FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Layered Particle-Based Fluid Model for Real-Time Rendering of Water

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Florian Bagar
Matrikelnummer 0500041

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Daniel Scherzer

Wien, 28.09.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Florian Bagar
Schönbrunnerstraße 29/A38
1050 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort, Datum:
Unterschrift:

Abstract

We present a physically based real-time water simulation and rendering method that brings volumetric foam to the real-time domain, significantly increasing the realism of dynamic fluids. We do this by combining a particle-based fluid model that is capable of accounting for the formation of foam with a layered rendering approach that is able to account for the volumetric properties of water and foam. Foam formation is simulated through Weber number thresholding. For rendering, we approximate the resulting water and foam volumes by storing their respective boundary surfaces in depth maps. This allows us to calculate the attenuation of light rays that pass through these volumes very efficiently. We also introduce an adaptive curvature flow filter that produces consistent fluid surfaces from particles independent of the viewing distance.

Kurzfassung

Wir präsentieren ein physikalisch basiertes Echtzeit-Wassersimulations- und Renderingverfahren, welches volumetrischen Schaum in den Echtzeitbereich einführt und den Realismus von dynamischen Flüssigkeiten maßgeblich verbessert. Dazu kombinieren wir ein partikelbasiertes Flüssigkeitsmodell, welches das Entstehen von Schaum berücksichtigt. Der Ansatz dieses Modells ist auf drei Schichten aufgebaut und bezieht die volumetrischen Eigenschaften von Wasser und Schaum ein. Die Bildung von Schaum basiert auf der sogenannten „Weber-Nummer“, wobei die Schaumbildung ab einem gewissen Schwellwert einsetzt. Die Wasser- und Schaumoberfläche wird durch Tiefenbilder repräsentiert auf deren Grundlage die jeweilige Dicke der Schichten berechnet wird, was wiederum eine effiziente Berechnung der optischen Eigenschaften des Wassers zulässt. Des Weiteren führen wir das sogenannte *Adaptive Curvature Flow Filtering* ein. Es ermöglicht uns eine Wasseroberfläche zu generieren, die, unabhängig vom Betrachtungsabstand, immer gleich viel Detail aufweist.

Contents

Chapter 1

Introduction

Over the last decade, simulation and rendering of complex natural phenomena such as fire, smoke, cloud or fluid has been an active and one of the most important research areas in computer graphics. Among these phenomena, water may be the most fascinating and challenging problem. Fluids such as water are an essential substance in our daily life and have attracted the attention of many researchers. Although the visual quality has improved, the lack of realism still offers a lot of room for improvement.

1.1 Scope of the work

Under the assumption that a fluid is at rest, it can be represented as a flat surfaces. For instance, this representation has been used to realistically render ocean scenes in computer games. This is a reasonable assumption in the field of games, but not sufficient for computer-generated films or motion pictures (and even in modern computer games one would prefer a more advanced method). Because as the realism of a scene increases, also the fluid has to be simulated and rendered in a more realistic way. The problem is that a fluid can move in a very complex way and even topologically changes can occur if the fluid is separated because of turbulent motion. Furthermore, the visual appearance of a fluid like water is based on complex optical effects, caused by reflection and refraction. As well as caused by caustics, which are complex patterns of light that can be seen on surfaces in presence of water (for instance, those formed on the floor of a swimming pool).

Real water also has additional features such as foam, bubbles and spray, appearing through advection, created by interaction with wind or formed by a water jet mixing gas with liquid. In general as proposed by Joseph [18], foam is a two-phase medium of gas and liquid with a particular structure consiting of gas pockets trapped in a network of thin liquid films and plateau borders. Taking these features, including the optical effects, in consideration extremely enhances the realism of scenes including water. But as one can

imagine, due to the complexity of fluid physics the computational cost is very high. This is especially true for offline rendering, although it achieves realistically and detailed results.

For offline rendering the focus is very high-quality and thus it is not interactively. For instance, objects or the view point cannot be moved by the viewer within the scene because the rendering time for a single image (referred to as frame) is too long. Also interaction with a fluid included in the scene is not possible because of this circumstance. Contrary to offline rendering, performance has the highest priority concerning real-time rendering. Thus a real-time application is able to synthesize the frames fast enough to keep the viewer immersed in the scene and also interaction with fluids is possible. For example, the viewer can throw objects into the fluid which float on the fluid sink. An application is classified as real-time if it renderes at least 15 *frames per second* (fps) [1], which permits the viewer to distinguish individual frames. In this context the increasing performance of today's GPUs is an important factor because it enables real-time methods to achieve improved visual quality, simulate complex physics on the GPU, and even adapt approaches from offline rendering methods. For instance, the fluid simulation used in this thesis is carried out on the GPU resulting in a high performance gain compared to a CPU fluid simulation.

Simulation of fluids like water can be classified into Eulerian and Lagrangian approaches. The former build on a fixed grid in space, and are an obvious choice for GPU-based calculation, since calculations on the cells of a grid are easily parallelizable. However, these approaches are not intuitive for flows because they limit the simulation to the space where the grid is defined. Lagrangian-approaches, like *Smoothed Particle Hydrodynamics* (SPH), simulate a fluid by moving discrete volume elements, and are therefore not restricted concerning the simulation space. Furthermore, particle-based fluid simulations are also suited for simulating topologically changes of water surfaces.

The remaining part of this chapter provides a motivation, introduces a real-time fluid simulation and rendering system developed for this thesis, and shows the main contributions and the structure of this thesis.

1.2 Motivation

Dynamic fluids are a desirable element of many real-time applications. So far, the mathematical complexity of realistically simulating and rendering the behavior and interaction of fluids with the environment has hindered their

widespread use. One promising approach would be to render the results of SPH simulations using splatting, but the locally high curvature of spherical splatting primitives results in an unrealistic jelly-like appearance.

Only recently, van der Laan et al. [37] proposed curvature-based screen space filtering for rendering the result of SPH simulations. The approach alleviates sudden changes in curvature between the particles and creates a continuous and smooth surface. While this method is a significant step towards realistic fluid rendering in real time, there is room for improvement. First, the screen-space curvature flow formulation is highly dependent on viewer distance. While fluids farther away from the viewer are overly smoothed, fluids near the viewer almost completely retain the undesirable spherical particle structure. Second, there exists as yet no realistic real-time method to create foam, which is an important visual element in most situations where real-time fluids are used (see Figure 1.1).

The main algorithm discussed in this thesis was also published in a paper [3] at the "21st Eurographics Symposium on Rendering".

Fig. 1.1: A scene rendered with simple noise-based foam [37] *(left)* and with our new method *(right)*;

1.3 Main Contributions

This thesis presents a real-time fluid simulation and rendering system that overcomes these drawbacks:

- We introduce an *adaptive curvature flow* filtering algorithm for *smoothed particle hydrodynamics* rendering which accounts for perspective due to is independence from the viewpoint and avoids over- or under-smoothing as present in previous methods. The construction of the water surface and the filtering is not based on polygonization, and thus does not suffer from the associated tessellation artifacts.

- We introduce a fast *physically based foam rendering* approach using Weber number thresholding and a volumetric layer-based rendering system (see Figure 3.3). Foam can appear as top-most layer or between two water layers, as it is the case when a turbulent stream, like a waterfall, immerges into resting water with high impact. We calculate the thickness of each layer and perform volumetric back to front compositing along the viewing rays. Thus, objects of the background scene become less visible depending on the amount of water and foam that is in front of them.

- Our method is almost as fast as previous approaches and has comparable performance with the benefit of improved image quality.

- Our method is highly configurable from an artistic point of view, and thus can produce a multiplicity of visual appearances which match the background scene and create the desired atmosphere.

- In addition, the algorithm is simple to implement and integrate into existing rendering engines. Furthermore, all rendering steps are calculated using shaders and intermediate render targets on the GPU.

1.4 Thesis Structure

The thesis is structured into different chapters as follows:

- Chapter 2 gives an overview about previous work. First, the chapter describes the *smoothed particle hydrodynamics* formalism. Then, recent *offline methods* are described and finally an overview of *real-time approaches* is given.

- Chapter 3 explains our method which is separated into our *adaptive curvature flow* filtering, and *real-time foam* rendering.

- Chapter 4 deals with the implementation of our method. The chapter describes the used *application programming interfaces*, gives an top-level overview of the involved classes, and presents a detailed description of the rendering algorithm.
- Chapter 5 shows results that are achieved using our simulation and rendering method and provides a detailed performance comparison. The Chapter also discusses limitations of our method.
- Chapter 6 summarizes the thesis' contents by drawing a conclusion and discussing possible future enhancements.

Chapter 2

Previous Work

Several offline and real-time methods exist to simulate and render dynamic fluid flows. This chapter is a collection of some of these approaches which are related to our work. As stated in chapter 1 the simulation of fluids can categorized into two types, the Eulerian- and the Lagrangian- approach.

2.1 Fluid Simulation

In general, the dynamic behavior of incompressible fluids is described by the so-called Navier Stokes equations [14]. Methods based on the Eulerian approach subdivide the simulation space into a regular grid. This grid is used to simulate the fluid dynamics, and the Navier Stokes Equations are solved by discretizing the equations, using this grid. Each cell of this grid contains certain quantities such as velocity, pressure or density needed for the simulation and in every simulation step these values are updated depending on the flow. The Navier Stokes equations are also suitable for simulating phenomena such as clouds, foam or smoke. One problem that arises by using a fixed grid, is that small-scale phenomena like foam, bubbles or spray can hardly be tracked because too coarse grid resolutions are used. One could try to refine the grid that is being used, but the drawbacks are increased performance and memory requirements and for this reason only limited applicable for real-time methods. Thus, many methods based on the Eulerian approach decouple small-scale effects from the main fluid simulation and offers ~~them~~ to simulate effects such as foam or bubbles on a much smaller scale than the underlying grid. This is especially the case for offline methods. Another drawback is that the fluid is restricted to stay with the grid and thus cannot flow everywhere within a scene. Common methods to solve the Navier Stokes equations for the Eulerian case are finite element techniques [34].

In the Lagrangian formulation of a fluid, particles are used to completely define the fluid and to solve the Navier Stokes equations. Compared to the Eulerian approach where the fluid is tracked at fixed position in space, in the

case of the Lagrangian approach the particles move with the fluid and thus track the fluid. Particle-based fluid simulations such as SPH which is used by the algorithm presented in this thesis (see section 2.3) have several advantages compared to grid-based methods. For instance, the fluid is not restricted to a fixed volume in space and even topological changes of the fluid surface are handled because of their particle-based nature. Furthermore, collision detection with the surrounding environment and also with dynamic objects can easily ~~be~~ determined because each particle carries position and velocity information, which are sufficient to calculate any collision. However, the major drawback of particle-based methods is the high particle amount that is required to simulate realistically behaving fluids. Moreover, the particles should be small in size to ensure that small-scale effects can be treated in an appropriate way ~~and~~ thus the particle count has to be increased to achieve realistic~~ally~~ results. Furthermore, it is ~~a~~ difficult to extract a surface for rendering.

2.2 Fluid Rendering

To achieve realistically looking renderings including optical effects such as reflection and refraction the fluid surface has to be reconstructed. A common method for surface extraction is the *marching cubes* [19] algorithm which is applicable to both grid-based and particle-based methods. The algorithm extracts a polygonal mesh that approximates the iso-surface from a three-dimensional grid. The drawback with respect to particle-based methods is that the simulation space has to be subdivided into a grid. Thus, the computational cost of the reconstruction becomes quite high. Another common approach to reconstruct the fluid surface are *level set methods* [31] which reconstruct the surface on a grid whose resolution and computation are completely independent of the fluid simulation. This reconstruction method is suitable for particle-based fluids, but has inherent problems with creating sharp boundaries when the fluid is in contact with solid objects.

Today's GPUs support point primitives ~~used~~ for rendering instead of polygons, which enables an application to perform hardware-accelerated surface splatting [4]. In contrast to polygonal meshes, point primitives are more flexible and do not have to care about topological changes. Hence, point primitives are ~~acutely~~ applicable for real-time rendering when the underlying simulation is particle-based. Due to the spherical nature of splatting primitives, they suffer from jelly-like artefacts on the surface and produce unrealistic appearance as mentioned in Section 1.2. To overcome this drawback, the surface has to be filtered after the splatting has been carried out

(see Section 2.5.3). The major advantage of using point primitives for rendering is that the method is not based on polygonization, and thus does not show grid discretization artifacts in coherent frames as present in grid-based methods.

The actual rendering of fluids can be separated into methods for offline rendering and ~~such~~ suited for real-time rendering. The former basically render the results from grid- or particle-based simulations via ray tracing methods such as *Monte Carlo path tracing* [31] or *Photon Mapping* [17] because ~~they~~ are well suited for volumetric phenomena such as water, foam and bubbles. Due to this characteristic they are able to produce realistic images including physically correct reflections, refractions and caustics. Even effects like light scattering can be achieved. Real-time methods such as the one presented in this thesis are based on scanline rendering and try to perform most of their calculations on the GPU. Furthermore, they have a limited time buffer wherein the simulation- and rendering- calculations have to fit into. Because of this limitation most of the optical effects are approximated for real-time rendering. For instance, a common method to achieve reflections on a fluid surface like water is to use static environment mapping. Especially advanced fluid effects such as foam and spray are hardly achievable in a physically correct manner for real-time rendering at the time of writing. Even so, non-physical but realistic approximations can be used to render these effects in real-time as shown in this thesis.

Particle-based fluid simulations are an active research topic, but the real-time rendering of the simulation results is an ongoing challenge. In the remaining part of this chapter we will briefly describe the Smoothed Particle Hydrodynamics formalism, then we will give an overview of related offline particle-based fluid rendering methods and afterwards we will describe related real-time particle-base methods. Finally, we will give a brief overview of relevant APIs used by our implementation.

2.3 Smoothed Particle Hydrodynamics

SPH is a formalism for simulating highly deformable bodies like fluids with a particle system. It was introduced by Monaghan [23] and is used by physicists for cosmological fluid simulations. Desbrun et al. [9] has later extended the SPH approach to be used for computer graphics. The particles of the SPH simulation, which can be viewed either as matter or sample points scattered in a soft substance, represent small volume elements of inelastic material that move over time. SPH does not need a grid to calculate the spatial derivatives,

instead they are found by analytical differentiation of interpolation formulas. SPH is an interpolation method which allows any function to be expressed in terms of its values at a set of disordered particles. The interpolation is based on the theory of integral interpolants using kernels that approximate the Dirac delta function [10]. Using the identity

$$A(\vec{r}) = \int A(\vec{r}\,')\delta(\vec{r} - \vec{r}\,')d\vec{r}\,', \tag{2.1}$$

in which δ is the Dirac delta function and $\vec{r}$ is the position one is interested in, the integral interpolant A_I of any function $A(r)$ is defined by

$$A_I(\vec{r}) \approx \int A(\vec{r}\,')W(\vec{r} - \vec{r}\,', h)d\vec{r}\,', \tag{2.2}$$

where the integration is over the entire space, and the Dirac delta function δ is approximated by an interpolating kernel W to be able to construct a differentiable interpolant. h is a parameter that defines the size of the kernel support and is refered to as the smoothing length. The interpolating kernel W has the following properties

$$\int W(\vec{r} - \vec{r}\,', h)d\vec{r}\,' = 1, \tag{2.3}$$

and

$$\lim_{h\to 0} W(\vec{r} - \vec{r}\,', h) = \delta(\vec{r} - \vec{r}\,'), \tag{2.4}$$

where equation 2.3 states that W is normalised and the limit is to be interpreted as the limit of the corresponding integral interpolants as proposed by [23]. For the discrete case the integral is approximated by a summation, which gives the summation interpolant

$$A_S(\vec{r}) = \sum_{b=1}^{N} m_b \frac{A_b}{\rho_b} W(\vec{r} - \vec{r}_b, h), \tag{2.5}$$

where N is the particle count and m_b is the mass, $\vec{r}_b$ is the position and ρ_b is the density of a single particle b. The value of any quantity A at position $\vec{r}_b$ is denoted by A_b. Kernel functions commonly used are based on the Gaussian function,

$$W(\vec{r} - \vec{r}\,', h) = \frac{1}{h\sqrt{\pi}} e^{-\frac{(\vec{r}-\vec{r}\,')^2}{h^2}}, \tag{2.6}$$

and on cubic splines, which are computationally more efficient.

To calculate quantities such as acceleration or viscosity, one needs a differentiable interpolant. As mentioned by Monaghan [23], the essential point

is that a differentiable interpolant of a function can be constructed from its values at the particles (interpolation points) by using a kernel which is differentiable. This means that the derivatives can be exactly calculated if the kernel W is differentiable and there is no need to use finite differences or a grid to approximate the derivatives. For instance, if one wants ∇A,

$$\nabla A(\vec{r}) = \sum_{b=1}^{N} m_b \frac{A_b}{\rho_b} \nabla W(\vec{r} - \vec{r}_b, h) \tag{2.7}$$

can be used (for further details and examples see [23] and [9]).

In summary, a smoothing length h is used, over which the properties of the particles are smoothed by a kernel function W. By summing the relevant properties of all the particles, which lie within the range of this kernel, the physical quantities (such as velocity, density, or viscosity) can be approximated. Furthermore, a SPH simulation can include physical field values like pressure or temperature. For Computer Graphics, Desbrun et al. [9], modified the expression of pressure to animate materials with constant density at rest and they use a slightly different kernel, which mimics the Gaussian function and has finite radius of influence. In contrast to a Gaussian kernel, their kernel prevents clustering between particles which ensures a constant density.

2.4 Offline Methods

Offline methods include effects like foam, bubbles and spray. They try to simulate all these effects physically correctly and treat them in different ways. For instance, Mihalef et al. [22] propose different models for droplets and bubbles in their framework. In contrast, our method and other real-time approaches do not differentiate between these effects. Another important aspect of offline methods is the number of particles they can handle. For example, Losasso et al. [20] use up to 645 million particles to simulate scenes like the one illustrated in Figure 2.1. In contrast, real time simulation methods only treat several thousand particles (see Section 2.5). In this thesis we adapt some of their elements for our real-time method.

2.4.1 Two-Way Coupled SPH and Particle Level Set Fluid Simulation

Losasso et al. [20] propose a two-way coupled simulation framework. They mix the Eulerian and Lagrangian approaches to generate realistic fluids including spray and foam. Where the Eulerian particle level set method is used

to efficiently model dense liquid volumes. The particle level set method is based on the level set method which is used to model and animate implicit functions that dynamically change over time. With an implicit formulation the method is able to simulate topological changes of the fluid's shape (such as splitting up into two fluid volumes).

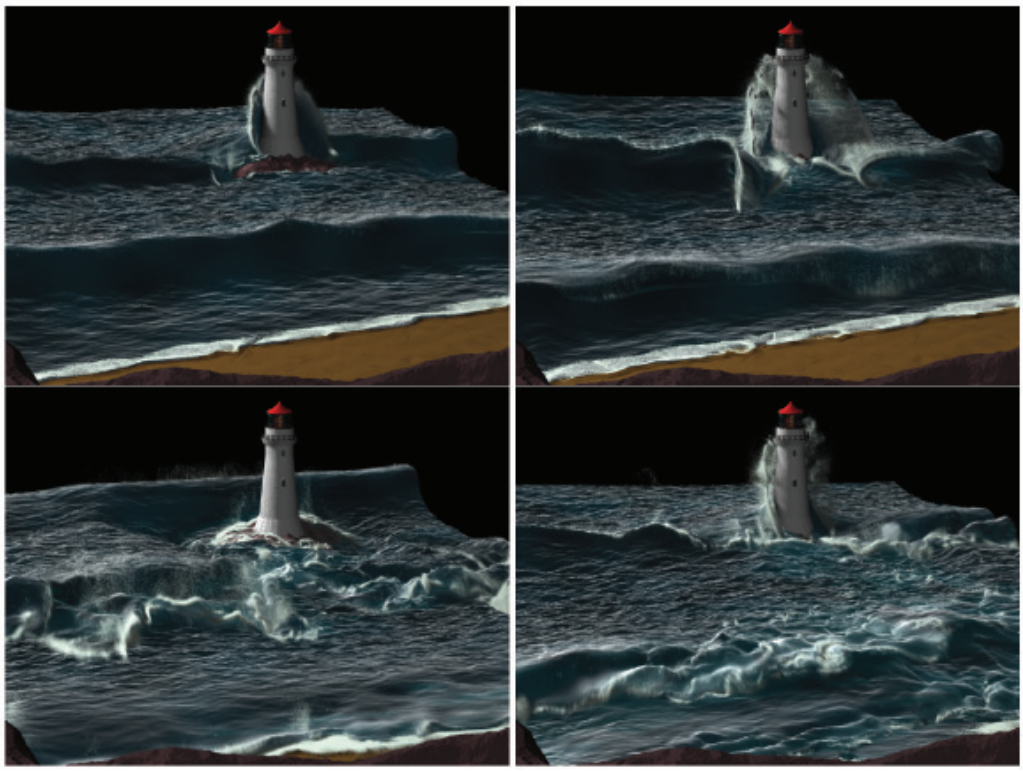

Fig. 2.1: Ocean scene simulated with two-way coupling between the SPH and the particle level set method [20].

The main drawback of the level set method is that because of the coarse grids (e.g. $100 \times 100 \times 100$) that are used the method is prone to volume loss because of numerical errors. Thus the particle level set method extends the level set method by introducing particles which are used to correct any volume loss. Beside the particle level set method, a Lagrangian SPH method is used to simulate diffuse regions such as spray and foam. Furthermore they extend the SPH simulation to take the surrounding air into account, which is used to simulate diffuse phenomena such as mixtures of spray and air.

Losasso et al. [20] state that their method introduces unwanted noise, due to the fact that particles have wildly varying velocities and that their particle density computation is unreliable near the air/liquid interface, where

SPH particles do not have neighbors on all sides. They experimented with several strategies to reduce noise in these areas, but achieved only limited success. Figure 2.1 shows an ocean scene simulated with this approach, where a $560 \times 120 \times 320$ grid is used. The simulation time lies between 30 seconds and 3 minutes per frame and the rendering is done with Pixar's RenderMan with a deep-water texture applied to the surface.

2.4.2 Simulation of Two-Phase Flow with Sub-Scale Droplet and Bubble Effects

Mihalef et al. [22] adapt the *Two-Way Coupled SPH and Particle Level Set Fluid Simulation* method and replace the Lagrangian SPH approach with a simple particle system to include droplets and bubbles. In contrast to Losasso et al.'s [20] method, their method uses a much simpler framework without SPH involved, which avoids the "graininess" usually associated with SPH methods. The sub-scale droplets and sub-scale bubbles generation is separated into two models. The former are evolved in a Newtonian manner, using a density-extension drag force field. Bubbles are evolved using a model based on Stokes' Law. Because their method makes use of subscale dynamic models and Lagrangian dynamics, it achieves an enhanced accuracy. They introduce the so-called Weber number to the computer graphics domain, which is used to control the generation of droplets and bubbles. Furthermore, they propose a switching parameter which is called gamma parameter, which is a combination of the absolute value of the mean curvature and an abstraction of the Weber number which is proportional to the Weber number. They abstract the Weber number because the density and surface tension coefficients are fixed for their simulation. The switching from the large-scale level set method simulator to the small-scale particle-based simulator is achieved by thresholding the gamma parameter.

In this thesis we adapt the Weber number thresholding approach for creating foam in real time, but in contrast we do not use an abstracted version of the Weber number. Instead we use the whole Weber number formula as described in Section 3.2.1. Figure 2.2 shows different time steps of a simulation where a diver jumps into a pool. For this simulation a $72 \times 72 \times 144$ grid is used and the computation time is 7 minutes for a single frame.

2.4.3 Realistic Animation of Fluid with Splash and Foam

Takahashi et al. [35] use the *Cubic Interpolated Propagation* (CIP) method [40] as the base fluid solver and a particle system is used to model splashes

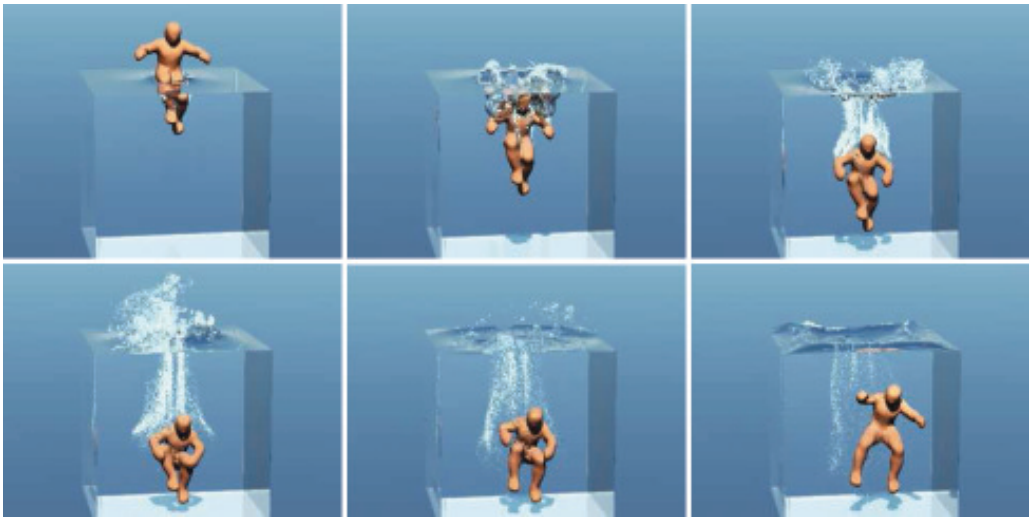

Fig. 2.2: Diver performing a jump into a pool [22].

and foam. By using a uniform grid cell and the CIP method, the full Navier-Stokes equation is solved. Furthermore the particle system obtains information on the simulation space from the base fluid solver, which means that the velocity of the water and the velocity of the air is used by the particle system.

The generation and transition of foam is controlled using state change rules, which work in a similar way as our separation of the particles into water- and foam-particles as described in Section 3.2. Particles are created when the curvature of the water surface of the base fluid simulation exceeds a given threshold th. Their state is either *SPLASH* if they are above the water surface or *FOAM* if they are on the water surface. Their initial velocity is obtained from the velocity field of the base fluid simulation and their position is updated according to the velocity field. *SPLASH* particles are influenced by gravity and *FOAM* particles are not. Instead, *FOAM* particles are restricted to floating on the surface of the fluid.

Although the approach is an offline method, Takahashi et al. [35] propose a polygonal-based rendering approach because of the acceleration gained by modern graphics hardware. They propose to first render the geometry of the environment, then the the polygonal-based fluid surface, including caustics, refraction and reflection, and finally the particles representing the foam. The particles are sorted front-to-back with respect to the light source and rendered using a technique originally developed for rendering of clouds [12]. This enables their method to render shadows of foam. Results of this method can be seen in Figure 2.3. In this scene a $75 \times 60 \times 50$ grid is used and a maximum

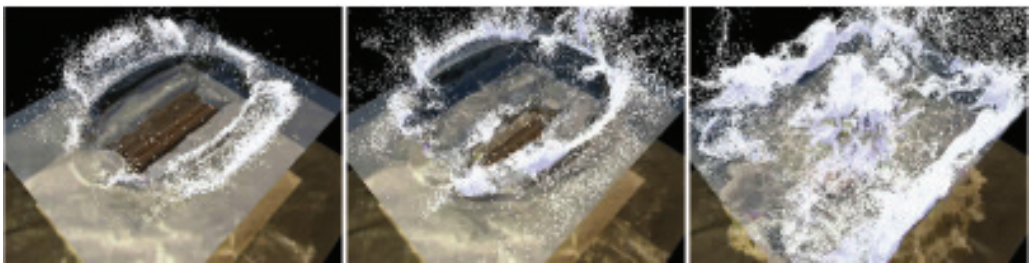

Fig. 2.3: A piece of lumber falling into water [35].

of around $270k$ particles are generated in one frame. The simulation of one frame takes 170 seconds and the rendering additionally takes 100 seconds.

2.4.4 Bubbling and Frothing Liquids

Cleary et al. [8] extend SPH by considering the dissolved gas within the fluid. This dissolving of gas is common for carbonated liquids like beer and champagne. The bubbles are represented as discrete spherical bodies and are coupled to the SPH simulation. Each particle of the SPH simulation contains a certain amount of dissolved gas, which is transferred from the SPH fluid model to the discrete bubble model. During the simulation bubbles rise in the fluid and grow by gathering more dissolved gas from the SPH simulation. Furthermore, collision of bubbles with each other and with other solids and boundaries is taken into account.

Fig. 2.4: Ale and Stout pouring into a beer mug [8].

Similar to our work they use a layered representation where the different parts of the fluid volume are separately rendered and composed into the final image. The method divides the rendering into a liquid pass, which

renders only the results from the underlying SPH simulation, and a bubble pass, which renders the bubbles that have been generated during the coupled discrete bubble-SPH based simulation. The final image is achieved by composing the background scene, the liquid and the bubbles. The rendering is performed via *Maya* using the *Mentalray renderer* and the compositing is achieved using *After Effects*. Note that the compositing of the different parts is done by hand which allowed more control over specific elements of the simulation while compositing. Figure 2.4 shows image sequences of different sorts of beer pouring into a mug.

2.5 Real-Time Methods

Current *real-time approaches* are usually limited in the number of particles they can handle, and do not include realistic foam. The amount of particles that can be simulated in real time at the time of writing is around $64k$. The first method that is described in this section performs simulation and rendering. In contrast, the methods proposed by Müller et al.'s [24] and van der Laan et al. [37] assume that a SPH particle simulation has already been carried out.

2.5.1 Real-Time Simulations of Bubbles and Foam within a Shallow Water Framework

Thürey et al. [36] present a shallow water framework which is coupled to a particle-based bubble simulation. Furthermore, the bubble simulation interacts with a SPH simulation to handle foam floating on the fluid surface. The shallow water simulation is a reduction of the Navier-Stokes equations to a simplified two-dimensional height field representation and represents the main volume of the liquid. Bubbles beneath the fluid surface are simulated as particles and are coupled to the shallow water simulation. This enables the method to simulate breaking waves induced by bubbles reaching the fluid surface. Foam is simulated with an SPH simulation, whereas each SPH particle represents exactly one foam bubble. Foam bubbles are generated from bubbles reaching the fluid surface from beneath and the emergence is controlled by a user-defined probability. Simulating individual foam particles is an expensive task, in contrast our method uses the SPH particles to simulate a volume of foam and the emergence is achieved in a physical manner.

Figure 2.5 shows an example scene simulated and rendered with the shallow water framework, whereas a 50×80 grid is used for the shallow water simulation. The maximum count used for bubbles and particles is around

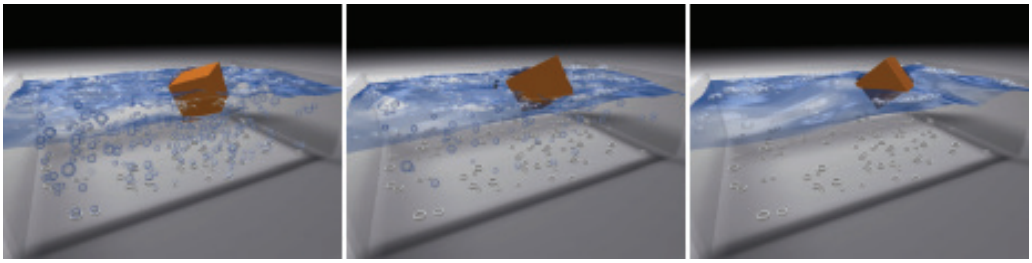

Fig. 2.5: Bubbles and foam within a Shallow Water Framework including interaction of an obstacle with the foam and the bubbles [36].

960 and the lowest frame rate is 34.3 FPS.

2.5.2 Screen Space Meshes

One way to render the water surface from the results of a SPH simulation are Müller et al.'s [24] *screen space meshes*, created using a marching squares technique on a depth map. Compared to the full 3D *marching cubes* [19] algorithm the method has several advantages, like view-dependent level of detail and the possibility to apply filtering in screen space. Another important aspect is that the method only generates a surface where it is visible.

The algorithm starts by computing a depth map in screen space. In addition, depth values are computed where silhouettes cut the depth map grid. These inner and outer silhouettes are indicated by large z-differences of adjacent depth values and are used during the screen space mesh construction. Figure 2.6 illustrates the concept of the depth map including the generation of silhouette nodes.

After the depth map has been computed, the next step is to generate the vertices and triangles of the screen space mesh. Each valid depth value in the depth map and each outer silhouette edge generates exactly one vertex. For an inner silhouette edge two vertices are generated, as illustrated in the left and middle part of Figure 2.7. Triangles are generated by looking at each grid cell, whereas each edge of the cell is either a silhouette edge or a regular edge. Depending on the constellation, this leads to one of 16 cases as illustrated in the right part of Figure 2.7. From the triangles shown, only those are generated for which all three vertices exist, which results in a triangle mesh with correct connectivity and vertices in screen space.

Because the base geometry of the method consists of spheres, the results after the triangulation look bumpy. Müller et al. propose a separable binomial filter that smoothens the surface. Special care has to be taken near silhouettes, because the applied filtering should not change the shape of the

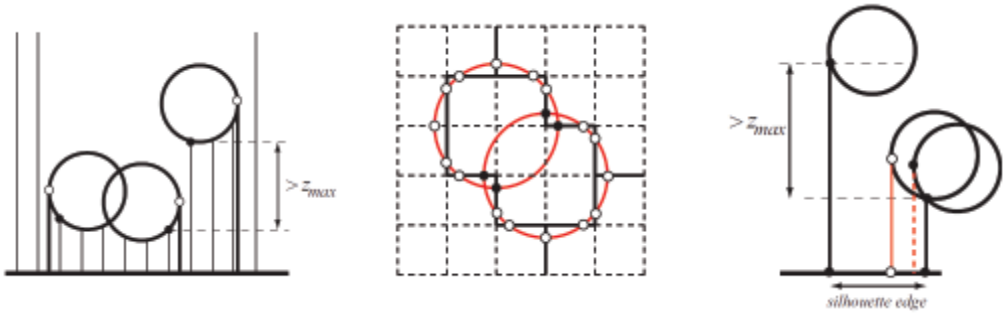

Fig. 2.6: Left: Side view of depth map. Between adjacent nodes at most one additional node (white dot) is stored to indicate the silhouette (the middle white dot represents an inner silhouette and the two other white dots outer silhouettes). Middle: Top view of the grid. Right: Side view. The cut (white point) furthest from the end with the smaller depth value is taken [24].

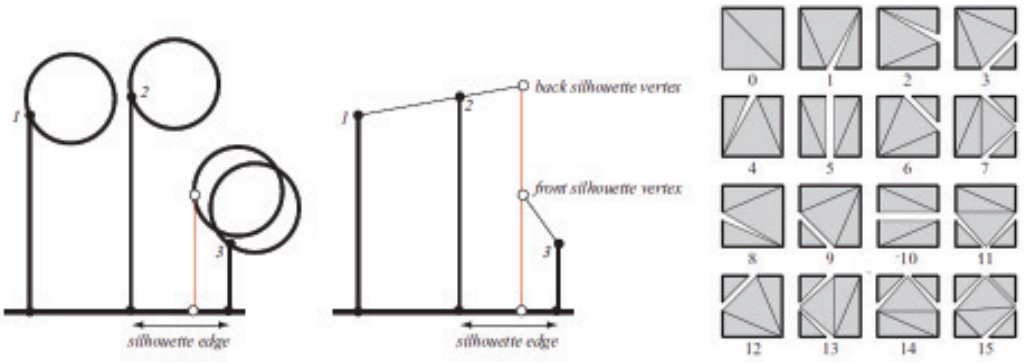

Fig. 2.7: Left: Side view of the grid. Middle: The vertices generated for this configuration, whereas vertices with different depth values are generated for the silhouette node (white point). Right: All the cases for the generation of a 2D triangle mesh from cut edges [24].

silhouettes, which is achieved by only considering depth values within a z_{max} range (see Figure 2.6).

For rendering, the resulting screen space mesh is transformed back to 3D world space and per vertex normals are computed. Finally the triangles and normals are sent to the standard graphics pipeline and are rendered including reflections, refraction, and specular highlights. Figure 2.8 shows a car wash scene consisting of $16k$ SPH particles running at 20 FPS.

Although the algorithm provides view-dependent level of detail and filtering in screen space, rendering foam with this approach is prohibitive, because of the large amount of geometry that needs to be generated.

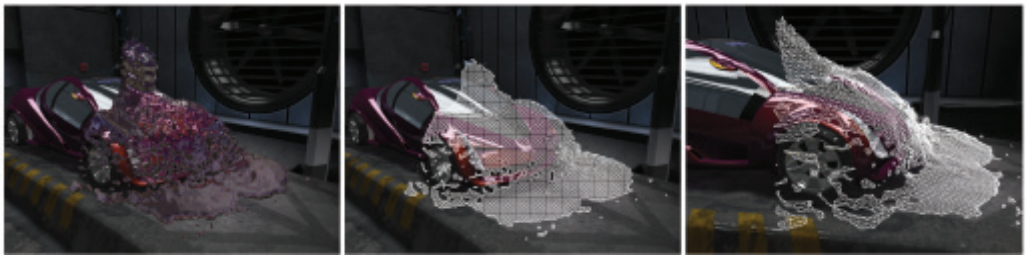

Fig. 2.8: Final rendering of the screen space mesh (including a rotated view of the mesh to show its dependence on the viewing direction) [24].

2.5.3 Screen Space Fluid Rendering with Curvature Flow

van der Laan et al. [37] present an approach for rendering particle-based fluids directly using splatting instead of performing a polygonisation and thus avoid the associated tessellation artifacts that come with grid-based approaches. They use screen space curvature flow filtering to conceal the sphere geometry of the particles and to prevent the fluid from looking jelly-like. All the processing, rendering and shading steps are done directly on the graphics hardware and the method achieves real-time performance.

First of all the front-most surface of the fluid from the viewpoint of the camera is determined. This representation is obtained by rendering all particles as spheres into a depth map as illustrated in Figure 2.9. This step is similar to the visibility splatting pass described by [4], but in the case of *screen space fluid rendering with curvature flow* the depth values of the point-sprites are replaced by the geometry of a sphere. The hardware depth test ensures that only the nearest pixels to the viewpoint are stored in the depth maps. Splatting normal vectors and material properties, as it is done by [4] in the attribute pass, is not practicable at this point, because the depth values of the depth map are manipulated in the following smoothing pass.

Rendering particles as point-sprites with sphere geometry results in a blobby appearance. To prevent the water from looking artificial, filtering of the surface is performed in screen-space. An obvious approach is to use a Gaussian blur, but this introduces artefacts like blurring over silhouette edges and plateaus of equal depth when using large kernels. A more suitable method is the so-called curvature flow introduced by [21], which is extended by van der Laan et al. [37] for their method. The smoothing is an iterative method where in each iteration the z-values in the water depth map are moved proportional to the mean curvature,

$$\frac{\partial z}{\partial t} = H, \tag{2.8}$$

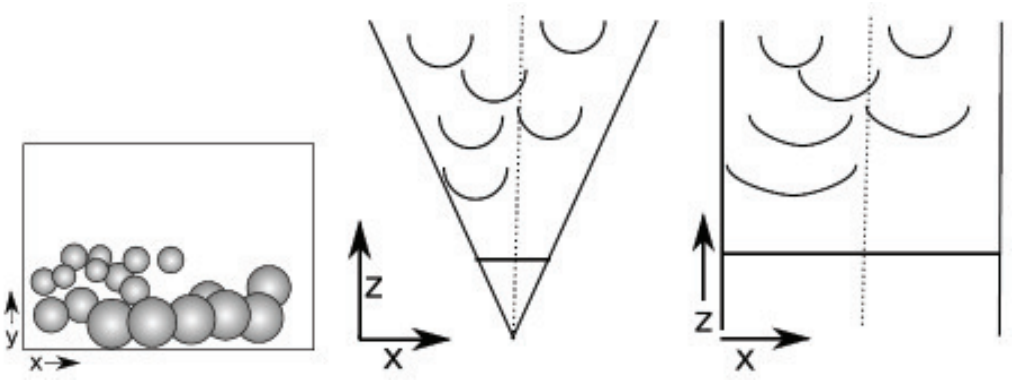

Fig. 2.9: Left: drawing particles as spheres; middle: front view in view-space; right: after perspective projection [37].

where t is a smoothing time step and H is the mean curvature. For a surface in 3D space the mean curvature is defined as follows:

$$2H = \nabla \cdot \hat{n} \tag{2.9}$$

where $\hat{n}$ is the unit normal of the surface. The normal is calculated by taking the cross product between the derivatives of the view space position P in x and y direction (for details see [37]), resulting in a representation of the unit normal:

$$\hat{n}(x,y) = \frac{n(x,y)}{|n(x,y)|} = \frac{(-C_y \frac{\partial z}{\partial x}, -C_x \frac{\partial z}{\partial y}, C_y z)^T}{\sqrt{D}} \tag{2.10}$$

where

$$D = C_y^2 \left(\frac{\partial z}{\partial x}\right)^2 + C_x^2 \left(\frac{\partial z}{\partial y}\right)^2 + C_x^2 C_y^2 z^2 \tag{2.11}$$

Finite differencing is used to calculate the spatial derivatives, and C_x and C_y are the viewpoint parameters in x and y direction respectively. They are computed from the field of view and the size of the viewport V_x and V_y as shown in Equation 2.12 and 2.13.

$$C_x = \frac{2}{\tan\left(\frac{FOV}{2}\right) * V_x} \tag{2.12}$$

$$C_y = \frac{2}{\tan\left(\frac{FOV}{2}\right) * V_y} \tag{2.13}$$

The unit normal $\hat{n}$ from Equation 2.10 is substituted into Equation 2.9, which enables the derivation of H, leading to,

$$2H = \frac{\partial \hat{n}_x}{\partial x} + \frac{\partial \hat{n}_y}{\partial y} = \frac{C_y E_x + C_x E_y}{D^{\frac{2}{3}}} \tag{2.14}$$

in which

$$E_x = \frac{1}{2}\frac{\partial z}{\partial x}\frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2}D \tag{2.15}$$

$$E_y = \frac{1}{2}\frac{\partial z}{\partial y}\frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2}D \tag{2.16}$$

In one iteration, an Euler integration of Equation 2.8, including the mean curvature H, is used to modify the z values of the water depth map, whereas the number of iterations is chosen depending on the smoothness that is desired. However, using a fixed iteration count results in different filtering strength depending on the view distance (see Section 3.1). Discontinuities of the water depth map are handled by simply forcing the spatial derivatives to zero, which prevents smoothing over silhouettes in screen space. Furthermore, all smoothing is computed at half instead of full resolution, which is a good tradeoff between quality and performance.

Water is a volumetric phenomenon, so the thickness of the water in front of the opaque scene has to be taken into account. The thickness is used to correctly compute visual attributes like color attenuation, transparency and refraction. To accomplish this, the particles are regarded as spheres of fluid with a fixed size in world space and are rendered similarly as the particles for computing the depth values in the visibility pass described above, but instead of writing the view-space depth value, the thickness of a particle at it's projected position is written,

$$T(x,y) = \sum_{i=0}^{n} d(\frac{x - x_i}{\sigma_i}, \frac{y - y_i}{\sigma_i}) \tag{2.17}$$

where d is the depth kernel function, x_i and y_i are the projected position components of the particle, x and y are the screen coordinates and σ_i is the projected size. The summation as illustrated in Equation 2.17 is achieved by using additive blending, and the particles are rendered with enabled depth test and disabled depth write to ensure correct visibility with respect to the scene geometry behind the fluid.

For real-time applications, the amount of particles that can be simulated with a SPH simulation is limited. Therefore the particles are relatively large in proportion to the complete volume of the fluid. One expects that water has fine micro structures on its surface which move along when the fluid flows. van der Laan et al. [37] additionally propose an approach based on Perlin noise [28] that generates noise that is advected by the fluid and is of a higher frequency and a smaller scale than the underlying SPH simulation. They assign one octave of noise to each projected particle to establish a certain pattern of noise moving along with each particle (see Figure 2.10).

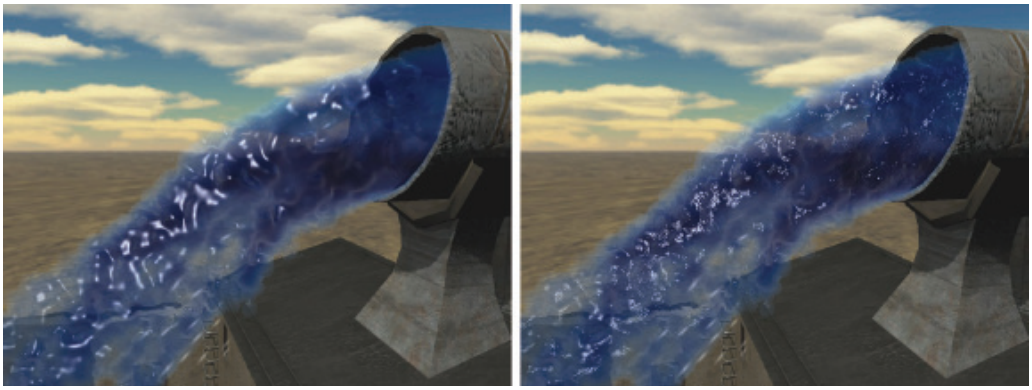

Fig. 2.10: Screen space curvature flow without (left) and with (right) surface noise [37].

Using a Gaussian noise kernel which is multiplied by an exponential fall-off and additive blending results in a Perlin noise texture which can be used for rendering. The exponential fall-off is based on the depth below the surface and ensures that particles contribute less as the submerge into the fluid.

Finally, all intermediate results, like the smoothed surface depth, the thickness and the surface noise, are composed into a final image. This is done in a simple screen-space rendering pass by rendering a full-screen quad. To shade the surface of the fluid, the view-space normals $\vec{n}$ are calculated using the finite differences of the surface depth as shown in Equation 2.10. van der Laan et al. [37] propose to choose the smallest absolute finite difference (when a discontinuity is detected, by comparing the difference in depth to a threshold) instead of simply using the finite differences in one direction, which would result in artifacts along the silhouettes. Furthermore, the Perlin noise texture is used to perturb the normals to add wave-like micro structures by adding the partial derivatives of the noise texture to the calculated normals. In addition to this, van der Laan et al. [37] propose a noise-based surface foam effect by adding a grayish color depending on the magnitude of the noise (see Figure 1.1). The computed thickness enables the method to attenuate the background color, in the sense that the thicker the fluid is, the less of the background scene is shown through,

$$a = lerp(c_{fluid}, S(x + \beta\vec{n_x}, y + \beta\vec{n_y}, e^{-T(x,y)}) \tag{2.18}$$

where c_{fluid} is the color of the fluid medium, $S(x, y)$ is the scene without the fluid rendered into a background texture and $\beta = T(x, y)\gamma$ is used to perturb the texture coordinates of the sampled background scene texture, whereas γ is a user parameter describing the refractive properties of the

fluid. The optical properties of the fluid are based on the Fresnel equation and a Phong specular highlight,

$$C_{out} = a(1 - F(\vec{n} \cdot \vec{v})) + bF(\vec{n} \cdot \vec{v}) + k_s(\vec{n} \cdot \vec{h})^{\alpha} \tag{2.19}$$

where a is the fluid color (containing the refracted background scene color) from Equation 2.18, b is the reflected scene color (sampled from an environment map), k_s and α are constants for the specular highlight and F is the Fresnel function controlling the composition of the reflection and the refraction component. $\vec{n}$ is the screen space surface normal, $\vec{h}$ is the half-angle vector between the camera and the light and $\vec{v}$ is the camera vector.

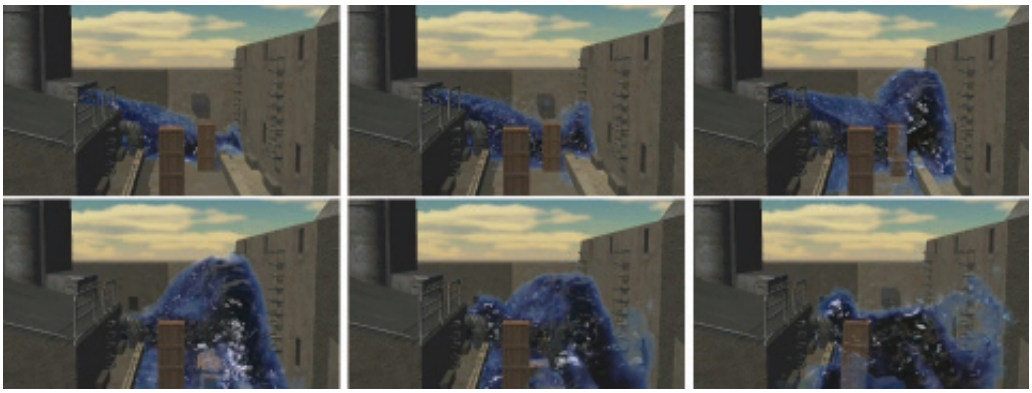

Fig. 2.11: Different time steps showing the visual results that can be achieved with the method presented by van der Laan et al. [37].

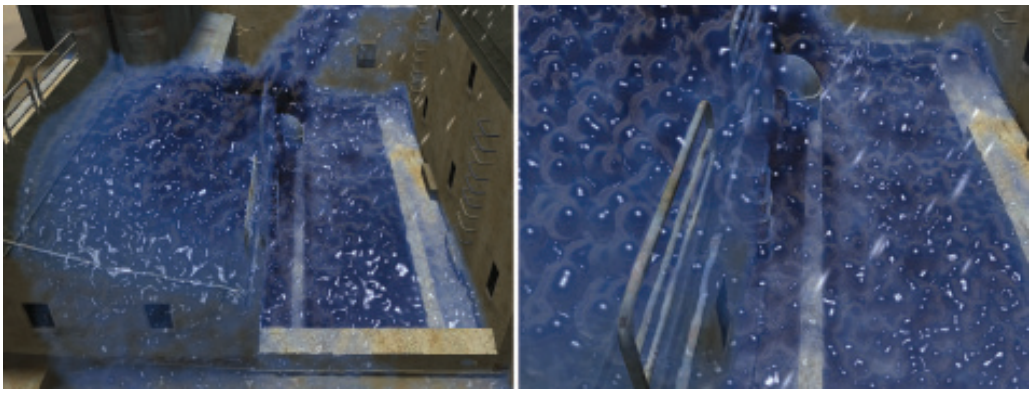

Fig. 2.12: Left: side view onto the waterfall scene showing the curvature flow filtered fluid surface; right: closeup view [37].

Figure 2.11 shows an image sequence of the screen space fluid rendering method flooding a corridor scene. The SPH simulation for this scene consists

of 60k particles and an iteration count of 40 is used. The computation time is around 19.6 ms without noise and foam (filtering is done at half resolution). Including noise and surface-based foam, the computation time is around 40 ms, where the simulation time is not included in these results. Figure 2.12 shows the corridor scene from different view points. The curvature flow filtering smoothes the surface and prevents the fluid from looking "jelly-like". However, the curvature flow filtering is dependent on the view distance (as one can see in Figure 2.12), and the proposed simple noise-based surface foam effect does not have a volumetric appearance (see Figure 1.1).

2.6 APIs

This section gives a brief overview of APIs realated to and used in our implementation. An Application Programming Interface (API) is a software interface which enables a developer to access the underlying hardware and represents a standard for accessing and programming the hardware.

2.6.1 OpenGL

The *graphics API* used in the implementation is *OpenGL* (Open Graphics Library) [26], which is a cross-platform API for writing applications that produce 2D and 3D computer graphics and enables a developer to program the graphics hardware.

2.6.2 Shader Authoring Language

In the field of computer graphics, a shader is a program executed on the *graphics processing unit* (GPU). Shaders supersede the fixed-function pipeline, are used to calculate rendering effects and are classified as follows:

- Vertex Shaders: Are executed exactly once for each vertex that is given to the GPU. The primary operation a vertex shader performs is to transform the world space coordinates to screen space.
- Geometry Shaders: Are used to procedurally generate new graphics primitives from those primitives that are processed by the vertex shader. These new graphics primitives can be points, lines or triangles.
- Fragment Shaders: Are used to calculate the output color of pixels delivered from the rasterizer. Common effects that are calculated with fragment shaders are *Per Pixel Lighting*, *Bump Mapping*, and more.

For OpenGL, the standardized high-level shading language is the *OpenGL Shading Language* (GLSL) [27]. It was development by the OpenGL ARB as a replacement for the low-level *OpenGL Assembly Language*. GLSL was originally introduced as an extension to OpenGL 1.4 and since the introduction of OpenGL 2.0 it is part of the core. Microsoft developed the *High Level Shading Language* (HLSL) [16] for use with the *DirectX* API [11]. Furthermore, HLSL is used to develop shaders on Xbox and Xbox360. The third kind of shading language is the *Cg Shading Language* [5] developed by NVIDIA [25]. This shading language is not bound to a specific graphics API and can be used with OpenGL and DirectX. Cg shader code is portable to a wide range of platforms and the Cg compiler optimizes code automatically. Because of these benefits our implementation uses the *Cg Shading Language*.

Chapter 3

Method

Our method builds on the screen-space fluid rendering approach with curvature flow [37]. Similar to this method, we start from an SPH simulation calculated using a hardware physics engine (PhysX [29]), which provides a non-sorted 3D point cloud as input. Apart from the particle's position x we will also use the density ρ and velocity v for foam thresholding and the lifetime for varying the Perlin noise on a foam particle.

The original algorithm calculates the water depth by splatting the particles, then smoothes the depth buffer using curvature flow filtering, then calculates water thickness by accumulating particle depths in a separate thickness buffer, and finally composites the results. Our algorithm extends this by adapting the curvature flow filter for the viewer distance, and by adding a foam layer that can lie between two water layers.

Our algorithm then performs the following steps once per frame after the scene has been rendered into a texture (see Figure 3.1):

1. Calculate water and foam depth (Section 3.2.2)
2. Smooth the water depth using the new adaptive curvature flow algorithm (Section 3.1)
3. Calculate water and foam thickness (Section 3.2.2)
4. Composite water and foam layers using intermediate results (Section 3.2.3)

In this chapter we will describe our advanced particle based water and foam rendering approach. First of all we will introduce the *adaptive curvature flow* method which extends the curvature flow filtering introduced by van der Laan et al. [37], then we will describes our *real-time foam* approach. Our foam approach is separated into the *foam formation*, which is carried out in a physically based manner, the *layer creation* and finally the *layer composition*, which ensures that all the intermediate results are composed into the final image.

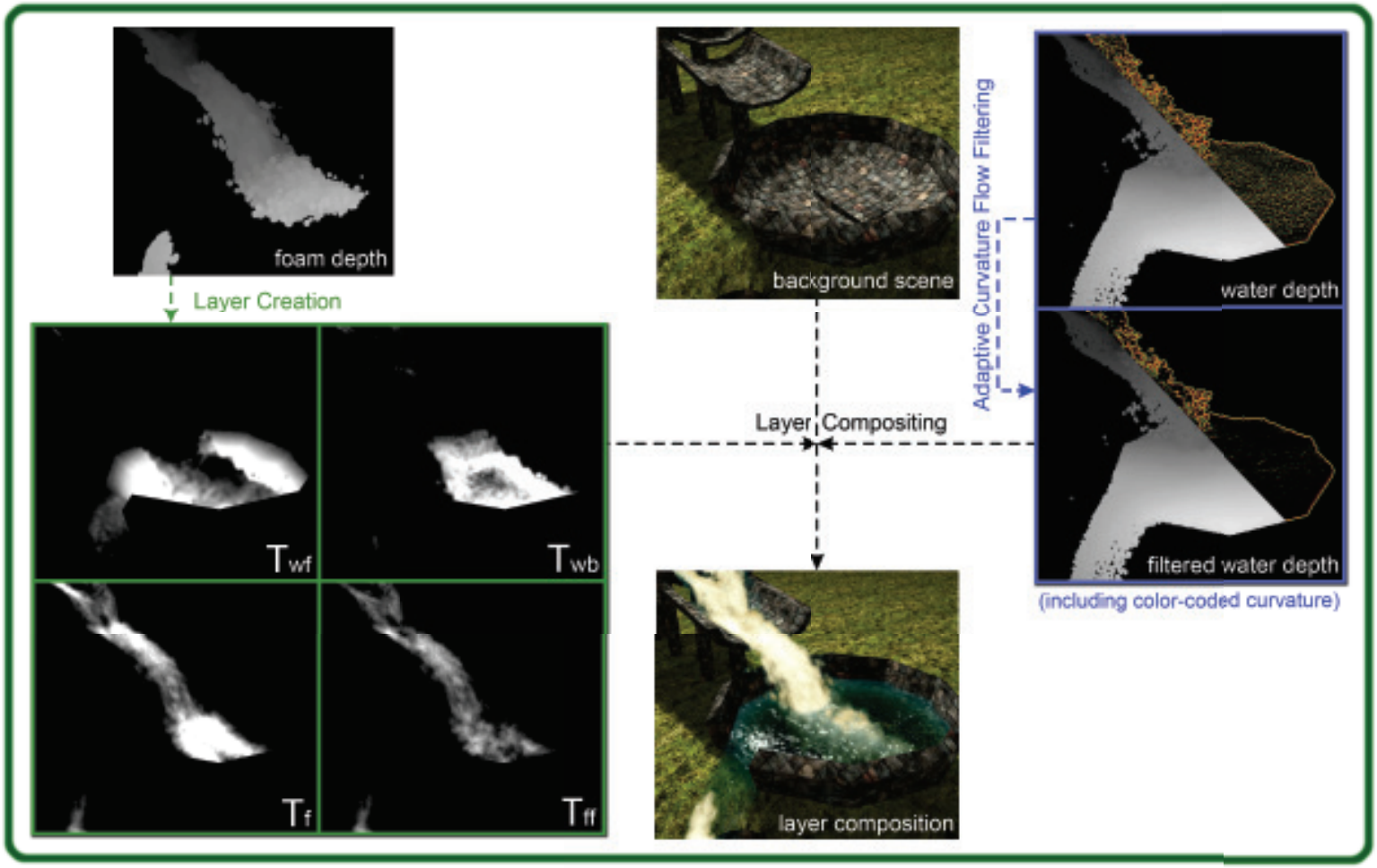

Fig. 3.1: Overview of the buffers used in our method: T_{wf}: thickness of the water in front of the foam; T_{wb}: thickness of the water behind the foam; T_f: foam thickness; T_{ff}: thickness of the front foam layer;

3.1 Adaptive Curvature Flow

The first step in rendering a fluid using particles is to create the fluid's frontmost surface. This is done by splatting the particles into the depth buffer using point sprites, with the depth values replaced by the geometry of a sphere. In order to avoid a "jelly-like" or "blobby" appearance due to the spherical particles, it is important to *smooth* the depth surface. Our *adaptive curvature flow* filtering is based on van der Laan et al.'s [37] *screen space curvature flow* filtering as described in Section 2.5.3.

However, close inspection of the screen-space curvature formulation reveals that the reference coordinate system for calculating the curvature is the window coordinate system. This means that, given equal iteration sizes, a particle that appears larger on screen (because it is closer due to perspective) will have a significantly larger radius in this coordinate system and therefore significantly lower curvature than a particle that is farther away. The resulting artifact is that smoothing will have a lower effect on closer particles, which therefore retain the unwanted spherical appearance, whereas particles far from the viewer will be overly smoothed, so that the fluid surface loses its defining characteristics such as highlights. This can be observed in Figure 3.2, left.

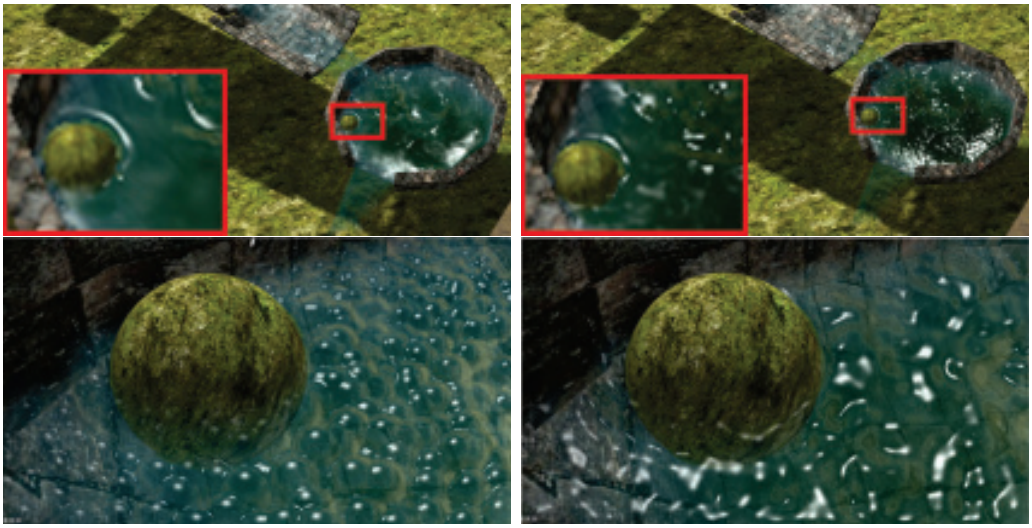

Fig. 3.2: In [37] *(left)*, distant water is over-smoothed (top) and near water is under-smoothed (bottom). Our new method *(right)* maintains the same amount of smoothing regardless of the distance.

One possible solution would be to remap the curvatures into a common reference coordinate system, for example by dividing H_s by z for each eval-

uation of H_s. However, our experiments have shown that this makes the integration very unstable, because the screen-space evaluation for larger particles is very noisy due to depth quantization. On the other hand, depth correction would make the resulting curvatures large in magnitude, leading to oscillation.

Therefore, we approach the problem from a different direction and interpret each integration step as a filtering step with a 3x3 kernel. Obviously, repeated filtering leads to an increased screen-space kernel radius r_s of the hypothetical overall filter – in fact, the number of iterations corresponds exactly to r_s. The main idea is now to vary the number of iterations depending on the view space distance z in order to obtain a roughly equal overall filter kernel size r_w in world space, making sure that a similar world-space neighborhood is taken into account when calculating the curvature flow. So r_s can be calculated from a desired world-space kernel radius r_w through

$$r_s = \frac{r_w}{z}\frac{FV}{2} \tag{3.1}$$

where F is the focal length, V is the viewport width in pixels, and z is the eye-space z-distance. So in iteration i, an Euler iteration step is only applied to a pixel if $r_s < i$. For optimization, the user can specify a maximum iteration count. Furthermore, an occlusion query is issued for each iteration to check whether any depth value was actually modified. If that is not the case, all pixels are already converged and no further iteration is necessary.

Smoothing is only applied to the frontmost water surface. Beside this water depth map, a second depth map (see Figure 3.1) is computed as described in the next Section, but smoothing for this depth map is not necessary because it is only used as a helper depth map during the layer creation.

3.2 Real-Time Foam

In this section we describe how to incorporate foam into real-time fluid rendering. We define water foam as a substance that is formed by trapping air bubbles in the liquid. Foam is usually observed as spray or bubbles above the surface of a turbulent water stream. However, we have also observed that a significant visual effect is caused by foam that occurs behind a water surface, usually due to a turbulent water stream that immerges into resting water with high impact (see Figure 3.4).

In order to capture these two main effects in a real-time setting, we separate foam particles from water particles and arrange the resulting foam and water particles in separate layers and render them using volumetric back-to-front compositing. Although our layered representation does not account for

discontinuity in the fluid volume which occurs if there are several layers of water and foam, the two most common cases mentioned above are covered by this model.

3.2.1 Foam Formation

First, we classify particles as water or foam. Following [22], we base the classification on the Weber number [33], which is a dimensionless physical quantity that describes the relative influence of the inertia of a fluid to its surface tension. The Weber number is defined as the ratio of the kinetic energy to the surface energy:

$$We = \frac{\rho v^2 l}{\sigma}, \tag{3.2}$$

where ρ is the density, v is the relative velocity between the liquid and the surrounding gas, l is the characteristic length, and σ is the surface tension. For larger We, the kinetic energy of the water is greater than the surface energy, causing water to intermix with the surrounding air, which results in foam formation. Thus, we separate particles into water and foam particles by thresholding the Weber number. In practice, we use a linear transition area where the particle is counted both as water and foam particle to ensure a smooth emergence and disappearance of foam. The new foam particle starts out as a point and expands, while the corresponding water particle shrinks.

Similar to [22], we assume that the surface tension and the characteristic length are fixed for the SPH simulation. We also assume that the characteristic length l is the particle diameter, which is a simplification for our real-time purposes, and that the surrounding air is not moving, and therefore the relative velocity v is the velocity of the particles. The velocity v and the density ρ are obtained from the physics simulation package.

3.2.2 Layer Creation

Now that particles have been classified as either particle or foam, we partition the fluid into *layers*, as shown in Figure 3.3.

By using two water layers, one in front and one behind the foam layer, we can simulate foam inside water, as happens at the end of a waterfall (see for instance Figure 5.2, or Figure 3.4). We first determine the front water surface and the front foam surface by splatting water and foam particles into separate depth buffers (the splatting step was described in Section 3.1). Curvature flow is only applied to the front water surface.

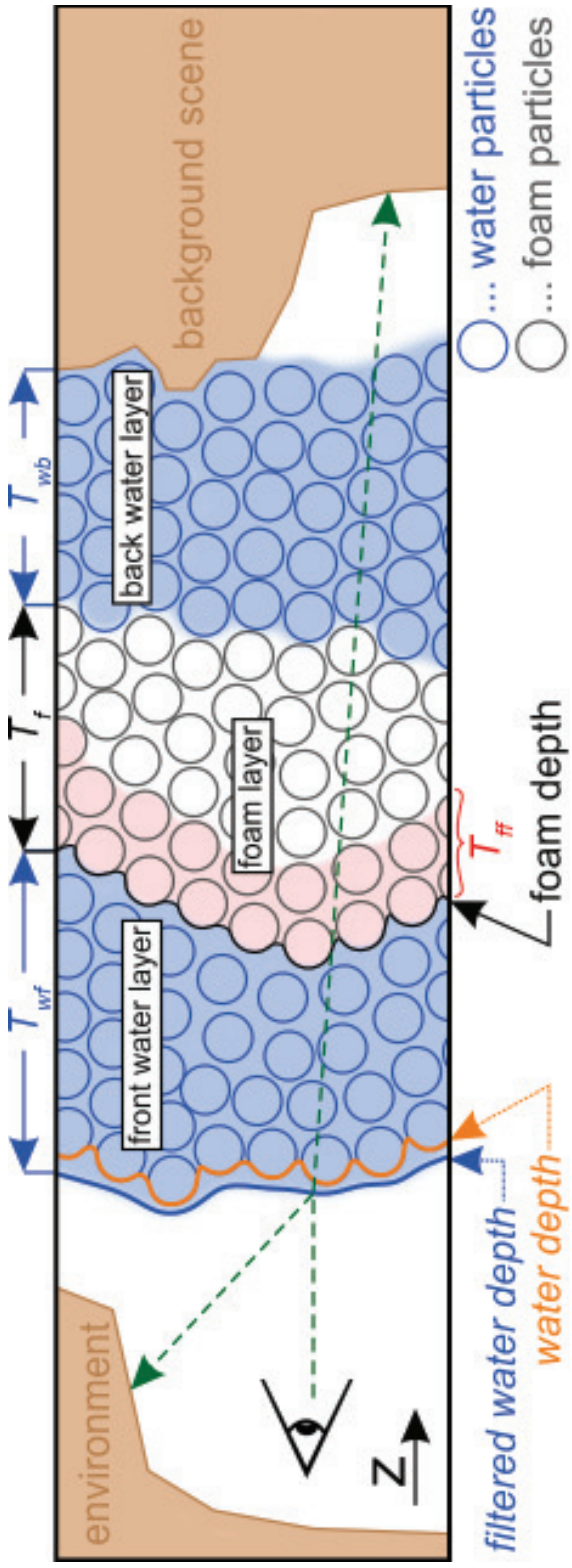

Fig. 3.3: A cross-section of our layered water model: The volumetric appearance of the result is achieved by not only accounting for the water thickness T_{wb} at each pixel as previous approaches [37], but also for the foam thickness T_f and the thickness of water in front of the foam T_{wf}. We also partition foam into two differently colored layers (T_{ff}) to achieve more interesting foam.

Since water and foam are volumetric phenomena, the amount of water respectively foam between two layer surfaces needs to be determined in order to allow correct compositing and attenuation. Similar to [37], the thickness of a layer is determined by additively splatting every particle belonging to the volume into a buffer. In contrast to the depth surface calculation, the splat kernel gives the thickness of the particle at each particle sampling point. Accumulating particle thicknesses is a reasonable approximation because particles from the physics simulation can be assumed to be largely non-overlapping.

$$T(x,y) = \sum_{i=0}^{n} t(\frac{x - x_i}{\sigma_i}, \frac{y - y_i}{\sigma_i}) \tag{3.3}$$

where t is the particle thickness function, x_i, y_i are the projected position of the particle, x and y are screen coordinates and σ_i is the projected size. The particle thickness function is given by

$$t = 2N_z \; l \; e^{-2r} \tag{3.4}$$

where N_z is the z component of the view space normal, l is the particle diameter and r is the radius, calculated form the texture coordinates, on the point sprite.

In comparison to [37], we not only calculate the water thickness, but the thickness of:

- the foam surface T_f, by considering only foam particles. For the foam particles, the splat kernel is also multiplied by a 3D Perlin noise value, which is varied with the lifetime of the particle, to add sub particle detail.
- the front water surface T_{wf}, by considering only water particles that are in front of the foam layer (by comparing the particle depth with the depth of the front foam surface).
- the back water surface T_{wb}, by considering the other water particles.
- the front-most layer of the foam surface T_{ff}, to allow partitioning foam into two different-colored layers.

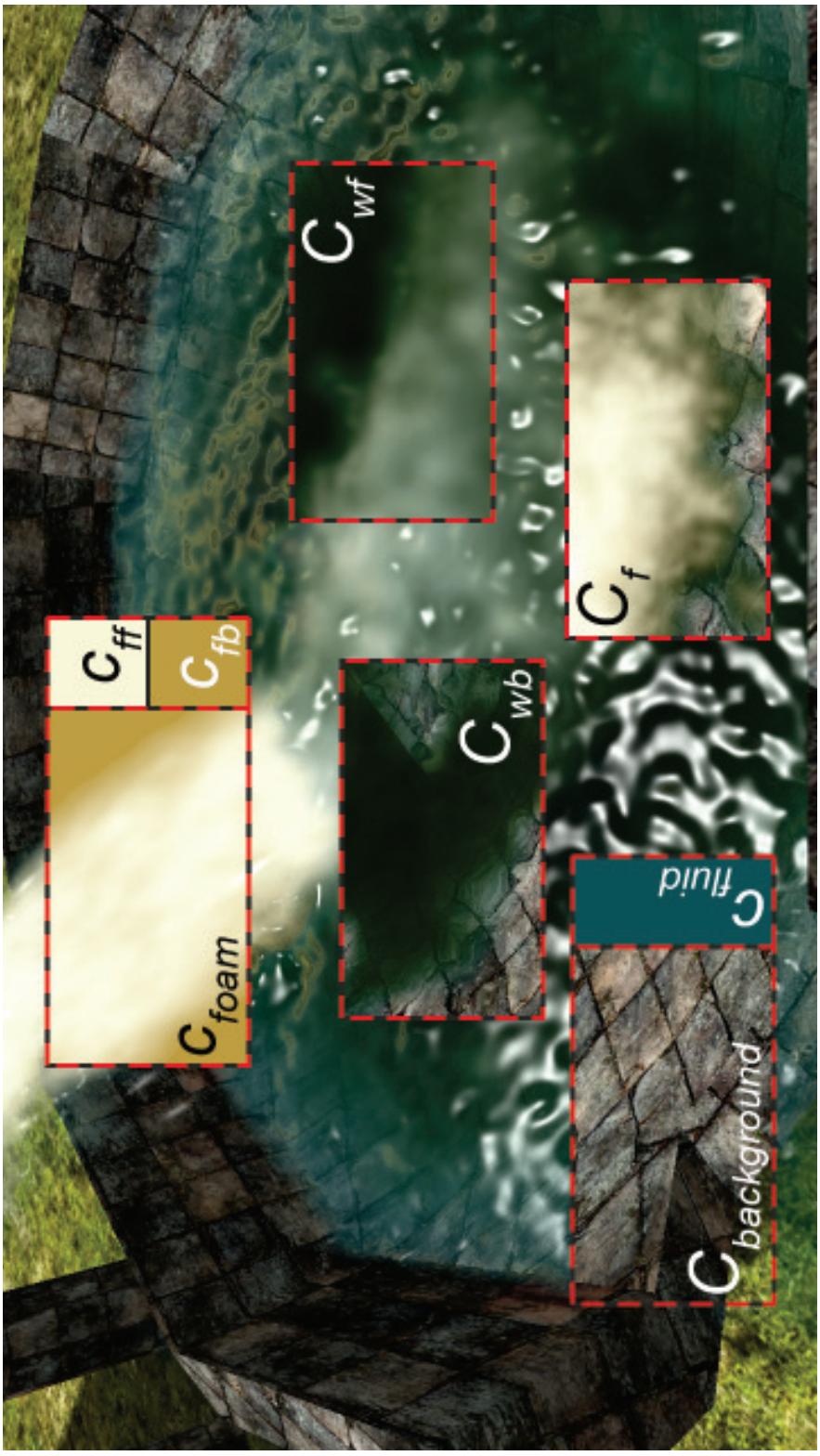

Fig. 3.4: User defined colors (c_{fluid}, c_{ff}, c_{fb}) and resulting colors from the compositing steps ($C_{background}$, C_{wb}, C_f, C_{wf}).

3.2.3 Layer Compositing

Finally, to account for the attenuation caused by the previously calculated layers, the actual pixel color is calculated by volumetric compositing. Figure 3.1 gives an overview of the buffers that are used for the compositing.

Compositing along a viewing ray back to front, we have (see Figure 3.3):

$$C_{wb} = \text{lerp}(c_{fluid}, C_{background}, e^{-T_{wb})} \quad (3.5)$$
$$C_f = \text{lerp}(c_{foam}, C_{wb}, e^{-T_f}) \quad (3.6)$$
$$C_{wf} = \text{lerp}(c_{fluid}, C_f, e^{-T_{wf}}) \quad (3.7)$$

where c_{foam} and c_{fluid} are the colors of the medium. Figure 3.4 shows the individual steps and colors used in the compositing.

In addition to attenuation, we also calculate reflection with a Fresnel term, and a highlight at the front water surface, as well as refraction, similar to [37]. For reflection and highlight, care needs to be taken because the front water surface might be behind the foam. So if $T_{wf} = 0$, we have $C_{wb} = illuminate(C_{wb})$, otherwise $C_{wf} = illuminate(C_{wf})$, where

$$illuminate(x) = x(1 - F(\mathbf{nv})) + rF(\mathbf{nv}) + k_s(\mathbf{nh})^\alpha, \quad (3.8)$$

i.e., the standard Fresnel (F) reflection calculation (r is a lookup into an environment map) and a Phong term. We also carry out refraction for the whole water surface, so $C_{background}$ is sampled from the scene background texture perturbed along the normal vector, scaled using T_{wb}+T_{wf} (see Section 2.5.3).

Finally, we have found that foam can be made to look more realistic by blending two different user defined colors, c_{ff} and c_{fb}. The thickness T_{ff} is calculated along with the foam thicknesses T_f by accumulating just the foam particles within a user defined constant range δ_{foam}. So c_{foam} is actually calculated as

$$c_{foam} = \text{lerp}(c_{fb},\ c_{ff}, e^{-T_{ff}}). \quad (3.9)$$

By considering only foam particles which are close behind the foam surface, we obtain a foam pattern that has a controllable thickness, achieving the benefit that the pattern moves along with the particles and exhibits fine micro-structures in its visual appearance.

Chapter 4

Implementation

This chapter describes the implementation of our *layered particle-based fluid model for real-time rendering of water*, including all technical implementation details. All provided source/shader code listings are taken from our implementation, which has been developed for this thesis, and are included in the Appendix A.

The primary logic concerning the water simulation and rendering is separated into two classes:

- Fluid Class
- ScreenSpaceCurvature Class

The fluid methods and data structures provided by the physics engine are encapsulated in the *Fluid* class. Additionally, the physics engine provides a mechanism to fragment the simulation data into packets. Each packet holds particles from the simulation and their location in space is represented by an *axis-aligned bounding box* (AABB), which can be used for view-frustum culling (see Section 4.3). The class which represents the implementation of the algorithm presented in this thesis is called *ScreenSpaceCurvature*. The basic program flow from the applications point of view is as follows:

- Update foam simulation (parallel to the physics sim. update; see Section 4.2)
- Update ScreenSpaceCurvature
- Render background/main scene into texture (see Section 4.4)
- Render intermediate results (see Section 4.6 and 4.8)
- Compositing (see Section 4.9)

- Render full-screen quad of rendering results (post-processing can be applied in this step)

Similar to the fluid class this class is used to encapsulate the Weber number based simulation update, and the different shaders involved. An overview about the program flow of the involved shaders is shown in Figure 4.1. Additionally, the class manages rendering parameters such as particle size, water base color, falloff color, and more, which control the visual appearance of the water. Furthermore, the view frustum, represented by six planes which are calculated from the current view-projection matrix, is stored and also parameters like field of view, window size or the light position are passed to the class by appropriate member functions.

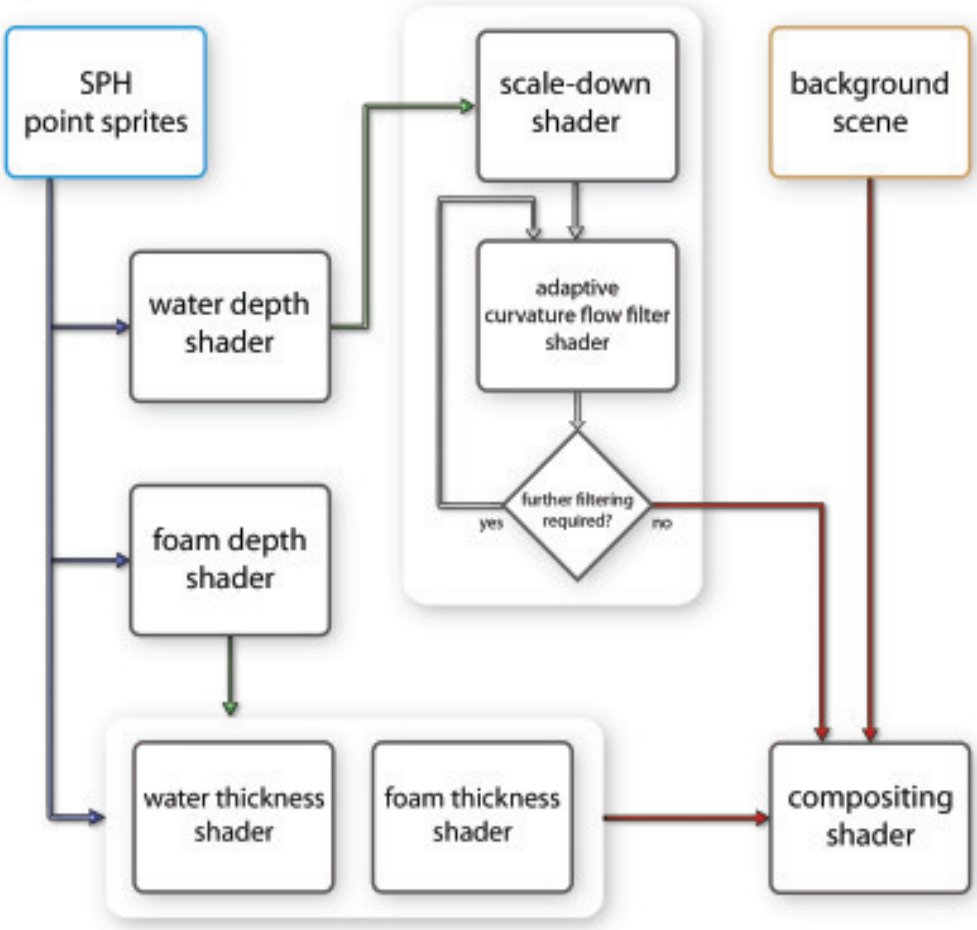

Fig. 4.1: Program flow of involved shaders.

4.1 Textures and Render Targets

As described in Section 3 and illustrated in Figure 3.1, all intermediate results are stored in textures. By default, the texture filtering modes for minification and magnification are set to *GL_NEAREST* to prevent unwanted filtering during the composition pass. The same applies to the texture repeat modes, in which *GL_CLAMP_TO_EDGE* is used to prevent black borders around the texture.

Texture	Internal Format	Format
sceneTexture	GL_RGBA8	GL_RGBA
depthTexture	GL_LUMINANCE32F_ARB	GL_LUMINANCE
foamDepthTexture	GL_LUMINANCE32F_ARB	GL_LUMINANCE
thicknessTexture	GL_RGB16F_ARB	GL_LUMINANCE
foamThicknessTexture	GL_RGB16F_ARB	GL_LUMINANCE
noiseTexture	GL_LUMINANCE16F_ARB	GL_LUMINANCE
resultTexture	GL_RGBA8	GL_RGBA

Tab. 4.1: Internal format and formate of textures used for intermediate results.

Table 4.1 illustrates the textures used for storing the intermediate results of the algorithm. In the case of the *thicknessTexture* and the *foamThicknessTexture*, *GL_RGB16F_ARB* is used as an internal format, although only two color channels of the three available are used. This is because there is no internal format with two color channels which is usable for the algorithms purposes. Furthermore, the texture target is *GL_TEXTURE_RECTANGLE_ARB* for all intermediate textures and the texture dimension is equivalent to the viewport size.

For the adaptive curvature flow filtering approach additional textures are needed to perform the downsampling (see Section 4.7). The number of required downsampling textures depends on the resolution the filtering is carried out. For instance, if the filtering is done at half resolution, one additional texture is required. In the case of performing the filtering at quarter resolution two additional textures are needed to perform the downsampling. The actual filtering needs one more additional texture, which has an resolution equivalent to the one at which the filtering is done. This texture and the lowest resolution texture from the downsampling textures are used to perform *multipass ping-pong* rendering.

The last category of textures required by the algorithm of this thesis are helper textures, used for pattern generation and improving rendering performance. Table 4.2 shows the textures including the parameters used

for creation. The *foamPerlinNoise* texture is an important quantity for the foam rendering approach. As described in Section 3.2.2 it is multiplied by the splat kernel. The texture is based on Perlin noise introduced by Ken Perlin [28] and the dimension is $64x64$ as illustrated in Figure 4.2.

Texture	Target	Internal Format	Format
foamPerlinNoise	TEX_3D	GL_RGB8	GL_UNSIGNED_BYTE
squareRootRamp	TEX_1D	GL_LUMINANCE	GL_FLOAT

Tab. 4.2: Target, internal format and formate of used helper textures.

Fig. 4.2: FoamPerlinNoise texture.

The *squareRootRamp* is used during rendering to calculate the square root of an given value in the range of $[0, 1]$ by performing a texture lookup. This gives an important performance speedup, especially if the square root calculation is frequently done in the fragment shader, because a texture lookup is much faster in that case. Although the accuracy of this method is not as accurate as the accuracy of the $sqrt(x)$ function provided by the *Cg Standard Library* [5], the accuracy is sufficiently precise for rendering tasks and does not introduce any visible artefacts.

As indicated in Table 4.2 the *squareRootRamp* is a one-dimensional texture, it has a width of 128 pixels and during initialization of the application it is filled with the square root values between 0 and 1. Figure 4.3 gives an illustration of an *squareRootRamp* with a width of 16 pixels (to illustrate the discrete steps). As one can see, the resulting diagram shows a polyline,

which is achieved by linear interpolate pixel values if the texture lookup falls between two pixels.

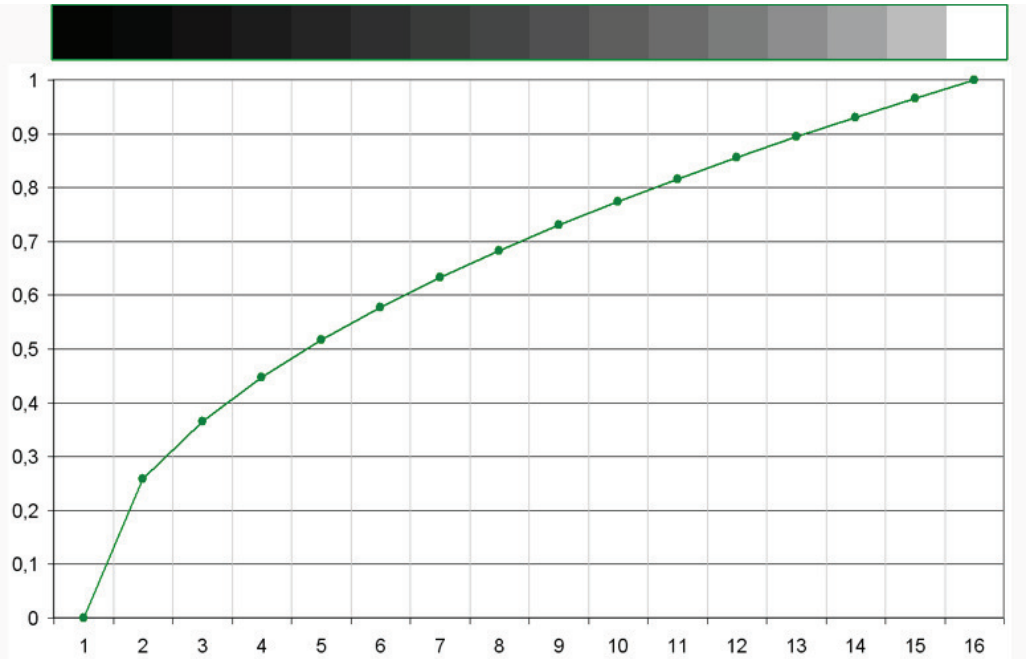

Fig. 4.3: Square root texture used during rendering; Top: Debug rendering of the *squareRootRamp* texture; Bottom: Diagram illustrating the progress of the square root function.

To redirect the rendering output into rendering destination buffers the *GL_ARB_framebuffer_object* (FBO) extension [15] is used. This feature enables the algorithm to directly render in each pass into one of the intermediate textures.

4.2 Simulation Update

The first thing that has to be carried out at the beginning of each frame is to update the simulation. This includes to simulate one step of the physics simulation as well as to update the Weber number-based foam simulation. The simulation data is a non-sorted 3D point cloud and each of the particles has the following properties: position, density, velocity, lifetime, and a unique number identifying the particle.

In each simulation step of the physics simulation, the properties of the particles are updated by performing an SPH simulation. Because we build

on an existing physics engine which supports SPH fluid simulation, the foam simulation is separated from the physics simulation (but could also be part of the physics simulation if the source code is available or a custom SPH implementation is used). As described in Section 3.2.1 the foam simulation is responsible for classifying the particles as either water particles or foam particles. The foam simulation uses its own data structure:

- *FOAM*: value used for fading and scaling of foam particles
- *LIFETIME*: lifetime that a particle has, depending on its current phase
- *TIMER*: timer used for accumulating the current lifetime
- *PHASE*: current phase of the particle

The *foam phase* describes in which state a particle currently is. At the beginning a particle is counted as a water particle and its state is *FP_NONE*. If the Weber number of the particle exceeds the user defined threshold its state changes to *FP_WATER_TO_FOAM*. During this phase the foam value of the particle increases from 0 to 1. After the foam value has increased to 1 the particle enters the *FP_FOAM* phase and stays in this state as long as the Weber number is above the threshold. If the Weber number falls below the threshold the *FP_FOAM_TO_WATER* phase is entered and the particles foam value is decreased until it reaches 0, whereas a user-defined lifetime is used during this phase. Listing A.1 and A.2 show the main loop of the foam simulation update.

Because of performance reasons the foam simulation is done in parallel to the physics simulation which consists of the general rigid body simulation and the fluid simulation. The drawback of this parallelism is that the foam simulation lags one frame behind because both simulation steps work on the same data basis and the SPH simulation buffer is locked during the simulation step. However, close inspection of the difference between the iterative approach and the parallel one reveal that the resulting error can be neglected. The concept of the parallelism is shown in Listing A.3.

4.3 View Frustum Culling

After the simulation has been updated the data in the simulation buffers is ready for rendering. An important performance optimization is to use *view frustum culling* [7]. For that purpose the physics engine provides a functionality which enables one to query for spatial data that is connected to the fluid simulation. This spatial data is presented by packets, whereas

one packet is described by the *NxFluidPacket* structure as shown in Listing A.4. This meta data enables the algorithm to avoid ~~to~~ send particles down the rendering pipeline because they are outside the camera's frustum and therefore are ~~being~~ outside of the unit cube in clipping space. Because the *axis-aligned bounding boxes* (AABBs) instead of the particle positions are tested against the view frustum the number of performed frustum tests is quite small. Figure 4.4 shows a rendering of the waterfall scene including the debug rendering of the AABBs.

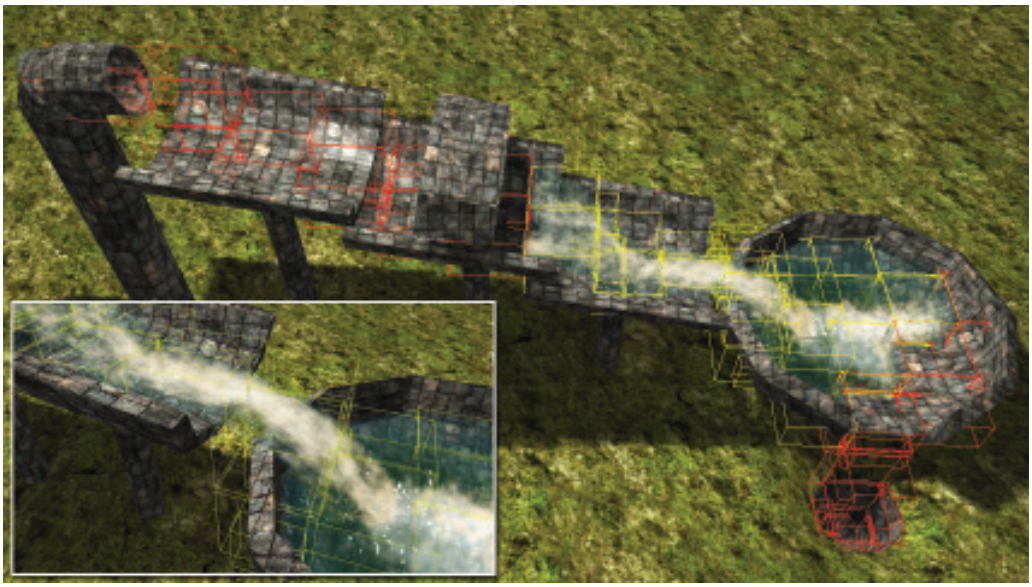

Fig. 4.4: Renderings showing the AABBs provided by the physics engine. Both figures use the same camera viewpoint for the culling. The red AABBs are culled.

The view frustum is represented by six planes including the far and near clip planes (defined by the standard plane equation $Ax + By + Cz + D = 0$, where~~as~~ $\vec{n} = (A\ B\ C)$ is the normal vector to the plane). As shown in Listing A.5 the algorithm iterates over all packets and tests each packet's AABB against the view frustum. If the packet's AABB is fully or partly inside the view frustum the start index and the particle count of the packet is stored for later rendering. All packets that are fully outside the view frustum are discarded and are ignored by all rendering passes of the algorithm.

After the simulation data has been updated and the view frustum culling has been carried out, the resulting data is copied to a vertex buffer object [38] (VBO). One major advantage of a VBO is that one can specify the usage of the VBO and the VBO memory manager will choose automatically the best memory place for the buffer. This enables the algorithm to store the

SPH particle data including the results of the foam simulation directly in high-performance memory on the GPU for rendering. Attributes such as density, velocity, lifetime and foam are handled as texture coordinates.

4.4 Main Scene Rendering

The first rendering task which is performed by the algorithm is to render the main scene without the water into a texture as illustrated in Figure 4.5. Listing A.6 shows the concept of the main scene rendering, whereas everything that is rendered between a call to *BeginRenderScene* and *EndRenderScene* is stored in the scene texture. Also the depth values of the main scene are stored to ensure proper visibility.

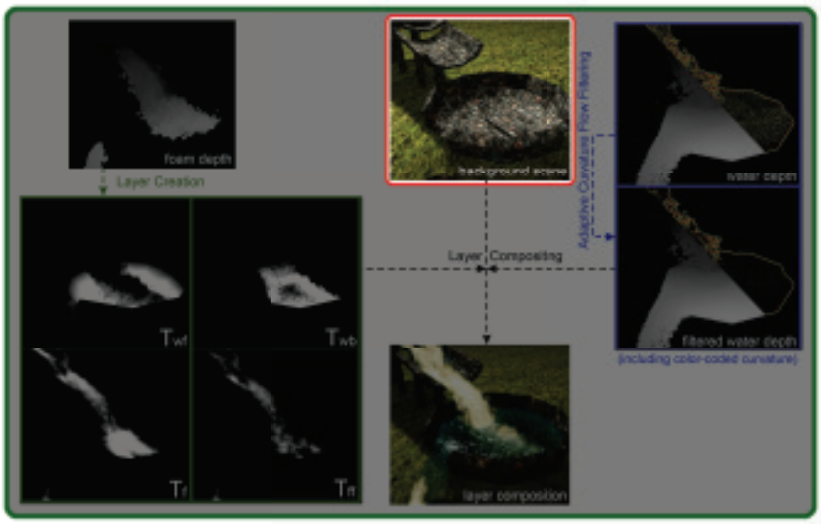

Fig. 4.5: Background scene rendered at beginning of the rendering process.

By using this concept the algorithm of this thesis can be seen as a post-processing method. A post-processing pipeline [30] applies post-processing effects to an already rendered scene and has become a standard in modern rendering engines. Because of this analogy that both work in the same way by rendering the main scene into a texture and performing additional computations on this data afterwards, our algorithm can be well integrated into a post-processing pipeline.

4.5 Point Sprites

The primary primitives used by the rendering passes as described later in this chapter are point sprites. Point sprites are an useful functionality if thousands of particles have to be rendered. Because our method can be classified as a splatting method, point sprites are an essential choice and thus are used by our implementation. Furthermore, point sprites give a significant performance improvement because only one vertex has to be send down the rendering pipeline for each particle of the SPH simulation instead of four vertices as it would be the case with ordinary billboard rendering.

The point sprite rendering functionality is encapsulated by the *RenderParticles* member function of the *ScreenSpaceCurvature* class as shown in Listing A.7. An important implementation detail which has to be addressed is that *GL_VERTEX_PROGRAM_POINT_SIZE_ARB* has to be enabled to be able to calculate the point sprite size in the vertex shader. As described in Section 4.3 we also use view frustum culling to reduce the particle amount that is processed by the algorithm. *vboStartIndices* and *vboIndexCount* are arrays holding the start indices and the index count of the packets that have passed the view frustum culling test, *visiblePacketCount* holds the overall packet count.

4.6 Depth Passes

The second rendering task that is done by our algorithm is to extract the frontmost water- and foam- surface. For this purpose the particles are splatteted into two depth textures, one for the frontmost water surface (see Section 4.6.1) and one for the frontmost foam surface (see Section 4.6.2). As illustrated in Figure 4.6 the depth passes are carried out after the background scene has rendered into a texture.

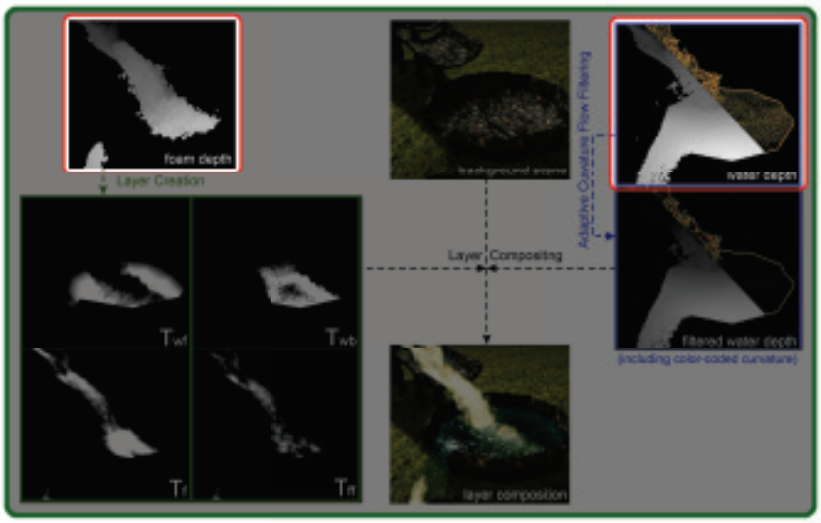

Fig. 4.6: Water and foam surface extraction.

4.6.1 Water Depth

As described above the algorithm needs to calculate the frontmost water surface. At the beginning of this pass the depth texture is cleared to -10000.0 which indicates that the water surface is infinitely far away. Furthermore, blending is disabled and the hardware depth test is enabled which ensures that the closest value at each pixel is retained. The particles are rendered by calling the *RenderParticles* function.

The main task of the vertex shader used in this pass (see Listing A.8) is to transform each particle and calculate the window-space point size. Additionally all particles with a density below a user-defined threshold are culled (see Section 4.9.5) and particles with a density slightly above this threshold are scaled. All other particles use the same world-space particle size. Although the point size calculation is only an approximation it does not produce any noticeable artefacts.

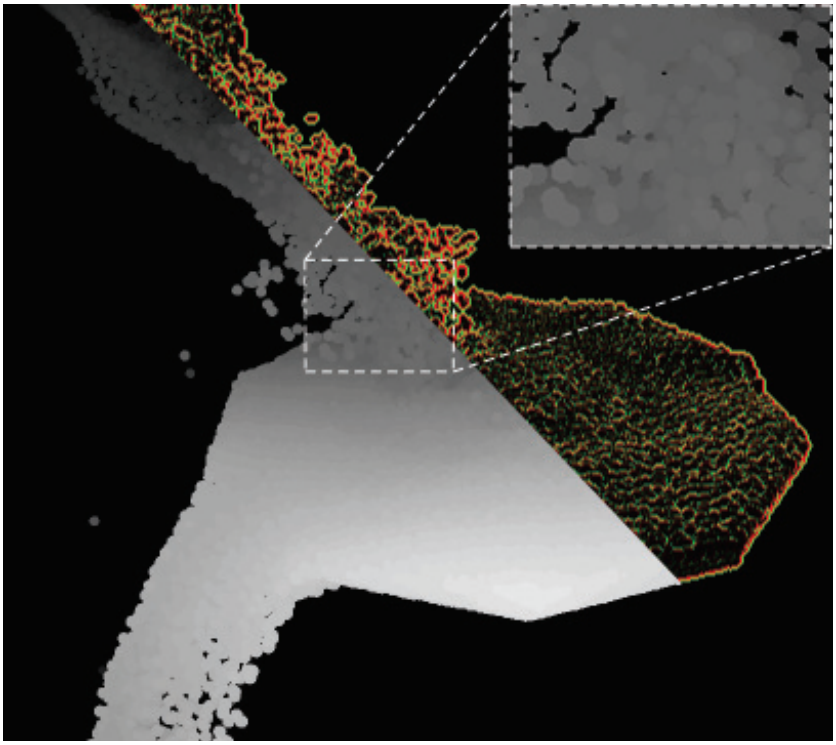

Fig. 4.7: Water depth pass results including close-up view of the water depth and color-coded curvature.

The fragment shader (see Listing A.9) used in this pass calculates the eye-space normal vector for every pixel of the point sprite and discards all pixels that are outside the inscribed circle. Then the depth values of the screen-aligned point sprite are moved towards the viewer to exactly present the geometry of a sphere. Finally the fragment shader outputs the linear eye-space (view-space) depth and the clip-space position which is needed for the hardware depth test.

Figure 4.7 illustrates the water depth texture after the water depth pass has been carried out. Note that the particles are rendered as spheres which form the frontmost water surface. However, the water surface has high curvature which can be observed in the close-up view.

4.6.2 Foam Depth

Beside the water depth our algorithm also calculates the depth of the frontmost foam surface. This is important to be able to separate the water and foam particles into layers as described in Section 3.2.2. In contrast to the water depth, this pass also uses the geometry shader to orient the foam

particles toward their velocity direction. This is done by calculating four vertices which lie on a plane parallel to the view plane and aligning them toward the velocity direction (see Listing A.10). Flickering artefacts ~~because of~~ low velocity are prevent by thresholding the norm of the velocity and the orientation. In this case the fallback is to render a simple screen-aligned quad. Finally the geometry shader calculates the view/eye space positions, transforms them to post-perspective space and emits four new vertices. The alignment is useful because it results in a kind of motion blur for the final foam rendering which is a reasonable assumption for foam rendering.

Fig. 4.8: Foam depth pass results.

In comparison to the water depth pass results the primitives are no longer ~~squared~~ as can be observed in Figure 4.8.

4.7 Adaptive Curvature Flow Filtering

This section describes how to perform our *adaptive curvature flow* filtering which is based on the *screen space curvature flow* filtering proposed by van der Laan et al. [37]. As described in Section 3.1 the purpose of our *adaptive curvature flow* filtering is to hide the particle-based nature of the SPH

simulation data by smoothen out sudden changes in curvature between the particles. The *adaptive curvature flow filtering* is performed after the depth passes have been finished as shown in Figure 4.9 because the input for the filtering are the depth values stored in the water depth texture. Note that the filtering is only applied to the water depth because the foam depth is only used during the layer creation process and not directly for rendering, and thus filtering is not necessary.

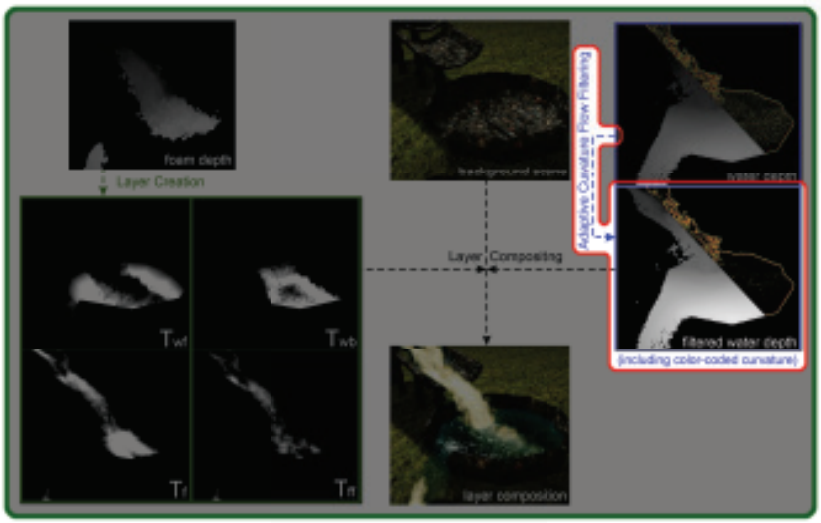

Fig. 4.9: *Adaptive curvature flow* filtering applied to the water depth.

The *screen space curvature flow* filtering is an iterative process and each iteration step corresponds to an Euler integration. As illustrated in Figure 4.10 using a fixed iteration count smoothens the surface and prevents the fluid from looking blobby or "jelly-like", but introduces over-smoothing for far view distances and under-smoothing for near view distances respectively. In contrast our *adaptive curvature flow* filtering maintains the same amount of smoothing regardless of the view distance (see Figure 3.2).

The rendering of the *adaptive curvature flow* filtering is separated into two passes.

- Depth texture scale down
- Depth texture smoothing

The scale down is done because of efficiency reasons. Performing the smoothing on a lower resolution (half or quarter resolution) results in a huge performance gain. The scaled down depth texture is calculated based on the

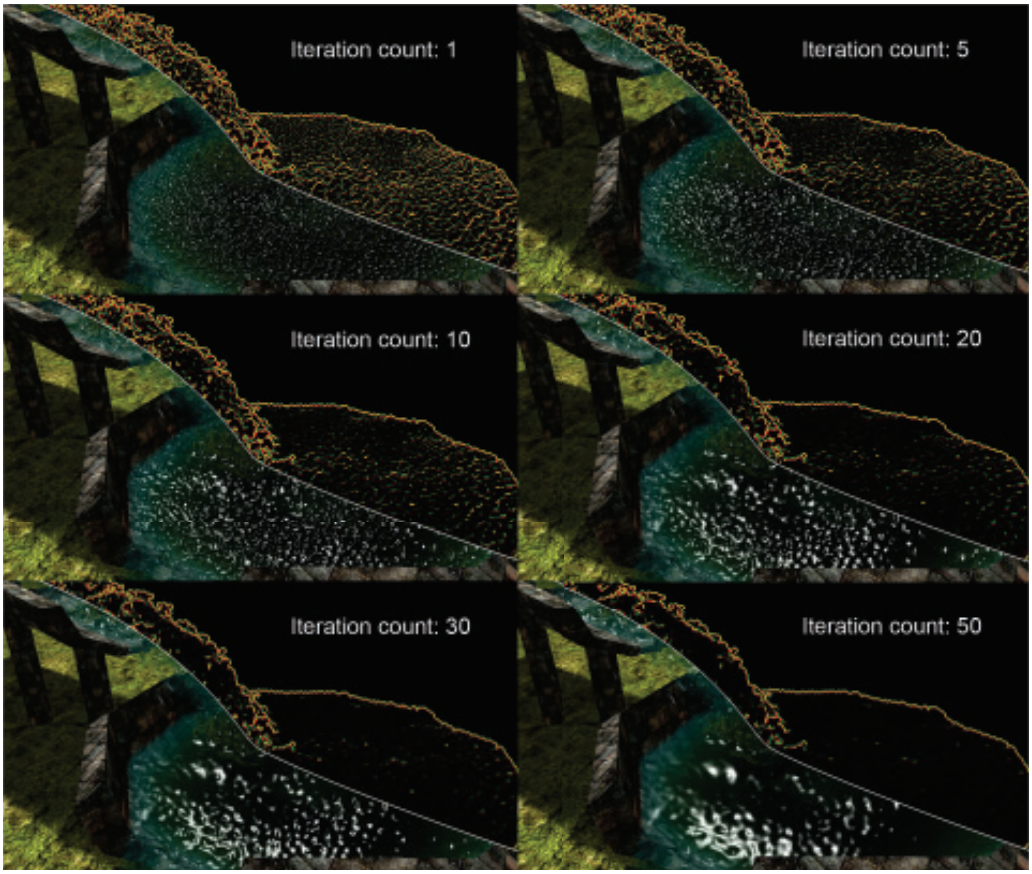

Fig. 4.10: Comparison of the *screen space curvature flow* filtering using different fixed iteration counts as proposed by van der Laan et al. [37].

full resolution water depth texture by rendering a full-screen quad with a viewport half as large as the full screen resolution.

The next step is to calculated the actual *adaptive curvature flow* filtering. The filtering is performed until no further change happens or an upper bound of 256 iterations has been reached as illustrated in Listing A.11. The required iteration count is determined by using the occlusion query functionality provided by the *OpenGL* API. An occlusion query is started by a call to the *glBeginQueryARB* function, whereas the query is only done for every fifth iteration in our current implementation because of performance reasons. After this the required shader parameters like the current iteration count (which is needed by the fragment shader to determine if further filtering is required for a single pixel) are applied and a fullscreen quad is rendered. The occlusion query is stooped by a call to the *glEndQueryARB* function afterwards.

Because the filtering fragment shader uses the *discard* instruction of the *Cg Shading Language* [5] and our upper bound calculation is based on occlusion queries the ping-pong rendering is not carried out in a traditional way. After the actual filtering of the water depth texture, the filtering result is only copied to the other render target instead of directly filtering again. In this way it is ensured that the filtering result is correct and not overwritten by mistake.

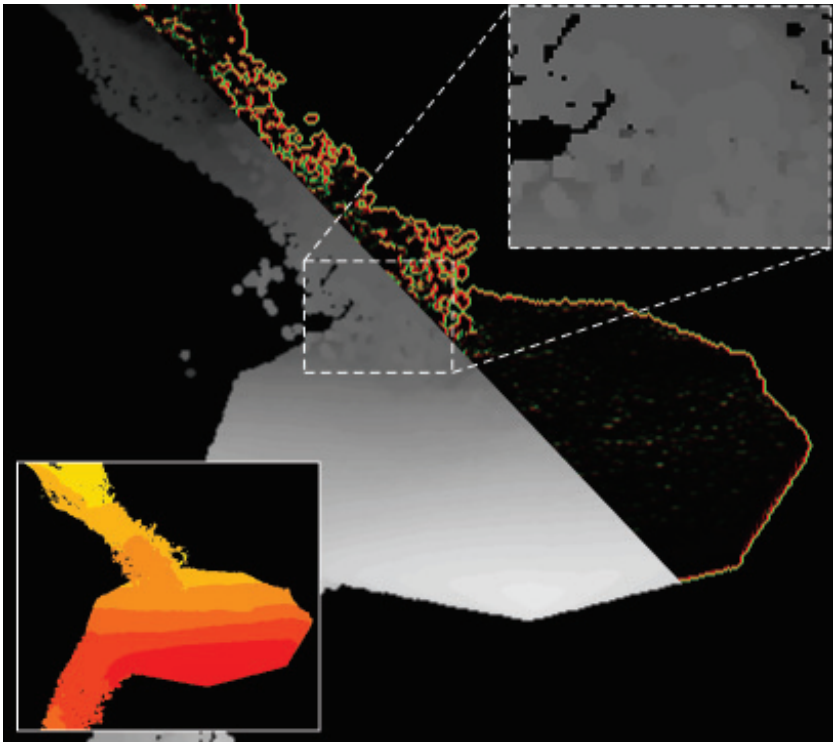

Fig. 4.11: *Adaptive curvature flow* filtered water depth including a close-up view of the depth values, color-coded curvature, and iteration count.

The fragment shader used to perform the *adaptive curvature flow* filtering calculates the necessary iteration count and smooths out sudden changes in curvature. The fragment shader is shown in Listing A.12 and Listing A.13. The first part of the fragment shader look-ups the water depth from the water depth texture and calculates the necessary iteration count depending on the eye-space depth as described in Section 3.1 and illustrated in Equation 3.1. Because all parameters of Equation 3.1 except the eye-space depth are constant during runtime their result can be pre-calculated (stored in the shader parameter named *depthFilterParam*). The minus in front of the depth value (*coo*) is required because in *OpenGL* [26] eye-space depth values are

negative in front of the camera. Based on the calculated iteration count and the current iteration count (passed as a shader parameter) a decision is made if further filtering is necessary or the fragment has to be discarded because it has already converged. Additionally, eye-space depth values are discarded that do not belong to the water surface (which is indicated by a depth value less than $-9999.0f$).

The second part of the fragment shader calculates the curvature flow filtering. First, the neighborhood depth values are looked up and the derivatives are calculated by central differencing. Additionally, the directional derivatives are calculated and tested against an user-defined depth falloff to prevent filtering across the silhouettes of the water surface. These boundary conditions are enforced where large changes in depth occur between one fragment and the next. At these boundaries the derivatives are forced to be zero to prevent any smoothing from taking place as proposed by van der Laan et al. [37]. Finally, the fragment shader calculates the mean curvature as described in Section 2.5.3 and modifies the eye-space depth values of the water depth texture. Because this process corresponds to an Euler integration the mean curvature is weighted by an user-defined parameter named *epsilon* to prevent the filtering from oscillating.

Figure 4.11 shows the result of the *adaptive curvature flow* filtering. As one can observe, the curvature of the water surface (see color-coded curvature on the right side) is much lower in comparison to the unfiltered water depth as illustrated in Figure 4.7. Also the close-up view shows less edges and has a smoother appearance. The subfigure shows the iteration count for this example viewpoint, whereas a red color indicates a higher iteration count and a yellow color a lower iteration count.

4.8 Layer Thicknesses

So far the algorithm has acquired the main scene color and extracted the water and foam surface from the SPH data. Furthermore, the water depth has been filtered to hide the particle-based nature of the SPH particles. The next step is to separate the particles into layers. As described in Section 3.2.2, our algorithm separates the particles into three layers, two water layers and one foam layer in between. As shown in Figure 4.12, the input data for this pass is the previously extracted foam surface and the SPH particles. Based on this input data the algorithm does the separation and calculates the thickness for each of these layers by splatting the particles and accumulating the result into the thickness textures. This thickness-based rendering approach gives the viewer important information about the water deep.

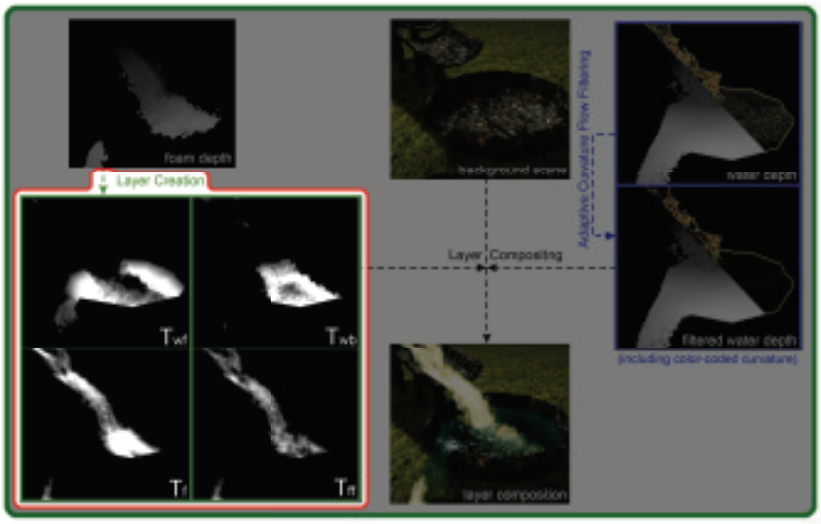

Fig. 4.12: Layer creation based on extracted foam surface.

4.8.1 Water Layer

The water thickness pass is responsible for creating the two water layers which we call *front water layer* and *back water layer* (see Figure 3.3). From the rendering point of view this pass is similar to the water depth pass (see Section 4.6.1) because this pass also renders the entire set of particles. The difference is that this pass uses additive blending to accumulate the contribution of each single particle that belongs to one of the water layers into the thickness texture and also the depth-write functionality is disabled. Another important difference is that the depth buffer of the main scene is used at this point for the hardware depth test. This ensures that the thickness of the layers is forced to zero in regions where the water is occluded by the geometry of the main scene.

The water thickness vertex shader is based on the water depth vertex shader (see Listing A.8). Beside the transformation and point-sprite size calculations the vertex shader also performs the layer separation as shown in Listing A.14. For this purpose the shader gets the *foam* parameter as an input (see Section 4.2) and outputs the layer to which the particle belongs to. The actual decision to which layer the particle belongs to is calculated by using the *step* function provided by the *Cg Shading Language* [5]. Whereas the *step* functions returns 0 if the foam depth is greater than the particles eye space depth and 1 if less or equal.

The fragment shader calculates the particle thickness function as illustrated in Figure 3.4. Instead of calculating the eye space position of the

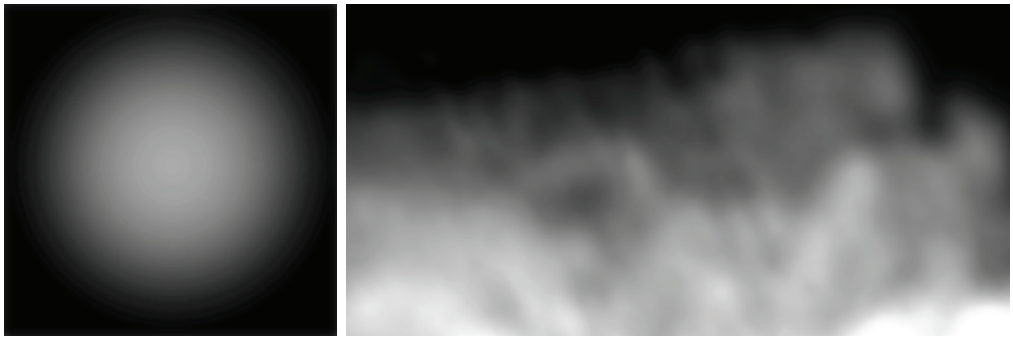

Fig. 4.13: Splat kernel used for the water thickness; left: single particle; right: close-up of the back water layer thickness.

fragment on a sphere and use it as an output, the fragment shader outputs two thickness values. As shown in Listing A.15, the thickness for each layer is calculated based on an exponential falloff and the layer separation flag *foam*. In so doing, each water particle contributes to the correct water layer.

Figure 4.13 illustrates the exponential ~~based~~ splat kernel calculated by the water thickness fragment shader A.15. On the left side the splat kernel for a single particle is shown and on the right side a close-up view. Note that by using an exponential ~~based~~ splat kernel the accumulated thickness does not show any edges or discontinuities as one can observe in the close-up view.

Figure 4.14 illustrates the results of the water thickness pass. On the left side the thickness of the *back water layer* is shown and on the right side the thickness of the *front water layer* ~~respectively~~. Note that the water thickness is zero for regions that are occluded by the geometry of the main scene. This can be observed at the bottom of both figures where the water is occluded by the pool geometry.

4.8.2 Foam Layer

In this pass the thickness of the *foam layer* is calculated (see Section 3.2.2). Along with the foam thickness also the thickness in a constant range behind the foam surface is calculated to be able to render a more realistic foam color by blending two different user-defined colors (see Section 3.2.3). Equivalent to the water thickness pass (4.8.1) additive blending, to accumulate the foam particles, and the hardware depth test, to ensure proper occlusion by the geometry of the main scene, are enabled and also the depth write functionality is disabled. The main differences between the water thickness pass and the foam thickness pass are that the geometry shader is used to orient the foam

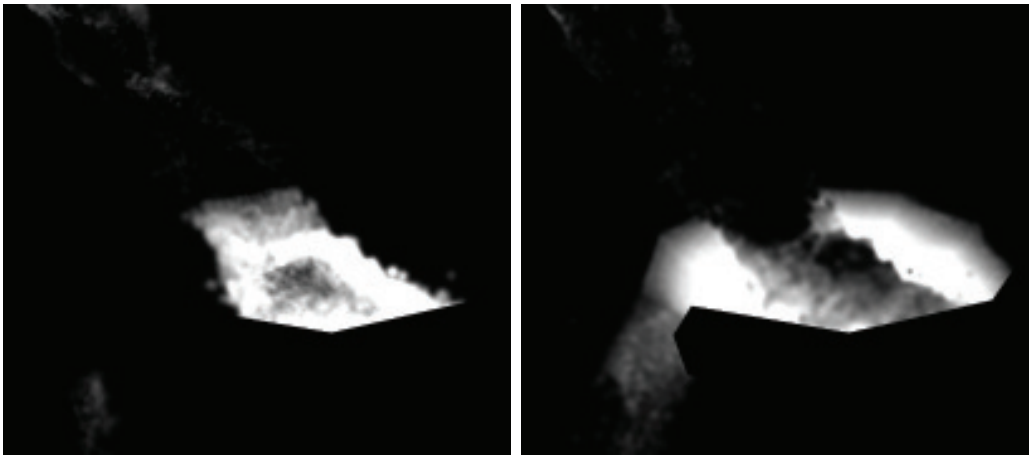

Fig. 4.14: Water thickness pass results; left: back water layer thickness; right: front water layer thickness.

particles toward their velocity direction (as ~~it~~ is done in the foam depth pass 4.6.2) and the splat kernel is multiplied by 3D Perlin noise.

Similar as for the water thickness vertex shader some additional calculations are carried out by the foam thickness vertex shader. The vertex shader also looks up the foam depth to separate the foam particles which are within a constant range behind the foam surface. This distance is passed to the shader as a parameter and defined by the user. As mentioned above the geometry shader is used to orient the foam particles toward their velocity direction ~~equivalent as~~ it is done in the foam depth pass (see Section 4.6.2). For this purpose the foam depth geometry shader as shown in Listing A.10 is used because this shader does all the calculations that are required.

The foam thickness calculation done by the fragment shader~~,~~ is similar to the calculation done for the water thickness. Additionally, the exponential falloff is multiplied with 3D Perlin noise. A different noise pattern for each particle is achieved by using the lifetime of a particle for the texture look-up. The output of the foam thickness fragment shader are two thickness values weighted by Perlin noise. Listing A.16 shows the additional calculations done by the foam thickness fragment shader.

Figure 4.15 illustrates the splat kernel calculated by the foam thickness fragment shader A.16. On the left side the noise-weighted splat kernel for a single particle is shown. Note that this pattern is only a snapshot because the pattern changes over time for each particle. The right side shows a close-up view including sub-particle detail.

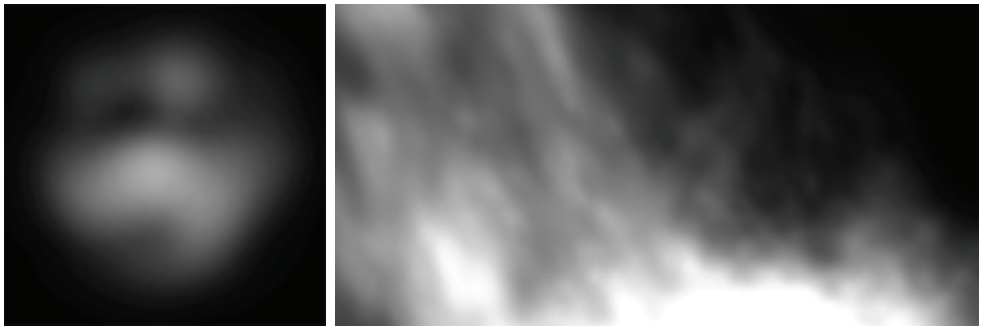

Fig. 4.15: Splat kernel used for the foam thickness; left: single particle; right: close-up of the foam layer thickness.

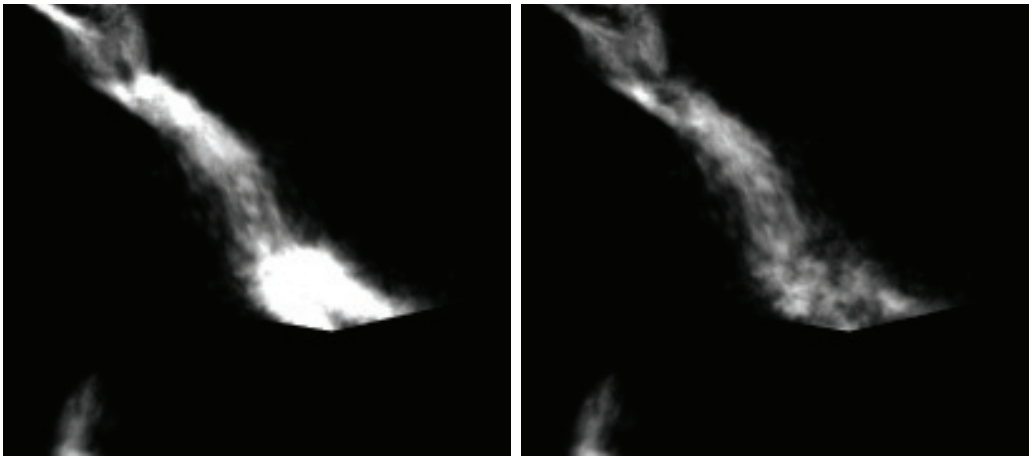

Fig. 4.16: Foam thickness pass results; left: foam layer thickness; right: foam thickness in a constant range behind the foam depth.

Figure 4.16 illustrates the results of the foam thickness pass. On the left side the thickness of the *foam layer* is shown and on the right side the thickness of the *frontmost foam layer* respectively. In comparison the *frontmost layer* thickness is at most as thick as the depth threshold *fdt*, hence it is easy to normalize the *frontmost layer* thickness for blending by dividing it by the depth threshold *fdt* (see Section 4.9). In contrast, the *foam layer* thickness does not have an upper bound.

4.9 Water Rendering including Foam

In the final rendering pass all intermediate results such as the background scene color, the filtered water depth and the layer thicknesses are used to compose the actual pixel color of the water including foam (see Figure 4.17). As described in Section 3.2.3 the actual pixel color is calculated by volumetric compositing along a viewing ray from back to front. The compositing pass also includes real-time refraction of the background scene behind the water, static reflections of the environment (looked up from an static cubic environment map) and specular highlights based on the water surface normal and the light position.

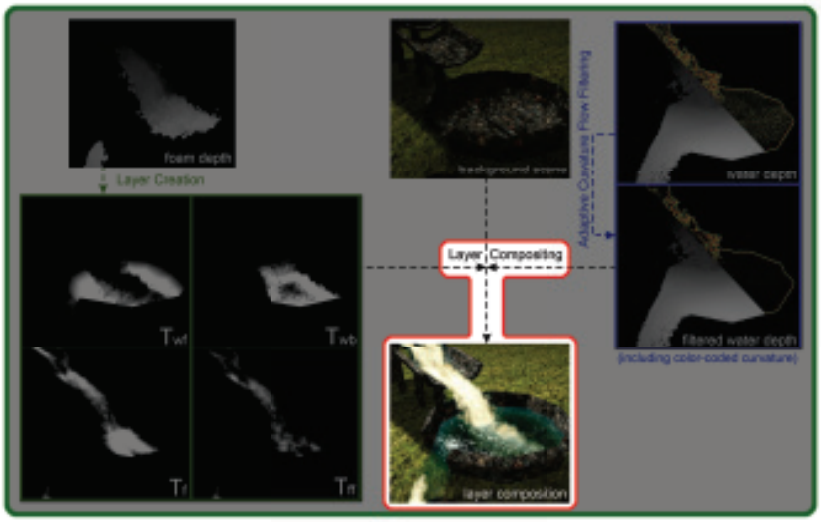

Fig. 4.17: Compositing based on the previously calculated intermediate results.

For the compositing pass a full-screen quad is sent down the rendering pipeline similar as done for the *adaptive curvature flow* filtering as described in Section 4.7. The calculations done in this pass are structured into different parts as follows:

- Description of helper functions used for calculating eye-space positions and interpolated normals.
- Description of the *shade* function (see Listing A.19 and A.20) which calculates the volumetric compositing as described in Section 3.2.3.
- Description of the compositing fragment shader (see Listing A.21) called by the main programme.

- Description on how to incorporate *shadow mapping* [39] into our compositing pass.
- Description of spray particle rendering.

4.9.1 Helper Functions

Listing A.17 shows helper functions to calculate an eye-space position. The *uvToEye* function takes texture coordinates in the range $[0, 1]$ and an eye-space depth value to calculate the eye-space position. The *getEyeSpacePos* function only takes a set of texture coordinates which are used to look-up the eye-space depth value from the water depth texture.

The *ipnormal2* function as shown in Listing A.18 calculates the partial derivatives of a normal vector based on a set of texture coordinates, a directional parameter *d1*, and the fractional part of the texture coordinates. For this purpose, the function look-ups three eye-space depth values from the water depth texture (eg.: left, center, middle pixel if *d1* is $float2(1.0f, 0.0f)$) and calculates as set of interpolated eye-space depth values. The output are the interpolated partial derivatives of the water surface at the given position, which are calculated by an approximation of the central difference because of performance reasons.

4.9.2 Back-to-Front Compositing

Our method composes the different layers back to front along the viewing rays. An overview about the intermediate calculations is illustrated in Figure 3.4. Note that the shader variables holding the intermediate calculation results correspond to the labels used in this figure. Listing A.19 and A.20 show the core calculations performed in the final rendering pass. The input parameters required to calculate the actual pixel color are the partial derivatives *_dxx* and *_dyy*, the eye-space position of the fragment, the texture coordinates of the fragment, and a shadow term which holds information about shadowing. A detailed description on how to calculate the shadowing is given later in this Section.

First of all, the partial derivatives in x- and y-direction respectively are used to calculate the normal vector of the water surface. Then the eye-space position and the light position (also in eye-space and passed as a shader parameter) are used to calculate the light direction. Based on the light direction and the view vector the specular highlight amount is calculated. Note that the shadow amount is multiplied by the specular highlight amount to suppress specular highlights in regions which are shadowed. Furthermore,

the specular highlight amount is forced to zero for fragments that do not have a valid eye-space depth value. This is required to avoid artefacts along the boundary of the water surface.

The next step in calculating the actual pixel color is to calculate reflections of the environment. In our current implementation, static reflections based on a cubic environment mapping and a Fresnel term which depends on the viewing angle onto the water surface are used. The reflection vector is simply calculated with the *reflect* function (provided by the Cg Toolkit [5]) based on the viewing direction and the surface normal. Note that the resulting reflection vector has to be transformed to world-space because this is the native space for environment maps. The Fresnel term is approximated as described in Chapter 7 of *The Cg Tutorial* [6] (which is also a good overview for *environment mapping techniques* and effects such as *chromatic dispersion*).

As described in Section 3.2.3 the compositing accounts for the attenuation caused by the calculated layers (see Section 4.8.1 and 4.8.2). Hence, the next step is to look-up the water thickness (representing the *back water layer* and the *front water layer*) and the foam thickness (representing the *foam layer* and the *frontmost foam layer*). Note that not all textures have the same dimension because the depth textures are scaled down during filtering (see Section 4.7). Thus, the texture coordinates have to be scaled to ensure proper texture look-ups. Based on the back water thickness and the front water thickness a refraction factor is calculated, and also the color of the background scene is retrieved from the scene texture which has been filled during the main scene rendering pass as described in Section 4.4. The effect of a warped background that can be seen through the water is achieved at this point by perturbing the texture coordinates by the surface normal vector before the look-up (see Equation 2.18).

The next step is to calculate the color of the *back water layer* and the *front water layer* which are based on a user-given water color. Thus the algorithm can produce a variety of water colors from an artistic point of view. Beside the water layer colors, the frontmost foam surface thickness is used to calculate the foam color as shown in Equation 3.9. Note that the frontmost foam surface thickness is normalized by the depth threshold as mentioned in Section 4.8.2. Additionally, the foam color is shadowed based on an ambient color and the shadow term.

Finally, the actual pixel color is calculated by blending the intermediate color values together (as shown in Equation 3.7) by using the attenuation weights, which are based on the layer thickness values. At this point also the the reflection color and the specular highlight are added.

4.9.3 Compositing Shader

The next part of this Section describes how the compositing fragment shader uses the *shade* function as described above, to calculate the entire water surface including foam. The primary task of the fragment shader is to separate the fluid surface into a border-region and an inner-region. This enables the fragment shader to avoid edge artefacts because of depth discontinuities. Next to edges colors are blended, and in the middle of the water the surface normals are interpolated for a smooth surface. The compositing fragment shader is illustrated in Listing A.21.

As mentioned above the fragment shader separates the fluid surface into a border-region and an inner-region. The separation is based on the second derivative for three points and the derivatives are compared against a user-defined threshold.

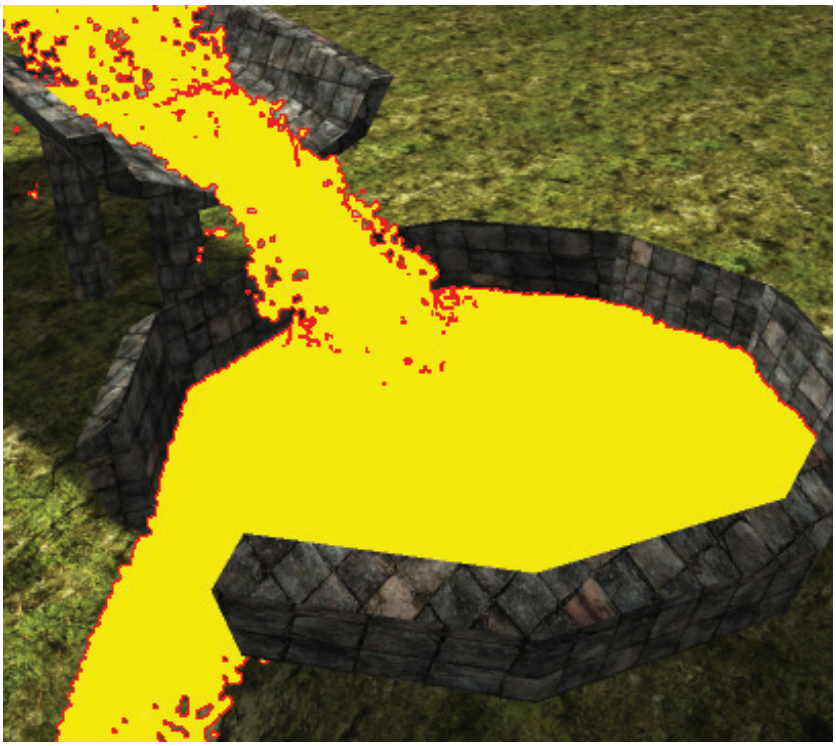

Fig. 4.18: Border- and inner-regions separated by the compositing fragment shader.

Figure 4.18 shows an example of a separation calculated by the compositing fragment shader. For the red colored regions the actual pixel color is calculated by interpolating the pixel colors of the 2×2 neighborhood around an fragment and interpolating them afterwards. Each color of the neighborhood is calculated based on the partial derivatives in x and y direction and

the back to front compositing described above. The process of interpolating samples is shown in Listing A.23 and A.24.

The yellow colored regions as illustrated in Figure 4.18 represent the inner parts of the fluid surface. The difference to border-regions is that for those regions the surface normals are interpolated instead of colors. The normals are represented by their partial derivatives in the horizontal direction and the vertical direction respectively. These partial derivatives are interpolated based on the fragment's location and used to calculate the actual pixel color for the fluid surface.

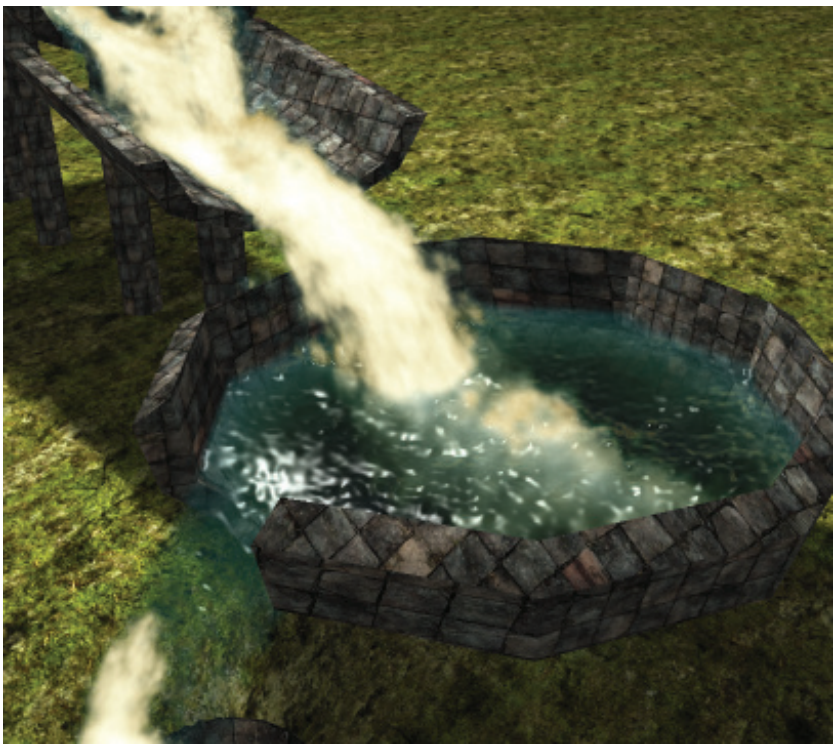

Fig. 4.19: Compositing pass results stored in the result texture.

Finally, the compositing fragment shader outputs the actual pixel color which includes specular highlights, reflection based on the *Fresnel Effect*, refraction of the background scene, a water surface which attenuates the background scene based on the thickness, and volumetric foam. Figure 4.19 illustrates the results of the compositing pass. Note the volumetric appearance achieved by doing back-to-front compositing.

4.9.4 Shadowing

As mentioned at the beginning of this section, our current implementation also handles shadows which are cast by the environment onto the water and foam. The first part of the shadow calculation accumulates the water depth values of the 3×3 neighborhood and calculates an average water depth value. Because there could be the case that the fluid has regions with no water at all (in case if both water thickness values are zero and especially near borders because the foam particles are orient towards their velocity direction) it has to be checked if the average water depth value is valid. In case of an invalid average water depth value the routine is repeated, but instead the water depth values the foam depth values are used for the accumulation. This results in an approximation of the fluid surface depth regardless of the layer setup which is used to calculate the shadowing (see Listing A.22).

Note that we do not go into detail ~~onto~~ the actual shadow calculation at this point because the fragment shader is structured in a way that any shadow algorithm based on *shadow mapping* [39] is suitable and it would exceed the scope of this thesis. Our current implementation uses *convolution shadow maps* proposed by Annen et al. [2].

4.9.5 Spray Particles

Spray particles which are indicated by a low density value normally do not form part of any surface. As mentioned by van der Laan et al. [37] it is desirable to exclude those particles from rendering by putting a threshold on the density obtained from the simulation. This is done in our implementation as described in Section 4.6.1 (and Listing A.8). Note that the culling of low density particles is carried out in all passes which render the SPH simulation data. The rendering of the spray particles is done immediately after the compositing pass. ~~Thereto~~ the result texture from the previous pass stays bound and the spray particles are combined with the compositing results by additive blending. The vertex shader used in this pass is very similar to those used for particle rendering in this ~~C~~chapter expect that all particles with a density above the user-defined threshold (instead of below) are culled. The geometry shader is also very similar to the one used during the foam depth pass (see Listing A.10) for example. ~~In which~~ the spray particles are orient toward their velocity direction and a user-defined spray color is forwarded to the fragment shader. The fragment shader simply outputs a user-defined spray color multiplied by an exponential ~~-based~~ falloff.

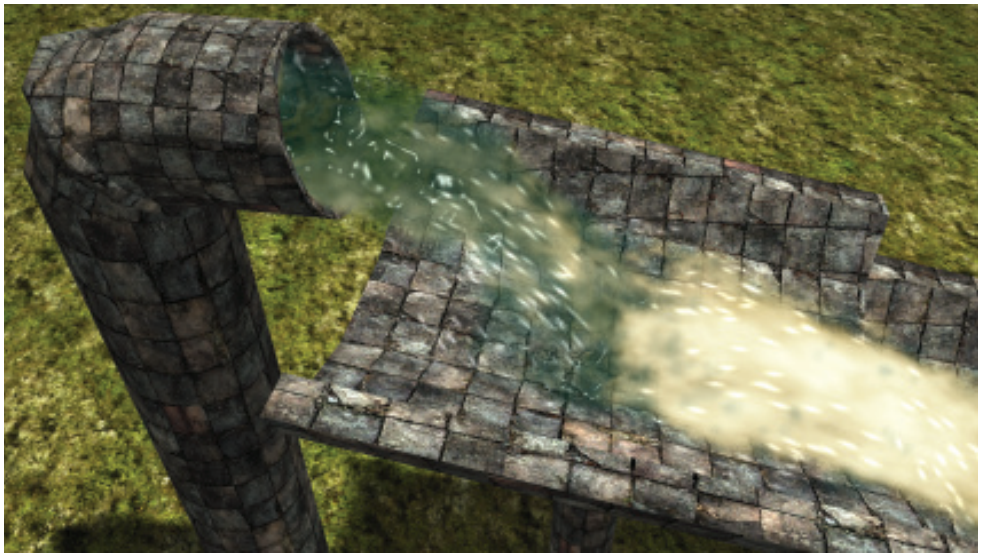

Fig. 4.20: Spray particles added after the compositing pass.

Figure 4.20 illustrates the results of the spray pass. Note that the spray particles are blended additively with the compositing results and therefore are much more visible on darker background colors.

Chapter 5

Results

This chapter presents results that can be achieved with our new method. First, our test setup is described, then an overview about our three scenes is given, and finally the chapter presents a detailed performance comparison.

We have used an Intel Q9450 CPU with a GeForce GTX 280 graphics card. The SPH simulation was done with NVIDIA PhysX. Particle counts range from 20k to 64k, depending on the scene. All images were taken at 1280×720 resolution. The curvature flow filtering step was done at half resolution. We use off-screen buffers to store our various intermediate results: 32 bit float for the water depth, 16 bit float for the foam depth, and 16 bit each for T_{wb}, T_{wf}, T_f and T_{ff}. This results in a total of 112 bit per pixel.

5.1 Scenes

We have tested our approach in three different scenes. The *Corridor* scene has many obstacles and therefore creates a turbulent water flow with a lot of foam and spray. The particle count used to flood this scene is around 64k and the particles are evenly distributed in the scene. Figure 5.1 illustrates the *Corridor* scene. The required iteration count for the viewpoint used for this Figure is between 27 and 52, which means that regions close to the viewpoint (as it is the case at the bottom of Figure 5.1) are filtered with 27 iterations by our *adaptive curvature flow* filtering and those farther away with up to 52 iterations.

The *Waterfall* scene is less turbulent, but due to its simplicity, artifacts are easily detected by visual inspection. Here, rendering of foam is essential for realistic results. In contrast to the *Corridor* scene, the particles are not evenly distributed in this scene, which means that most of the 64k particles used in this scene are located in the basin at the end of the ramp construction (see Figure 5.2). Another important geometric part of this scene is the pillar that holds the center ramp and splits the water flow into two streams. Due to the high compression that occurs at that position, the Weber number

Fig. 5.1: Corridor scene (27–52 iterations).

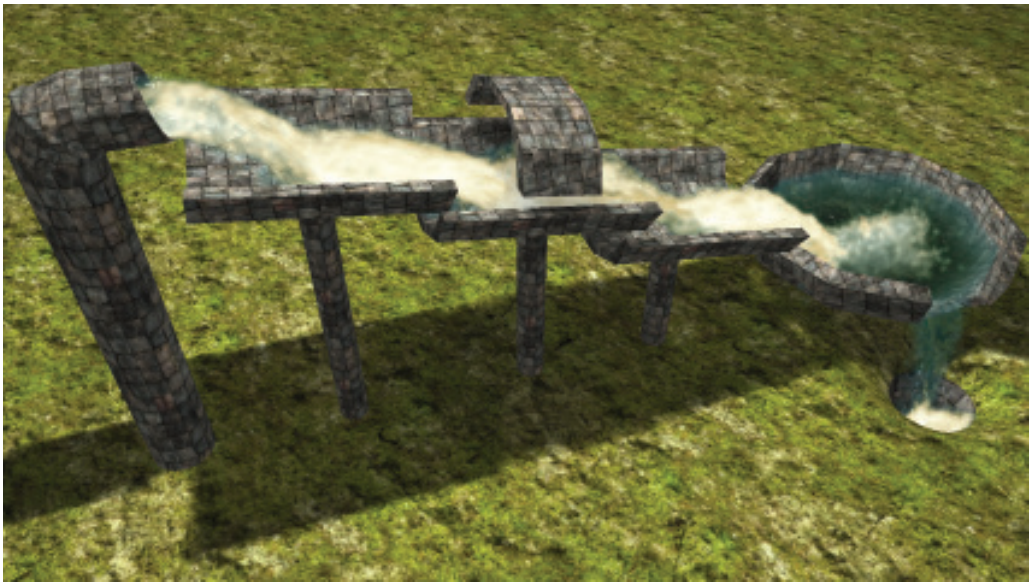

Fig. 5.2: Waterfall scene (15–20 iterations).

thresholding can be well observed. Figure 5.3 shows a comparison of the *Waterfall* scene using different values for the Weber number threshold ranging from 110 to 50 (see label contained in each image of Figure 5.3). Note that the lower the Weber number threshold, the easier foam is formed, which can be observed especially after the pillar.

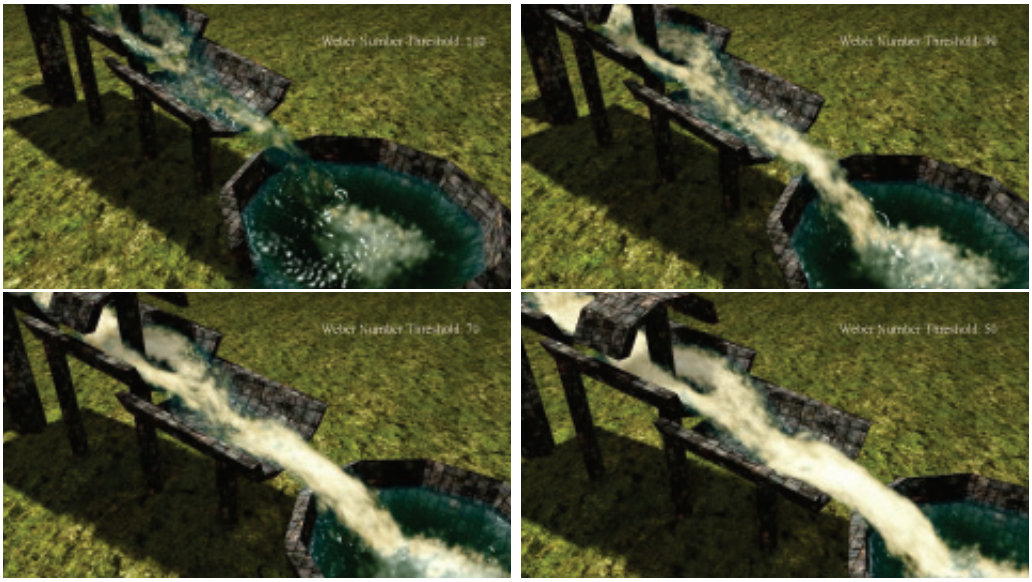

Fig. 5.3: Impact of Weber number thresholding.

Figure 5.4 shows a close-up view of the *Waterfall* scene. Here, one can observe that the water surface and the foam around the pillar are shadowed. Note that in case of the water surface, the specular highlights are suppressed and the foam itself is rendered with an ambient term.

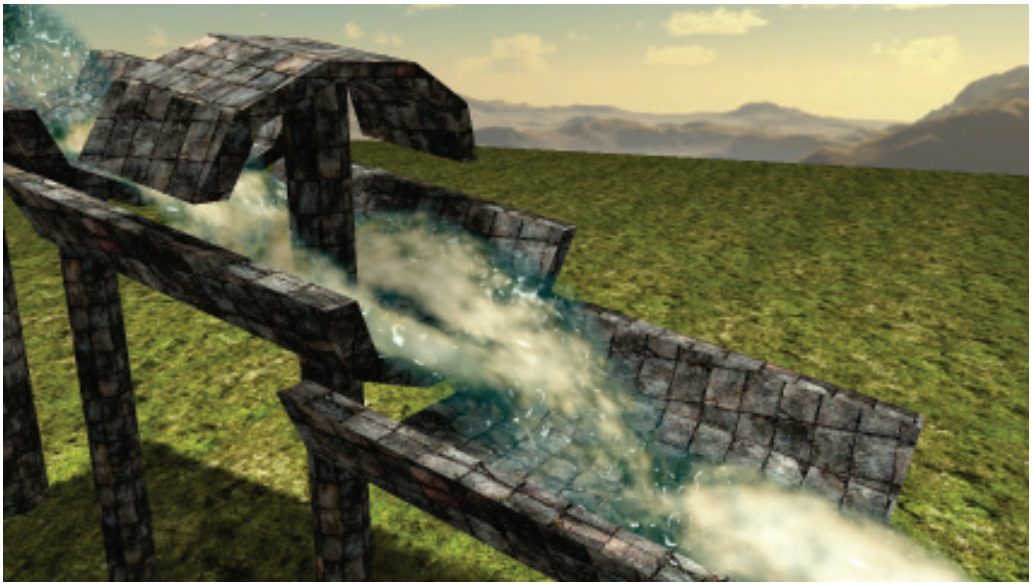

Fig. 5.4: Waterfall scene including shadowing (20–44 iterations).

The *Bamboo* scene consists of a bamboo construction and has dynamic elements that interact with the water as illustrated in Figure 5.5. The bamboo is slowly filled with water, till the water weighs it down and is emptied again. Although such a scene does not produce any foam in reality, it shows that our method can be used in scenes where the water interacts with dynamic elements. Figure 5.6 shows a close-up of the bamboo construction. Note the smooth transition between water and foam which can be observed when the construction is emptied again.

Fig. 5.5: Bamboo scene (22–40 iterations).

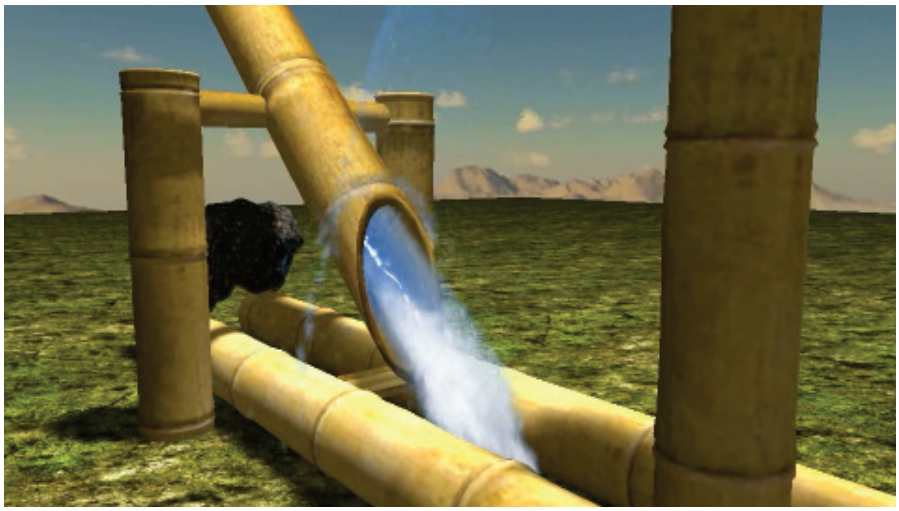

Fig. 5.6: Bamboo scene showing water to foam transition (32–44 iterations).

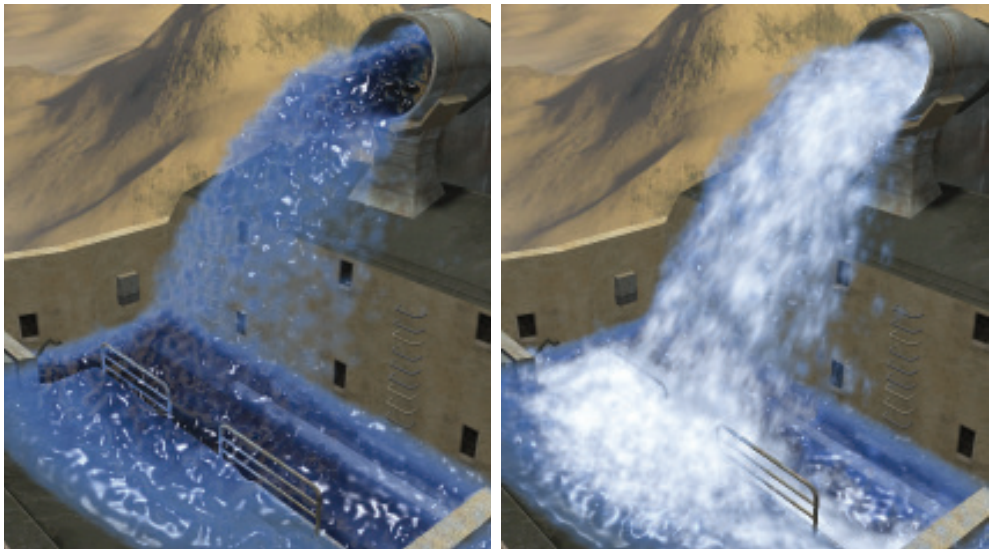

Fig. 5.7: Corridor scene without/with foam (26–50 iterations).

Figure 1.1 shows the benefit of our physically guided foam generation over simple noise-based foam [37]. Figure 5.7 demonstrates that foam is an important visual element when rendering fluids. Figure 5.8 compares a photograph of a real waterfall with our method. As one can also observe in the photograph, the foam is visible below the surface when a turbulent water stream immerges into resting water.

Fig. 5.8: Comparison between a photograph of a real waterfall (left) and our new method (right). The rectangle marks an area where foam occurs below the water surface.

5.2 Performance

Table 5.1 compares the *computational cost* of [37] with our method (SPH simulation time not included). The indicated running times are an average for a default camera movement. Our method has comparable performance with the benefit of improved image quality especially at near or far viewpoints. Even foam does not significantly increase running time for our method.

Scene	[37]	without foam	with foam
Waterfall	14 ms	14 ms	16 ms
Corridor	11 ms	12 ms	15 ms
Bamboo	12 ms	13 ms	17 ms

Tab. 5.1: Performance comparison between [37] (without foam) and our method with and without foam.

It takes on average 23.12% of the computation time to render the water and foam depth, 24.4% for the thickness passes, 27.43% for the *adaptive curvature flow* filtering and 25.05% for the composition (including update of data structures). This measurement represents the mean breakdown of 6,000 frames using different viewpoints in the waterfall scene. Figure 5.9 presents the computational cost of [37] and our method using the example of a camera zoom movement in the *Waterfall* scene. The slope on the right side of the curve is because of increased pixel overdraw which arises during rendering of close viewpoints.

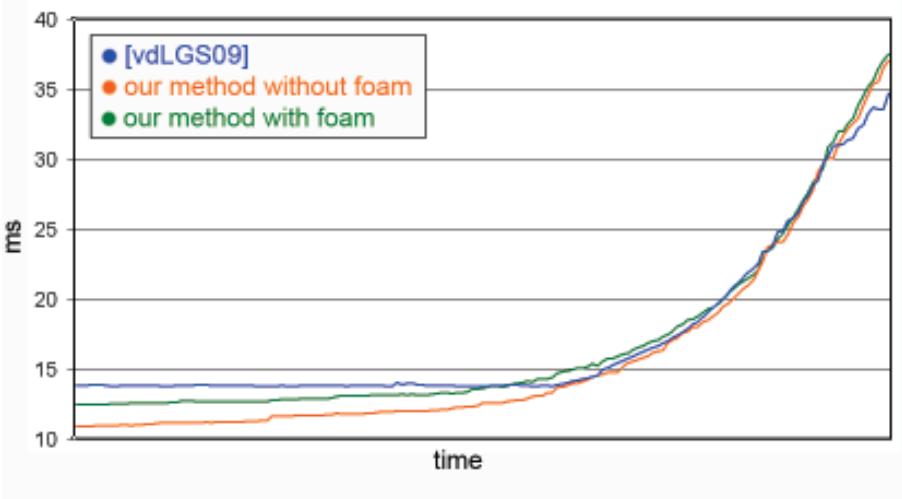

Fig. 5.9: Performance comparison of a camera zoom movement in the Waterfall scene.

Figure 5.10 presents the computational cost using a standard camera movement through the *Waterfall* scene. Note that our method without foam is faster for most of the viewpoints, only if the viewpoint is close to the basin our method has increased rendering times (see right spike in Figure 5.10).

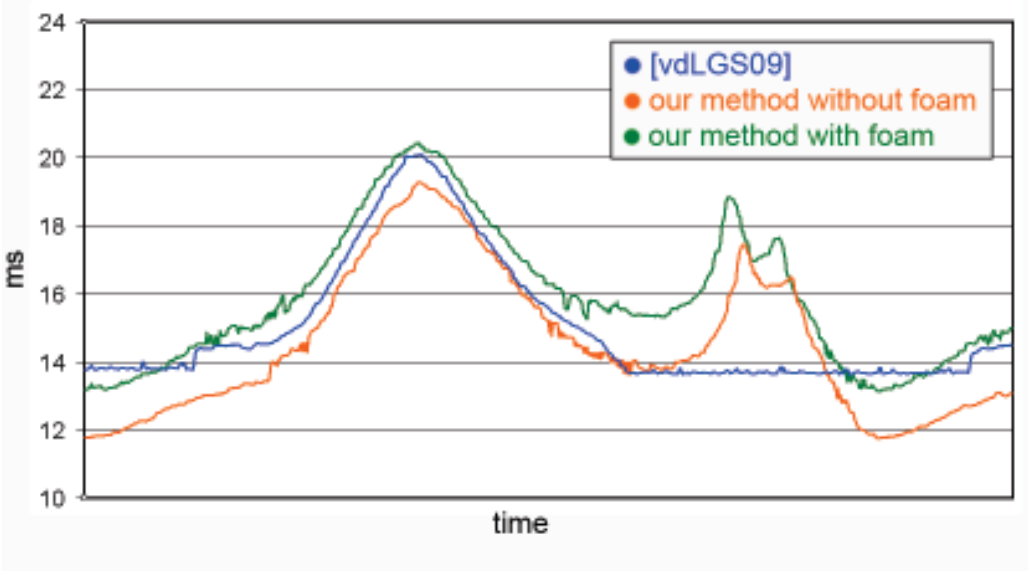

Fig. 5.10: Performance comparison of a camera movement in the Waterfall scene.

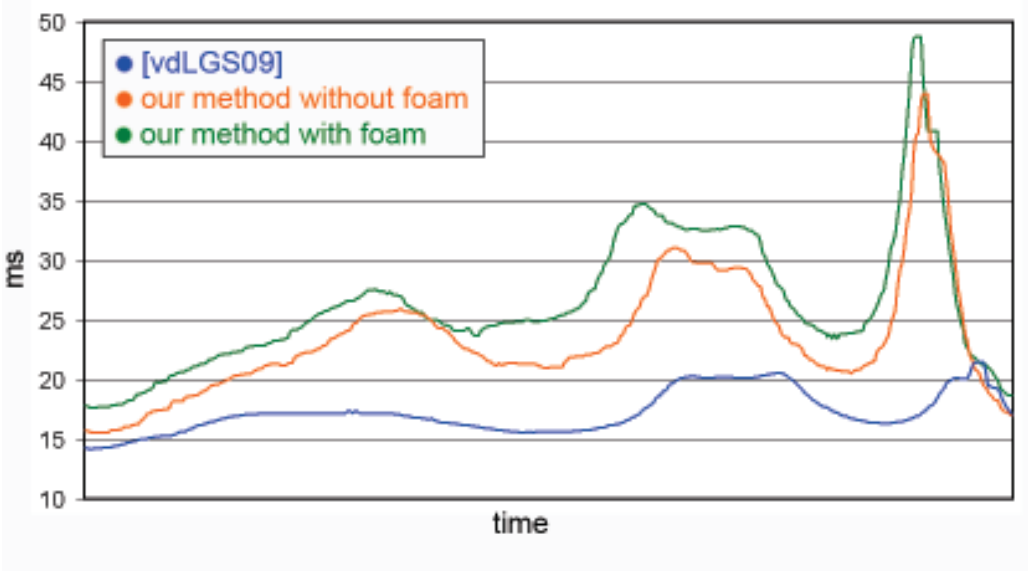

Fig. 5.11: Performance comparison of a camera movement in the Corridor scene.

Figure 5.11 presents the computational cost using a standard camera movement through the *Corridor* scene. Because the viewpoints of the used camera movement are close to the water surface in general and the particles are evenly distributed across the scene our method is slower.

Finally, Figure 5.12 presents the computational cost of the *Bamboo* scene. Note that the measurement for this scene includes the *computation times* of the physics simulation to account for the dynamic behavior and its influence.

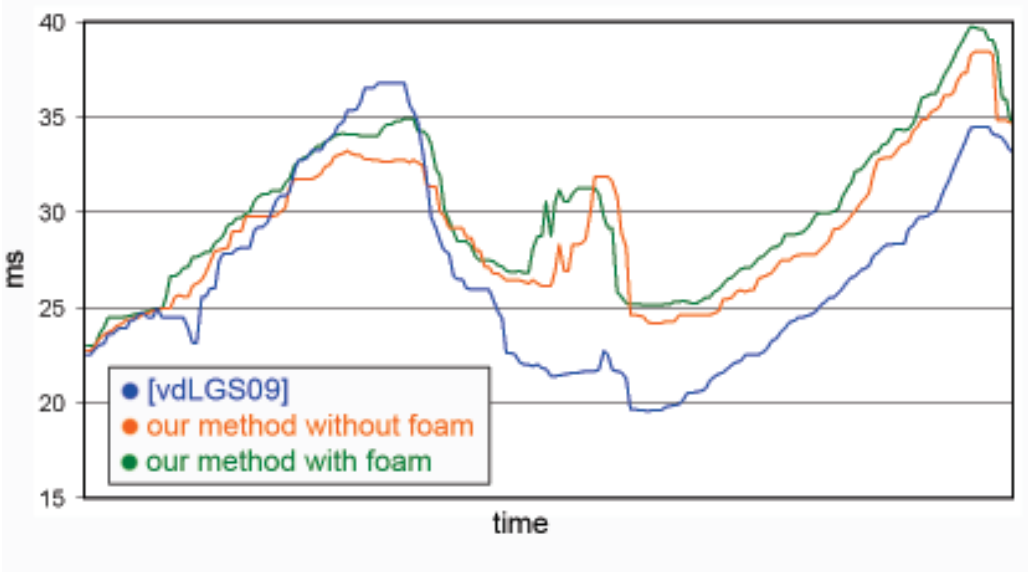

Fig. 5.12: Performance comparison of a camera movement in the Bamboo scene, this time including physics simulation time.

5.3 Limitations

As described in Section 3.2.2, our layer model separates the fluid volume into three layers (two *water layers* and one *foam layer* in between). This is a reasonable approximation for most situations that appear as shown above. However, there are situations which are not covered by our layer model as shown in Figure 5.13. Here, multiple overlapping *water layers* are separated by air and there is no *foam layer* at all. Because of this combination the entire water thickness is covered by the *front water layer*, which results in an inaccurate thickness calculation. To solve this artefact, it would be necessary to calculate a filtered water surface for every *water layer* that occurs, which is in fact too expensive for real-time rendering at the moment. Another possible solution would be to separate the fluid volume with an approach

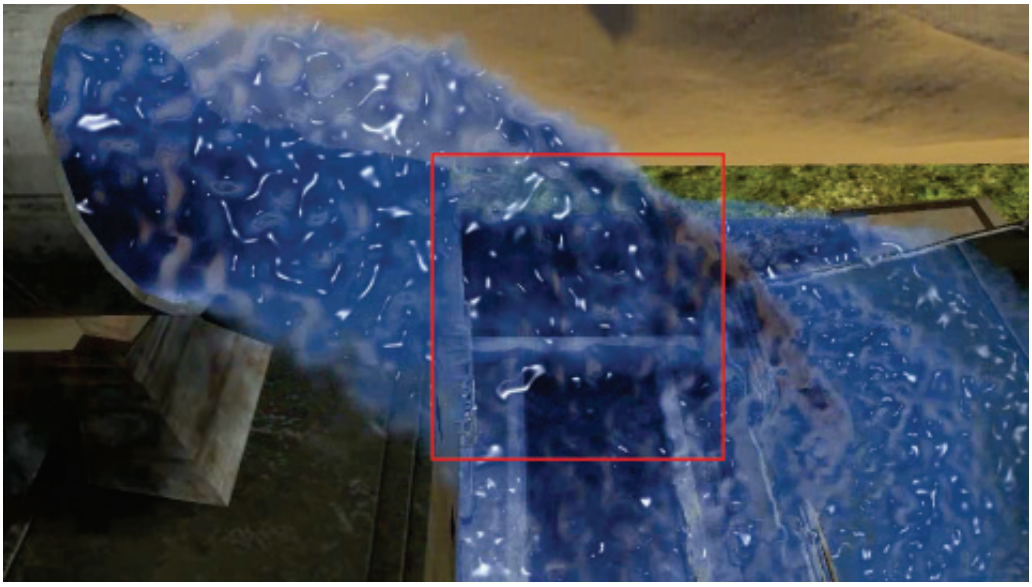

Fig. 5.13: Overlapping water layers separated by air. The rectangle marks an area where the thickness calculation is inaccurate.

based on voxelization as proposed by Eisemann et al. [13] for example. Their approach performs a voxelization along the viewing rays based on framebuffer blending. One problem of voxelization is that not all particles might be captured because of their relatively small size with respect to the scene size and the floating point precision not sufficient for this scale. However, using a *foam layer* as mentioned above helps to hide those thickness artefacts as one can observe in Figure 5.7 for example.

Chapter 6

Summary and future work

In this last chapter, a summary of the thesis is given, and ideas for future work are explored.

6.1 Conclusion

In this thesis, we presented a new physically based real-time method for rendering particle-based fluids that brings volumetric foam to the real-time domain. Foam is an important visual element in most situations where real-time fluids are used and there exists as yet no realistic real-time method for volumetric foam. Our algorithm is based on a previously implemented screen space fluid rendering approach which renders the surface of particle-based fluids and smoothes the surface to prevent the fluid from looking jelly-like.

The first contribution is an adaptive curvature flow smoothing method for SPH rendering which accounts for perspective. Instead of using a fixed iteration count which introduces view-dependent artefacts, our adaptive method varies the number of iterations depending on the view distance. Thus over- or under-smoothing as present in previous methods is avoided, and a consistent fluid surface is produced which is independent of the viewing distance.

Our second contribution is a fast physically guided foam rendering algorithm and a layer-based volumetric compositing algorithm. The former classifies particles as water or foam based on Weber number thresholding, which takes the density and velocity calculated by the physics engine into account. Our layer-based model is capable of treating cases such as foam only, water behind foam, and water in front of and behind foam. By partitioning the fluid into a foam layer an two water layers, one in front and one behind the foam, we can simulate foam inside water, as happens at the end of a waterfall. For each layer the amount of water respectively foam is determined by additively splatting every particle belonging to the volume. These thickness values are used by the volumetric compositing algorithm to calculate correct compositing and attenuation along a viewing ray back

to front. Furthermore, the actual pixel color also includes reflection of the environment and specular highlights.

Our approach provides more realistic fluid rendering at comparable cost to previous methods with the benefit of improved image quality especially at near or far viewpoints. Even foam does not significantly increase running time for our method. Our method is simple to implement and because of the analogy that the algorithm can be seen as a post-processing method it can be well integrated into existing rendering engines.

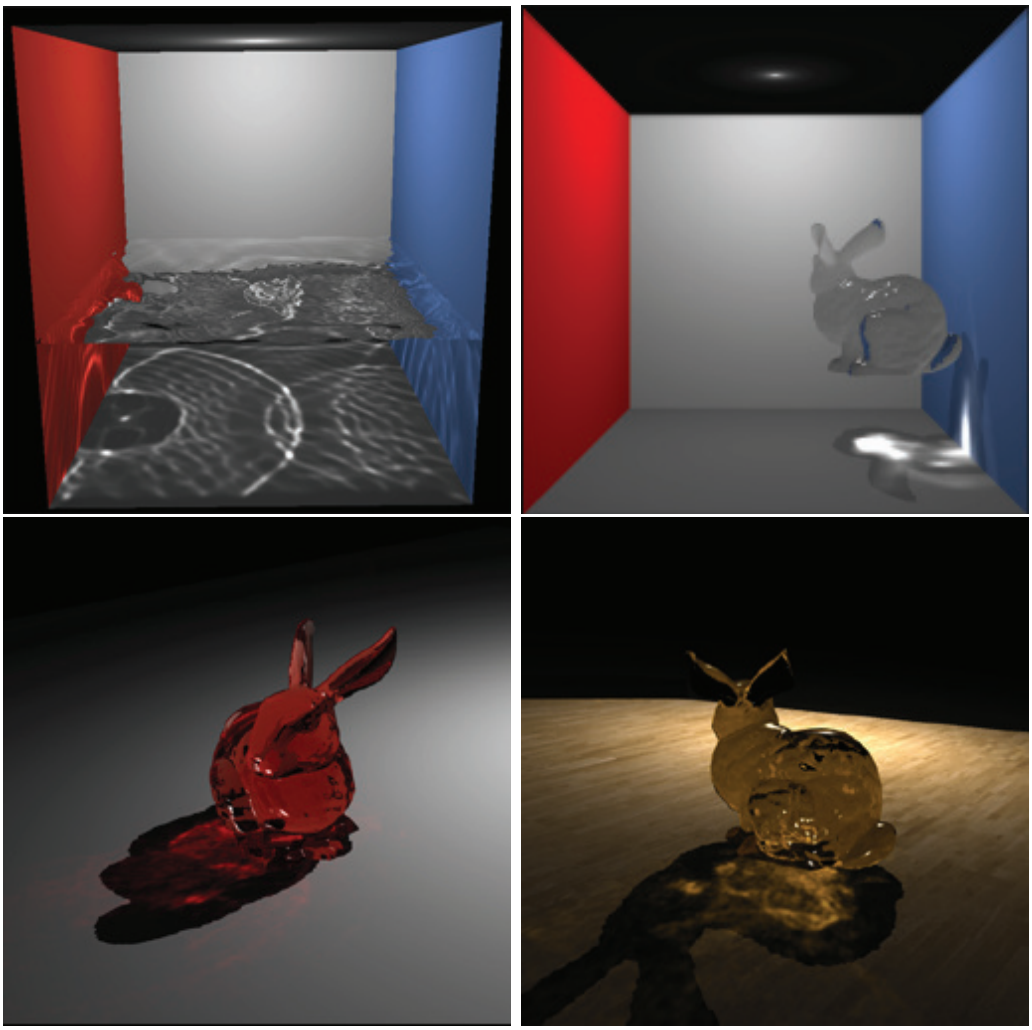

Fig. 6.1: Real-time rendering of caustics from reflective and refractive objects ([32]).

6.2 Future Work

In future work, we plan to account for situations that require more than 3 layers. This could be achieved by extending our layer approach to consider for multiple water and foam layers. The challenge with this approach is to avoid the cost that arises if multiple water surfaces have to be filtered with our *Adaptive Curvature Flow* filtering. Another solution to this problem could be an approach based on voxelization along the viewing rays as mentioned in Section 5.3.

In addition, we would like to investigate the achievable improvement of using caustics, which are an important visual element of realistic fluids. Recent publications like Shah et al. [32] have shown that caustics can be calculated efficiently in real time and with great visual quality as illustrated in Figure 6.1. Their approach avoids performing expensive geometric tests, such as ray-object intersections, and requires no pre-computation, which is important when dealing with fluids because the simulation is highly dynamic.

Attention has to be paid to the case if multiple water layers are overlapping as mentioned above. Also for caustics this has to be considered because each layer reflects and refracts the incident light depending on its surface normal vector.

Appendix A

Shader Code

A.1 General

```
unsigned int fluidBufferNum = fluid->GetFluidBufferNum();
Fluid::FluidParticle* fluidBuffer = fluid->GetFluidBuffer();

const float threshold = renderDescription.foamThreshold;
const float lifetime = renderDescription.foamLifetime;

unsigned int i;
for (i=0; i<fluidBufferNum; i++)
{
  const Fluid::FluidParticle& particle = fluidBuffer[i];
  float weberNumber = particle.velocity.Length() *
    particle.velocity.Length() *
    particle.density;

  MetaData* metaData = &fluidMetaData[particle.id];

  switch (fluidMetaData[particle.id].phase)
  {
  case FP_NONE: // do nothing as long as weber number is low

    if (weberNumber >= threshold)
    {
      metaData.foam = 0.0f;
      metaData.lifetime = Math::RandomFloat(0.25f, 0.5f);
      metaData.timer = metaData.lifetime;
      metaData.phase = FP_WATER_TO_FOAM;
    }
    break;

    ...
```

Listing A.1: Main loop of the foam simulation update.

```
    ...

    case FP_WATER_TO_FOAM: // fade-in foam

      metaData.timer -= deltaTime;

      if (metaData.timer <= 0.0f)
      {
        metaData.foam = 1.0f;
        metaData.lifetime = metaData.timer = 0.0f;
        metaData.phase = FP_FOAM;
      }
      else
        metaData.foam = 1.0f-metaData.timer/metaData.lifetime;

      break;

    case FP_FOAM: // as long as weber number is high keep foam

      if (weberNumber < threshold)
      {
        metaData.foam = 1.0f;
        metaData.lifetime = Math::Clamp(lifetime*Math::
            RandomFloat(0.5f, 1.5f), 0.0f, Math::MAX_FLOAT);
        metaData.timer = metaData.lifetime;
        metaData.phase = FP_FOAM_TO_WATER;
      }
      break;

    case FP_FOAM_TO_WATER: // fade back to water phase

      metaData.timer -= deltaTime;

      if (metaData.timer <= 0.0f)
      {
        metaData.foam = 0.0f;
        metaData.lifetime = 0.0f;
        metaData.timer = 0.0f;
        metaData.phase = FP_NONE;
      }
      else
        metaData.foam = metaData.timer/metaData.lifetime;

      break;
    }
  }
```

Listing A.2: Main loop of the foam simulation update. (cont.)

```
...

// run collision and dynamics for delta time since the last
// frame on the gpu (hardware)
// note that this function is non-blocking, thus the
// application can perform calculations in parallel
physicsScene->simulate(deltaTime);

// update meta data on the cpu (software)
screenSpaceCurvature->UpdateMetaData(deltaTime);

// wait for physics (fetchResults is blocking)
NxU32 error;
physicsScene->flushStream();
physicsScene->fetchResults(NX_RIGID_BODY_FINISHED, true, &
   error);

...
```

Listing A.3: Parallelism concept.

```
struct NxFluidPacket
{
  // AABB of all particles which are inside the same packet.
  NxBounds3 aabb;

  // index of first particle for a given packet.
  NxU32   firstParticleIndex;

  // number of particles inside the packet.
  NxU32   numParticles;

  // the packet's Identifier.
  NxU32   packetID;
};
```

Listing A.4: Data structure to represent a bounding box and its associated particle data of a fluid packet [29].

```
unsigned int numPackets = fluid->GetFluidNumPackets();
const NxFluidPacket* packets = fluid->GetFluidPackets();

unsigned int vpc = 0; // visible packet count

unsigned int i;

for (i=0; i<numPackets; i++)
{
  // test intersection with view frustum
  if (FrustumAABBIntersect(frustumPlanes,
    Vector3(packets[i].aabb.min),
    Vector3(packets[i].aabb.max)))
  {
    vboStartIndices[vpc] = packets[i].firstParticleIndex;
    vboIndexCount[vpc] = packets[i].numParticles;
    vpc++;
  }
}
```

Listing A.5: Frustum culling functionality.

```
...

// setup render target
screenSpaceCurvature->BeginRenderScene();
{
  // clear results of previous frame
  glClearColor (0.0, 0.0, 0.0, 1.0);
  glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

  ...

  // render scene (eg. static level geometry)
  RenderManager::Instance()->Render();

  ...
}
// finalize rendering
screenSpaceCurvature->EndRenderScene();

...
```

Listing A.6: Main scene rendering concept

```
void ScreenSpaceCurvature::RenderParticles(void)
{
  // Enable point sprite rendering
  glEnable(GL_POINT_SPRITE_ARB);

  // Enable point sprite size manipulation by the
  // vertex shader
  glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_ARB);

  // Render point sprites...
  glMultiDrawArrays(GL_POINTS, vboStartIndices, vboIndexCount,
      visiblePacketCount);

  glDisable(GL_POINT_SPRITE_ARB);
  glDisable(GL_VERTEX_PROGRAM_POINT_SIZE_ARB);
}
```

Listing A.7: Point sprite rendering function

A.2 Depth Passes

```
void FluidParticleVP(float4 position  : POSITION,
            float2 texCoord           : TEXCOORD0,
            float density             : TEXCOORD1,
            out float4 outPos         : POSITION,
            out float outPointSize    : PSIZE,
            out float2 outTexCoord    : TEXCOORD0,
            out float4 eyeSpace       : TEXCOORD1)
{
  outPos = mul(glstate.matrix.mvp, position);
  outTexCoord = texCoord;

  const float pointShrink = 0.5f;
  float scaledPointRadius = particleSize * (pointShrink +
     smoothstep(densityThreshold, densityThreshold*2.0f,
     density)*(1.0-pointShrink));

  // calculate window-space point size
  float4 eyeSpacePosition = mul(glstate.matrix.modelview[0],
     position);

  float dist = length(eyeSpacePosition.xyz);

  outPointSize = scaledPointRadius * (particleScale / dist);

  eyeSpace = float4(eyeSpacePosition.xyz, scaledPointRadius);

  // cull particles with density below threshold
  if (density < densityThreshold)
    outPos.w = -1.0;
}
```

Listing A.8: Water depth vertex shader

```
FragShaderOutput FluidDepthFP(float2 texCoord : TEXCOORD0,
                              float4 eyeSpace : TEXCOORD1)
{
  FragShaderOutput OUT;

  // calculate eye-space normal from texture coordinates
  float3 normal;
  normal.xy = texCoord.xy*float2(2.0f, -2.0f) +
    float2(-1.0f, 1.0f);

  float r2 = dot(normal.xy, normal.xy);

  // kill pixels outside circle
  if (r2 > 1.0f)
    discard;

  // look-up the square root for the given radius
  normal.z = tex1D(sqrtMap, 1.0f-r2).x;

  // position of this pixel on sphere in eye space
  float4 eyeSpacePos = float4(eyeSpace.xyz +
    normal*eyeSpace.w, 1.0f);

  float4 clipSpacePos = mul(glstate.matrix.projection,
     eyeSpacePos);

  // output eye-space depth
  OUT.color = float4(eyeSpacePos.z, 0.0f, 0.0f, 0.0f);
  OUT.depth = (clipSpacePos.z / clipSpacePos.w)*0.5f+0.5f;

  return OUT;
}
```

Listing A.9: Water depth fragment shader

```
POINT TRIANGLE_OUT void FluidFoamDepthGP(
      AttribArray<float4> position      : POSITION,
      AttribArray<float4> color         : COLOR0,
      AttribArray<float4> prevPosition  : TEXCOORD0,
      AttribArray<float> pointSize      : TEXCOORD1,
      AttribArray<float> density        : TEXCOORD2)
{
  if (density[0] > densityThreshold)
  {
    float3 pos = position[0].xyz;
    float3 prevPos = prevPosition[0].xyz;

    float3 motion = pos - prevPos;
    float3 direction = normalize(motion);
    float len = length(motion);

    float3 viewVector = normalize(-pos);
    float facing = dot(viewVector, direction);

    float3 x = direction*pointSize[0]*2.0f;
    float3 y = normalize(cross(direction, viewVector))*
       pointSize[0];

    const float motionThreshold = 0.5f;
    if ((len < motionThreshold) || (facing > 0.95f) ||
      (facing < -0.95f))
    {
      prevPos = pos;
      x = float3(pointSize[0], 0.0, 0.0);
      y = float3(0.0, -pointSize[0], 0.0);
    }

    float4 eyeSpace0 = float4(pos + x + y, 1.0f);
    ... // the same for eyeSpace1, eyeSpace2, and eyeSpace3

    float4 p0 = mul(glstate.matrix.projection, eyeSpace0);
    ... // the same for p1, p2, and p3

    emitVertex(p0, color[0], float4(0, 0, 0, 0), eyeSpace0);
    emitVertex(p1, color[0], float4(0, 1, 0, 0), eyeSpace1);
    emitVertex(p2, color[0], float4(1, 0, 0, 0), eyeSpace2);
    emitVertex(p3, color[0], float4(1, 1, 0, 0), eyeSpace3);
  }
}
```

Listing A.10: Foam depth geometry shader which outputs four vertices that are aligned toward the velocity direction

A.3 Adaptive Curvature Flow

```
bool breakLoop = false; bool doQuery = false;
const unsigned int queryInterval = 5;

for (int i=0; i<256 && !breakLoop; i++)
{
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, frameBuffer);
  glFramebufferTexture2DEXT(...);

  doQuery = (i%queryInterval == (queryInterval-1));
  if (doQuery)  // start occ. query
    glBeginQueryARB(GL_SAMPLES_PASSED_ARB, occQuery);

  ... // setup shader stuff (eg.: current iteration count)
  RenderQuad(scaleDownWidth, scaleDownHeight);

  if (doQuery)
    glEndQueryARB(GL_SAMPLES_PASSED_ARB);

  srcTex = 1 - srcTex;
  glFramebufferTexture2DEXT(...);

  ... // copy filtering result ('discard' and occ. query)

  if (doQuery)
  {
    unsigned int done = 0;
    while (!done) // wait for occ. query result
      glGetQueryObjectuivARB(occQuery,
          GL_QUERY_RESULT_AVAILABLE_ARB, &done);

    GLuint fragmentCount; // get occ. query result
    glGetQueryObjectuivARB(occQuery , GL_QUERY_RESULT_ARB, &
        fragmentCount);

    breakLoop = (fragmentCount == 0);
    doQuery = false;
  }

  srcTex = 1 - srcTex;
}
```

Listing A.11: Iteration loop performing the *adaptive curvature flow* filtering using the occlusion query functionality

```
FragShaderOutput FluidSmoothingFP(float2 tc : TEXCOORD0)
{
  FragShaderOutput OUT;

  // lookup depth value
  float coo = texRECT(depthMap, IN.tc.xy).x;

  // calculate necessary iteration count
  float it = (depthFilterParam / -coo) + 1.0f;

  // check if depth is valid and further filtering is
  // necessary
  if ((coo < -9999.0f) || (currentIteration > it))
  {
    discard;
  }
  else
  {
    ...

    // calculate curvature flow filtering
    // (see Listing A.13)

    ...
  }

  return OUT;
}
```

Listing A.12: *Adaptive curvature flow* filtering fragment shader (showing the view depended iteration count calculation)

```
...

// samples
float com = texRECT(depthMap, IN.tc.xy+float2( 0.0f,-1.0f)).x;
float cmo = texRECT(depthMap, IN.tc.xy+float2(-1.0f, 0.0f)).x;
float cpo = texRECT(depthMap, IN.tc.xy+float2( 1.0f, 0.0f)).x;
float cop = texRECT(depthMap, IN.tc.xy+float2( 0.0f, 1.0f)).x;

// derivatives
float dx = (0.5f*(cpo - cmo));
float dy = (0.5f*(cop - com));
float dxx = (cmo - 2.0f*coo + cpo);
float dyy = (com - 2.0f*coo + cop);
float dxy = 0.0f;

// directional derivatives
float dx1 = coo - cmo;      float dx2 = cpo - coo;
float dy1 = coo - com;      float dy2 = cop - coo;

if (abs(dx1) > depthFalloff || abs(dx2) > depthFalloff)
{
  dx = 0.0f;      dxx = 0.0f;
}

if (abs(dy1) > depthFalloff || abs(dy2) > depthFalloff)
{
  dy = 0.0f;      dyy = 0.0f;
}

float depth = min(coo, thresholdMin);

float a = camera.x; float b = camera.y; float c = camera.z;

// calculate curvature
float bl = a*a*dx*dx + b*b*dy*dy + c*c*depth*depth;
float H =  a*dxx*bl - a*dx*(a*a*dx*dxx + b*b*dy*dxy + c*c*
    depth*dx) +b*dyy*bl - b*dy*(a*a*dx*dxy + b*b*dy*dyy + c*c*
    depth*dy);
H /= pow(bl, 3.0f/2.0f);

// evolve under curvature
OUT.color = float4(coo + epsilon * H, 0.0f, 0.0f, 0.0f);

...
```

Listing A.13: *adaptive curvature flow* filtering fragment shader (showing the curvature flow filtering)

A.4 Layer Thicknesses

```
...

// cull particles with density below threshold and particles
// that belong to the foam layer
if ((density < densityThreshold) || (foam > 0.0f))
  outPos.w = -1.0;

// calculate texture coordinates
float3 postPerspPos = outPos.xyz / outPos.w;
float2 uv = (postPerspPos.xy*0.5f + 0.5f)*scaledDownSize;

// lookup foam depth and separate into layers
float depth = texRECT(foamDepthMap, uv).x;
outFoam = step(depth, eyeSpacePosition.z);

...
```

Listing A.14: Water thickness vertex shader (showing only additional calculations to the water depth vertex shader A.8)

```
FragShaderOutput FluidThicknessFP(float2 texCoord : TEXCOORD0,
                                  float foam      : TEXCOORD2)
{
  FragShaderOutput OUT;

  // calculate eye-space normal (see Listing A.9)
  ...

  // calculate thickness with exponential falloff
  float t = normal.z*particleSize*2.0f*exp(-r2*2.0f);

  // output back- and front water layer thickness
  OUT.color = float4(t*(1.0f-foam), t*foam, 0.0f, 0.0f);

  return OUT;
}
```

Listing A.15: Water thickness fragment shader (showing only different calculations to the water depth fragment shader A.9

```
FragShaderOutput FluidThicknessFP(float4 color    : COLOR0,
                                  float2 texCoord : TEXCOORD0,
                                  float foam      : TEXCOORD2)
{
  FragShaderOutput OUT;

  // calculate eye-space normal (see Listing A.9)
  ...

  // calculate thickness with exponential falloff
  float t = normal.z*particleSize*2.0f*exp(-r2*2.0f);

  // lookup 2D slice from perlin noise tex based on lifetime
  float n = tex3D(perlinMap, float3(texCoord.xy, color.x)).x;

  // output foam thickness and frontmost foam thickness
  OUT.color = float4(t*n, thickness*n*color.y, 0.0f, 0.0f);

  return OUT;
}
```

Listing A.16: Foam thickness fragment shader (showing only different calculations to the water depth fragment shader A.9

A.5 Compositing

```
// convert [0,1] uv coords and eye-space Z
// to eye-space position
float3 uvToEye(float2 uv, float eyeZ)
{
  uv = uv * invViewport;
  uv = uv * float2(-2.0f, -2.0f) - float2(-1.0f, -1.0f);
  return float3(uv * invFocalLength * eyeZ, eyeZ);
}
//  convert [0,1] uv coords to eye-space- position
float3 getEyeSpacePos(float2 texCoord)
{
  float eyeZ = texRECT(depthMap, texCoord).x;
  return uvToEye(texCoord, eyeZ);
}
```

Listing A.17: Eye-space position calculations

```
float4 ipnormal2(float2 tc, float2 d1, float f1)
{
  float ep1m  = texRECT(depthMap, tc - d1).x;
  float ep1c  = texRECT(depthMap, tc).x;
  float ep1p  = texRECT(depthMap, tc + d1).x;
  float2 dv1 = ipdepth(ep1m, ep1c, ep1p, f1);
  return float4(uvToEyeD(d1, ep1c + dv1.x, ep1c), dv1.y);
}

// interpolate depth values
float2 ipdepth(float ep1m, float ep1c, float ep1p, float f1)
{
  float2 d1 = float2(ep1c - ep1m, (ep1c + ep1m)*0.5f);
  float2 d2 = float2(ep1p - ep1c, (ep1p + ep1c)*0.5f);
  return lerp(d1, d2, f1);
}

// approx. to uvToEye(uv + uvdiff, eyeZ2) - uvToEye(uv, eyeZ)
float3 uvToEyeD(float2 uvdiff, float eyeZ2, float eyeZ)
{
  return float3(invCamera*uvdiff*eyeZ, eyeZ2 - eyeZ);
}
```

Listing A.18: Calculate interpolated partial derivatives for normal based on texture coordinates

```
// shade a pixel based on partial derivatives and eye position
float4 shade(float3 _ddx, float3 _ddy, float3 eyeSpacePos,
    float2 texCoord, float shadowTerm)
{
  float3 normal;
  normal = cross(_ddx.xyz, _ddy.xyz);
  normal = normalize(normal);

  float3 lightDir = normalize(lightPosEyeSpace.xyz -
     eyeSpacePos);

  float3 v = normalize(-eyeSpacePos);
  float3 h = normalize(lightDir + v);
  float specular = pow(max(0.0, dot(normal, h)),
     fluidShininess)*shadowTerm;

  // disable specular for pixels not part of the water surface
  specular *= step(-9999.0f, eyeSpacePos.z);

  float fresnelTerm = fresnelBias + fresnelScale*pow(1.0 +
    dot(v, normal), fresnelPower);

  // cubemap reflection
  float3 r = reflect(-v, normal);
  r = mul((float3x3)invView, r);
  float4 reflectColor = texCUBE(cubeMap, r);

    // lookup water thickness from the water thickness tex
  float4 thickness = texRECT(thicknessMap,
    texCoord.xy*lowResFactor);

  // lookup foam thickness from the foam thickness tex
  float4 thicknessFoam = texRECT(foamThicknessMap,
    texCoord.xy*lowResFactor);

  // refraction factor
  float refraction = (thickness.x+thickness.y)*
     thicknessRefraction;

  // scene color
  float4 c_scene= texRECT(sceneMap, (texCoord.xy*lowResFactor)
      + (normal.xy * refraction));

  ...
```

Listing A.19: Compositing calculation based on partial derivatives and eye position

```
    ...

    // water back and water front fluid color
    float4 c_fluid_wb = baseColor * exp(-thickness.x*
        falloffScale*colorFalloff);
    float4 c_fluid_wf = baseColor * exp(-thickness.y*
        falloffScale*colorFalloff);

    // foam color based on user-defined colors and blended
    // based on thickness in a constant range behind the
    // foam surface
    float3 c_foam = lerp(foamBackColor, foamFrontColor, pow(
        thicknessFoam.y*fdt, foamPowScale)).xyz;

    // apply shadow term to foam color
    float3 Ka = float3(0.6f, 0.6f, 0.6f);
    c_foam = Ka*c_foam*(1.0f-shadowTerm) + c_foam*shadowTerm;

    // attenuation factors
    float att_wb = saturate(c_wb.w);
    float att_foam = saturate(exp(-thicknessFoam.x*
        foamFalloffScale));
    float att_wf = saturate(c_wf.w);

    // composition
    float3 c_wb = lerp(c_fluid_wb.xyz, sceneColor.xyz, att_wb);
    float3 c_f  = lerp(c_foam.xyz, c_wb.xyz, att_foam);
    float3 c_wf = lerp(c_fluid_wf.xyz, c_f.xyz, att_wf);

    // calculate factor to suppress specular highlights if
    // foam is the frontmost visual element
    float spw = saturate(1.0f-att_wf + att_foam)*(1.0f-att_wf);

    // combine with fresnel and specular highlight
    float4 surfaceColor = float4(c_wf.xyz + (reflectColor.xyz*
        fresnelTerm + float4(fluidSpecularColor.xyz, 1.0f).xyz*
        specular)*spw, 1.0f);

    return c_surface;
}
```

Listing A.20: Compositing calculation based on partial derivatives and eye position (cont.)

```
FragShaderOutput FluidCompFP(float4 position  : POSITION,
                             float2 texCoord : TEXCOORD0)
{
  FragShaderOutput OUT;

  // center tex coords and fractional part of fragment
  float2 tc = floor(texCoord.xy) + 0.5f;
  float2 fr = frac(texCoord.xy);
  float4 fluidColor;

  // texture coordinate offsets for 3x3 neighborhood
  float2 texOff[9];
  texOff[0] = float2( 0, 0);  texOff[1] = float2(-1, 0);
  texOff[2] = float2( 1, 0);  texOff[3] = float2( 0,-1);
  texOff[4] = float2( 0, 1);  texOff[5] = float2( 1, 1);
  texOff[6] = float2( 1,-1);  texOff[7] = float2(-1, 1);
  texOff[8] = float2(-1,-1);

  // lookup depth values in 3x3 neighborhood
  float depth[9];
  for (int i=0; i<9; ++i)
    depth[i] = texRECT(depthMap, tc + texOff[i]).x;

  // calculate shadow term (see Listing A.22)
  float shadowTerm = 0.0f;
  ...

  // seperate fluid surface into border- and inner region
  if(abs(depth[1] - 2.0f*depth[0] + depth[2]) > depthThresh ||
     abs(depth[3] - 2.0f*depth[0] + depth[4]) > depthThresh ||
     abs(depth[7] - 2.0f*depth[4] + depth[5]) > depthThresh ||
     abs(depth[6] - 2.0f*depth[2] + depth[5]) > depthThresh)
  {
    // interpolate samples (see Listing A.23)
    ...
  }
  else
  {
    // interpolate normals (see Listing A.25)
    ...
  }

  OUT.color = float4(fluidColor.xyz, 1.0f);
  return OUT;
}
```

Listing A.21: Fragment shader calculating the actual pixel color

```
float accDepthValue = 0.0f;
float accDepthNorm = 0.0f;

// average depth values 3x3 neighborhood
for (int i=0; i<9; ++i)
{
  float stepValue = step(-9999.0f, depthValues[i]);
  accDepthValue += depthValues[i]*stepValue;
  accDepthNorm += stepValue;
}
accDepthValue /= accDepthNorm;

// if accDepthValue is invalid use foam depth instead
if (!step(-9999.0f, accDepthValue))
{
  accDepthValue = 0.0f;
  accDepthNorm = 0.0f;
  for (int i=0; i<9; ++i)
  {
    float foamDepth = texRECT(foamDepthMap, tc + texOff[i]).x;
    float stepValue = step(-9999.0f, foamDepth);
    accDepthValue += foamDepth*stepValue;
    accDepthNorm += stepValue;
  }
  accDepthValue /= accDepthNorm;
}
float shadowTerm = Shadow(uvToEye(texCoord, accDepthValue));
```

Listing A.22: Shadow term calculation

```
// calculate color for samples in 2x2 neighborhood
float4 s1 = sample(tc, texCoord, shadowTerm);
float4 s2 = sample(tc + texOff[2], texCoord, shadowTerm);
float4 s3 = sample(tc + texOff[4], texCoord, shadowTerm);
float4 s4 = sample(tc + texOff[5], texCoord, shadowTerm);

// interpolate samples based on fractional part of tex coords
fluidColor = lerp(lerp(s1, s2, fr.x),lerp(s3, s4, fr.x),fr.y);
```

Listing A.23: Sample interpolation for border regions

```
float4 sample(float2 tc, float2 texCoord, float st)
{
  float3 eye = getEyeSpacePos(tc);

  // calculate partial derivatives in x-direction
  float3 ddx = getEyeSpacePos(tc + float2(1.0f, 0.0f)) - eye;
  float3 ddx2= eye - getEyeSpacePos(tc + float2(-1.0f, 0.0f));
  if(abs(ddx2.z) < abs(ddx.z)) ddx = ddx2;

  // calculate partial derivatives in y-direction (see above)
  ...

  // return sample color
  return shade(ddx, ddy, eyeSpacePos, texCoord, st);
}
```

Listing A.24: Calculate fluid color for given texture coordinates

```
// calculate partial derivatives
float4 ddx  = ipnormal2(tc, float2(1.0f, 0.0f), fr.x);
float4 ddx2 = ipnormal2(tc + float2(0.0f, 1.0f),
  float2(1.0f, 0.0f), fr.x);
float4 ddy  = ipnormal2(tc, float2(0.0f, 1.0f), fr.y);
float4 ddy2 = ipnormal2(tc + float2(1.0f, 0.0f),
  float2(0.0f, 1.0f), fr.y);

// interpolate partial derivatives based on fractional part
ddx = lerp(ddx, ddx2, fr.y);
ddy = lerp(ddy, ddy2, fr.x);

// calculate eye-space position and actual color
float3 eyePos = uvToEye(texCoord, (ddx.w + ddy.w)*0.5f);
fluidColor = shade(ddx.xyz, ddy.xyz, eyePos, texCoord,
   shadowTerm);
```

Listing A.25: Normal interpolation for inner regions

List of Figures

List of Tables

List of Listings

Bibliography

[1] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering 2nd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[2] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. In *Rendering Techniques 2007: Eurographics Symposium on Rendering*, pages 51–60, Grenoble, France, 2007. Eurographics.

[3] Florian Bagar, Daniel Scherzer, and Michael Wimmer. A layered particle-based fluid model for real-time rendering of water. In *Computer Graphics Forum (Proceedings EGSR 2010)*, volume 29, June 2010.

[4] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Symposium on Point-Based Graphics 2005*, pages 17–24, June 2005.

[5] Cg Toolkit. `http://developer.nvidia.com/object/cg_toolkit.html`.

[6] The Cg Tutorial: Environment Mapping Techniques. `http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter07.html`.

[7] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[8] Paul W. Cleary, Soon Hyoung Pyo, Mahesh Prakash, and Bon Ki Koo. Bubbling and frothing liquids. *ACM Trans. Graph.*, 26(3):97, 2007.

[9] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: a new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 61–76, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[10] P. Dirac. *Principles of quantum mechanics*. Oxford at the Clarendon Press, 1958.

[11] Microsoft DirectX. `http://msdn.microsoft.com/directX`.

[12] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 19–28, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[13] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *GI '08: Proceedings of graphics interface 2008*, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[14] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics Based Animation*. Charles River Media, 2005.

[15] Framebuffer Object. `http://www.opengl.org/registry/specs/ARB/framebuffer_object.txt`.

[16] High Level Shading Language. `http://msdn.microsoft.com/en-us/library/bb509561%28v=VS.85%29.aspx`.

[17] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, Ltd, 2001.

[18] Daniel D. Joseph. Questions in fluid mechanics: Understanding foams and foaming. *Journal of Fluids Engineering*, 119:497–498, 1997.

[19] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[20] Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[21] Ravi Malladi and James A. Sethian. Level set methods for curvature flow, image enhancement, and shape recovery in medical images. pages 329–ff., 1997.

[22] Viorel Mihalef, Dimitris N. Metaxas, and Mark Sussman. Simulation of two-phase flow with sub-scale droplet and bubble effects. *Comput. Graph. Forum*, 28(2):229–238, 2009.

[23] J. J. Monaghan. Smoothed particle hydrodynamics. *araa*, 30:543–574, 1992.

[24] Matthias Müller, Simon Schirm, and Stephan Duthaler. Screen space meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 9–15, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[25] NVIDIA Corporation. `http://www.nvidia.com/page/home.html`.

[26] OpenGL. `http://www.opengl.org/`.

[27] The OpenGL Shading Language. `http://www.opengl.org/registry/doc/GLSLangSpec.4.00.7.pdf`.

[28] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[29] NVIDIA PhysX. `http://developer.nvidia.com/object/physx.html`.

[30] Creating A Post-Processing Framework. `http://www.gamasutra.com/view/feature/1812/creating_a_postprocessing_.php?page=1`.

[31] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross T. Whitaker. Particle-based simulation of fluids, 2003.

[32] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13:272–280, 2007.

[33] William A Sirignano. Fluid dynamics and transport of droplets and sprays, 1999. Incluye referencias bibliográficas (p. 287-307) e índice (p. 309-311).

[34] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[35] Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki. Realistic animation of fluid with splash and foam. *Comput. Graph. Forum*, 22(3):391–400, 2003.

[36] N. Thürey, F. Sadlo, S. Schirm, M. Müller-Fischer, and M. Gross. Real-time simulations of bubbles and foam within a shallow water framework. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 191–198, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[37] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98, New York, NY, USA, 2009. ACM.

[38] Vertex Buffer Object. `http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt`.

[39] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[40] T. Yabe. Universal solver cip for solid, liquid and gas. *Computational Fluid Dynamics Review 1997 Ed. By M. M. Hafez and K. Oshima*, 1997.

Acknowledgements

> "All we have to decide is what to do with the time that is given to us."
>
> – Gandalf the Grey, "The Lord of the Rings: The Fellowship of the Ring"

First, I want to thank Daniel Scherzer for making this diploma thesis possible and helping me to publish our new approach at the "21st Eurographics Symposium on Rendering" which took place from June 28th 2010 – June 30th 2010 in Saarbrücken.

Secondly, thanks to Michael Wimmer for giving me great support during the paper submission and helping to improve the quality of our paper.

Thanks to all my co-workers at Spriong for giving me the possibility to learn so much from them during the last four years. Extra acknowledgment to Raimund Schumacher for his great feedback which helped to enhance the quality of my work and to Johannes Graf which helped me with his artistic skills.

Last but not least, I would like to thank all my friends and my family for all their great support over the past years.