Perlin noise* function from the provided implementation¹. A given pixel’s world space position $\mathbf{p}$ and the neighboring pixel’s (in the x and y direction) world space positions are required to determine the normal vector at the mentioned pixel by calculating the cross product $\mathbf{n} = (\mathbf{p} - \mathbf{p}_x) \times (\mathbf{p} - \mathbf{p}_y)$. By altering these position according to the noise function’s value at the given position in the original pixel’s normal direction I am able to achieve the illusion of small indents in the fluid surface:

$$\mathbf{p} = f(\mathbf{p}_0) \cdot \mathbf{n}_0 \quad \mathbf{p}_x = f(\mathbf{p}_{x0}) \cdot \mathbf{n}_0 \quad \mathbf{p}_y = f(\mathbf{p}_{y0}) \cdot \mathbf{n}_0 \quad (3.6)$$

This results in a static wave pattern on the surface (figure 3.5), which can look a bit unnatural – however this issue is resolved by always sampling the noise function in two different places, and applying a time-dependent offset to both sample positions.

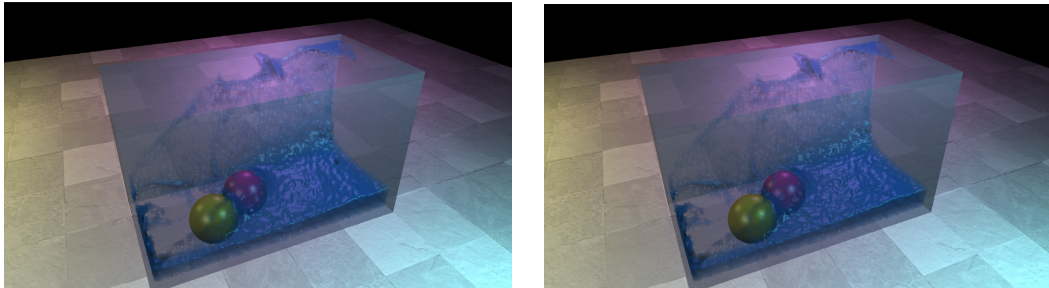


Figure 3.5: The transparent fluid without (to the left) and with the applied particle spray and noise (to the right)

¹<https://github.com/stegu/webgl-noise>

Chapter 4

Fluid state reconstruction

In this chapter I present the exact methodology and algorithms I have developed and used in order to achieve the desired ability to edit the fluid state based on a reference image. In the optimization world, the goal usually is to minimize the value of an error function by adjusting its (countless) input parameters. By computing the function's gradient it is possible to determine how the function's value changes at a certain location with a small change to its input parameters. An *error value optimizer* algorithm can then be used to minimize the error value by adjusting the input parameters according to the gradient. The method I present also aims to minimize the value of an error function.

4.1 The error function

The two possible error function candidates that I have used compute the error value using color and depth data. When broken down to a per pixel level the error functions are the following:

$$e_{(x,y)}(\theta) = |\mathbf{I}_{(x,y)}(\theta) - \mathbf{I}_{(x,y)}|^2 \text{ and} \quad (4.1)$$

$$e_{(x,y)}(\theta) = (\mathbf{D}_{(x,y)}(\theta) - \mathbf{D}_{(x,y)})^2, \quad (4.2)$$

where $e_{(x,y)}(\theta)$ is the error value of the pixel at (x, y) while using θ scene parameters for rendering, $\mathbf{I}_{(x,y)}(\theta)$ is the pixel's color and $\mathbf{D}_{(x,y)}(\theta)$ is the pixel's depth value when rendered using the aforementioned scene parameters, and the reference image's pixel color and depth values are $\mathbf{I}_{(x,y)}$ and $\mathbf{D}_{(x,y)}$ respectively.

Using these $e_{(x,y)}(\theta)$ error functions the image error function $e(\theta)$ can be computed as the sum of all pixel errors in both cases.

4.2 Gradient computation

Determining the gradient function of the error function is generally speaking a hard task. One possible solution is to compute it using a differentiable rendering engine (and a differentiable simulation engine in case of the physics based reconstruction described in subsection 4.3.4) such as the one *Kerbl et al.* present in their paper [14], but this requires heavy modifications and a lot of added complexity compared to a classic rasterization engine –

and we must also consider the fact that the engine might already be part of an application framework and thus hard to modify.

Another great solution was recently presented by *Deliot et al.* in their paper about stochastic gradient estimation [7], where they have shown a novel way to implement an efficient estimation procedure of the gradient function’s value at a given scene parameter position θ . In the following subsections I present the basic inner workings of the estimator and how I have adapted it for use in my screen space fluid state editing method.

4.2.1 Stochastic gradient estimation

Computing the error function gradient using a stochastic approach is trivial at first glance, but highly difficult achieve in an efficient manner. The classic approach is to add a small, randomly chosen offset to the scene parameters, compute the error function for these modified parameters, and finally determine the finite differential for each parameter independently – as if it was the only parameter that has been modified. This would be implemented as follows:

1. Perturb the scene parameter vector by adding and subtracting a randomly signed perturbation vector: $\theta_+ = \theta + \mathbf{s} \cdot \epsilon$ and $\theta_- = \theta - \mathbf{s} \cdot \epsilon$, where $\mathbf{s}$ is the randomly chosen sign vector (consisting of only 1’s and -1’s), and ϵ is the perturbation magnitude vector – this vector determines how much a given parameter is perturbed, the values for similar parameter types are usually the same.
2. The partial derivative in case of each parameter can be calculated by a finite differential: $\frac{df}{d\theta_i} = \frac{e(\theta_+) - e(\theta_-)}{\theta_{+i} - \theta_{-i}}$
3. By doing the partial derivative for each parameter the function gradient $\frac{df}{d\theta}$ is also determined, although with some noise in it, because the change in the function’s output can be caused by multiple parameters – but this can be solved by taking multiple samples with different $\mathbf{s}$ vectors.

The main problem with this approach is that given a large parameter vector it generally takes a very large number of samples to get even close to the real value. *Deliot et al.* have shown that this issue can be mitigated by only taking into account the error change that is directly caused by the given scene parameter while calculating the partial derivative of that parameter. In case of the previously mentioned error function for images, the function can be split into individual pixel errors. This yields the following equation:

$$\frac{df}{d\theta_i} = \sum_{(x,y)}^{\text{impacted by } \theta_i} \frac{e_{x,y}(\theta_+) - e_{x,y}(\theta_-)}{\theta_{+i} - \theta_{-i}} \quad (4.3)$$

According to this the only extra information I need from the rendering procedure to compute the stochastic gradient efficiently is a list of parameters that impact the error value in the given pixel – for all pixels.

4.2.2 Computing the per-pixel parameter lists

The per-pixel parameter list that impacts the error value in the given pixel is basically equivalent to the scene parameters of the object(s) that are visible in that pixel. This can

easily be computed in case of the particle visualization that is presented in section 3.1: by creating an extra render target (texture) and outputting to it the instance id (available in the vertex/pixel shaders) of the rendered particle – this way it can be determined which particles affect a given pixel, as can be seen on figure 4.1.

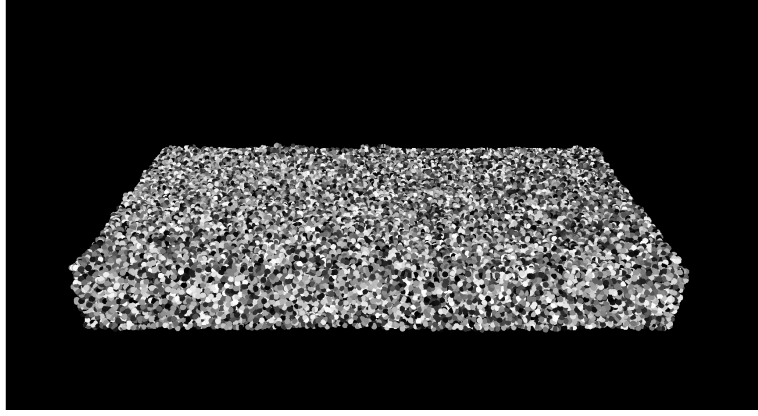


Figure 4.1: The parameter map: black is $id=0$ and white is $id=max$

In case of a fluid surface a slightly more complicated procedure is needed. In a similar fashion to the particle visualization case, particle ids are rendered into the aforementioned parameter map while rendering the initial depth map. As discussed in section 3.2.1, the depth map is first smoothed in the x and then in the y direction with a constant (world space) sized kernel. This implies that the depth in a pixel of the resulting depth map is impacted by the position of multiple particles. The list of these particles is equivalent to the union of the particle ids found at the locations of the pixels that are taken into account by the smoothing kernel:

$$\mathbf{L}_{\mathbf{x}}(x_0, y_0) = \cup \{p(x, y) : (x, y) \in \mathbf{K}_{\mathbf{x}}(x_0, y_0)\} \text{ and} \quad (4.4)$$

$$\mathbf{L}(x_0, y_0) = \cup \{\mathbf{L}_{\mathbf{x}}(x, y) : (x, y) \in \mathbf{K}_{\mathbf{y}}(x_0, y_0)\}, \quad (4.5)$$

where $p(x, y)$ is the particle id value at the (x, y) pixel, $\mathbf{K}_{\mathbf{x}}(x_0, y_0)$ and $\mathbf{K}_{\mathbf{y}}(x_0, y_0)$ are the sets of pixel positions which are part of the smoothing kernel in the x and y directions accordingly at the pixel given by (x_0, y_0) , and finally $\mathbf{L}(x_0, y_0)$ is the final list of parameters that have an impact on the pixel value at x_0, y_0 .

These computations do not require any extra render passes – I have integrated all of the computation into the x and y direction smoothing passes. For storage I have chosen SSBOs that store a fixed size array of integers (the particle id list) and a single integer (the count of the array elements) for each pixel. The size of the arrays must be adjusted accordingly to the maximum expected number of parameters in case of a single pixel, in direct correlation with the smoothing kernel size in both directions.

Using this method it is possible to determine the ids of particles that influence the color of the diffuse fluid surface at any given pixel, but the method still does not take into account the particle spray and the fluid transparency (the fluid surface noise is not useful for the optimization procedure, as it introduces randomness into the fluid surface, however if the need would arise, it could be also easily incorporated into the stochastic gradient computation in the form of offset and scale scene parameters that affects all pixels).

For the particle spray the proposed solution is the following: during the render pass that adds the low density particles to the depth map I also create another parameter (particle id) map. Using this I overwrite the final parameter list $\mathbf{L}(x, y)$ at those locations where a value has been written to the parameter map with a list containing only the id from the parameter map. Observe that because the depth of the particle spray is not smoothed, a pixel in which there is a spray particle in the foreground is only affected by the parameters of that particle.

The case of fluid transparency is more challenging, as it introduces a large amount of parameters for the majority of pixels. The reason behind this is that due to the additive blending (as described in section 3.2.3) each pixel is affected by a large number of particles. This would multiply by a factor of 20-100 the per-pixel parameter count and result in a large increase in the necessary size for the parameter arrays in the $\mathbf{L}_x(x, y)$ and $\mathbf{L}(x, y)$ lists as well as in a major increase in processing time. As a result, from now on I will only consider the diffuse fluid surface and particle spray during fluid state reconstruction.

4.2.3 Gradient smoothing

It can be derived from equation 4.3 that $\frac{df}{d\theta_i} = 0$ for all scene parameters that do not have an impact on any of the pixels in the viewport. As a result, considering that the optimization targets are images rendered using the particle or diffuse fluid surface approach, the previously mentioned partial derivative is going to be zero for each particle that cannot be seen from the viewport. This means for example that on figure 3.3 the majority of particles will receive zero gradients. As a consequence, the optimizer will not move or otherwise change the parameters of these particles – meaning that only the topmost few layers of the fluid are optimized, which can lead to slower optimization times.

Another issue can be seen on figure 4.2, namely that due to stochastic gradient estimation the gradient can have significant variations even on neighboring particles that should otherwise have a similar gradient vector (because they have to move in the same direction) even when using a lot of samples – 25 in this case. This can once again disrupt the optimizer, especially when using physics based reconstruction.

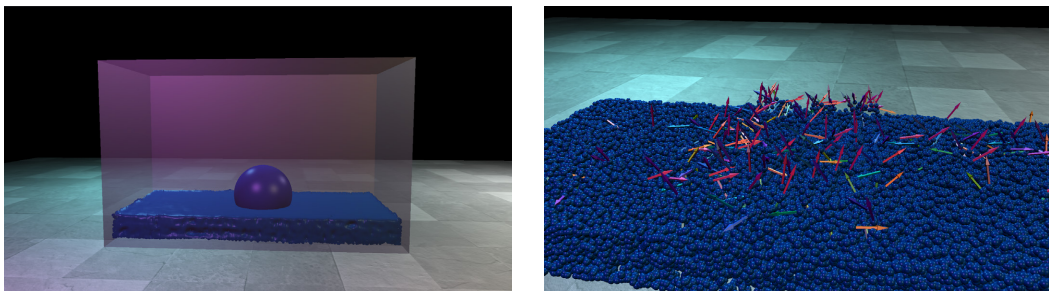


Figure 4.2: The reference image (to the left) and the position gradients at the start of the optimization (to the right)

A possible remedy is to smooth the gradient, i.e. make the gradient of the same parameters of neighboring particles similar. This can be achieved in various ways – my solution is based on simple Gaussian smoothing. The algorithm consists of two parts: first a 3D grid is computed with cell size equal to that of the smoothing kernel, each cell of which contains the particle indexes that are in the given cell. Then using this grid it is possible to efficiently apply the Gaussian smoothing to the particle gradient vectors. Further details can be seen in the algorithm’s pseudo code on listing 4.1.

```

input: k - kernel size, pos[n] - particle positions, grad[n] - particle gradient vectors
initialize 3D grid G[X][Y][Z]

foreach i in n:
    coord = pos[i] / k
    G[coord.x][coord.y][coord.z] = G[coord.x][coord.y][coord.z]  $\cup$  i

foreach i in n:
    grad[i] = 0
    weight_sum = 0
    coord = pos[i] / k
    cells = get_neighboring_cells(coord)
    foreach cell in cells:
        foreach index in cell:
            if |pos[index]-pos[i]| < k:
                w = Gauss(pos[index], pos[i], k)
                weight_sum += w
                grad[i] += grad[index] * w
    grad[i] /= weight_sum

```

Listing 4.1: Pseudo code for the gradient smoothing algorithm

Grid computation is done on the CPU, as it cannot be parallelized easily for running on the GPU, while the smoothing itself is done on the GPU. The result of the gradient smoothing can be seen on figure 4.3.

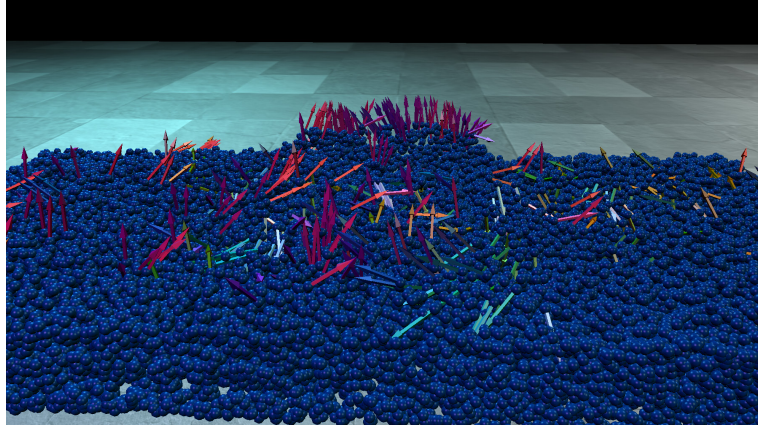


Figure 4.3: The smoothed particle position gradient at the start of the optimization

4.3 Optimization

The gradient estimation technique described in the previous section works well with gradient optimizers that were explicitly developed for use with stochastic gradients. One such optimizer is *Adam* [15], which is a well known method in the deep learning and AI community. In this chapter I demonstrate its use in my screen space fluid state editor and describe how the optimization is done on a high level.

4.3.1 Adam

The stochastic gradient optimizer developed by *Kingma et al.* generally speaking tries to find the minimum of the optimized function using the function's (stochastic) gradient.

One of the novelties of this approach is the use of moment vectors, which can help the optimizer avoid getting stuck in a local minimum of the function.

I use the first algorithm described in the paper [15] in my own optimization method, all computations are carried out in a compute shader on the GPU. As for parameters, based on practical experience the values described in the paper for β_1 , β_2 and ϵ work well, only the learning rate α needs to be adjusted.

4.3.2 Density control

During the fluid state reconstruction process it might occur that within given region the number of particles is simply insufficient to produce the desired shape. As a result, the reconstruction gets stuck in a local minimum that is hard to get out of because the optimizer would have to move a large number of particles by a lot to fill in the gaps. This problem is also described in other geometry optimization literature [14][7].

To alleviate the problems, I propose a density control method that moves particles from places where they are not needed to places where there are only few of them and the error is large at the same time. The algorithm consists of the following basic steps:

1. Compute the average particle movement magnitude in case of each particle during the last n optimization iterations using a rolling average (so that the most recent movements have the largest weight).
2. Sort the particles by average particle movement magnitude.
3. Move the bottom X% to the locations of the top X% with a small random offset, where X=10 works well in practice. Skip particles that have a density value higher than a predetermined threshold.

4.3.3 Geometry based approach

One possible way of achieving the ability to edit the fluid state from screen space is by direct movement of the particles. In this case the only modifiable scene parameters are the particle position vectors – this is a reasonable assumption as the goal is to edit the *fluid state* without changing for example its material properties. All of this can be achieved using the machinery described in the previous sections 4.2.1, 4.2.2, 4.3.1. The steps and the flow of data can be seen on figure 4.4.

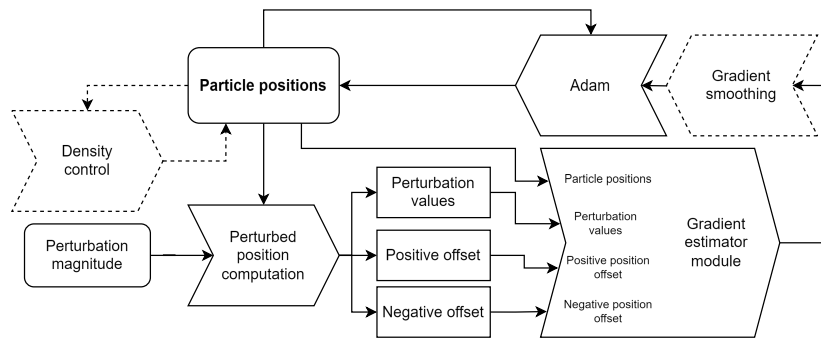


Figure 4.4: High level flowchart of the geometry based reconstruction

The *Gradient estimator module* is the one seen on figure 4.5.

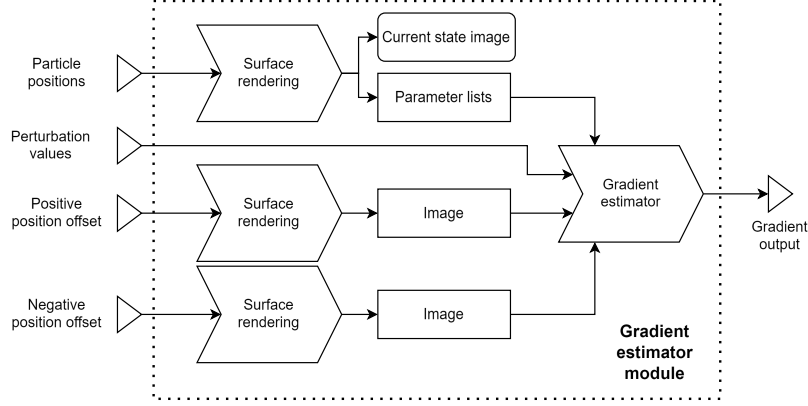


Figure 4.5: Stochastic gradient estimation module

The elements (computations) marked using dashed lines do not get always executed – they can either be configured as enabled or disabled, or get executed only after a certain number of iterations. Furthermore, utility steps that handle edge cases, such as the one that keeps the fluid inside the container’s boundaries have been omitted. All of the steps are executed almost exclusively on the GPU, meaning that there is no significant data movement between the CPU and GPU, which helps to improve performance.

4.3.4 Physics based approach

A seemingly more precise approach would be to integrate some physics into the fluid state reconstruction process. This need might arise from the fact that in some cases, although the geometry based approach generates a fluid state that looks almost the same as the reference image from the viewport, in reality the fluid’s state is physically implausible: it has various spikes and edges when viewed from another direction, as can be seen on figure 4.6. Now if the manipulation of the fluid would be done through the simulator by manipulating velocities for example, then the resulting fluid state should be more natural.

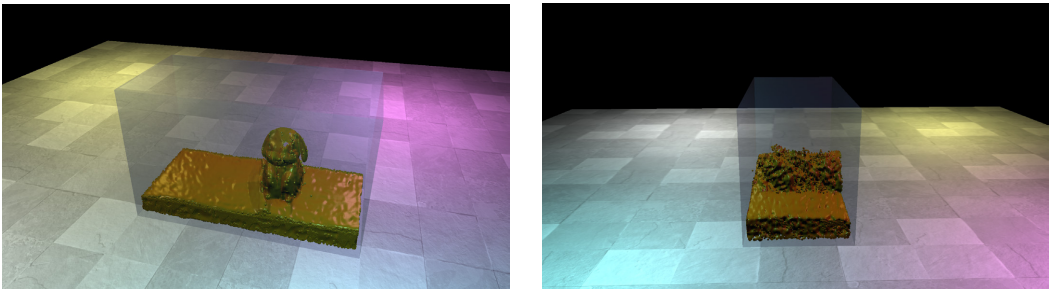


Figure 4.6: Reconstructed bunny viewed from different perspectives

The proposed physics based reconstruction approach uses particle velocities to adjust the fluid state. In this scenario the particle velocities (and not positions) are the scene parameters to be optimized. The stochastic gradient computation is also done through the simulator by the means of converting the velocity parameters into particle movements – one such conversion might have multiple simulation iterations with varying time steps. The error function and the parameter list calculations are the same as the ones described before, the simulation step is basically only a converter proxy between the optimizer and

the gradient estimator that transforms the velocity parameters into particle movement vectors. Another important property that must be considered is whether to reset the velocity parameters to zero after each optimization iteration or not – this will be analyzed in more detail in the evaluation chapter 6.

The flowchart of the method that can be seen on figure 4.7 is very similar to the one shown on figure 4.4, including the same *Gradient estimator module* shown on figure 4.5, with some notable extensions and modifications.

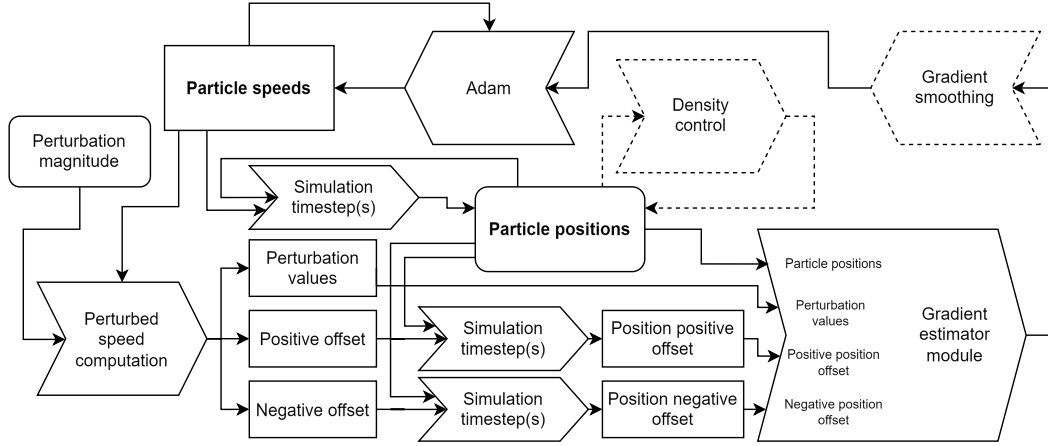


Figure 4.7: High level flowchart of the physics based reconstruction method

Chapter 5

Implementation details

In this chapter I present the used technologies, structure and inner workings of the test application and framework that I have developed during my research. All rendered images in the paper were generated using an application developed by me. The project source code is available in open source format on GitHub¹ – the base fluid simulation application (including the rendering engine) is available on the main branch, while the screen space fluid state reconstruction framework is on the `diff_rendering` branch. A small demo video is available on YouTube² that demonstrates the capabilities of the simulation application itself.

The project is written in C++, using as much of the modern C++17/20 features as possible. I have attempted to make the code modular and the individual components reusable.

5.1 Architecture

The project is structured into components, some of which are reusable utility components, while others use the aforementioned components. These components are:

- `RenderEngine` – this component is a collection of different OpenGL and GLFW functions. It wraps the C-style API of these libraries into a much more user friendly, (semi) object oriented interface, while also including some utility functions and classes for rendering.
- `Simulator` – this component contains the physics simulation engine and some helper classes.
- `imgui` – this is an open source, easy to use UI controls library called Dear ImGui³, with a small extension that I have made in order to make rendering all the controls easier in case of a large class hierarchy.
- `Visualization` – this component handles all of the fluid and scene rendering as well as it also contains the machinery needed for the fluid state reconstruction, including the gradient estimator, optimizer and the optimization methods as well.

¹https://github.com/dkiovics/fluid_simulator/tree/diff_rendering

²<https://youtu.be/Br7949G7z0o>

³<https://github.com/ocornut/imgui>

- Application – this is the wrapper component, that connects most of the other components together. It also handles the switching between different simulation and visualization modes.

5.2 Render engine

As it was mentioned in the previous section, the `RenderEngine` component contains all of the graphics related classes and functions. `OpenGL` and `GLFW` are used for rendering and creating the application window respectively. The most notable functions of this component are:

- GPU program classes – compute and shader,
- different types of storage classes like textures, framebuffers and render target textures,
- basic 2D and 3D geometries, a mesh loader and an instanced geometry rendering utility class,
- 2D and 3D camera, light and render object classes – these set the uniforms automatically according to their inner state.

5.3 Fluid simulation engine

This component is responsible for the fluid simulation itself, with a hybrid FLIP/APIC simulator at its core. All computations are done on the CPU using the OpenMP parallelization library. While moving these computations to the GPU might be possible it would also be quite challenging. In practice the CPU implementation was sufficiently performant for the current use case – the simulation of 60000 particles is possible at around 60 FPS (frames per second).

Most simulation parameters can be configured, including the exact type of simulation method (PIC/FLIP/APIC) as well as all of the simulation constants. The simulator can also handle solid spherical and rectangular obstacles, particle source and sink objects and more. There are two main operation modes: 2D and 3D – both use the same implementation, the only difference is that while running in 2D mode the particles can only move in a 2D plane.

In order to make the UI more responsive even in case of large scenes (over 100000 particles), the UI runs on a thread separate from those of the simulator. This requires some means of synchronization and communication between these threads, which is handled by the management class. This class is basically an interface between the inside of the component and all the other components and also handles the connections between the simulator, MAC grid, and particle classes as well.

5.4 Visualization

This component can be split into two subcomponents: one responsible for the actual scene and fluid rendering and one that contains the optimization methods and all of the related code. The former renders the scene with the transparent box and the moveable obstacle(s)

in both 2D and 3D, and also contains the particle and fluid surface renderer described in subsection 3.

The latter is responsible for the error function computation, the stochastic gradient estimation and for the parameter optimization. All of this blends seamlessly into the rest of the simulation application using the proxy design pattern. The subcomponent's most notable parts are the following:

- `Optimizer` – this is my implementation of the Adam optimization algorithm.
- `DensityControl` – my density control algorithm described in section 4.3.2.
- `GradientSmoothing` – my gradient smoothing algorithm in section 4.2.3.
- `GradientEstimatorPos` and `GradientEstimatorSpeed` – the gradient estimator implementation for geometry and physics based reconstruction respectively. They are not the same as show on figures 4.4 and 4.7, because the latter also contains the velocity-to-position conversion, but their interface is common.
- `DiffRenderProxy` – this class is the heart of the optimization submodule, it connects all of the listed classes as well as controls the reconstruction process according to the user settings entered into its own UI.

As mentioned before, the majority of the computationally intense tasks in this module are done on the GPU with some minor exceptions (for example the simulation step in the physics based reconstruction mode). This results in a highly efficient execution and low CPU load.

Chapter 6

Evaluation

In this chapter I evaluate the achieved results using different approaches and settings. I also discuss the performance and limitations of the methods that I have developed.

6.1 Gradient estimation correctness

For any optimization method it is critical that the error function gradient is correct, because this is the source of information about how the error changes with the change of input parameters. In my case, the stochastic gradient will always have some noise in it by nature, but obviously the smaller the average deviation of the computed stochastic gradient from the actual gradient the better.

6.1.1 Evaluation based on gradient visualization

One of the best ways to estimate the correctness of geometric parameters is to visualize them and compare the results to some sort of ground truth. In this case it is possible to estimate the ground truth using a fitting reference image and initial state. I have chosen a simple initial state where the fluid is resting, and the reference image is the same as the initial state with the addition of a rectangle in the middle of the container. All of this can be seen on figure 6.1.

Using this data, I am able to estimate where the gradient (or more precisely the gradient's opposite) should point in case of the geometry based optimization's *position* $\rightarrow$ *error* function. Considering that the fluid must be moved upwards in order to fill the rectangle seen on the reference image and reduce the error, it is trivial that the gradient must also point upwards where the rectangle is visible. This is marked as the red arrow on the initial fluid state image.

As I show on figure 6.1, the particle's gradients point as expected mostly upwards (in the area marked by red) following the rectangle's silhouette after a couple of optimization iterations. Although there is significant noise and deviation in between the gradient of neighboring particles, I can state that the gradient estimation is correct and produces the expected noisy output.

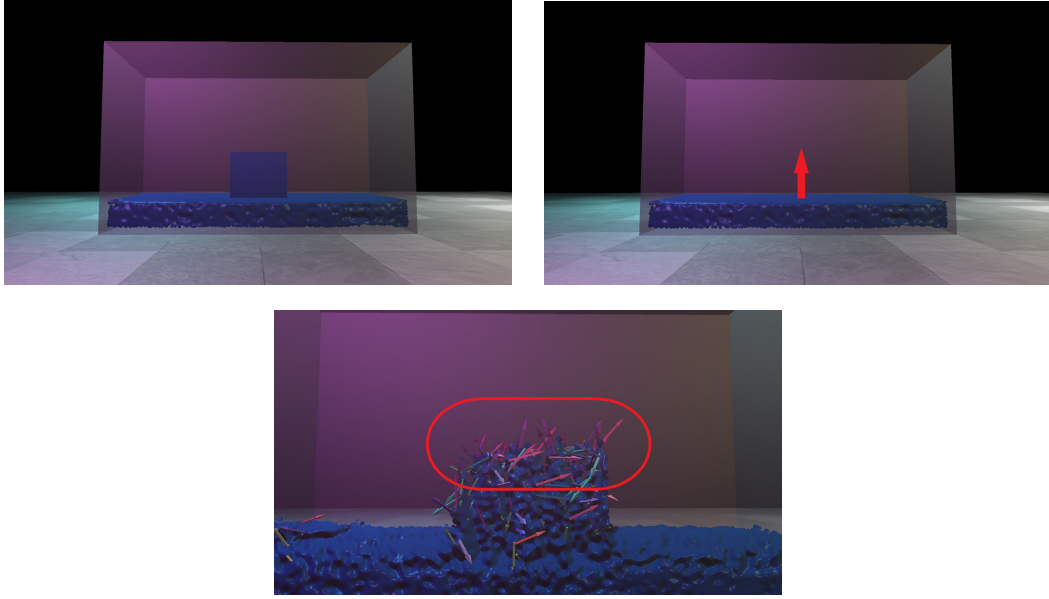


Figure 6.1: The reference image (top left), the initial state with the expected gradient (top right) and the actual gradient after a couple of optimizer iterations (bottom)

6.1.2 Error function value

Now that I have proven that the gradient estimation works correctly, it is also necessary to show that the optimizer really is doing what it is supposed to: changing the scene parameters in such a way so that the error function's value is decreasing. This statement can be proven by plotting the error function value during an optimization sequence. For the sake of simplicity, the reconstructed scene is the same as the one shown on figure 6.1.

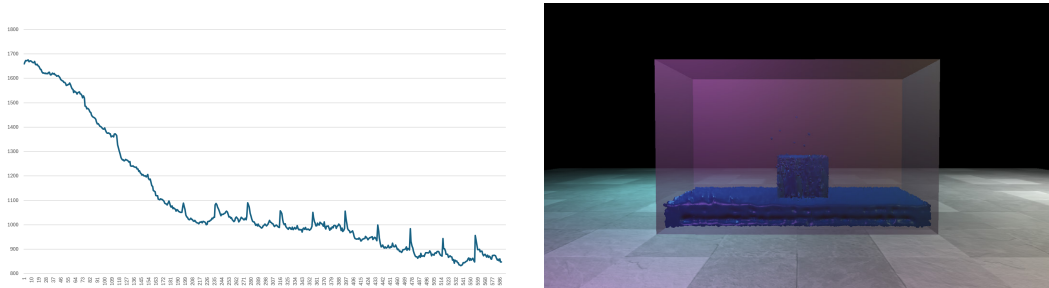


Figure 6.2: The error value's dependence on the number of iterations (left) and the result of the reconstruction (right)

The error function plot and the resulting reconstructed fluid state can be seen on 6.2. It can be seen that the error value has a strong downward tendency as expected with the increase of the iteration count. The noise in the function can be attributed to the fact that the gradient estimation is stochastic, while the periodic spikes are caused by my density control algorithm 4.3.2 – it rearranges the particles periodically, which causes a slight increase in the error for a short amount of time, but after this the error quickly decreases even further down than before. The reason why the error value stagnates at around 850 is that the final fluid state is reached and no more improvement is possible, because of multiple facts: the rectangle's color is slightly different than that of the fluid,

such sharp edges cannot be formed by the fluid surface and it is really hard to make the surface completely smooth.

6.2 Fluid state reconstruction

In this section I present different potential variations of my screen space fluid state editing method. It is important to mention, that although a lot of the reference images are simply rendered by the same fluid visualization technique (and in most cases with the addition of some object mesh), they are still *only* used as reference images, without any additional information other than the pixel information of the images – in other terms, these reference images could have been made by a skilled artist using a basic image editing program like Illustrator or Photoshop. Another small caveat is that in order to achieve nice reconstructed states the learning rate has to be reduced and the density control turned off near the end of the reconstruction – however, this can be easily automated.

6.2.1 Depth vs. RGB based error function

The stochastic gradient estimation algorithm presented in subsection 4.2.1 is able to use not only the color, but also the depth value in each pixel for the error function’s computation. Although only the color based approach is suitable for easy and intuitive fluid state editing from screen space using basic drawing methods, the depth based approach can also be useful if the user wants to recreate a mesh using the fluid’s surface – as this cannot be achieved easily by just arranging the particles on the boundaries of the mesh.

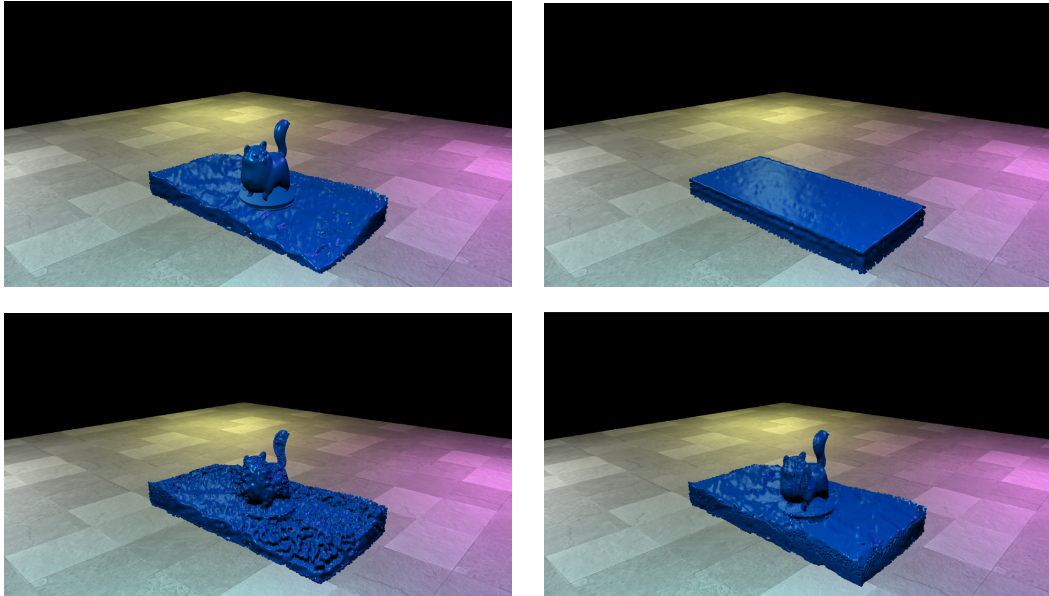


Figure 6.3: The reference image (top left), the initial fluid state (top right), the depth based result (bottom left) and the color based result (bottom right)

The reference image and the result of the reconstruction procedure in both cases can be seen on figure 6.3. While the depth based error function resulted in a slightly faster convergence, it is evident that the generated fluid state’s quality is not as good as in the

RGB case – this can be attributed to the fact that a small amount of variation in the depth seen in a given pixel might result in a much larger difference in terms of how the fluid surface looks.

6.2.2 Optimizing using particle vs. surface based visualization

The fluid surface visualization significantly enhances the fluid-like look compared to the particle based visualization, but the particle based visualization might still have some advantages when it comes to my the fluid state reconstruction technique. Firstly, there is a significant performance increase mostly because the parameter lists described in section 4.2.2 do not have to be calculated; secondly, particles can handle sharp edges in the reference image better in most cases, as they do not suffer from over smoothing.

One more considerable approach is to use the particle based visualization during the reconstruction phase, and switch to the fluid surface visualization only after it is done. In some rare cases this may be the best "middle ground" approach, although in this particular case seemingly the best option is to use the fluid surface visualization during reconstruction as well. The reference image (which is a fair bit more complex than the previous one, including sharp edges and small details), the results of the particle and surface based approaches, and the result of the aforementioned combination of the two can be seen on figure 6.4 – the initial fluid state is not shown, but is basically the same as on figure 6.3.

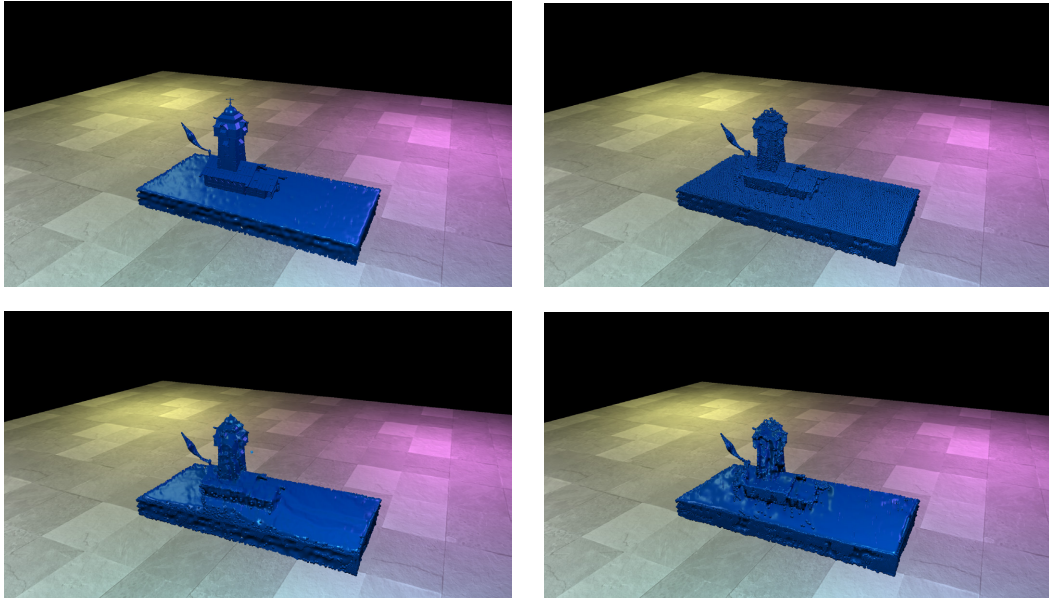


Figure 6.4: The reference image (top left), the particle based result (top right), the surface based result (bottom left) and particle-surface based result (bottom right)

6.2.3 Geometry vs. physics based reconstruction

In the previous subsections all of the tests were performed using the geometry based reconstruction approach, with sufficient results. But there are cases when a reconstruction approach such as the one presented in subsection 4.3.4 is preferable, as it also takes the

physics of the fluid into account. One such case might be when a more uniform distribution of particles is needed: in the previous figures, there was no guarantee that the particle density is anything close to constant in the reconstructed state – this can sometimes be a problem when continuing the simulation from the reconstructed fluid state, as it can lead to fluid implosions, explosions and weird behavior in general.

A couple of important considerations have to be taken into account when using the physics based fluid state reconstruction technique:

- the gradient must be smoothed at least a bit, because otherwise the optimization algorithm will only try to move the outer layers of the fluid, as described in subsection 4.3.2,
- whether or not to reset back to zero the particle velocities after each optimization iteration,
- and how many simulation iterations have to be done in between the optimization iterations and what should be the simulation time step.

Unfortunately, all combinations of the above mentioned considerations have some major flaw that stops them from working correctly and results in some form of undefined fluid state that (although physically correct) hardly resembles the target image as can be seen on figure 6.5. One of my theories as to why this is happening is that my hybrid fluid simulator presented in section 2.1 is not differentiable trivially in such a stochastic way. The other possible reason is that both resetting and not resetting the particle velocities after each iteration have their own major flaws:

- if the particle velocities are reset, then even with the gradient smoothing the fluid cannot build up enough velocity in order to overcome the internal friction introduced by the particle-grid-particle transfers,
- and if the velocities are not reset, then they get out of control – simply put, there is too much inertia and the optimizer cannot handle this.

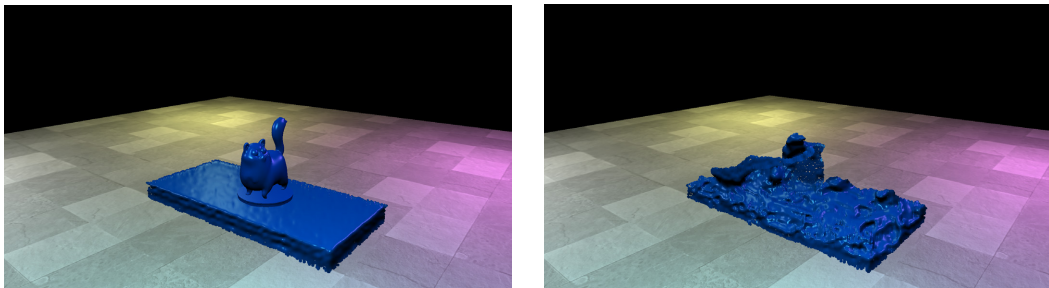


Figure 6.5: The reference image (left) and the image produced by the physics based optimizer without resetting the velocities (right)

6.2.4 Simple drawings as reference images

As mentioned at the beginning of this section 6.2, the reference consists of only pixel data, and as such one can easily edit the current fluid state in an image editor. On figure 6.6 an example can be seen for this use case. Although at first glance this is a simple reference image, in reality it is hard to achieve such a fluid state where the colors of the surface are as

uniform as on the reference image – still my fluid state reconstruction method performed as expected.

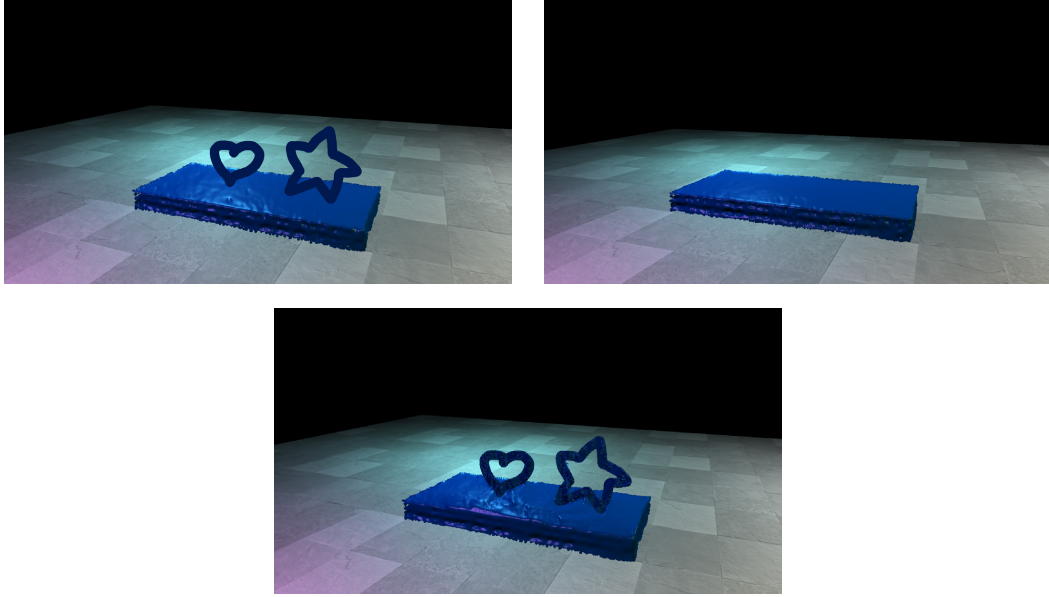


Figure 6.6: The reference image (top left), the initial fluid state (top right) and the fluid state after the reconstruction (bottom)

6.3 Simulation from the reconstructed fluid state

After the fluid state reconstruction from the reference image has achieved a satisfactory result, the resulting fluid state can be trivially transferred to the fluid simulation engine in order to continue the simulation from the edited state. Furthermore, it is also possible to enable once again the fluid transparency and the surface noise, as the reconstruction method has finished running. This can be seen in figure 6.7, which is the simulation from the reconstructed state achieved on figure 6.6. A couple of live demonstration of the reference image $\rightarrow$ state reconstruction $\rightarrow$ simulation can be seen on YouTube¹ (the reconstruction phase is sped up). Also, there are further sample images from different reconstruction scenarios in appendix A.

6.4 Performance

In this section I present the performance of my fluid visualization and fluid state reconstruction methods. All of the tests were done on a laptop with an AMD Ryzen 9 5900HS CPU and an Nvidia RTX 3060 (100W TDP) GPU in it – generally speaking, this is mediocre hardware at best for gaming and most computationally intensive tasks in the fall of 2024, so the results are representative of what an average user would at least get. In terms of scene complexity, the demonstrated performance is achieved on a medium sized scene consisting of around 50000 particles.

The fluid visualization is at the core of both displaying the fluid simulation (chapter 3) and also the fluid state reconstruction methods (chapter 4) because of the stochastic

¹<https://youtu.be/gm7qkTVk-nw>

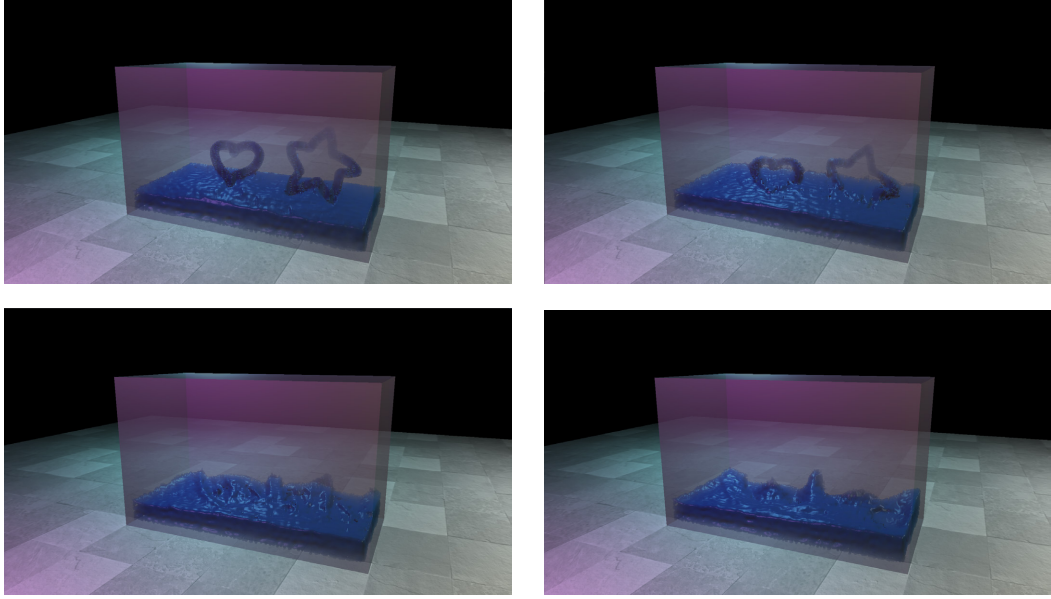


Figure 6.7: Simulation from the reconstructed fluid state

gradient estimation algorithm (subsection 4.2.1) – this is the reason why its performance is critical. The method I have presented relies heavily on the GPU and most of the computations are also done on it. It yields acceptable performance for most rendering applications, and probably would not cause any significant bottlenecks even in games, both in the particle and the fluid surface based visualization cases (with spray, transparency and surface noise enabled). The average FPS can be seen on table 6.1.

	Rendering	Geometry-based gradient estimation	Physics-based gradient estimation
Particle based vi- sualization	392 FPS	111 SPS	9.1 SPS
Surface based vi- sualization	307 FPS	55 SPS	7.2 SPS

Table 6.1: Rendering and stochastic gradient estimation performance

The stochastic gradient estimator’s performance can be measured in SPS (sample per second). As described in the fluid state reconstruction chapter 4.3.3 4.3.4, there are two possible approaches: geometry based and physics based. In case of the geometry based, all of the computation is done on the GPU, the most costly step is determining the per-pixel parameter lists in case of the surface visualization method, but this only needs to be done once at the beginning of every gradient sampling batch after the scene parameters have been altered by the optimizer – so I do not take this step into account during my performance measurement. As can be seen on table 6.1, the performance of the physics based reconstruction method’s gradient estimation is significantly worse. The reason for this is that the fluid simulation engine must be ran two times² for each sample for at

²The velocity to position transfer is performed by running 2 simulation iterations, each with a 20ms time step.

least one iteration (but most likely more) – in my implementation this requires moving data back and forth between the CPU and the GPU a lot and furthermore, the simulation engine’s CPU implementation itself is not fast enough for this application.

When running the fluid state reconstruction methods to achieve a fluid state that resembles the reference image as closely as possible, there is no real objective way to evaluate performance. On table 6.2 I record the time it takes to achieve a fluid state that mostly resembles the reference image and the time it takes to achieve the final state from which point on there is no significant improvement – the former is unfortunately a highly subjective metric. The tested reconstruction scenario is the one shown on figure 6.3 (the color based approach).

	Time until halfway done	Time until done
Particle based visualization	35s	142s
Surface based visualization	127s	331s

Table 6.2: Fluid state reconstruction performance

6.5 Comparative analysis

As I have mentioned before, I suspect that my approach is novel, so I was not able to find any closely related literature about editing the state of a fluid simulator using only a single RGB reference image and differentiable screen space fluid surface rendering. Thus, there is no other method I am trying to compete with. The closest related work is probably the Gaussian splatting approach [14], as in that case they also try to optimize the position of 3D gaussians (in this paper we optimize the position of particles) – but using multiple images while doing so instead of one as in this case. Furthermore, their error function gradient computation requires that the renderer is automatically differentiable, while my method has no such requirement.

6.6 Limitations

One of the major downsides of my fluid state reconstruction using a single reference image method is that the resulting scene parameters only make a sensible image when viewed from a perspective similar to the one used during reconstruction. This is not necessarily a problem by itself, but – at least in case of the geometry based reconstruction approach – the reconstructed scene might have a particle arrangement from which continuing the simulation would result in unexpected behavior. An example is when a lot of particles are clumped up together or there is only a very thin layer of particles instead of the expected particle volume.

The performance is far from ideal, as it can sometimes take a significant amount of time to achieve a satisfactory fluid state. This statement is even more true in the case of optimizing the state of the fluid visualized using the transparent method, as in that case the per-pixel parameter lists take a lot of time to compute. Finally, the performance of the

physics based reconstruction method is currently not suitable for any practical application – even if the reconstruction method would work as expected.

Another problem worth mentioning is the case of disconnected fluid masses on the reference image: if there is significant distance in between them and only one of them has particles in it in the initial state, than it might occur that the particles never get to the place of the other fluid mass because it would result in a large increase of the error for a number of optimization iterations.

Chapter 7

Conclusion

Manipulating the behavior of fluids is a challenging task that usually requires a lot of parameter fine tuning and manual arrangement of the fluid volumes. In this paper I have demonstrated a novel approach that partly solves this problem: using my method it is possible to edit the fluid’s state just by defining how the fluid should look like from a certain point of view. Such a reference image can then be easily created by a skilled artist. My method achieves visually appealing results while not being overly complex, it also does not require an automatically differentiable renderer.

Furthermore, I have presented how such a fluid simulation framework is constructed, including the algorithmic background of the used methods, two different fluid visualization techniques and some background on hybrid fluid simulation methods.

7.1 Future work

Although the method I have developed can already be useful in certain scenarios, there is a lot of research left to be done in this area, as this paper has only explored a small fraction of all possibilities. Probably the most notable subjects are the following:

1. Currently my method only supports editing the initial fluid state which is the starting point of the fluid simulation. But combining it with a method that is able to introduce keyframes into the fluid simulation itself would enable the creation of keyframed fluid animations by only using my screen space editing technique. Such a method has been presented by *Pan et al.* [18], and it could be integrated into my method with only minor modifications.
2. Further research could also go into my physics based fluid state reconstruction approach in order to draw a conclusion if it is a viable option. If so, then the computations done by the fluid simulation engine would need to be ported to the GPU in order to achieve usable performance.
3. As I have mentioned in the Limitations section 6.6, the resulting state after the reconstruction sequence only looks good from a point of view that is close to the one used during the reconstruction. However, this could be solved without much effort by defining two or more reference images from different perspectives – the only notable changes that this would require are minor ones to the error function’s and parameter list’s computations, otherwise the same method should work (even though the performance would likely suffer by a considerable amount).

Bibliography

- [1] Volker Aurich and Jörg Weule. Non-linear gaussian filters performing edge preserving diffusion. In Gerhard Sagerer, Stefan Posch, and Franz Kummert, editors, *Mustererkennung 1995*, pages 538–545, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-642-79980-8.
- [2] Haimasree Bhattacharya, Yue Gao, and Adam W. Bargteil. A level-set method for skinning animated particle data. *IEEE Transactions on Visualization and Computer Graphics*, 21(3):315–327, 2015. DOI: 10.1109/TVCG.2014.2362546.
- [3] J. Brackbill, Douglas Kothe, and H.M. Ruppel. Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48:25–38, 01 1988. DOI: 10.1016/0010-4655(88)90020-3.
- [4] Robert Bridson. *Fluid Simulation*. A. K. Peters, Ltd., USA, 2008. ISBN 1568813260.
- [5] O. Buneman. Dissipation of currents in ionized media. *Phys. Rev.*, 115:503–517, Aug 1959. DOI: 10.1103/PhysRev.115.503. URL <https://link.aps.org/doi/10.1103/PhysRev.115.503>.
- [6] John Dawson. One-Dimensional Plasma Model. *The Physics of Fluids*, 5(4):445–459, 04 1962. ISSN 0031-9171. DOI: 10.1063/1.1706638. URL <https://doi.org/10.1063/1.1706638>.
- [7] Thomas Deliot, Eric Heitz, and Laurent Belcour. Transforming a non-differentiable rasterizer into a differentiable one with stochastic gradient estimation. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 7(1):1–16, May 2024. ISSN 2577-6193. DOI: 10.1145/3651298. URL <http://dx.doi.org/10.1145/3651298>.
- [8] Yutao Feng, Xiang Feng, Yintong Shang, Ying Jiang, Chang Yu, Zeshun Zong, Tianjia Shao, Hongzhi Wu, Kun Zhou, Chenfanfu Jiang, et al. Gaussian splashing: Dynamic fluid synthesis with gaussian splatting. *arXiv preprint arXiv:2401.15318*, 2024.
- [9] Mark Richardson Ian McEwan, David Sheets and Stefan Gustavson. Efficient computational noise in glsl. *Journal of Graphics Tools*, 16(2):85–94, 2012. DOI: 10.1080/2151237X.2012.649621. URL <https://doi.org/10.1080/2151237X.2012.649621>.
- [10] Filomen Incahuanaco and Afonso Paiva. Surface reconstruction method for particle-based fluids using discrete indicator functions. *Computers & Graphics*, 114:26–35, 2023. ISSN 0097-8493. DOI: <https://doi.org/10.1016/j.cag.2023.05.015>. URL <https://www.sciencedirect.com/science/article/pii/S0097849323000705>.

- [11] Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. The affine particle-in-cell method. *ACM Trans. Graph.*, 34(4), July 2015. ISSN 0730-0301. DOI: 10.1145/2766996. URL <https://doi.org/10.1145/2766996>.
- [12] Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342896. DOI: 10.1145/2897826.2927348. URL <https://doi.org/10.1145/2897826.2927348>.
- [13] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey, 2020. URL <https://arxiv.org/abs/2006.12057>.
- [14] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023. URL <https://arxiv.org/abs/2308.04079>.
- [15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL <https://arxiv.org/abs/1412.6980>.
- [16] Xiangyu Kong. *Differentiable fluid simulation and rasterization*. PhD thesis, McGill University, 2023.
- [17] Xiaoying Nie, Yong Hu, Zhiyuan Su, and Xukun Shen. Fluid reconstruction and editing from a monocular video based on the sph model with external force guidance. *Computer Graphics Forum*, 40(6):62–76, 2021.
- [18] Zherong Pan, Jin Huang, Yiyang Tong, Changxi Zheng, and Hujun Bao. Interactive localized liquid motion editing. *ACM Trans. Graph.*, 32(6), November 2013. ISSN 0730-0301. DOI: 10.1145/2508363.2508429. URL <https://doi.org/10.1145/2508363.2508429>.
- [19] David Tskhakaya, Konstantin Matyash, Ralf Schneider, and Francesco Taccogna. The particle-in-cell method. *Contributions to Plasma Physics*, 47:563 – 594, 12 2007. DOI: 10.1002/ctpp.200710072.
- [20] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, page 91–98, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584294. DOI: 10.1145/1507149.1507164. URL <https://doi.org/10.1145/1507149.1507164>.

Appendix A

Every image column is a separate fluid state reconstruction session. In each column the topmost image is the reference image (in case of the second column it is drawn by hand), the second image is the initial fluid state, the third image is the reconstructed fluid state, and the rest of the images show the simulation from the reconstructed fluid state. Transparency and fluid surface noise is only enabled after the state reconstruction.

