Leopold-Franzens-Universität Innsbruck

Institute of Computer Science
Interactive Graphics and Simulation Group

Bachelor Thesis

Real-Time Screen Space Fluid Rendering with Scene Reflections

Andreas Moritz
andreas.moritz@student.uibk.ac.at

advised by
Prof. Dr. Matthias Harders

Innsbruck, November 14, 2016

Abstract

In this thesis we explore the real-time capability of rendering SPH simulated fluid with light attenuation approximation. We extend upon existing approaches by adding the capability for reflecting scene geometry with screen space ray tracing. Using binary refinement, we are able to trace reflections across high distances while keeping the performance impact low and still providing good results.

Keywords SPH, Screen Space Rendering, Fluid, Reflection, Real-Time.

Contents

List of Figures

List of Tables

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signature: _____ Date: _____

Chapter 1

Introduction

Simulation of fluid is used in a wide area of applications such as medical simulations for training, special effects for movies, games, flood simulations, petroleum simulations and many more. Most simulation models can generally be broken down into two categories. The Eulerian approach defines a fixed grid in space. Eulerian methods then observe the change of quantities (such as pressure) at the voxels as fluid passes through. On the contrary in Lagrangian methods the fluid is modeled by a loose set of particles, each carrying information about its own quantities. This has the advantage that we are not bound to a finite space for simulating fluid such as the Eulerian methods. Furthermore particles can intrinsically collide with an existing geometry in a physics engine, like any other object. These properties make Lagrangian methods preferable to Eulerian ones in dynamic and interactive applications in computer graphics, although hybrid solvers do exist that try to take the best from both models [18, 1, 19]. For a more in-depth explanation of the various fluid simulation models we refer to the extensive book of Robert Bridson [8].

SPH Smoothed Particle Hydrodynamics is a widely used Lagrangian fluid solver. Originating in the field of astrophysics [10], Desbrun and Gascuel [9] introduced the SPH formalism into the field of computer graphics. Mueller et al. [16] extended SPH for use in interactive applications. Since then it has been extensively researched and is nowadays used for state-of-the art fluid animations, including the simulation of foam, air bubbles and other features. Simulation in complex scenes (e.g. for movies and games) can consist of a large number of particles (up to a million and more). However, the loose set of particles that represent the fluid do not form an implicit surface that can be used for rendering. The main challenge hereby lies in achieving smooth surfaces without eliminating surface details such as capillary waves. To date numerous techniques have been presented to extract such surfaces, for offline as well as real-time applications [13].

In this paper we will implement a real-time rendering method for SPH fluids, based on the proposed screen space methods of van der Laan et al. [22] and Green [11]. When we speak of real-time we mean effective rendering times of <16.666 ms per frame for the fluid, preferably less to account for rendering of the scene geometry and other application specific tasks on the GPU. The SPH simulations for this thesis are created with RealFlow 2015, an "industry-standard, out-of-the-box fluid simulation software" [21]. However, our methods are not restricted to RealFlow and can be applied to any arbitrary SPH simulation.

In Chapter 2 we will explain some related methods for surface extraction and rendering of SPH fluids. As Ihmsen et al. [13] provide an extensive recent state-of-the art survey about

SPH simulation as well as rendering, we will only briefly explain the methods that are already covered by their survey and refer to it for further details.

In Chapter 3 we will provide a quick overview about our method and follow with a more in-depth explanation.

In Chapter 4 we will discuss the performance of our test scenes as well as the general memory requirements.

Chapter 2

Related Work

2.1 Mesh Triangulation with Scalar Fields

The Marching Cubes method [14] is one of the most utilized techniques for surface reconstruction due to its simplicity. It triangulates the surface of a point cloud by creating a voxel grid around the specified data of interest. In essence, for each voxel's corner vertices we assign a scalar value in relation to the surface value. These scalars at the vertices determine whether the vertex is inside/on or outside of the surface. Using the information of this scalar field allows for the creation of various triangle configurations, making up the polygonized surface. The surface quality of Marching Cubes depends highly on the resolution of the grid and the chosen method for the scalar field computation [3]. Producing high quality surfaces requires high resolution grids, resulting in large memory footprints and slow computation times. Zhou et al. [25] provide a parallelized algorithm that runs entirely on the GPU and is based on octree data structure for increased performance. Akinci et al. [4] use a different method of subdividing the uniform grid into a three level grid system depending on the surface curvature. This allows for preserving surface details during the triangulation while reducing memory consumption and computation time.

While providing high quality results, these methods still require substantial computation time and memory, which makes them not suitable for real-time applications without sacrificing much of their quality.

2.2 Explicit Meshes

Because the fluid particles are changing every simulation step, a new mesh for the fluid surface has to be constructed every frame. Explicit meshes circumvent this restriction. Yu et al. [24] create an initial surface. The vertices of this surface are changed by nearby particle velocities and thus carry the mesh forward in time.

2.3 Screen Space Methods

The aforementioned methods all reconstruct the fluid surface in 3D world space. Reconstructed fluid meshes can be potentially huge, containing more than a million triangles and requiring multiple GB of memory [4]. What we are generally interested in is the front-most surface that can be seen from the camera. By operating mainly in 2D, screen space methods are much more efficient and can be utilized for real-time applications. Usually screen space methods

start to render the particles into a depth map as spheres and apply some smoothing to reduce the resulting bumps. Mueller et al. [17] smooth the depth map by using a binomial filter and proceed to create a two dimensional triangle mesh by utilizing a Marching Square algorithm, a 2D version of the Marching Cubes. The resulting mesh is then transformed back into world space and rendered as a normal 3D mesh. Our method is based on [22, 11], where the image buffers are directly rendered to the screen without the creation of an intermediate mesh, by reconstructing the normals and view space position from the smoothed depth buffer. Bagar et al. [6] introduce a physically foam based rendering method for water. By separating the water into a front and back layer, foam can appear underneath the surface, causing a volumetric appearance.

2.4 Volumetric Methods

For rendering shadows and advanced lighting, refractions and reflections, we require volumetric information about the fluid to approximate scattering and other fluid properties. Only considering the front-most surface is not good enough for this. Van der Laan et al. and Green [22, 11] generate a volumetric representation by blending the particles with a thickness kernel. This can be used for approximating light attenuation through the fluid. Zirr and Dachsbacher [26] present a promising screen space voxelization technique that generates a true volumetric view dependent representation of the fluid in a reasonable amount of time.

Chapter 3

Methods

In this chapter we present and describe the individual rendering stages of our method. Our rendering is based on an SPH simulation. We assume that the simulation has already been carried out and provides us with the particles as input. The background scene is rendered separately to a color and depth texture and composited with the fluid at the end.

For the following sections we assume a right handed coordinate system. This means that the negative z-axis points from the camera towards the screen. We use a column-major perspective projection matrix $\mathbf{M}$ of the form:

$$\mathbf{M} = \begin{pmatrix} f_x & 0 & 0 & 0 \\ 0 & f_y & 0 & 0 \\ 0 & 0 & \frac{z_f+z_n}{z_n-z_f} & \frac{2\cdot z_f\cdot z_n}{z_n-z_f} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \tag{3.1}$$

where $f_x = \frac{\cot(fov\cdot 0.5)}{aspect}$ and $f_y = \cot(fov \cdot 0.5)$ are the focal lengths in the x and y dimension, $aspect$ is the aspect ratio of the viewport and fov is the field of view in the y dimension. z_n is the distance from the camera to the near plane and z_f the distance from the camera to the far plane.

3.1 Overview

Figure 3.1 shows a high level overview of our rendering stages:

1. **Particle Splatting:** The particles are rendered as spheres. The view space depth is used for a depth map. To approximate the fluid volume we are blending the particles to a thickness map, usually rendered at a lower resolution.

2. **Smoothing:** The depth map is smoothed, either with a separated bilateral filter or curvature flow, to reduce the bumpy appearance of the spheres. The thickness map is smoothed with a bilateral filter to provide a smoother thickness distribution.

3. **Shading:** We recreate the view space surface normals and view space positions from the smoothed depth map. The fluid color is a composition of screen space reflections, light attenuation, approximate refractions and specular highlights. The stages can all be done in one render-pass, for better clarity we separated them in Figure 3.1.

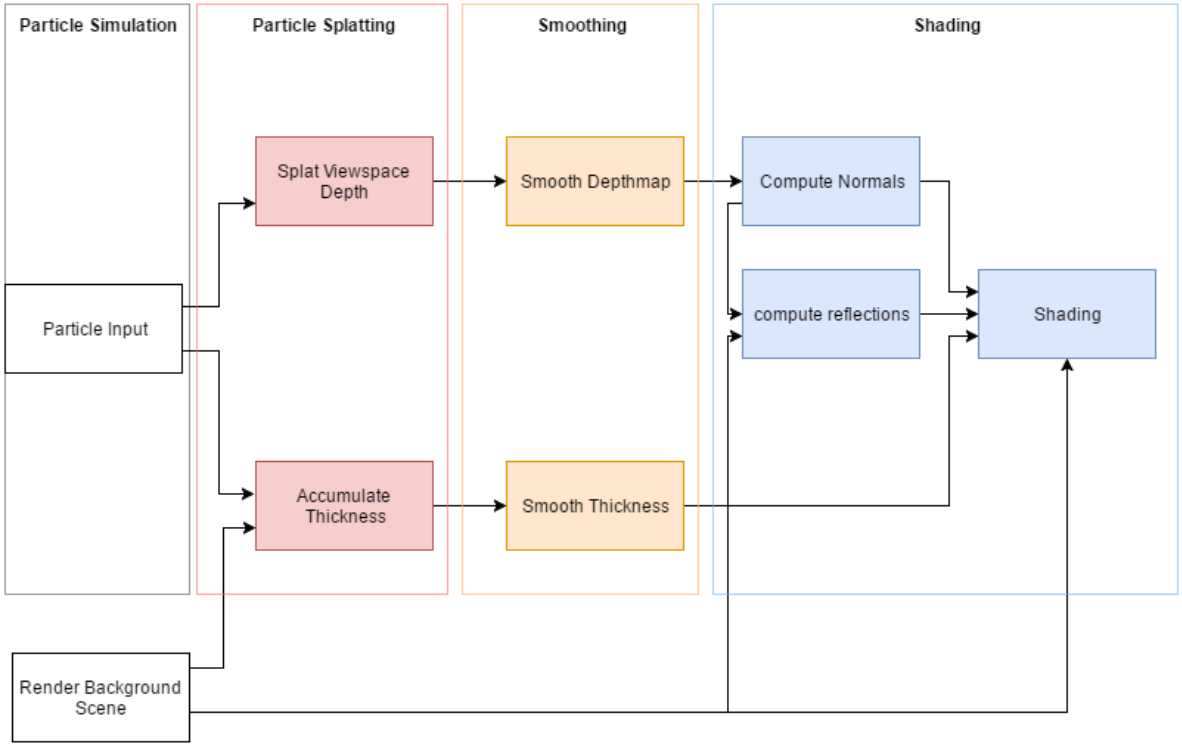

Figure 3.1: Overview of the rendering stages

3.2 Particle Splatting

To avoid rendering expensive geometry when splatting the particles as spheres, we render them as impostor spheres instead. For every particle a screen-aligned quad (point sprite) is created. Depending on the distance to the camera the point sprite size is scaled, so that nearer particles are larger than those far away. The quad is then passed to the fragment shader. We calculate the x and y component of the quad normal $\mathbf{nq}$ from the texture coordinates $(u, v) \in [0, 1]$ of the quad.

$$\mathbf{nq} = \begin{pmatrix} nq_x \\ nq_y \\ nq_z \end{pmatrix} = \begin{pmatrix} u \cdot 0.5 - 0.5 \\ v \cdot 0.5 - 0.5 \\ nq_z \end{pmatrix} \tag{3.2}$$

We can then discard the pixels that lie outside of the projected sphere radius. This is the case if $\mathbf{nq}_{xy} \cdot \mathbf{nq}_{xy} > 1$. For pixels inside the sphere radius we calculate the z component of $\mathbf{nq}$, so that

$$nq_z = \sqrt{1 - \mathbf{nq}_{xy} \cdot \mathbf{nq}_{xy}} \quad . \tag{3.3}$$

3.2.1 Depth Map

To create the illusion of a sphere the view space position $\mathbf{P}'$ of the point sprite at pixel (x, y) is adjusted towards the viewer in relation to the quad normal.

$$\mathbf{P} = \mathbf{P}' + \mathbf{nq} \cdot p_r \tag{3.4}$$

where $\mathbf{P}$ is the adjusted view space position for the sphere and p_r is the particle radius. If the particle is near to the camera, the new depth may be outside the near plane. We cut off the

depth in these cases:

$$P_z = \begin{cases} -z_n, & \text{if } |P_z| < z_n \\ P_z, & \text{otherwise} \end{cases} . \tag{3.5}$$

For proper depth testing we also have to replace the fragment depth of the shader. The view space depth P_z is then saved to our depth map. Saving the position and normal is not required because we have to recalculate them anyway after smoothing the depth map later on.

3.2.2 Thickness Map

The thickness map is an approximation for the volume of the fluid at the current viewpoint. We use it for attenuating the color and refracting the background scene later on. The per pixel thickness value T of a particle is calculated as

$$T = p_r \cdot nq_z \cdot 2. \tag{3.6}$$

If a particle is behind the background scene we set the thickness to 0 as the particle is not visible and thus does not contribute to the volume from the camera's perspective. We save T to our thickness map, with blending on and depth test off for proper accumulation. The thickness map only has to be an approximation, so rendering the particles at half or even quarter resolution still produces sufficient results (Figure 3.2). To create a smoother thickness distribution we additionally apply bilateral filtering.

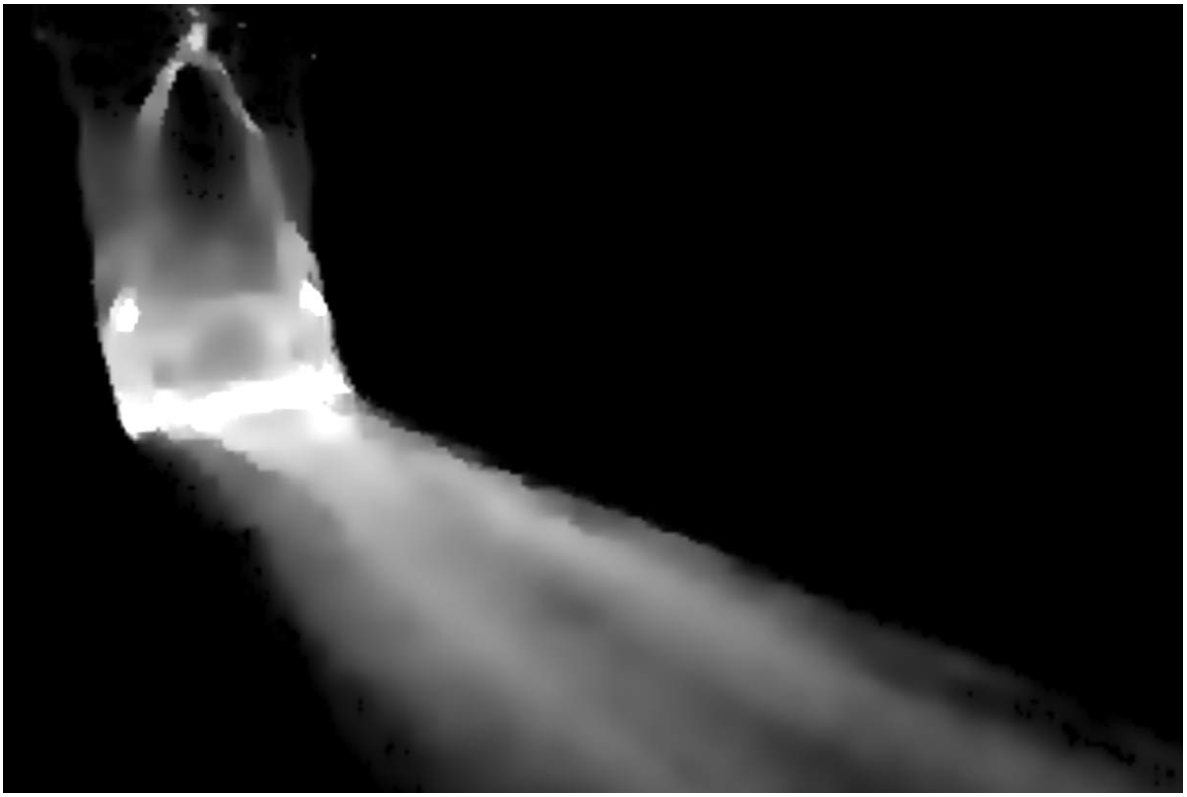

Figure 3.2: Thickness map of fluid at quarter resolution with bilateral filtering applied

3.3 Depth Smoothing

The resulting surface of the depth map is very bumpy due to rendering the particles as spheres. To create a more natural surface, the depth texture is smoothed by applying a smoothing kernel S to each pixel (x, y).

$$z(x, y) = S(x, y) \tag{3.7}$$

Since we are smoothing in screen-space, nearer particles cover more pixels than distant particles in the depth texture. A constant kernel radius will thus smooth far particles stronger than near particles and the fluid appears over-/under smoothed depending on the camera distance. To approximate a constant smoothing in screen space we chose to vary the number of smoothing iterations per pixel depending on the view space depth as in [6]. The number of required iterations $it(x, y)$ at pixel (x, y) is calculated based on the viewport height h, focal length and view space depth $z(x, y)$.

$$it(x, y) = \frac{h \cdot f_y \cdot \varrho}{z(x, y)}, \tag{3.8}$$

in which ϱ is the desired overall world space kernel radius. The number of required iterations can be quite high for near particles. To save performance we limit the number of maximum iterations by it_{max}. Thus a smoothing step i at pixel (x, y) is only applied if

$$it_{max} > it(x, y) > i.$$

Figure 3.3 shows the difference between constant smoothing and adaptive smoothing. At far distances we avoid over smoothing the surface by reducing the number of applied iterations and thus the kernel radius. In the following subsections we describe different smoothing kernels that can be applied.

3.3.1 Spatial Filter

The spatial filter kernel is simply replacing each depth value with a weighted sum of the pixel neighborhood in a certain radius. Spatial filters (e.g. box filter, Gaussian filter) can be implemented in a separable way, allowing the image to be blurred in one dimension and the resulting image in the other dimension.

This reduces the complexity to $(O)(2 \cdot kernel_{size} \cdot texture_{width} \cdot texture_{height})$ as opposed to a non-separable pass with complexity of $(O)(kernel^2_{size} \cdot texture_{width} \cdot texture_{height})$. Since only spatial information is considered, silhouettes and multiple patches of surface merge together, creating a single surface with low to no details. In most cases this is undesirable.

3.3.2 Bilateral Blur

A bilateral filter consists of a spatial kernel as well as a range kernel. Samples of similar intensity as the center have a stronger influence on the end result. This preserves silhouettes and surface details.

$$z(x, y) = \sum_{i,j=-N/2}^{N/2} G_s(i, j) \cdot G_r(|z(x, y) - z(x + i, y + j)|) \cdot z(x + i, y + j), \tag{3.9}$$

where G_s is the spatial kernel G_r the range kernel and N the kernel radius.

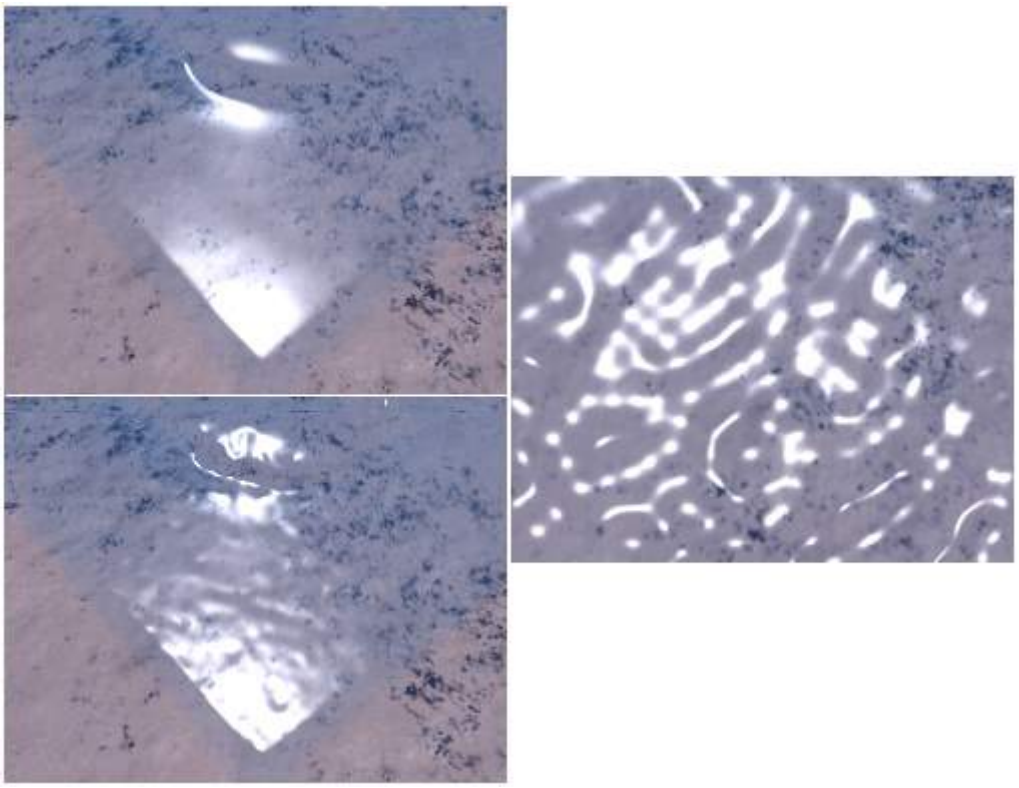

Figure 3.3: Comparison of constant- (top) and adaptive kernel radius (bottom).

The range component of this filter makes the blur non-separable. Implementing this nonetheless as a separated kernel leads to some artifacts, though their influence on the final appearance of the fluid is not very noticeable after applying the remaining shader stages [11], especially when looking at the fluid in motion.

3.3.3 Curvature Flow

Van der Laan et al. [22] address the smoothing by taking a different approach. Instead of blurring directly over the depth texture, they are minimizing the curvature that results from the spheres. In each integration step the view space depth values z are adjusted in time according to the mean curvature H. The integration is carried out using a simple forward Euler method. The spatial derivatives of z are calculated using finite differencing with central differences.

$$z = z + H \cdot dt, \tag{3.10}$$

where dt is the integration time step. The mean curvature is calculated from the divergence of the unit normal as

$$H = \frac{\nabla \cdot \hat{\mathbf{n}}}{2}.$$

The component $\frac{\partial \hat{n}_z}{\partial z}$ of the divergence is always zero, because z is a function of x and y and does not change when these are kept constant [22]. Thus we can rewrite the above equation as

$$H = \frac{\frac{\partial \hat{n}_x}{\partial x} + \frac{\partial \hat{n}_y}{\partial y}}{2}. \tag{3.11}$$

The normal is calculated by the cross product of the partial derivatives of the view space position $\mathbf{P}$ in the x and y direction.

$$\mathbf{n}(x,y) = \frac{\partial \mathbf{P}}{\partial x} \times \frac{\partial \mathbf{P}}{\partial y} \tag{3.12}$$

We can regain the view space position at pixel (x, y) from the depth texture by inverting the projection transformation. We first map the pixel coordinates back to normalized device coordinates:

$$ndc_x = \frac{2 \cdot x}{w} - 1, \quad ndc_y = \frac{2 \cdot y}{h} - 1, \tag{3.13}$$

where w and h are the viewport width and height respectively. We reverse the perspective division to regain clip space coordinates:

$$clip_x = ndc_x \cdot -z, \quad clip_y = ndc_y \cdot -z. \tag{3.14}$$

Then we regain the view space coordinates by dividing the clip space coordinates with the focal lengths:

$$view_x = \frac{clip_x}{f_x}, \quad view_y = \frac{clip_y}{f_y}. \tag{3.15}$$

The view space position is then

$$\mathbf{P}(x,y) = \begin{pmatrix} view_x \\ view_y \\ z(x,y) \end{pmatrix} = \begin{pmatrix} (\frac{1}{f_x} - \frac{2 \cdot x}{w \cdot f_x}) \cdot z(x,y) \\ (\frac{1}{f_y} - \frac{2 \cdot y}{h \cdot f_y}) \cdot z(x,y) \\ z(x,y) \end{pmatrix} = \begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} \cdot z(x,y). \tag{3.16}$$

Plugging Equation 3.16 into 3.12 gives us

$$\mathbf{n}(x,y) = \begin{pmatrix} C_x \cdot z(x,y) + W_x \cdot \frac{\partial z}{\partial x} \\ W_y \cdot \frac{\partial z}{\partial x} \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} W_x \cdot \frac{\partial z}{\partial y} \\ C_y \cdot z(x,y) + W_y \cdot \frac{\partial z}{\partial y} \\ \frac{\partial z}{\partial y} \end{pmatrix}, \tag{3.17}$$

in which $C_x = -\frac{2}{w \cdot f_x}$ and $C_y = -\frac{2}{h \cdot f_y}$. The terms that depend on W_x and W_y are ignored "because it simplifies the computations a lot and the difference is negligible as the contributions are small" [22]. Thus we get

$$\mathbf{n}(x,y) \approx \begin{pmatrix} -C_y \cdot \frac{\partial z}{\partial x} \cdot z(x,y) \\ -C_x \cdot \frac{\partial z}{\partial y} \cdot z(x,y) \\ C_x \cdot C_y \cdot z(x,y)^2 \end{pmatrix} = \begin{pmatrix} -C_y \cdot \frac{\partial z}{\partial x} \\ -C_x \cdot \frac{\partial z}{\partial y} \\ C_x \cdot C_y \cdot z(x,y) \end{pmatrix} \cdot z(x,y) \tag{3.18}$$

and

$$\hat{\mathbf{n}}(x,y) = \frac{\mathbf{n}(x,y)}{|\mathbf{n}(x,y)|} = \frac{(-C_y \cdot \frac{\partial z}{\partial x}, -C_x \cdot \frac{\partial z}{\partial y}, C_x \cdot C_y \cdot z(x,y))^T}{\sqrt{C_y^2 \cdot \frac{\partial z}{\partial x}^2 + C_x^2 \cdot \frac{\partial z}{\partial y}^2 + C_x^2 \cdot C_y^2 \cdot z(x,y)^2}} \tag{3.19}$$

Plugging Equation 3.19 into 3.11 finally leads to

$$H = \frac{C_y \cdot (\frac{1}{2} \cdot \frac{\partial z}{\partial x} \cdot \frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2} \cdot D) + C_x \cdot (\frac{1}{2} \cdot \frac{\partial z}{\partial y} \cdot \frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2} \cdot D)}{2 \cdot D \cdot \sqrt{D}}, \tag{3.20}$$

in which $D = C_y^2 \cdot \frac{\partial z}{\partial x}^2 + C_x^2 \cdot \frac{\partial z}{\partial y}^2 + C_x^2 \cdot C_y^2 \cdot z^2$.

Since an explicit integration scheme is used, this method becomes unstable at higher timesteps. Thus a low timestep at a high number of iterations is required. We empirically chose a timestep of 0.005 at up to 60 iterations.

We also found that particles near the camera (for $z > -3.0$ in our samples) tend to become unstable very fast. At these points we calculate D as

$$D = C_y^2 \cdot \frac{\partial z}{\partial x}^2 + C_x^2 \cdot \frac{\partial z}{\partial y}^2 + C_x^2 \cdot C_y^2 \cdot (z - \kappa)^2,$$

where κ is a user defined constant. We chose $\kappa = 3$ to sustain a stable integration at all depth values.

3.4 Normal Calculation

After smoothing the depth map, we have to recalculate the normals of the surface. The view space position at pixel (x, y) is regained with Equations 3.13–3.16. We get the normal $\mathbf{n}(x, y)$ with Equation 3.12 and calculate the unit normal $\hat{\mathbf{n}}(x, y) = \frac{\mathbf{n}(x,y)}{|\mathbf{n}(x,y)|}$ for every pixel.

The partial derivatives are calculated using finite forward differencing. At edges (where a large difference in depth occurs between one pixel and the next) the normals may not be well-defined for forward differencing, so we use the backward difference instead [11].
Figure 3.4 shows the calculated normals, encoded as RGB colors.

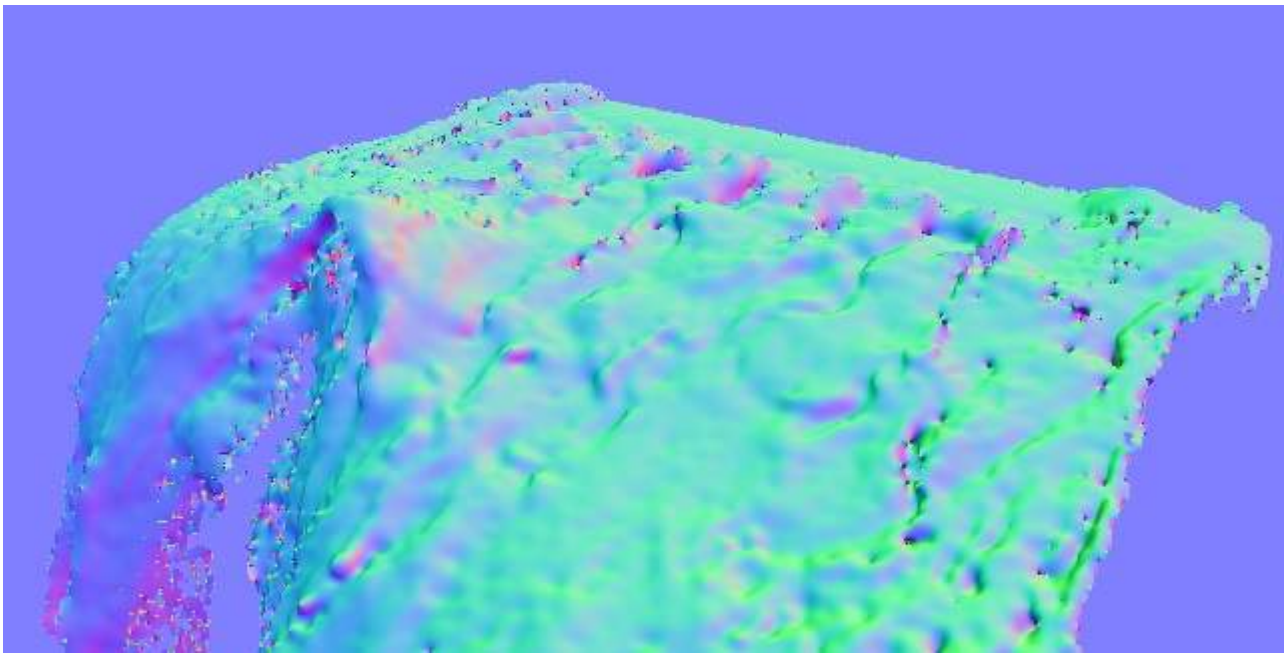

Figure 3.4: Calculation of normals from the view space position. The normals are visualized as $(red, green, blue)$ colors with minimum intensity of 0 and maximum intensity of 1. The mapping to the visualized color $\hat{\mathbf{c}}$ is done as $\hat{\mathbf{c}} = \hat{\mathbf{n}} \cdot 0.5 + 0.5$.

3.5 Rendering

The final color c_{final} is a composition of the attenuated fluid color c_{fluid}, the refracted scene color $c_{refract}$ and the reflected scene color $c_{reflect}$ with a specular highlight component σ.

$$c_{final} = mix(c_{fluid}, c_{refract} \cdot (1 - F(\hat{\mathbf{v}}, \mathbf{n}, \rho)), \tau) + c_{reflect} \cdot F(\hat{\mathbf{v}}, \mathbf{n}, \rho) + \sigma \tag{3.21}$$

in which $\hat{\mathbf{v}}$ is the view vector from the camera to the surface and $\mathbf{n}$ the surface normal. F is the Fresnel term, resulting in stronger reflections at grazing view angles and stronger refractions when looking towards the surface normal. The amount of reflection and refraction also depends on the refractive index of the fluid (we assume that the surrounding medium is air with a refractive index of ≈ 1. For our sample scenes we used the refractive index of water (1.33) and employed Schlick's approximation [20] for the Fresnel term. The specular highlight

$$\sigma = k_s \cdot (max(0.0, \mathbf{n} \cdot \mathbf{h}))^{\alpha}$$

is computed from the surface normal, halfway vector and specular constants.

3.5.1 Light Attenuation

As light travels through a sample of fluid, a percentage of it gets absorbed and scattered. This diminution of radiant energy is referred to as light attenuation. The amount of attenuation increases in relation to the extinction coefficient η and the sample depth z in meters [23].

Because η differs for each wavelength the transmitted light I_z at depth z is of a different color than the incoming light I_0 at the fluid surface. Table [3.1] and Figure [3.5] show the approximate attenuation of wavelengths in the visible spectrum for water, causing its color to appear mostly as blue at increasing depths. The percentile transmittance is given by the ratio of I_z to I_0 [23]:

$$\frac{I_z}{I_0} = e^{-\eta \cdot z} \cdot 100\,\% \tag{3.22}$$

By putting z at the eye we can relate the sample depth directly to our thickness map T. Thus we get the fluid color $c_{fluid}(x, y) = I_z(x, y)$ by

$$I_z(x, y) = e^{-\eta \cdot T(x,y)} \cdot I_0, \tag{3.23}$$

where I_0 is the incident light; We assume that the main light source is a directional light, leading to a constant light color I_0.

Table 3.1: Approximate extinction coefficients and percentile attenuation for water; Data taken from [23]

Wavelength [nm]	η	attenuation $[\% \cdot m^{-1}]$
680.0 (red)	0.555	42.59
526.2 (green)	0.041	4.017
465.0 (blue)	0.0208	2.058

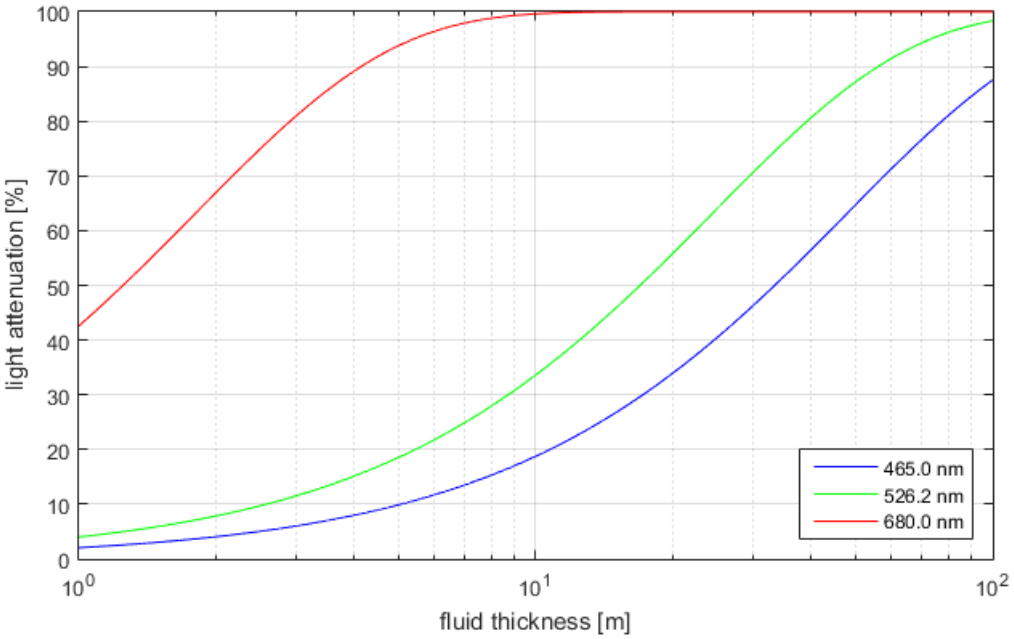

Figure 3.5: Attenuation of light traveling through water
Plotted data from Table 3.1

3.5.2 Reflection

Reflections of the scene geometry on the fluid can substantially enhance the final result, as can be seen in Figure 3.6. For each pixel (x, y) we send a ray from the camera to the view space position $\mathbf{P}$ of the fluid surface at that pixel and calculate the reflected direction

$$\mathbf{R} = \hat{\mathbf{v}} - 2 \cdot \hat{\mathbf{n}} \cdot \hat{\mathbf{v}} \cdot \hat{\mathbf{n}}, \tag{3.24}$$

in which $\hat{\mathbf{v}} = \frac{-\mathbf{P}}{|-\mathbf{P}|}$ is the view vector. Van der Laan et al. and Green [22, 11] sample the reflected color directly from an environment map. We extend upon this by implementing a screen space ray tracer based on [15] to also reflect the scene geometry.

Let us look first at a conventional linear 3D ray tracer in view space. Our ray starts at the view space position $\mathbf{P}$. Knowing the ray direction we can calculate the endpoint $\mathbf{E}$ by multiplying $\mathbf{R}$ with the desired maximum ray distance d and add it to $\mathbf{P}$. Taking the delta of the two points $\mathbf{E} - \mathbf{P}$ and dividing it by the desired number of trace iterations gives us a step vector $d\mathbf{P}$. In each tracing iteration we can advance along $\mathbf{R}$ with $\mathbf{P} = \mathbf{P} + d\mathbf{P}$ and check for a depth buffer intersection by sampling the depth buffer at the current point. If the ray is behind the sample (within some range) we consider it an intersection and return the scene color at this point. This approach has one major drawback. Linear marching in view space does not equal linear marching in screen space. Many points along $\mathbf{R}$ may be mapped to the same pixel. A large amount of iterations, thus small step sizes, will lead to oversampling the same pixels in screen space while larger step sizes may miss depth buffer intersections completely.

Linearly traversing the ray in 2D as in [15] circumvents this problem by guaranteeing that in each iteration we move at least one pixel in screen space. We start by projecting the start and end point of our ray using a standard projection matrix $\mathbf{M}$ such that $\mathbf{H_0} = \mathbf{M} \cdot \mathbf{P}$ and $\mathbf{H_1} = \mathbf{M} \cdot \mathbf{E}$ is in homogeneous clip space. After perspective divide we get the respective

start and end point in screen space. Rasterization and tracing of the line between these two points is in effect a simple 2D line rendering problem and can be done by using standard, well known and understood, DDA (Digital Differential Analyzer) methods such as Bresenham's [7] line algorithm. This method considers eight cases. Due to the excessive branching, the performance on a GPU is considerably limited. McGuire and Mara [15] provide an optimized version for the GPU that unifies these cases and avoids expensive branching by ensuring that the primary traversal axis in every iteration is the x-axis. Linear interpolation of a property in homogeneous space will result in linear interpolation of the corresponding 3D value [15]. This means that we can keep track of the view space depth of any point on the line by linearly interpolating between $k_0 = H_{0w}^{-1}$ and $k_1 = H_{1w}^{-1}$ as we traverse it. We get the corresponding view space depth z for value k on a given point as

$$z = \frac{-1.0}{k}. \tag{3.25}$$

Checking z against the depth buffer for our given point, like in the 3D approach, we can determine if the ray intersects the buffer and, if so, return the scene color for this point as our reflected color. If no intersection is detected we sample from an environment map instead.

$$c_{reflect}(x, y) = \begin{cases} Scene(\mathbf{HP}), & \text{if } raytrace(\mathbf{P}, \mathbf{R}, I) \\ Env(\mathbf{R}), & \text{otherwise} \end{cases}, \tag{3.26}$$

in which $Scene(x, y)$ returns the scene color of a point and $Env(x, y, z)$ the environment color of a vector. I is the maximum amount of steps, after which we abort the trace. **HP** is the pixel at which we determined an intersection with the depth buffer.

We can further improve the performance of the ray tracer by sacrificing visual quality. Scaling the derivatives of the DDA with a stride value ($d\mathbf{P} \cdot stride$ and $dk \cdot stride$), we increase the number of pixels that we traverse at each step. Some of the lost quality can be regained by using a simple binary search refinement [12]. If our stride is >1 and an intersection with the depth buffer is detected we take one step back and halve the stride. Then we check if the depth buffer is still intersected. If so, take another step back and repeat. If not, we take a step forward and repeat. Each time we detect an intersection **HP** is updated accordingly. The regained accuracy greatly improves the final result (Figure 3.7) with almost no impact on performance (Table 4.3).

Although we stop tracing after one hit, the method can be trivially extended to allow for more bounces by additionally returning the respective 3D point of the hit pixel.

3.5.3 Refraction

While the screen space ray tracer could also be used for refractions, we have no accurate volume information of the fluid to take scattering into account. Approximating the refractions as in [22] gives good results with the added benefit of saving computational expenses. The sample coordinates for the background scene are offset in accordance to the view space normal and thickness, so that increased thickness leads to higher refraction. We limit the amount of refraction by β to reduce highly unrealistic refraction at high thickness values.

$$c_{refract} = Scene((x, y) + \mathbf{n}_{xy} \cdot max(T(x, y) \cdot \alpha, \beta)), \tag{3.27}$$

in which α is a fluid-specific constant to control the amount of refraction. Figure 3.8 shows the approximated refractions when the scene is covered by high thickness fluid.

Figure 3.6: Comparison of skybox-only reflections (left) and screen space scene reflections (right) for smooth (top) and rough (bottom) fluid

3.5.4 Transparency

We expect the background scene to be increasingly hidden behind larger amounts of fluid. The transparency τ is used to interpolate between the fluid color and the refracted scene color in relation to our thickness map. Because the accumulated thickness may theoretically have any value in the range $[0, \infty)$ we map it into the range $[0, 1]$ by using an exponential function

$$\tau(x, y) = e^{-T(x,y)\cdot\gamma}, \tag{3.28}$$

where γ can be used to control the falloff, depending on the fluid opaqueness.

(a) Stride = 1

(b) Stride = 8;
No binary search refinement

(c) Stride = 8;
With binary search refinement

Figure 3.7: Reflection quality with different stride settings. Some of the lost quality that is caused by higher strides can be regained with binary search refinement.

Figure 3.8: Approximated refractions through thickness map

Chapter 4

Results and Discussion

In this chapter we will present and discuss the performance of our three test scenes in various situations and the general memory requirements of our textures.

4.1 Memory

The required memory for our data structures is fairly low, since we work mainly with textures. We use openGL color formats [2] to describe the memory usage of the textures. For example, $R32F$ denotes a single channel texture with 32 bit/pixel, $R16F$ uses 16 bit/pixel and an $RGBA8$ texture has four channels, each with 8 bit/pixel. The required memory per texture for our application is:

- ***Depth Texture***: $TextureSize_{depth} \cdot R32F$
- ***Smoothed Depth Texture (PingPong rendering)***: $2 \cdot TextureSize_{smoothed} \cdot R32F$
- ***Thickness Texture***: $TextureSize_{thickness} \cdot R32F$
- ***Normal Texture***:$TextureSize_{normal} \cdot RGBA32F$
- ***Scene Color Texture***:$TextureSize_{scene} \cdot RGBA8$
- ***Scene Depth texture***: $TextureSize_{sdepth} \cdot R32F$

If we are rendering the fluid at a resolution of $1920 \times 1080 = 2\,073\,600$ pixel, an $R32F$ texture would require

$$4\,\text{B/pixel} \cdot 2\,073\,600\,\text{pixel} \approx 7.91\,\text{MB}.$$

An $RGBA8$ texture would require the same amount, because each channel uses 1 B/pixel, leading to a total of 4 B/pixel. The RGBA32F normal texture amounts to

$$16\,\text{B/pixel} \cdot 2\,073\,600\,\text{pixel} \approx 31.65\,\text{MB}.$$

Rendering the thickness texture at half resolution ($960 \times 540 = 518\,400$ pixel) requires

$$4\,\text{B/pixel} \cdot 518\,400\,\text{pixel} \approx 1.98\,\text{MB}.$$

All textures combined would then require

$$5 \cdot 7.91\,\text{MB} + 31.65\,\text{MB} + 1.98\,\text{MB} \approx 83.08\,\text{MB}.$$

Considering the possible performance breakdown for rendering at 1920×1080 (Table 4.4), we usually render everything at half resolution. Thus a more realistic value would be

$$6 \cdot 1.98\,\text{MB} + \frac{31.65\,\text{MB}}{4} \approx 19.79\,\text{MB}.$$

In addition to the textures, we also have to account for the particles that are required in the splatting stage. Each particle comes with three float values for the position. Considering 4 B per float, that would be $3 \cdot 4\,\text{B} = 12\,\text{B}$ per particle.

Considering that modern GPUs come with at least 4 GB of VRAM, the memory footprint of our data structures is negligible at half-resolution.

4.2 Performance

We used three scenes for measuring the required render time in milliseconds per frame.

a) The wave scene starts as a cube of 143.944k particles and quickly dissolves into a stream, forming small-scale waves along the way. We also provided a few environment objects for this scene to show the application and performance of screen space reflections in comparison to the other scenes without much scene geometry.

b) The crown scene (Figure 4.2) exists in two versions. One has a low amount of particles (38.146k) distributed over a relatively large space. The second version has a higher particle resolution (157.944k with a smaller particle radius over the same amount of space), thus leading to a more detailed surface and more potential individual surfaces through splashes and water tension.

c) The waterfall scene (Figure 4.3) has a large amount of particles (819.417k) over a large space, giving a good idea about the scaling of our method in regards to the number of particles.

Tables 4.1 and 4.2 show the performance measurements of the various scenes at full and half resolution. In all scenes the fluid made up about a quarter of the screen. The screen space reflections were done with a maximum of 100 tracing steps and a stride of 8 with four binary refinement iterations. We used curvature flow smoothing with up to 60 iterations. The measured times are purely for the fluid rendering, not taking the scene rendering into account. All measurements are reproducible on home systems as they were taken on an Intel Core i7-5820K CPU 3.30 GHz with 16GB RAM and an NVIDIA GTX 980.

Table 4.1: Performance in our three sample scenes at full resolution

Scene	Wave	Crown 1	Crown 2	Waterfall
Fluid Resolution	1920×1080	1920×1080	1920×1080	1920×1080
Particles [#]	143944	38146	157944	819417
Splat [ms]	0.387	0.465	0.752	1.801
Smoothing [ms]	3.101	2.991	3.058	3.627
Normals [ms]	0.122	0.123	0.124	0.132
Raytrace [ms]	0.650	0.72	1.003	1.302
Composition [ms]	0.225	0.23	0.247	0.307
Total [ms]	*4.485*	*4.529*	*5.184*	*7.169*

Table 4.2: Performance in our three sample scenes at half resolution

Scene	Wave	Crown 1	Crown 2	Waterfall
Fluid Resolution	960×540	960×540	960×540	960×540
Particles [#]	143944	38146	157944	819417
Splat [ms]	0.342	0.386	0.674	0.999
Smoothing [ms]	0.925	0.884	0.924	0.906
Normals [ms]	0.036	0.036	0.0359	0.034
Raytrace [ms]	0.579	0.799	0.962	1.217
Composition [ms]	0.209	0.192	0.235	0.306
Total [ms]	*2.091*	*2.297*	*2.831*	*3.462*

Table 4.3 shows the performance impact on ray tracing by stride and binary refinement. Increased stride allows for a reduction in maximum trace iterations while providing at least the same tracing distance. All measurements were taken for our wave scene.

Table 4.3: Performance for various reflection settings in wave scene

Stride	1	4	4	8	8
Binary refinement [iterations]	0	0	2	0	4
Max steps [iterations]	280	180	180	100	100
Raytrace [ms]	*1.913*	*1.055*	*1.065*	*0.627*	*0.650*

We conclude that the performance of our rendering method is mainly influenced by four factors:

Particle Radius Because we are rendering the particles as impostor spheres, we have to overwrite the depth value in the fragment shader. This will turn off early depth tests for the hardware, thus resulting in many unnecessary fragment shader invocations. A higher particle radius covers more fragments per particle and will result in more wasted shader invocations. This is especially true for very dense sets of particles where most particles would be hidden by another. Enabling early depth tests (no depth replacement in shader) for the worst case scene (Table 4.4) increased the splatting performance from 7.865 ms to 1.380 ms.

Amount of Splatted Fluid Most of our shaders work directly in screen space. Texels that do not contain any fluid information can be mostly ignored, save for silhouette detection. The computational cost increases substantially in relation to the amount of visible fluid, because the shaders have more work to do. This point has the most impact on the final performance, because it influences the more cost extensive shader stages. The main performance bottleneck is the smoothing stage, because it has to be run multiple times (up to 60 iterations in our sample scenes) to achieve a smooth surface without disregarding surface details.

Particle Distance to the Viewer In direct relation to the amount of visible particles, their distance to the viewer also influences the performance of the smoothing stage. As explained in Section 3.3 near particles have a larger screen space radius than far particles and require a higher kernel radius for the same amount of smoothing.

Average Ray Trace Distance For each fluid texel on the screen we trace a reflection ray as described in Section 3.5.2. Depending on the amount of rays not hitting any geometry,

the amount of iteration steps can increase substantially. We found that by choosing the function parameters carefully, the performance drop can be mitigated considerably while still providing sufficient quality. Increasing the stride has the most impact, because it allows for a reduction in maximum iteration steps while maintaining the same tracing distance. Although increased stride causes a drop in quality, some of it can be recovered with binary refinement (Figure 3.7) at little additional cost.

Thus we can conclude that the worst case performance happens when a screen is completely filled with fluid that hides a lot of particles behind its surface. We produced such a scenario by taking the starting cube of fluid in the wave scene and zooming in until it covered the whole camera. Although these cases should be rare in an actual application (unless the camera is positioned inside the fluid), the performance drop at these occasions is very notable, as shown in the table entry. To sustain real-time performance of our application throughout the simulation, save for the most extreme situations, we propose to render the fluid at half resolution. The upscaling results in artifacts, mainly at the edges, and blurs the silhouettes slightly as can be seen in Figure 4.4. However, we found that these artifacts are barely visible in the final shading, especially when the fluid is in motion. Table 4.4 shows the performance for our worst case scenario at full and half resolution.

Table 4.4: Performance in our worst case scenario
(Fluid as a dense cube and filling up whole screen)

Fluid Resolution	1920×1080	960×540
Splat [ms]	7.865	2.364
Smoothing [ms]	8.031	2.813
Normals [ms]	0.235	0.060
Raytrace [ms]	1.831	1.544
Composition [ms]	0.329	0.101
Total [ms]	*18.291*	*6.882*

Figure 4.1: Wave test scene: 143.944k particles, rendered at 1920×1080 in 5.79 ms

Figure 4.2: Crown test scene: 38.146k particles, rendered at 1920×1080 in 6.22 ms

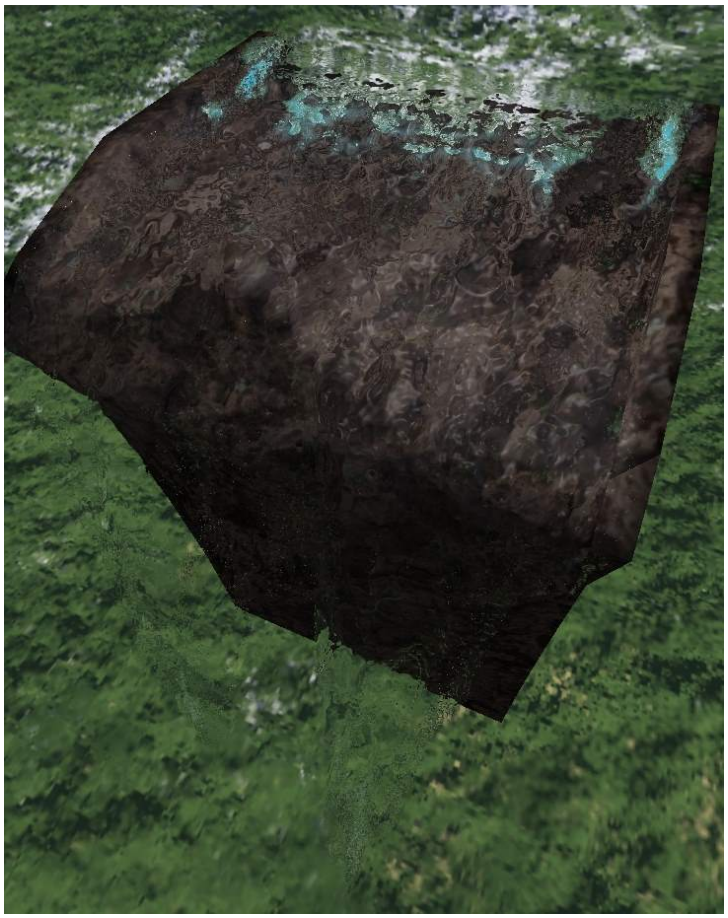

Figure 4.3: Waterfall test scene: 819.417k particles, rendered at 1920×1080 in 7.18 ms

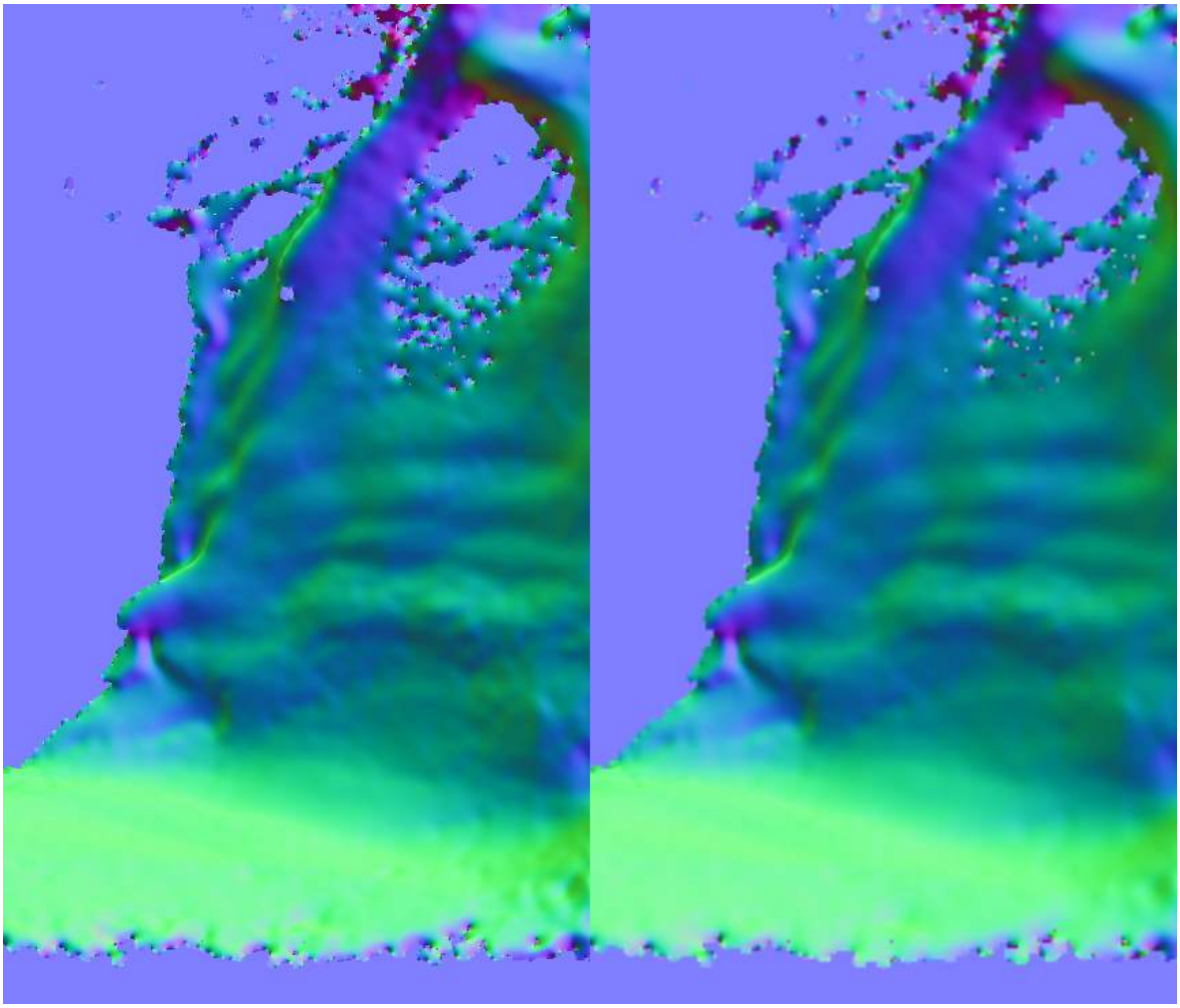

Figure 4.4: Comparison of fluid at full resolution (left) to fluid at half resolution (right). Normal maps, as in Figure 3.4, are used for the comparison because they show the artifacts most intensely (especially at the hole in the top right corner).

Chapter 5

Conclusion & Future Work

In this thesis, we implemented a real-time application for rendering SPH fluids with screen space reflections. Because of the missing information in screen space, only scene geometry that is actually visible by the camera can be reflected. The performance can be expected to be < 7 ms per frame at half resolution (960x540) throughout the application. This allows for additional extensions and work on the GPU while still providing 60 frames per second. The memory usage is also fairly low since we are only operating in screen space and should not influence the overall application noticeably. Our method does not take transparent scene geometry into account. To extend it to one transparent layer we have to splat the fluid into two separate textures for fluid in front of and behind the transparent object.

In the future we would like to add additional effects such as proper foam [6, 5] and bubbles which greatly improve the visual result of highly advected fluids (such as our waterfall scene). We also think that shadows and caustics may be plausible at real-time, although this requires to render and smooth the fluid again from the light's viewpoint, which may be expensive because we cannot utilize early depth tests due to the impostor sphere rendering.

Bibliography

[1] A note on hybrid Eulerian/Lagrangian computation of compressible inviscid and viscous flows.

[2] openGL Color Formats, 2015.

[3] G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner. Parallel Surface Reconstruction for Particle-Based Fluids. *Comput. Graph. Forum*, 31(6):1797–1809, September 2012.

[4] Gizem Akinci, Nadir Akinci, Edgar Oswald, and Matthias Teschner. Adaptive surface reconstruction for sph using 3-level uniform grids. 2013.

[5] Nadir Akinci, Alexander Dippel, Gizem Akinci, and Matthias Teschner. Screen Space Foam Rendering. pages 173–182, 2013.

[6] Florian Bagar, Daniel Scherzer, and Michael Wimmer. A Layered Particle-Based Fluid Model for Real-Time Rendering of Water. *Computer Graphics Forum*, 29(4):1383–1389, 2010.

[7] J. E. Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Syst. J.*, 4(1):25–30, March 1965.

[8] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, 2nd edition, 2015.

[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] R.A. Gingold and J.J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. 181:375–389, 1977.

[11] Simon Green. Screen Space Fluid Rendering for Games, 2010.

[12] Ben Hopkins. Screen Space Reflectionsin Unity 5, 2015.

[13] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH Fluids in Computer Graphics. In Sylvain Lefebvre and Michela Spagnuolo, editors, *Eurographics 2014 - State of the Art Reports*. The Eurographics Association, 2014.

[14] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[15] Morgan McGuire and Michael Mara. Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, December 2014.

[16] Matthias Müller, David Charypar, and Markus Gross. Particle-based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[17] Matthias Müller, Simon Schirm, and Stephan Duthaler. Screen Space Meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pages 9–15, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[18] Artur Palha, Lento Manickathan, Carlos Simao Ferreira, and Gerard van Bussel. A hybrid Eulerian-Lagrangian flow solver. *arXiv preprint arXiv:1505.03368*, 2015.

[19] Saket Patkar, Mridul Aanjaneya, Dmitriy Karpman, and Ronald Fedkiw. A Hybrid Lagrangian-Eulerian Formulation for Bubble Generation and Dynamics. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 105–114, New York, NY, USA, 2013. ACM.

[20] Christophe Schlick. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[21] Next Limit Technologies. Real Flow, 2015.

[22] Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen Space Fluid Rendering with Curvature Flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 91–98, New York, NY, USA, 2009. ACM.

[23] Robert G. Wetzel. *Limnology*. Saunders College Publishing, 2nd edition, 1983.

[24] Jihun Yu, Chris Wojtan, Greg Turk, and Chee Yap. Explicit Mesh Surfaces for Particle Based Fluids. *Comput. Graph. Forum*, 31(2pt4):815–824, May 2012.

[25] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Data-Parallel Octrees for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, May 2011.

[26] Tobias Zirr and Carsten Dachsbacher. Memory-efficient On-the-fly Voxelization of Particle Data. In *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*, PGV '15, pages 11–18, Aire-la-Ville, Switzerland, Switzerland, 2015. Eurographics Association.