LiU-ITN-TEK-A--10/058--SE

High-resolution simulation and rendering of gaseous phenomena from low-resolution data

Gabriel Eilertsen

2010-09-28

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LiU-ITN-TEK-A--10/058--SE

High-resolution simulation and rendering of gaseous phenomena from low-resolution data

Examensarbete utfört i medieteknik
vid Tekniska Högskolan vid
Linköpings universitet

Gabriel Eilertsen

Handledare Michael Root
Examinator Jonas Unger

Norrköping 2010-09-28

Linköping University
INSTITUTE OF TECHNOLOGY

Linköping University

ITN – Department of Science and Technology

Master's Thesis

High-resolution Simulation and Rendering of Gaseous Phenomena from Low-resolution Data

Author:
Gabriel Eilertsen*

Supervisor:
Michael Root

Examiner:
Jonas Unger

Norrköping, Friday 1st October, 2010

*e-mail: gabei429@student.liu.se

Abstract

Numerical simulations are often used in computer graphics to capture the effects of natural phenomena such as fire, water and smoke. However, simulating large-scale events in this way, with the details needed for feature film, poses serious problems. Grid-based simulations at resolutions sufficient to incorporate small-scale details would be costly and use large amounts of memory, and likewise for particle based techniques.

To overcome these problems, a new framework for simulation and rendering of gaseous phenomena is presented in this thesis. It makes use of a combination of different existing concepts for such phenomena to resolve many of the issues in using them separately, and the result is a potent method for high-detailed simulation and rendering at low cost.

The developed method utilizes a slice refinement technique, where a coarse particle input is transformed into a set of two-dimensional view-aligned slices, which are simulated at high resolution. These slices are subsequently used in a rendering framework accounting for light scattering behaviors in participating media to achieve a final highly detailed volume rendering outcome. However, the transformations from three to two dimensions and back easily introduces visible artifacts, so a number of techniques have been considered to overcome these problems, where e.g. a turbulence function is used in the final volume density function to break up possible interpolation artifacts.

Keywords: Computer graphics, Fluid simulation, Simulation refinement, Simulation optimization, Slice simulation, Volume rendering, Gaseous phenomena, Smoke, Fire

Acknowledgements

I would like to thank all of the talented people at Tippett Studio for giving me this opportunity, for making me feel at home, and for making my internship a really great time. Special thanks goes to the people at R&D, and in particular Michael Root, Andrew Gardner and Michael Farnsworth for guiding me through my thesis work and giving me valuable advise and discussions. A special thank also to Rebecca Byars who handled all of the important internship details, enabling my time at Tippett.

I also would like to thank Jonas Unger at Linköping's University for making the internship at all possible, introducing me to the people at Tippett Studio, and for helping me in outlining a master thesis work.

For economical support I would like to thank Tippett Studio, as well as John Lennings Stipendiestiftelse för Norrköpings Näringsliv and Holmen Paper who provided me with scholarship funds.

Last, but certainly not least, I would like to thank my girlfriend, Jenny Nilsson, for supporting me through my time in California, and for always being there for me. And thank you for spending a wonderful vacation with me in San Francisco and Los Angeles, recharging me for the final period of work. I love you.

Contents

Chapter 1

Introduction

The work described in this thesis was carried out as an internship at the Research and Development department at Tippett Studio[1], Berkeley, California, USA. The internship took place between March and August, 2010, and constitutes the Master Thesis for a Master of Science in Media Technology and Engineering at Linköping University, Campus Norrköping, Norrköping, Sweden.

To give an introduction to the work, the motivation and background will be treated in the following sections. The structure of the report will also be laid out, as well as its limitations.

1.1 Background

Gaseous phenomena, such as smoke and fire, are commonly appearing elements in the real world. Thus, to recreate realistic scenes in visual effects, such phenomena often have to be considered and modeled. The problem with these effects are not only in the rendering of the participating media, but also in the fluid behavior of the gas or liquid itself. Simulating this behavior and approximating it for use in computer graphics is well-studied but expensive when it comes to large-scale simulations where the small-scale details still are needed, i.e. when large simulation resolutions are needed. This expense is in terms of computation time and memory, and since both are valuable in visual effects production solutions to this problem is an active area of research.

When it comes to rendering of participating media, there are costly calculations involved in the estimation of the light scattering through the media. Since the light interaction is not restricted to the surface of the media, the whole three-dimensional space has to be considered. Inside the volume, light can be scattered multiple times when interacting with the small particles constituting the media.

[1] http://tippett.com

The work in this thesis intend to investigate different techniques for achieving volume simulations and renderings suitable for feature films. This, in an attempt to enable such capabilities in an efficient manner integrated into the pipeline at Tippett Studio. Efficient in this case means shorter simulation and rendering times and less memory usage than direct high-resolution simulations and costly ray-tracing methods in the rendering.

To be suitable for the production pipeline at Tippett Studio some conditions have to be met. The input data is given as a particle simulation, limited in the number of particles, enabling fast low-resolution simulations that are easy for an artist to work with. The input particles should undergo a simulation refinement to create fine details, and the output of the simulations should be visualized with volume rendering techniques suited for the rendering environment at hand.

1.2 Objective

As described in the previous section there are problems in generating natural volume phenomena simulations in a brute-force manner – that is using numerical methods in grid-based simulations on large grid sizes, or with particle based simulations using a large number of particles. Therefore, a method is sought that can reduce calculation times and memory usage and still produce the fine details needed from the simulations. Given the pipeline conditions at Tippett Studio, the method should be a simulation refinement stage where a low-resolution simulation governs the over-all movement of the media, and where simulations are performed to add fine high-resolution details.

The input to the simulation refinement stage is a particle simulation, restricted in the number of particles, and the output should be a high-resolution density function, possibly with additional attributes to use in the rendering stage.

The rendering of the simulations should also be restricted in terms of calculation times and memory, without sacrificing important volume lighting phenomena; the renderings should be of such quality that they could be used in visual effects in feature films. Concepts such as multiple scattering and self-shadowing therefore have to be considered.

The input to the volume rendering is the density function from the simulation stage. This simulation could be placed in an arbitrary environment, and the rendering should account for the light interactions between the volume and its surroundings.

To summarize, the purpose of the work outlined is to find an efficient way to simulate and render natural gaseous phenomena. The simulation should convincingly capture the fine details in large-scale effects, such as in clouds or in smoke from a forest fire. The simulation should start from a low-resolution simulation with a limited set of particles, and effectively capture and add the inherent fine details of turbulence, such as swirls and vortices, at high resolution. The simulation should finally be rendered in such way to incorporate the lighting

effects associated with participating media detailed enough to suit visual effects in feature films.

1.3 Method

To achieve the established goals, a number of different techniques were considered, and some tests were done, before the decision was made to use a slice simulation technique. The technique performs a high-resolution refinement on two-dimensional view-aligned planes, effectively capturing the fine detailed movements of the fluid. It was introduced by [Horvath and Geiger, 2009] for simulation and rendering of fire, for which it has shown to be successful. The biggest concern is how to translate the technique to simulation and rendering of smoke, which according to the investigation done for this thesis work has never been done before. Since fire is emissive, a lot of the artifacts inherent in the use of this method are effectively masked out. To account for this, and to suit the development environment at hand, the method proposed differs in many respects from the one in [Horvath and Geiger, 2009].

The method is based on the fact that the important details of a participating media are those in the plane perpendicular to the viewing direction; in the view direction, along the camera axis, much less detail can be distinguished. By setting up a relatively small amount of planes aligned with the camera plane, these can be simulated separately in 2D to make the process significantly faster and use much less memory. The input coarse particle simulation is projected, or splatted, onto the planes before simulation, and during the simulation new data is interpolated into the simulation to make it conform to the movement of the original data. In this way an initial low-resolution particle simulation with limited detail can be used to create a set of refined high resolution slice simulations for rendering.

Since the technique described in [Horvath and Geiger, 2009] was intended for simulation and rendering of fire, the simulation slices could be used in a simple GPU blending stage to create fast outputs, which seems sufficient for fire simulation. When dealing with non-emissive media though, this is not an appropriate solution since the light interactions with the volume have to be calculated. Therefore, the slice data have to be transformed into a three-dimensional density function suitable for volume rendering methods. To do so, an interpolation is performed, where the slices are used for lookup. This approach may give visual artifacts due to the rough discretization along the viewing direction, since there is no emissive component smoothing the artifacts out. To compensate for this a turbulence function can be used to perturb the interpolation positions, and thereby add some lost information back into the volume.

A problem remaining is with storage and rendering. Since storing a high-resolution 3D data structure proposes a significant memory issue, the interpolation and turbulence should be evaluated on the fly during rendering. This means that the set of high resolution 2D textures may use to much memory if

loaded simultaneously during rendering, and some sort of strategy of loading chunks of the data must be considered.

The problem statement can now be divided into four general goals, where the first two are part of the simulation stage of the process, and the second two are part of the rendering stage:

- Convert a particle simulation to a set of two-dimensional density functions.
- Simulate the two-dimensional slices using new data for each frame.
- Formulate a three-dimensional density function from the slices, with 2D to 3D transformation artifacts suppressed.
- Render the density function using volumetric methods.

1.4 Structure

To show what has been done earlier, and how this thesis work is connected to current research, Chapter 2 will describe available fundamental methods in the area and some of the previous and related work. In Chapter 3 the underlying theory used to accomplish the specified goals is explained. How this theory is then used in the proposed method, and the details of the method itself, are explained in Chapter 4. The implementation specific details of the method are treated in Chapter 5. Finally, the results of the work are presented in Chapter 6, and discussed in Chapter 7.

1.5 Prerequisites

To fully comprehend the different techniques in the thesis, the reader is assumed to have at least basic knowledge in calculus, physics and computer graphics, including knowledge about commercial computer graphics software. However, the proposed and implemented method can be understood without the underlying math and physics, and the thesis can therefore be read in such manner to only focus on the method itself and the computer graphics aspects of it. In that case, the reader is suggested to skip the theory in Chapter 3, and possibly the background description in Chapter 2.

1.6 Limitations

The work described is aimed at developing a system to handle large-scale simulations and renderings of gaseous phenomena in general. However, the main work is concerned with creating smoke visualizations and how to extend the technique described by [Horvath and Geiger, 2009] to support visually convincing smoke simulations and renderings. This means that the possibilities of using the method for other phenomena, such as fire, dust, sand etc., are not investigated, but the method should be able to include those kind of effects.

Chapter 2

Background description

Mimicking the characteristics of gaseous phenomena in computer graphics is a challenging task, both when it comes to simulation of the complex turbulent movements and when rendering the light interactions with the three-dimensional translucent volume.

In this context, a gaseous volume consists of small particles suspended in a gaseous flow. Examples of such particles could be dust, salt or water. It is the particles that generate the lighting interaction phenomenas, while the gas carries the particles in a turbulent motion. The particles are often referred to as *particulates*, or a *particulate matter*, while *aerosol* is used to describe the combination of the particles and the gas. The sizes of the particles in a particulate matter are in the order of magnitude $< 10\mu m$ in diameter, while particles with diameter $< 0.1\mu m$ are called *ultrafine* particles.

Due to the gas, the particulate matter inherits a fluid-like motion. However, this motion does not entirely conform to the motion of a pure fluid; the small particles may be distributed in such fashion to affect the over-all motion. Since generation of gaseous phenomena in computer graphics not is aimed at creating an exact simulation, this behavior can be ignored without any visible loss.

The nature of gaseous effects have long been an area of active research; many new techniques to optimize or approximate the heavy calculations involved, or to generate visually more convincing renderings, appears every year. The following sections will try to summarize available basic concepts for achieving these effects, and point out some of the important work that has been done so far in the area – a broad field with many different approaches.

2.1 Sprites

Camera facing sprites, or billboards, has long been used in computer graphics, both for real-time applications and for feature films. Having textured planes, with their normals always facing the camera, present a fast way of compositing

effects into a filmed or computer generated scene. These planes can be textured with e.g. turbulence functions or any other computer generated material, or they can contain filmed elements of staged effects.

Sprites are fast to render and widely used in the VFX industry, but they have many limitations. If filmed elements should be used, there are practical issues in capturing the wanted effects. Furthermore, the composited sprites are unable to interact with the environment; this goes both for the movement of the effects and for the interaction with the light. Another problem is how to create a scene where the camera is moving around the effects, since the sprites only are supposed to be seen from one direction. The resolution of the sprites may also present a problem when dealing with large-scale effects where fine detail is needed.

2.2 Turbulence functions

The use of turbulence functions was first introduced in computer graphics by [Perlin, 1985], and since then the concept has been extensively used to achieve a large set of effects in the area.

Turbulence functions are seldom used on their own; they are most often used in conjunction with other concepts to create effects, or to add extra complexity to other methods. One common use is to combine turbulence functions with metaballs, as described by [Perlin and Hoffert, 1989], to form hypertextures. Metaballs, or implicit surface models, was introduced in computer graphics as *algebraic surface drawing* in [Blinn, 1982] to model molecular structures. Later on, [Wyvill et al., 1986] proposed an efficient way of using the concept to model "soft" objects, such as fabrics, mud and water. A metaball is described as a volume density field around a center point, where the density is formulated as a function of the distance from this point, and where the density reaches 0 outside a specified radius of influence. By modulating this density function with a three-dimensional turbulence model, interesting and complex volumetric effects can be modeled.

The use of turbulence functions results in very fast calculation times, and since the functions are described for the continuous space even the smallest details can be incorporated efficiently. The continuous density also makes rendering intuitive, with density at arbitrary points evaluated on the fly. However, it is difficult to control the behavior of the turbulence, and to achieve the desired shapes and motions; some way of controlling the over-all motion is likely needed. Another problem is the lack of connection to the expected physical behaviors of fluids, and the simulations may therefore not be believable.

2.3 Particle simulations

The modeling of particle systems for use in computer graphics origin in [Reeves, 1983], where stochastic processes are used in a model to simulate the movement

of the particles, with emission and extinction of particles as part of the simulation. The particles are rendered as point lights sources to avoid expensive calculations of shadows and scattering, restricting the use to emissive effects, and they are unable to interact with the environment. Although this was the first attempt to simulate the behavior of gaseous phenomena using particles, particles have been used in [Csuri et al., 1979] to model and render a column of smoke. They describe an approach where the intensity for rendering at a position (x, y, z) is determined by the density of the particles at this point.

While particles are fast to simulate, they are not very intuitive to render correctly due to the discontinuous representation. To render the particle volume, incorporating volume effects such as multiple scattering and shadowing, a three-dimensional density function is needed; this can be realized by instantiating a metaball at every particle position, but when many particles are used this may be very slow.

Pure particle simulation also has the problem that the details are restricted to the motion of the particles, which makes capturing small-scale details problematic; to simulate these details for large-scale effects, an enormous amount of particles is needed.

2.4 Voxel grids

To store a three-dimensional field, such as a density function, a voxel grid is commonly used. The voxel grid represent a discretization of space, where each bucket, or voxel, can hold a set of properties, such as density, velocity, texture coordinates etc. A more in-depth description of voxel grids can e.g. be found in [Wrenninge et al., 2010].

In contrast to particle data, voxel grids offer easy and fast access to data for arbitrary positions in space. This can be done using some interpolation techniques to store and lookup in continuous space, or by just associating positions with their nearest voxel.

Voxel grids are often used in medical visualization, where the data could come from e.g. a CT scan, but in computer graphics the data has to be created. This modeling can be done in many ways, for example using fluid dynamics, by splatting data into the grid from some geometry etc.

One problem when using voxel grids is memory; if large-scale volumes should be stored with high detail, large grid sizes are needed, resulting in significant memory usage. Consider for example a grid resolution of $2048 \times 2048 \times 2048$ voxels, where only one 4 byte float value is stored at each voxel, which results in a memory usage of $\approx$ 34GB. This is clearly a problem, especially when other properties also need to be stored at the voxels. Sometimes this is also a waste of memory, since there may be voxels that contain meaningless data that does not contribute to the volume representation. To overcome this problem to some extent, there have been alternatives proposed involving sparse data structures, for example the open source alternative Field3D[1] mentioned in [Wrenninge et al.,

[1] http://field3d.googlecode.com

2010].

2.5 Fluid dynamics

The dynamics of fluids is a well-studied area, where computational methods have been known for long. The famous Navier-Stokes equations were formulated by Claude-Louis Navier in 1822, and by George Stokes in 1845. These equations have been extensively used in many areas, to model behaviors such as wind flow in aircraft design, blood flow, movement of currents in oceans, and many more.

Computational fluid dynamics (CFD) in the context of computer graphics, distinguishes itself from many other areas in that it does not seek to find an accurate solution to the fluids flow; since the movement of the fluid is difficult to predict, an approximation that the human eye is unable to separate from an accurate flow is enough. The use of CFD in computer graphics dates back to [Kajiya and Von Herzen, 1984], but with the computational power available at that time the results were very limited in resolution.

In CFD, the Navier-Stokes equations are used both in Lagrangian approaches, where the motion of the particles in a particle system is controlled, and in Eulerian approaches to solve for a fluid flow in a grid representation.

In the first use of CFD simulations since [Kajiya and Von Herzen, 1984], except for some attempts in two dimensions, an Eulerian approach was taken by [Foster and Metaxas, 1996, 1997]. Their method solves for the Navier-Stokes equations, but is only stable for small time steps, which means long simulation times. An efficient and unconditionally stable solver was introduced in [Stam, 1999] which meant that large time steps could be used. However, the simulation does not capture the small-scale movements of the fluid since it suffers from numerical dissipation. To overcome this problem [Fedkiw et al., 2001] introduced a vorticity confinement step where lost energy is injected back into the simulation.

The use of the Navier-Stokes equations to simulate particle systems origins in [Gingold and Monaghan, 1977] where the smoothed particle hydrodynamics (SPH) concept was introduced for astrophysical calculations. The first use of SPH in computer graphics was described in [Desbrun and Cani, 1996], and since then the method has drawn much attention in computer graphics research. In recent research there has been a growing interest in parallel implementations of SPH for particle systems interest, achieved by decoupling the particle calculations. Since the processing power of graphics processing units (GPUs) has increased rapidly in later years, this is an interesting area that allows for fast simulations and where many real-time systems has been presented.

The obvious advantage with CFD is the simulations relation to the real flow of a fluid, and using some of the many approximations for solving the Navier-Stokes equations one can get simulations that, at least to the human eye, looks accurate. However, since the fluid simulations are physically based, they can

be hard to control, and it can therefore be difficult to get the wanted effects. One of the most serious problems is also the calculation times and the memory usage; if small-scale movements need to be captured in large-scale effects, large grid sizes or many particles, are needed, which is expensive.

2.6 Hybrid methods

Combinations of the methods described above are often used to benefit from their different advantages, and thereby achieve greater realism, better control, efficiency etc.

2.6.1 Particles and sprites

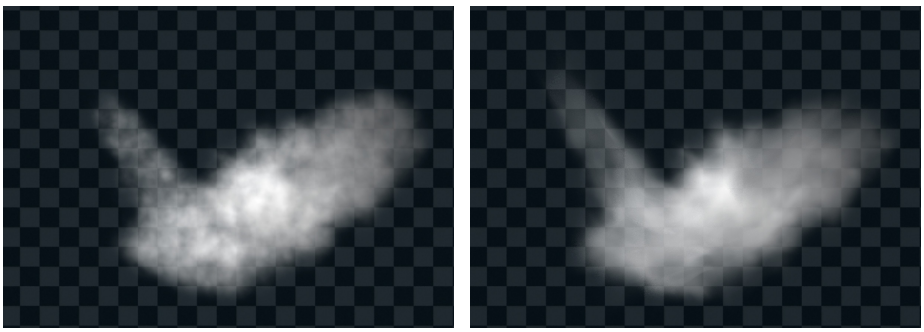
Sprites are commonly used together with particle simulations to visualize the particles, which successfully can create effects of gaseous phenomena given the right setup. This technique has been used in many feature films, e.g. for the fire on the Balrog creature in "The Lord of the Rings" trilogy [Aitken et al., 2004], where 6000-30000 particles in each frame were rendered as sprites with filmed elements of fire composited onto the plate.

2.6.2 Particles and turbulence

Since particle systems by themselves cannot capture the fine simulation movements with a reasonable amount of particles, they are often combined with techniques to add extra detail onto the simulation. This can for example be done by instantiating a hypertexture at each particle position during rendering.

Figure 2.1 shows an example of a rough particle simulation where the fine details are created using hypertextures. Each particle is assigned a local coordinate system oriented along the particles velocity vector, and the lookups in a turbulence function are done in this frame of reference. This locally oriented noise technique enables the turbulence to follow the translations and rotations of the individual particles in the simulation. It furthermore allows for easy implementation of motion blur, as can be seen in Figure 2.1(b), where a particles position is randomly selected multiple times along its velocity vector.

The technique presents a fast and easy way of adding detail, but suffers from several limitations. It cannot add new movement outside the radii of the particles, which makes it difficult to achieve the fine details wanted in the simulation. New particles can be advected through the simulation to create such effects, but this will in the limit of high detail contradict the goal of a restriction in the number of particles – the rendering time using a hypertexture for each particle position increases exponentially with the number of particles. Additionally, since this ad-hoc technique does not have a physical connection, the calibration may be extensive and not very straight-forward, and the detail does not capture a realistic turbulent flow associated with fluids. One would



(a) Without motion blur (b) With motion blur

Figure 2.1: Locally oriented noise simulations.

like to have dynamically simulated behavior in the added detail, which is the reason for the development of the method described in this thesis work.

An alternate – but somewhat related – approach was described by [Schechter and Bridson, 2008]. They proposed a method to bridge the gap between a pure turbulence model and dynamic simulations, in an attempt to generate more believable fine details. Their work introduced a turbulence model in which the kinetic energy and the net rotation are tracked per octave during simulation, and these entities are used to control a flow noise turbulence function. This model is used to add details to a rough particle simulation, in a similar way as the locally oriented noise simulation above. The method is capable of generating fine details that are somewhat related to fluid motion, but the simplistic approach may fall short in many situations, and the simulation still heavily relies on the input particle simulation.

2.6.3 Particles and grids

In [Selle et al., 2005], a method is described that performs a traditional grid-based Navier-Stokes solve, but where additional forces are added using a particle based approach. These forces are added to achieve highly turbulent phenomena, such as explosions. The method mostly focus on incorporating the turbulence, and small-scale effect effects still remains a problem since they are limited by the base resolution of the simulation grid.

The combination of particles and grids can take many forms. Particles can be used to control the over-all motion of a grid-based fluid simulation to make it more controllable, but the problems with costly details still remains. The combination can also be in that detail is added to a rough particle simulations by performing a grid-based fluid simulation, such as in [Horvath and Geiger, 2009].

2.6.4 Grids and turbulence

To overcome the problem with simulation resolution in grid-based CFD, extra detail can be added to a low-resolution simulation by means of a turbulence function. This concept was presented in [Stam, 1999] and refined in [Neyret, 2003], where a procedural texture is advected through a grid-based simulation, thereby adding fine details. The technique described by [Neyret, 2003] is only described for the two dimensional case, and it suffers from the same problems as when using turbulence to add detail to particle simulations.

2.6.5 Particles, grids and turbulence

In [Rasmussen et al., 2003], a technique is described which combines several concepts to achieve detailed simulations of large-scale phenomena. The technique uses a set of 2D simulations, from which a 3D velocity field is created by sweeping the 2D planes through space. To break up some of the artifacts in this 2D to 3D transformation, a Kolmogorov spectrum [Stam and Fiume, 1993] is used. Finally, the velocity field is used to advect particles, which are rendered volumetrically by means of rasterization. The technique is capable of generating highly detailed effects with the high-resolution simulations governing the behavior, but it is restricted to effects that are symmetric by nature and thus can be described as swept planes, and it suffers from the problems of rendering particles.

The technique to which the work in this thesis is most closely related, is as earlier mentioned described in [Horvath and Geiger, 2009], where a rough particle simulation governs the over-all motion of the simulation, while high resolution 2D simulations are used as a refinement method. Together with texture data, the slice refinements are blended into a highly detailed result. The use of this technique overcomes many of the problems described in previous sections, but it is limited to emissive effects with its simplistic form of rendering.

The approach taken in this thesis work can be seen as an extension of the slice technique in [Horvath and Geiger, 2009]; an extension made in order to enable volume rendering for non-emissive volumes. It combines many of the earlier mentioned techniques in an attempt to overcome the shortcomings inherent when using them separately.

Chapter 3

Theory

As described in Section 1.3, the method developed incorporates a number of concepts to achieve the objectives in Section 1.2. These concepts will be treated in the following sections and include: *a*) grid-based *fluid dynamic* simulation for the refinement stage; *b*) *volume rendering* methods in the rendering of the simulations, and in particular the *Reyes* rendering algorithm; *c*) *deep shadow maps* to accomplishe effective shadowing of the volume; and *d*) a turbulence function to break up some of the interpolation artifacts when transforming the set of two-dimensional simulations to three dimensions. The input to this turbulence function is texture coordinates that have been advected through the simulation, thus making the turbulence conform to the movement of the fluid.

3.1 Fluid dynamics

The important transformation of the Navier-Stokes equations to an approximation suitable for computer graphics, can be found in [Fedkiw et al., 2001; Foster and Metaxas, 1996, 1997; Stam, 1999]. For an in-depth explanation of the techniques the reader is referred to that literature, and in particular the introduction of the *Stable Fluids* concept in [Stam, 1999]. A brief description of the theory behind grid-based fluid dynamics, according to the stable fluids approach, is given next.

Using the Navier-Stokes equations in Equation (3.1), a fluids flow is determined by a velocity field $\mathbf{V}$ and a pressure field p, where the fields could be defined on either two-dimensional or three-dimensional discretized grids. In Equation (3.1b), ν corresponds to the kinematic viscosity of the fluid, and ρ is the density. The force $\mathbf{F}$ describes an external force acting on the fluid, which could be gravity, wind, or any other meaningful quantity influencing the flow.

The operator ∇ is the vector of partial spatial derivatives, $\nabla = (\partial/\partial x, \partial/\partial y)$ (in two dimensions). By enforcing the condition in Equation (3.1a) using the dot product of spatial derivatives the solution is ensured to be divergence free,

which implies an incompressible, volume conserving fluid.

$$\nabla \cdot \mathbf{V} = 0 \tag{3.1a}$$

$$\frac{\partial \mathbf{V}}{\partial t} = \mathbf{F} - (\mathbf{V} \cdot \nabla)\,\mathbf{V} + \nu \nabla^2 \mathbf{V} - \frac{\nabla p}{\rho} \tag{3.1b}$$

The problem now comes down to, given an initial state for the velocity and pressure fields at a time t, to solve for the state at time $t + \Delta t$. To do so, the different parts constituting Equation (3.1) can be solved in individual steps using an operator splitting. This splitting is depicted in Equation (3.2), and allows for a relatively easy way to approximate a solution as compared to finding a direct solution to the Navier-Stokes equations.

The outline for determining the state at $t + \Delta t$ is now performed as follows, in sequential order where the input to the next step is the result of the last step: *a*) add *external forces* to the flow; *b*) advect the flow by itself, that is solve for the *self-advection*; *c*) perform a *diffusion*, which solves for the internal frictions in the flow, governing the viscosity; *d*) *project* the velocity field onto its divergence free part using the condition in Equation (3.1a), thereby making the fluid incompressible.

$$\mathbf{V}_t \xrightarrow[\mathbf{F}]{\text{Add force}} \mathbf{V}_1 \xrightarrow[(\mathbf{V}\cdot\nabla)\mathbf{V}]{\text{Advect}} \mathbf{V}_2 \xrightarrow[\nu\nabla^2\mathbf{V}]{\text{Diffuse}} \mathbf{V}_3 \xrightarrow[\frac{\nabla p}{\rho},\nabla\cdot\mathbf{V}]{\text{Project}} \mathbf{V}_{t+\Delta t} \tag{3.2}$$

The result after the last step in Equation (3.2) is the sought solution of the flow after one time step, and the solution at any, in steps Δt discretized, time can be found by performing a number of such calculation chains.

3.1.1 External forces

The external forces calculation step in Equation (3.2) is the easiest term to solve for. To incorporate the influence of the forces, a simple first order Euler time integration scheme can be used as approximation, which is shown in Equation (3.3). This is a reasonable approximation as long as the forces can be assumed to not vary considerably during one time step.

$$\mathbf{V}_1 = \mathbf{V}_t + \Delta t\,\mathbf{F} \tag{3.3}$$

3.1.2 Self-advection

The term $-(\mathbf{V} \cdot \nabla)\mathbf{V}$ makes the Navier-Stokes equations non-linear, and describes how the fluid is advected, or convected, by its own flow. That is, given a disturbance in the velocity field, this term accounts for the propagation of this flow through the fluid, creating effects such as vortices and similar.

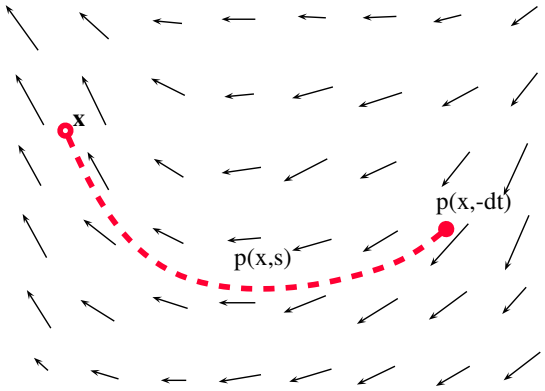


Figure 3.1: Self advection term solved by tracing backward in the velocity field, along the curve $\mathbf{p}(\mathbf{x}, s)$

For the self-advection, the velocity $\mathbf{V}_2$ need to be solved for in the partial differential equation in Equation (3.4). This can be done using a finite differencing method as described by [Foster and Metaxas, 1996, 1997], but the method is only stable when the time step is less than the fraction of the grid spacing to the velocity magnitude, $\Delta t < \Delta x / |\mathbf{V}|$. This makes the approach very slow with fine grids and large velocities, so [Stam, 1999] proposed using an unconditionally stable method to solve for the self-advection.

$$\frac{\partial \mathbf{V}_2}{\partial t} = -(\mathbf{V}_1 \cdot \nabla)\mathbf{V}_1 \tag{3.4}$$

The method described by [Stam, 1999] uses the *method of characteristics* to solve the partial differential equation. As the self-advection accounts for the motion of the velocity field with itself, it can be modeled as a simple advection through the velocity field. This is accomplished by back-tracing along the field line from the current position, to the position at $t - \Delta t$, where the velocity at the new position is used as the advected velocity $\mathbf{V}_2$. If the field line is defined as $\mathbf{p}(\mathbf{x}, s)$, see Figure 3.1, the self-advection step can be formulated as in Equation (3.5).

$$\mathbf{V}_2(\mathbf{x}) = \mathbf{V}_1(\mathbf{p}(\mathbf{x}, -\Delta t)) \tag{3.5}$$

This method of solving for the self-advection assumes a time independent velocity field, so regardless of parameters it will always be approximate. However, since the new velocity comes from the velocity field itself, it can never increase to a magnitude larger than the largest in the field, making the solution unconditionally stable. This property of the method, together with the fact that it is easy to implement, makes it an attractive way for solving the self-advection.

3.1.3 Diffusion

The term $\nu\nabla^2\mathbf{V}$ describes the effect of internal frictions, making the fluid flow with different viscosities for different values of ν. The viscosity is solved for using the partial differential equation in Equation (3.6), which describes a diffusion.

$$\frac{\partial \mathbf{V}_3}{\partial t} = \nu\nabla^2\mathbf{V}_2 \tag{3.6}$$

The diffusion equation can be solved in a straight-forward, explicit, manner using discretization of the diffusion operator ∇^2 and stepping through time; that is, a discretization in both time and space. This method however has the same drawbacks as finite differencing in the self-advection step, which makes it unstable for large time steps and large viscosities. [Stam, 1999] instead proposed using an implicit approach to achieve a stable solution. By only keeping the time discretized, one can arrive at the equation in Equation (3.7), where $\mathbf{I}$ is the identity operator. The equation results in a sparse linear system when the diffusion operator is discretized, which can be solved efficiently.

$$\frac{\mathbf{V}_3 - \mathbf{V}_2}{\Delta t} = \nu\nabla^2\mathbf{V}_2 \quad \Leftrightarrow \quad (\mathbf{I} - \nu\Delta t\,\nabla^2)\mathbf{V}_3 = \mathbf{V}_2 \tag{3.7}$$

3.1.4 Projection

The last term, $\frac{\nabla p}{\rho}$, in Equation (3.1b) is used together with Equation (3.1b) to enforce incompressibility of the fluid.

According to the *Helmholtz-Hodge Decomposition* theorem, a vector field can be decomposed into a curl free part ∇q and a divergence free part $\mathbf{U}$, as depicted in Equation (3.8). Since $\mathbf{U}$ is divergence free, it satisfies the condition in Equation (3.1a), $\nabla \cdot \mathbf{U} = 0$. To enforce incompressibility, i.e. a divergence free solution, it is thus sufficient to find the divergence free part of the Helmholtz-Hodge decomposition for the last steps result, $\mathbf{V}_3$. This means that, by substituting the term $\frac{\nabla p}{\rho}$ for the curl free part ∇q of the decomposition, the last term in Equation (3.1b) is accounted for in such fashion to make the final solution incompressible.

$$\mathbf{V}_3 = \mathbf{U} + \nabla q, \qquad \nabla q = \frac{\nabla p}{\rho} \tag{3.8}$$

The last step of solving the Navier-Stokes equations can be seen as the projection of $\mathbf{V}_3$ onto $\mathbf{U}$, hence the naming as the *projection step*. If this projection is denoted $\mathbf{P}$ and $\mathbf{U}$ is substituted for the sought solution $\mathbf{V}_{t+\Delta t}$, the calculation needed for the last step can be formulated as in Equation (3.9). Together with Equation (3.8) the projection operator can now be defined using ∇q.

$$\mathbf{V}_{t+\Delta t} = \mathbf{P}(\mathbf{V}_3) = \mathbf{V}_3 - \nabla q \tag{3.9}$$

The projection step now comes down to finding q in order to evaluate Equation (3.9). Since the solution $\mathbf{V}_{t+\Delta t}$ is divergence free, the divergence operator,

∇, can be used at both sides of Equation (3.9) to cancel it out. The result in Equation (3.10) is a Poisson equation where the velocity field $\mathbf{V}_3$ is known from the previous step.

$$\nabla \cdot \mathbf{V}_{t+\Delta t} = \nabla \cdot \mathbf{V}_3 - \nabla^2 q \quad \Leftrightarrow \quad \nabla \cdot \mathbf{V}_3 = \nabla^2 q \tag{3.10}$$

To solve the Poisson equation in Equation (3.10) it can, similar to the diffusion term, be discretized into a sparse linear system for which there are efficient algorithms to find a solution.

When solving the Poisson equation boundary conditions are also taken into account, making the fluid interact with objects inserted into to the flow. These conditions are enforced by ensuring that no flow is propagated into, or out from, the boundaries.

3.2 Volume rendering

Rendering of the propagation of light through a participating media is a computationally heavy task. Since the radiance for a given point and direction in a volume can be affected by all possible light paths through the media, rendering in a brute-force manner using ray-marching integration can take considerable time. This section will first review the different calculations involved in the volume rendering, followed by a description of the particular approach used in this thesis work to optimize the process.

3.2.1 Light transport in participating media

The following explanation tries to give an overview of the important theory of lights interactions with a participating media. For further detail, especially when it comes to the rendering aspects when dealing with these effects, the reader is referred to [Jensen, 2001; Max, 1995; Wrenninge et al., 2010], which gives good and comprehensive descriptions.

To render a scene containing a participating media, the radiance $L(\mathbf{x} \rightarrow \Theta)$ at position $\mathbf{x} = (x, y, z)$ in direction Θ reaching a pixel, need to be found for each pixel. This is an integral along the line of sight through the media, and on this line a number of events can take place due to the the small particles constituting the volume. The light entering the media can either be absorbed, scattered, or it can continue unaffected by the particles. Additionally, light can be added to the path from either spontaneous or stimulated emission from the particles. The different events that can take place are illustrated in Figure 3.2.

To find the probability of extinction, that is the blue light paths in Figure 3.2, for a light beam entering a participating media, one can consider the particles constituting the media. The distribution of particles, and their sizes, projected onto a cross-sectional area, gives the fraction occluded by particles and thus the probability of extinction. The loss in radiance due to this probability can be described using the differential equation in Equation (3.11), where $\sigma_e(\mathbf{x})$ is

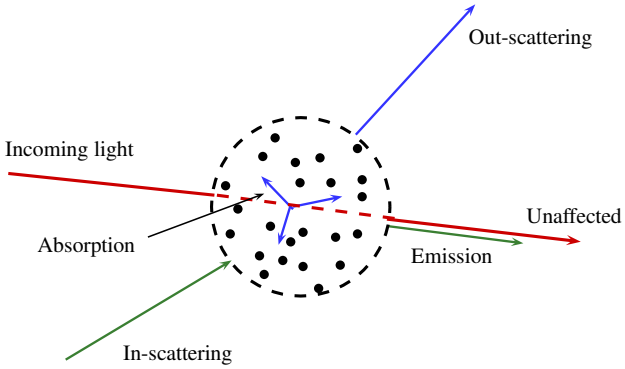


Figure 3.2: The different events light can undergo when interacting with a participating media. The radiance loss is illustrated in blue, while the radiance gain is in green.

a media specific extinction coefficient taking into account the properties of the particles.

$$\frac{dL(\mathbf{x} \to \Theta)}{ds}_{ext} = -\sigma_e(\mathbf{x})L(\mathbf{x} \to \Theta) \tag{3.11}$$

To model the light interactions with the media, the extinction coefficient is divided into one part representing the absorption, and one representing out-scattering. Equation (3.12) describes the radiance loss due to absorption while Equation (3.13) describes the loss from out-scattering. The absorption coefficient $\sigma_a(\mathbf{x})$ and the scattering coefficient $\sigma_s(\mathbf{x})$ then have to satisfy $\sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x}) = \sigma_e(\mathbf{x})$, and are also used to formulate the scattering albedo of the participating media, $\Omega = \sigma_s(\mathbf{x})/\sigma_e(\mathbf{x})$.

$$\frac{dL(\mathbf{x} \to \Theta)}{ds}_{abs} = -\sigma_a(\mathbf{x})L(\mathbf{x} \to \Theta) \tag{3.12}$$

$$\frac{dL(\mathbf{x} \to \Theta)}{ds}_{sca^-} = -\sigma_s(\mathbf{x})L(\mathbf{x} \to \Theta) \tag{3.13}$$

Due to the out-scattering events, radiance will also be added to a light path through the media, caused by such out-scattering events from other light paths. To formulate this in-scattering for a path $\mathbf{x} \to \Theta$, all incoming light to $\mathbf{x}$ ending up in direction Θ have to be calculated. An essential part of modeling a participating media now comes down to evaluating the fraction of light scattered from a given incoming direction Ψ at position $\mathbf{x}$, to the outgoing direction Θ. This fraction can be stated as in Equation (3.14), where $P(\mathbf{x}, \Theta \leftrightarrow \Psi)$ describes a so called *phase function*.

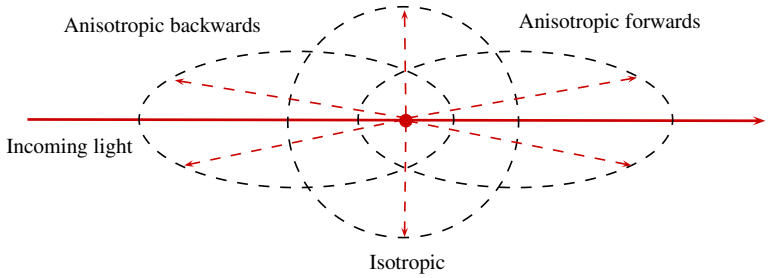


Figure 3.3: The difference between isotropic and anisotropic phase functions

$$P(\mathbf{x}, \Theta \leftrightarrow \Psi) = \frac{dL(\mathbf{x} \rightarrow \Theta)}{\frac{1}{4\pi}\int_{4\pi} L(\mathbf{x} \leftarrow \Psi)d\Psi} \tag{3.14}$$

The phase function models how light propagates through the media, and is crucial for good results. It controls if this propagation should be equal in all directions (isotropic scattering) or if the light scattering should be of higher probability in a certain direction (anisotropic scattering). The difference between isotropic and anisotropic scattering is illustrated in Figure 3.3, where the point in the middle of the light path represent a scattering event.

Using the phase function, the increase in radiance due to in-scattering can be stated as in Equation (3.15), where the integral is taken over the whole sphere surrounding $\mathbf{x}$. Since light cannot disappear or arise from the internal scattering events, the out-scattered light in the media must outweigh the in-scattered light. This is achieved by using the same scattering coefficient σ_s as for the out-scattering.

$$\frac{dL(\mathbf{x} \rightarrow \Theta)}{ds}_{sca^+} = \frac{\sigma_s(\mathbf{x})}{4\pi}\int_{4\pi} L(\mathbf{x} \leftarrow \Psi)P(\mathbf{x}, \Theta \leftrightarrow \Psi)d\Psi \tag{3.15}$$

The final form of phenomena governing the terminal radiance after interaction with the media is emission, which is given in Equation (3.16). Here L_b is the blackbody temperature of the media, and due to temperature equilibrium the amount of emitted light is described using the absorption coefficient $\sigma_a(\mathbf{x})$.

$$\frac{dL(\mathbf{x} \rightarrow \Theta)}{ds}_{em} = \sigma_a(\mathbf{x})L_b(\mathbf{x}) \tag{3.16}$$

Putting Equation (3.11) through Equation (3.16) together, the change in radiance per unit distance at a point $\mathbf{x}$ in direction Θ can be stated using the differential equation in Equation (3.17).

$$\begin{aligned} \frac{dL(\mathbf{x}\rightarrow\Theta)}{ds} &= \sigma_a(\mathbf{x})L_b(\mathbf{x}) \\ &- \sigma_e(\mathbf{x})L(\mathbf{x}\rightarrow\Theta) \\ &+ \frac{\sigma_s(\mathbf{x})}{4\pi}\int_{4\pi} L(\mathbf{x}\leftarrow\Psi)P(\mathbf{x},\Theta\leftrightarrow\Psi)d\Psi \end{aligned} \tag{3.17}$$

By finally integrating both sides of the expression in Equation (3.17) over a distance s, the volume rendering equation can be derived according to Equation (3.18), where the last term represent the light entering the volume. In this equation the term $\tau(\mathbf{x},\mathbf{x}') = \int_{\mathbf{x}}^{\mathbf{x}'}\sigma_e(t)dt$ describes the optical depth of the media; that is, the transparency between $\mathbf{x}$ and $\mathbf{x}'$ is given by $e^{-\tau(\mathbf{x},\mathbf{x}')}$.

$$\begin{aligned} L(\mathbf{x}\rightarrow\Theta) &= \int_0^s e^{-\tau(\mathbf{x},\mathbf{x}')}\sigma_a(\mathbf{x}')L_b d\mathbf{x}' \\ &+ \int_0^s e^{-\tau(\mathbf{x},\mathbf{x}')}\sigma_s(\mathbf{x}')\int_{4\pi} L(\mathbf{x}\leftarrow\Psi)P(\mathbf{x},\Theta\leftrightarrow\Psi)d\Psi d\mathbf{x}' \\ &+ e^{-\tau(\mathbf{x},\mathbf{x}+s\Theta)}L((\mathbf{x}-s\Theta)\rightarrow\Theta) \end{aligned} \tag{3.18}$$

Using ray-marching, the volume rendering equation can be evaluated recursively by approximating it with a Riemann sum. The multiple scattering term, Equation (3.15), inside this sum can also be evaluated recursively in the same manner. This means a recursion in an already recursive evaluation, and calculation of the volume rendering equation is obviously very demanding using such numerical schemes. The multiple scattering term is therefore often excluded in the calculations to get reasonable rendering times. However, this is not always a good approximation since this term can add considerable realism to the rendering, including the important effects of shadows.

3.2.2 The Reyes rendering algorithm

The heavy calculations involved in evaluating the volume rendering equation in Equation (3.18) makes optimizations and other approaches an area of active research. In [Cook et al., 1987], a technique was introduced for efficient rendering of photo-realistic images of complex scenes; a technique that employs vectorization and parallelism to render geometry by reducing it to so called micropolygons. The resulting renderer was given the name *Reyes* and it is currently used in Pixar's RenderMan[1] and SideFX's Mantra[2] among other rendering environments. It was recently extended to microvoxels to enable rendering of participating media in [Clinton and Elendt, 2009], and in [Zhou et al., 2009] a GPU implementation was proposed that enables interactive Reyes rendering for moderately complex scenes and is one order of magnitude faster than the implementation used in RenderMan.

The original Reyes polygon rendering concept will be described in this section, and volume rendering is done analogously.

[1] https://renderman.pixar.com
[2] http://www.sidefx.com

In order to overcome the problems of model complexity, shading complexity and speed, the Reyes algorithm introduced in [Cook et al., 1987] was developed to produce photo realistic renderings with minimal ray tracing. It decouples shading and image sampling by tessellating the geometric primitives into so called micropolygons, and utilizes parallel structures by vectorizing computations on those polygons. Non-local calculations are further approximated with extensive use of textures for efficiency.

The Reyes renderer was developed with the following goals:

- The renderer should be able to render arbitrary complex environments.
- A large amount of geometric primitives should be supported.
- Programmable shaders should be used to model complex surfaces.
- The renderer should use minimal amount of ray tracing since it is considered costly.
- The renderer should be able to render a two hour movie in one year assuming 24 frames per second.
- Image artifacts such as aliasing and Moiré patterns are not acceptable.
- The renderer should be flexible enough to be able to incorporate other rendering techniques.

Micropolygon rendering

To achieve the goals stated above, the design of the Reyes renderer is based on the use of some fundamental principles: *a*) first the calculations should be performed in a coordinate system suited for the calculation in question; *b*) the calculations should further be vectorized and parallelized to be able to calculate in a SIMD (single instruction, multiple data) fashion; *c*) the calculations should also be made locally on a common representation, which is the micropolygon primitive; *d*) the time to render should scale linearly with the size of the scene, and there should be no limit on the number of geometric primitives in the scene; *e*) to satisfy the goal of flexibility there should be a back door enabling other techniques in the rendering pipeline; and *f*) the texture accessing should be efficient since it should be extensively used to incorporate global effects, shadows, displacements etc.

The core in the Reyes algorithm is the use of micropolygons. These are created by tessellating, or dicing, the geometry into quadrilaterals of size approximately equaling 0.5 pixels, which is the Nyquist limit to avoid aliasing. The dicing of a primitive is stored in a grid and is done so that the micropolygons have sides parallel to the the texture coordinates, which makes texture lookups fast. The dicing is done in eye space with an estimate of the primitive's size on screen to decide the size of the micropolygons. The micropolygons in a grid are shaded together before any visibility calculations are done. This means that micropolygons that are occluded also are shaded, but the advantages of vectorization, use of efficient texture lookups for large blocks etc. makes this a reasonable trade-off. Additionally, since shading is done before perspective

transformations and the micropolygons are small enough to ignore perspective, costly inverse perspective calculations are not needed.

Algorithm outline

Primitives are processed one by one, which means that only the current object needs to be loaded into memory. The bounds of the primitive in screen space are determined. If the primitive is outside the viewing frustum or outside a hither plane, i.e. to close to the eye, it is culled. If the primitive spans a so called ϵ plane close to the eye and the hither plane, it is split into a set of primitives. This is repeated until no primitives span both these planes and the primitives outside the hither plane are culled. Primitives are also split if they are to large to be diced or if the primitive type is undiceable. This process is repeated until the primitives are diceable, which means that if an primitive is undiceable the split primitives must at some point be diceable.

The split primitives are loaded into memory one by one, and the current primitive is diced into a grid of micropolygons. The micropolygons in a grid are treated in parallel, and for each normals and tangent vectors are calculated, after which they are shaded. The shaded micropolygon grid is then broken into micropolygons so that each micropolygon can be checked. If a micropolygon is outside the hither plane or outside the screen space bounds it is culled, otherwise the z-depth of the polygon is compared with the z-value in the buffer and the sample in the buffer is replaced if it is smaller.

To remove aliasing, jittering – a form of stochastic sampling – is used where pixels are divided into a number of subpixels with positions randomly chosen.

Microvoxel rendering

Until recently volumetric effects were calculated using additional ray marching with the Reyes algorithm. Instead of this ray march, [Clinton and Elendt, 2009] extended the micropolygon concept to include microvoxels – the three-dimensional analogue to micropolygons. The dicing and shading is done in a similar manner to the 2D case but extended to 3D. The use of microvoxels is also developed to be flexible so that ray marching can be used to extend the rendering with further volumetric effects.

3.2.3 Deep shadow maps

Shadowing, both from a participating media onto surrounding environment and self-shadowing within the media, are very important to incorporate to achieve realism when rendering. As mentioned in Section 3.2.1, these effects are not accounted for when approximating the volume rendering equation by excluding the multiple scattering term.

To include shadows without having to resort to solving the expensive multiple scattering integral, separate ray marches can be performed. These are only done towards the light sources in the scene, to get an approximation of

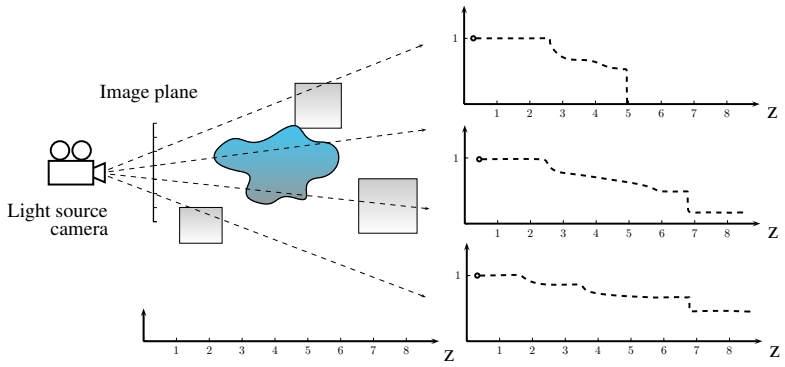


Figure 3.4: Example of visibility functions in a deep shadow map, where the object in the middle represents a participating media.

the amount of occlusion at the shading point. Using this technique, however, rendering can also take considerable time since costly ray marches have to be performed for each shading sample.

A cost-efficient, but approximate, solution to shadows is the use of *shadow maps*, which was introduced in [Williams, 1978]. Shadow maps have been extensively used within computer graphics, but they are restricted to the use in simple scenes not containing any translucent materials. [Lokovic and Veach, 2000] extended this concept to *deep shadow maps*, which can handle translucency, participating media and complex scenes. This section will explain how this technique enables shadowing of participating media.

Shadow maps uses the light sources in the scene as starting point to test the scene for occlusion, simply by rendering the z-depth from this point of view in a preprocessing step. In the subsequent rendering from the camera, this rendering, or shadow map, can be used as lookup. By comparing the depth stored in the map with the distance between the light source and the shading point, a fast evaluation can be made to see if the point is occluded or not.

Deep shadow maps uses a similar approach as regular shadow maps, but instead of storing a depth value for each pixel in the map, it stores a visibility function of the z-depth. This visibility function for a pixel, describes for each z-depth, how much light is carried through from the light source in question to that depth. A visibility value of 1 means full visibility, while 0 is no visibility. The visibility functions for three pixels are illustrated in Figure 3.4, where different events take place in the interaction with the opaque rectangles and the participating media in the middle.

To find the visibility function, the concept of transmittance functions $\tau(x, y, z)$ is considered. A transmittance function is calculated by performing a ray march from a position in the image plane at the light source camera, and for each step store the fraction of initial power remaining; that is, the transmittance. Using

such transmittance functions the visibility at a depth z for the pixel (i, j) can be calculated using a two-dimensional integration with some filter $f(s, t)$, as shown in Equation (3.19), where a filter radius of r is used.

$$V_{i,j}(z) = \int_{-r}^{r} \int_{-r}^{r} f(s,t)\tau(i + 0.5 - s, j + 0.5 - t, z)dsdt \tag{3.19}$$

By using a set of jittered positions for each pixel in the image plane, a sampled approximation of Equation (3.19) can be calculated for each pixel and depth by means of a simple summation.

One problem when using deep shadow maps is storage since a discretized function need to be stored for each pixel in the map. However, if an error threshold ϵ is defined, the visibility functions can be compressed to use less discretization points in such manner to still be within ϵ units from the original function. In this way an arbitrary accuracy and compression can be used, given a certain discretization.

Deep shadow maps present an elegant and efficient way of incorporating shadows in complex environments. Comparing the technique to regular shadow maps, deep shadow maps require considerably lower resolution to achieve the same shadowing quality, which is an effect of the pre-filtering process. This, and the depth feature itself, comes to the cost of significantly longer calculation times, but compared to standard ray marching to test for occlusion, this is a really fast technique. One of the largest disadvantages in using deep shadow maps are sensitivity to artifacts. These may arise due to e.g. resolution or the filter radii in the visibility calculations since the filtering assumes the same z-depth over the whole filter region.

3.3 Turbulence textures

Synthetic turbulence, as introduced in [Perlin, 1985], has been extensively used in the computer graphics area to model complex natural phenomena in a controllable way. This *Perlin noise* algorithm generates a pseudo-random gradient noise value given an input position, where pseudo-random means that for multiple noise lookups at the same position $\mathbf{x}$, the same result $S(\mathbf{x})$ is generated. Gradient here meaning that a smooth interpolation is done between a set of regularly spaced noise values. In one dimension this means an interpolation between two such values, in 2D between four values, and in N dimensions between 2^N values.

While the original Perlin noise enables fast generation of detail, it has some limitations. For example, since it is based on an interpolation from grid points in a space spanned by 2^N corners in N-dimensional space, the noise lookup complexity increases exponentially with the number of dimensions, resulting in poor scalability to higher dimensions. To overcome such limitations the concept of *simplex noise* was introduced in [Perlin, 2001] where the noise lookups are based on a space spanned by the simplest shapes possible – simplexes. In two

dimensions this means triangles, in three dimensions tetrahedrons, and in N-dimensional space shapes with $N+1$ corners. Thus, the calculation complexity is reduced to $O(N)$ instead of $O(2^N)$ as for the classic Perlin noise. For a good, more in-depth, explanation of simplex noise the reader is referred to [Gustavson, 2005].

Using a noise generator, such as simplex or Perlin noise, complex textures can be created by combining noise at different frequencies. In this way the texture will have features of different sizes, where the low frequencies can be made to control the over-all shape with larger magnitudes. This technique of adding noise at increasing frequency and decreasing amplitude can be formulated as in Equation (3.20), where $S(\mathbf{x})$ is the noise generator, and it creates an approximation of *fractal Brownian motion* (fBm) which is very useful for many different effects.

$$T(\mathbf{x}, t) = \sum_{i=0}^{O} (\frac{g}{2})^i \, S(l^i \, F \, \mathbf{x}, t) \tag{3.20}$$

In the fractal turbulence function definition $\mathbf{x}$ represent the input position for lookup, while the time t adds an extra dimension to move the turbulence through time. The amplitude of the different octaves in the noise summation is governed by the gain parameter g, which most often is set to 1 to reduce the amplitude by half for each octave. The frequency multiplier l, referred to as *lacunarity*, controls the size of "gaps" in the texture pattern and is usually set to a value around 2 to make the frequency of an octave the double of the previous. And while the lacunarity is used to control the change in frequency, F specifies the starting frequency for the noise summation.

Chapter 4

Technique

As stated in Section 1.3, the steps involved in simulating and rendering gaseous phenomena using the developed method are:

- **Splatting procedure**: Splat particle simulation onto view-aligned slices.
- **Simulation refinement**: Simulate slices.
- **Density function**: Transform slices to a 3D density function.
- **Rendering**: Render density function.

The different steps will be explained in detail in the following sections. To see a more organized description of the general outline of the developed method, it is given in a pseudo-code fashion in Appendix A.

The fundamental principle of the method is illustrated in Figure 4.1, which shows how the simulation volume is discretized into a set of view-aligned slices on which two-dimensional simulations are performed. In reality the number of slices to choose depends on the volumes complexity in the cameras viewing direction, but a common number is for example 64 or 128 slices.

4.1 Splatting procedure

The splatting of the input particle simulation onto view-aligned planes is an important step in the method developed. It is this procedure that enables grid-based simulations of particle data, and the over-all outcome is highly dependent on how this is done.

4.1.1 Bounding box definition

To enable the splatting, a simulation volume must be defined; that is, a bounding box encompassing all particles participating in the simulation. To find this

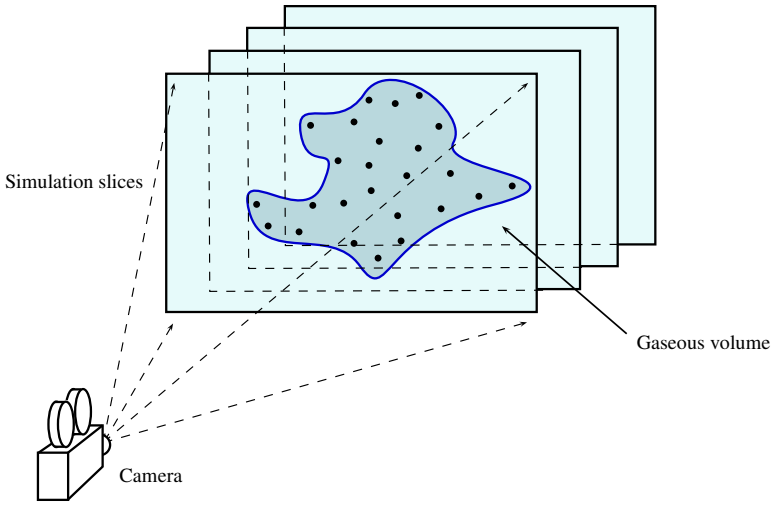


Figure 4.1: Simulation principle

volume all input particles in every frame are traversed and checked if being positioned outside the current bounding box, and if so it is extended. This process can take some time if there are many frames and particles, so an estimation of the bounding box can be specified manually, but generally the traversing of particles is very fast.

The formulated bounding box is then used together with the camera plane to define a transformation matrix from world space, where particles are defined, to simulation space. The simulation space is defined in such fashion that coordinates are in the range $(x, y, z) \in]0, 1[$ within the simulation space bounding box and 0 or 1 on its surface. The simulation space bounding box is found – as illustrated in Figure 4.2 – by formulating a volume encompassing the world space bounding box and which is aligned with the camera coordinate system. As shown in the figure, the new bounding box formulation could result in a volume that is not as tightly fitted to the input particles as the original bounding box, but since the two-dimensional slices normally should have some extra space for simulation this is not really a problem. And to find the tightly fitted bounding box in simulation space, a transformation of each particle would be needed when traversing for their bounds, which would increase preprocessing time.

4.1.2 Particle splatting

The simulation planes can now be defined by dividing the camera z-axis into N discretization planes that span the (x, y)-plane, located between $z = 0 \ldots 1$. To minimize the memory usage and to enable parallelization these slices are

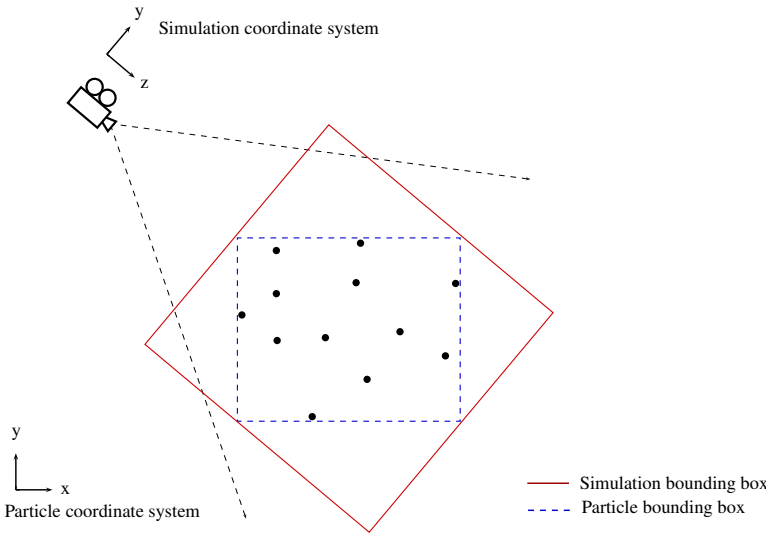


Figure 4.2: Bounding box transformation

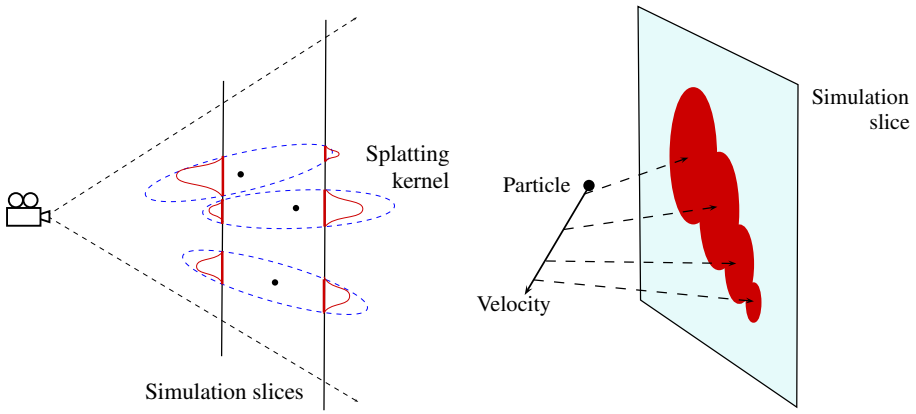
treated completely independently. For a specified slice all particles in a frame are transformed into simulation space and considered for splatting. The distance between the slice and a particle along the perspective direction – that is the vector originating at the camera position – is used in a Gaussian kernel. If this kernel weight is above a certain threshold the particle is used for splatting onto the slice.

The principles of the splatting are demonstrated in Figure 4.3. When a particle is going to be splatted onto a slice, the projected position $\mathbf{x}_p$ along the perspective direction is used as center point for a new Gaussian kernel, where the kernel width depends on the radius of the particle. The grid cells within the radius of the kernel from $\mathbf{x}_p$ are traversed and assigned density from the splatting. This density depends on the product of the two Gaussian kernels and a normalization scaling, as formulated in Equation (4.1), where $\mathbf{p}$ is the original particle position. The normalization scaling ω_s, as formulated in Equation (4.2), is a heuristic scaling to account for splatting radius r_{xy} on the planes, splatting radius along the splatting direction r_z and the depth between slices Δz. This in order to get about the same over-all density on the slices regardless of the specifications of the parameters.

$$I(\mathbf{x}) = I(\mathbf{x}) + \omega_s \exp(-0.25 \frac{|\mathbf{p} - \mathbf{x}_p|^2}{(\Delta z\, r_z)^2} - 5 \frac{|\mathbf{x} - \mathbf{x}_p|^2}{r_{xy}^2}) \tag{4.1}$$

$$\omega_s = \frac{500 \exp(\Delta z^{-0.27})}{N_s\, r_{xy}^2\, r_z^{0.8}} \tag{4.2}$$

In Equation (4.2), N_s denotes the number of consecutive splats a particle



(a) Splatting kernels (b) Multiple splatting

Figure 4.3: Particle splatting.

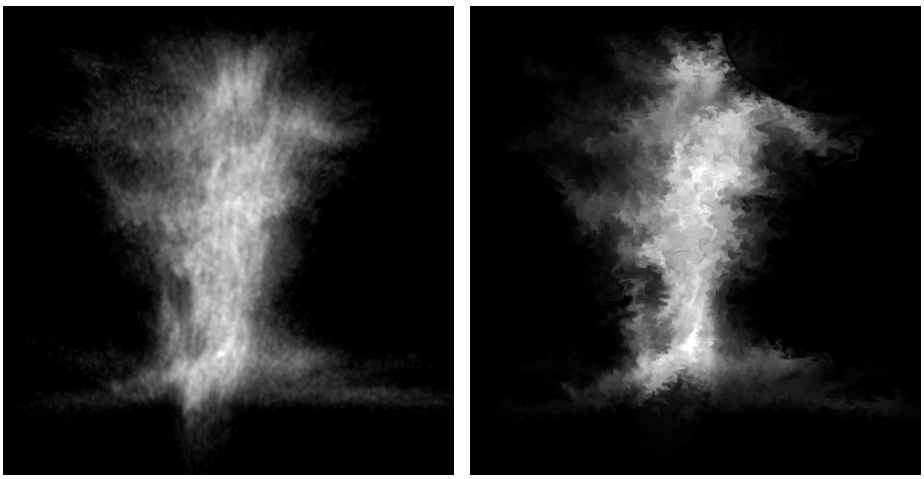
should undergo. This multiple splatting, as illustrated in Figure 4.3(b), is done at random positions along the particles velocity vector and it tends to smooth out the projections, making it more difficult to distinguish individual particles in the two-dimensional density functions. It also tries to emphasize the movement of the particles, which is especially important if the particles are moving in the direction of the camera z-axis.

The velocity at each grid cell is also specified for the simulation of a slice. This is, however, done in a different manner than for the density since a splatting according to the one explained above would smooth out the velocities to much, resulting in loss of fine, small-scale movements in the simulation. Instead the velocity at a grid cell is defined by accumulating a particles velocity to the cell if the particle is closer to the grid cell than the particle from which the last accumulated velocity came from. This technique seems to create a good compromise between continuity and velocity detail.

Summarizing the data for the splatting procedure, the minimum required properties stored for each particle, to make the splatting possible, is position and velocity. Additionally, as explained a radius can be used to enable different radii for the splatting kernels of the particles. An example of a density splatting is shown in Figure 4.4(a).

4.2 Simulation refinement

The set of slices, onto which density and velocity now have been splatted according to Section 4.1, are simulated separately with a grid-based two-dimensional fluid solver. The simulation for a frame is done with a specified time and time step, and for each new frame a splatting is done and interpolated into the cur-



(a) Splatted particles slice (b) Slice simulation refinement

Figure 4.4: The result of the particle splatting procedure and subsequent simulation.

rent simulation. This interpolation is both additive and multiplicative according to Equation (4.3), where I'_f and I_f is the density at frame f before and after simulation refinement, respectively, and where I_s is the splatted density to interpolate into the simulation. The reason for the multiplicative weight ω_m is to suppress simulated density outside the splatted density function, thus making the simulation conform more to the input density to avoid having an uncontrolled simulation that takes life on its own. Finally, the weight ω_d is used to control the dissipation of the simulation so that it does not increase or decrease in over-all density.

$$I'_f = (1 - \omega_a)(1 - \omega_m + \omega_m \omega_d I_s) I_{f-1} + \omega_a I_s \tag{4.3}$$

The formulation in Equation (4.3) is also used for the velocity interpolation, but where the weights can be specified in a different manner for separate control.

Except for the interpolation weights and the simulation times, a number of other options can be used to control the slice simulations. Margins around the simulation grid can for example be specified to assign some extra space for the simulation to expand in and an input scene can be used for interaction with this expansion. The scaling of the grid can also be changed to achieve simulations that have an over-all small- or large-scale appearance. A synthetic swirl can be added to the simulation to emphasize fluid behavior such as vortices and similar. The simulation can also be specified to continue without having new data interpolated, to get a regular uncontrolled fluid simulation. For a complete set of features, Appendix B lists all parameters that can be used, also when it comes to the other parts of the implementation.

Using different calibrations in the simulations, a large number of different effects can be simulated. The interpolation weights can for example be specified in such manner to have a simulation tightly controlled by the input particle simulation, or they can be set to yield a simulation with more "life of its own", where it even can be let free without any influence from the particles.

To illustrate the simulation, Figure 4.4(b) shows the result of simulating the splatted density in Figure 4.4(a), where a collision scene also has been specified for interaction with the simulation as can be seen in the upper right of the simulated density slice. The output of the simulation stage is, for each slice and for each frame, a density field and a velocity field which are stored to disk for later use in the rendering algorithm. Additionally, texture coordinates are advected through the simulations with a specified speed and saved to file along with the other simulation data. These coordinates are later used in the rendering as input for the turbulence function.

4.3 Density function

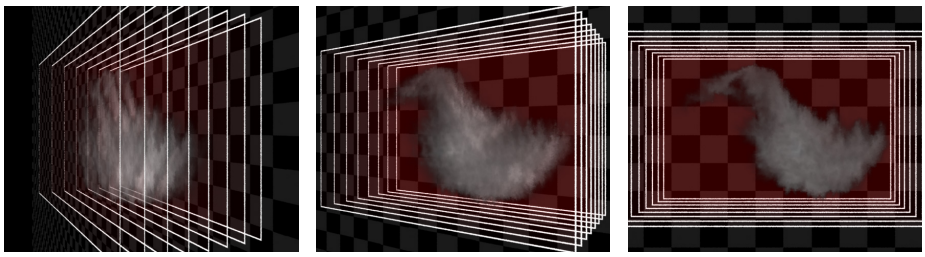
The major steps in the process, from which the total outcome is highly dependent, are the transformations between two and three dimensions. The splatting procedure – that is the transformation from three to two dimensions for simulation purposes – was treated in Section 4.1, and in this section the reverse transformation is described in more detail.

The motivation for the transformation to a three-dimensional representation can be explained as follows: The principle of the view-aligned refinement is illustrated in Figure 4.5, where the simulated slices with their simulation bounds are drawn from different angles. When viewed from the camera direction, Figure 4.5(c), the slices are effectively blended together to form an impression of a three-dimensional volume. This is, however, not appropriate for such effects as the one illustrated since it does not account for the light interacting with the media, and thus it would look the same regardless of the positioning of light sources in the scene. The simulated slices have to be transformed into a three-dimensional density representation to enable volume rendering calculations as described in Section 3.2.

4.3.1 Density interpolation

To formulate a three-dimensional high-resolution density function explicitly as a pre-calculation step would take up large amounts of disk space. Additionally, the rendering properties may not be known so the accuracy needed in the 2D to 3D transformation is not always specified. The method thus benefit from having an on the fly interpolation for this transformation, where speed is of high priority.

What the 2D to 3D transformation boils down to is how the interpolation should be done to avoid artifacts as much as possible. The implementation



(a) (b) (c)

Figure 4.5: Slice interpolation principle.

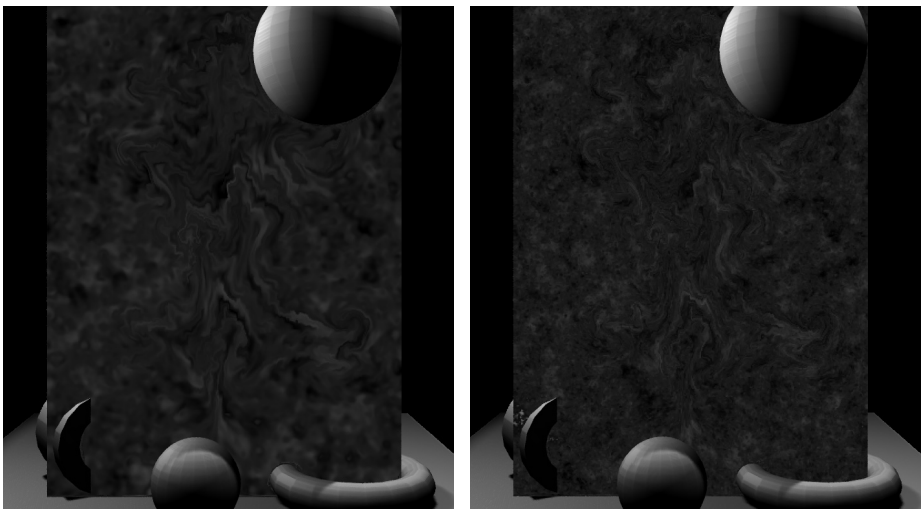
specific details of how to store and lookup are treated in Chapter 5, but the interpolation itself is described in this section.

The interpolation can for example be performed as a simple nearest neighbor lookup, which obviously has limitations when it comes to artifacts. Another alternative is to do a linear interpolation between the nearest grid cells in the two nearest slices for a point $\mathbf{x}$. It is also possible to do a more sophisticated lookup using e.g. a Gaussian kernel. Since speed is of great importance Gaussian interpolation may not be preferable, and the linear interpolation approach seems to generate the best compromise between speed and quality.

To break up possible artifacts in the interpolation, the simulation advected texture coordinates are now used. This is done by interpolating the texture coordinates (s, t) at the position $\mathbf{x}$, followed by a turbulence evaluation using these texture coordinates. The turbulence is defined as a fractal summation of simplex noise, as formulated in Equation (3.20). Since the texture coordinates have been advected on slices they are in two dimensions, but a three-dimensional turbulence function is used to get variations in all dimensions for the density volume definition. This is achieved by using the depth of the position as third input for the turbulence; that is, the input is (s, t, z). The time input dimension to the fractal in Equation (3.20) is not used since a movement of the turbulence through time is already captured in the texture coordinate advection.

The evaluated turbulence $S(s, t, z)$ at the position $\mathbf{x} = (x, y, z)$ is used to perturb the z-position since it is along this direction artifacts are likely to emerge, so when interpolating density the new position $(x, y, z + S(s, t, z))$ is considered. To illustrate how this perturbation looks for a certain depth, Figure 4.6 shows the fractal noise for one slice and with different settings for lacunarity and gain.

Similarly to the transformation from three to two dimensions in the splatting stage, the interpolation also uses multiple positions randomly selected along the velocity vector. These positions are used separately for density interpolation and the final result is the weighted average of these densities. The reason for the multiple interpolation points is to emphasize the movement of the fluid and to get a small motion blurred effect, which tends to help in suppressing possible interpolation artifacts.



(a) (b)

Figure 4.6: Different noise textures, advected through the simulation refinement.

4.3.2 View interpolation

The main limitation in using the view-aligned simulation refinement approach is if the scene contains a moving camera. The view-aligned simulation refinements should be able to keep up with the movement of a camera if it is moving slowly, but this is a relatively strict condition on the simulation and rendering. The method is thus best used in scenes with static cameras. However, this section will try to explain a possible solution to the problem, to enable cameras moving at arbitrary speed.

The way moving cameras are treated in the developed method is by letting the user specify certain cameras at which view-aligned refinements should be calculated. Using these different refinements, the interpolation from simulation slices to a three-dimensional density function can also be made with respect to the different simulation directions; that is, by performing another interpolation between the simulations. This interpolation is done according to the formulation in Equation (4.4).

The interpolation for a specific frame uses the two simulations where the simulation viewing directions Ψ_1 and Ψ_2 are closest to the rendering viewing direction Θ and where $\Theta \cdot \Psi_i > 0$. The interpolation exponent η is used to control how fast the interpolation should move between the different simulations when the rendering viewing direction moves between Ψ_1 and Ψ_2. The final, normalized weights in Equation (4.4c) and Equation (4.4d) are then used for the interpolation between the different simulations.

$$\omega_1 = (\Theta \cdot \Psi_1 - \|\Psi_1 \cdot \Psi_2\|)^\eta \tag{4.4a}$$

$$\omega_2 = (\Theta \cdot \Psi_2 - \|\Psi_1 \cdot \Psi_2\|)^\eta \tag{4.4b}$$

$$\Omega_1 = \frac{\omega_1}{\omega_1 + \omega_2} \tag{4.4c}$$

$$\Omega_2 = \frac{\omega_2}{\omega_1 + \omega_2} \tag{4.4d}$$

4.4 Rendering

When a density function can be defined on the fly, according to Section 4.3, a volume rendering can be performed with some approximation to the volume rendering equation in Equation (3.18). Specifics of the rendering environment used to accomplish this will be described in Chapter 5, but the general details are discussed in this section.

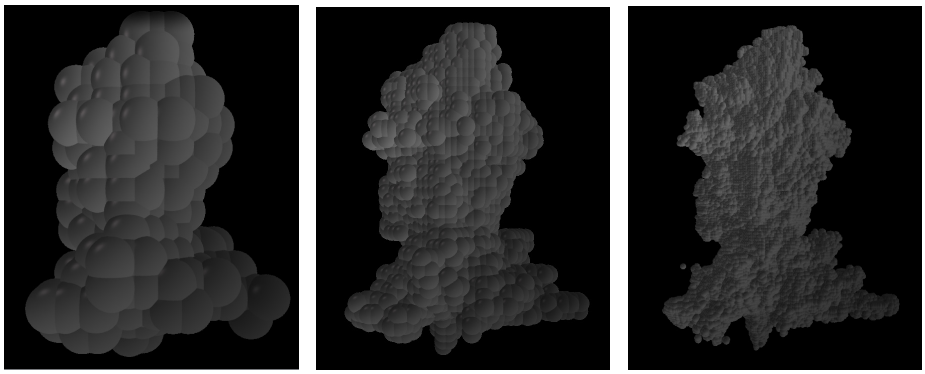
4.4.1 Rendering region

To avoid unnecessary calculations in the expensive volume rendering, where no interesting simulation data is present, a tightly fitted rendering region is wanted. In the implementation this region is specified using the concept of metaballs (Section 2.2).

A naive way of specifying such rendering region with metaballs is to simply use one large metaball that encompasses the whole simulation bounding box. This is obviously not a very efficient technique, which extends the region to consider to outside the simulation bounds rather than fitting it to the data.

Another technique is to use the original particle input to define where to instantiate metaballs. This can be done by considering the particles and specify metaballs at their positions with radii proportional to the particles radii. Since the simulated data conforms to the input particles this should fit the rendering region more tightly. However, the rendering complexity increases exponentially with the number of metaballs used, so normally only a subset of the input particles can be used for metaball instantiation to get a reasonable compromise between fitted rendering region and the number of metaballs. This may result in a rendering region that masks data that is important for the rendering result. Additionally, data may also be lost if the simulation refinement deviates to much from the input simulation. These properties of this technique makes it rather unstable and extensive calibration may be needed to find a good fitting of the rendering volume.

The third technique, which has shown to be most successful, is the direct use of the three-dimensional density function for metaball instantiation. By dividing the simulation bounding box into a $X \times Y \times Z$ voxel grid and traversing this grid, a lookup into the density function can be done at each voxel position. Using a threshold a decision can then be made if a metaball should be instantiated at that position. To make the metaball defined region continuous over the voxels,



(a) 20×20×20 (b) 100×100×100 (c) 160×160×160

Figure 4.7: Volume rendering region, defined using metaballs.

their radii have to be at least $\sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2}/2$, where ΔX, ΔY and ΔZ are the grid spacings. The result of this technique for three different grid resolutions is shown in Figure 4.7. It can be seen as a rendering volume defined by a rough level-set at the threshold density, which is the optimal region in regards to the density. As seen in the figure, the use of a higher resolution in the voxel grid results in a more tightly fitted rendering volume, but this also means that the rendering complexity and preprocessing time increases; a compromise between fitted region and the number of metaballs yields the fastest rendering times.

4.4.2 Volume rendering

The volume rendering is performed using programmable shaders, where each volume sample, or microvoxel, is treated. The shader accounts for the single scattering term in the volume rendering equation by making subsequent calls to light source shaders. Additionally, the calculated single scattering illumination can be baked to file for possible multiple scattering evaluations, and such evaluations can then be incorporated in the shading to account for the multiple scattering.

When dealing with the single scattering term there are different phase functions that can be used to approximate different light behaviors. For isotropic scattering the phase function can simply be defined as $P(\mathbf{x}, \Theta \leftrightarrow \Psi) = 1/4\pi$ to satisfy energy conservation. A more useful phase function is the empirical Henyey-Greenstein model, as formulated in Equation (4.5). Using $g \in]-1, 1[$, this phase function describes scattering that is backwards for $g < 0$, forwards for $g > 0$ and isotropic for $g = 0$.

$$P(\mathbf{x}, \Theta \leftrightarrow \Psi) = \frac{1 - g^2}{4\pi(1 + g - 2g\cos(\Theta \cdot \Psi))^{1.5}} \tag{4.5}$$

Although the Henyey-Greenstein phase function successfully can simulate many scattering behaviors, it is rather slow to evaluate cause of the 1.5 exponent in the denominator. An approximation to this phase functions is therefore often used; the Schlick phase function, as formulated in Equation (4.6). And the parameter $k \in]-1, 1[$ has approximately the same effects as the parameter g in the Henyey-Greenstein phase function.

$$P(\mathbf{x}, \Theta \leftrightarrow \Psi) = \frac{1 - k^2}{4\pi(1 + k\cos(\Theta \cdot \Psi))^2} \tag{4.6}$$

A third phase function, that is used to model the behavior of scattering in the atmosphere, is the Rayleigh phase function, given in Equation (4.7). This function is almost isotropic, but is used with a wave-length dependency to account for the increased scattering of blue light which makes the sky blue and the sunset red.

$$P(\mathbf{x}, \Theta \leftrightarrow \Psi) = 3\frac{1 + \cos(\Theta \cdot \Psi)}{16\pi} \tag{4.7}$$

To enable self-shadowing and global shadowing from the simulated volume – effects that are not accounted for when excluding the multiple scattering term or when approximating it using baked illumination – the concept of deep shadow maps is used. A deep shadow map is calculated in a preprocessing step and used in the shading to account for occluded and partially occluded shading samples, thus incorporating both self-shadowing and shadowing onto other objects in the rendering.

Chapter 5

Implementation

To enable an implementation of the method, as described in Chapter 4, there are a number of concepts needed: *a*) the simulation refinements have to be calculated using a grid-based fluid solver; *b*) the volume rendering need to be performed with effects such as shadowing and multiple scattering in consideration; and *c*) for input, intermediate and output data some sort of storage facilities must be used. This section specifies the different parts constituting the created framework for simulation refinement and subsequent rendering.

5.1 Fluid solver

The work in fluid simulation, such as [Fedkiw et al., 2001; Foster and Metaxas, 1996, 1997; Stam, 1999], culminated in fluid solvers such as the one in Autodesk® Maya®[1], which is used in the implementation to simulate the refinement slices.

The interaction with Maya is handled through the Maya API, which provides a C++ interface for using Maya's different features. With regards to the collaboration diagram in Figure 5.1, this is represented by the "Slicer" object that serves as such interface for the Maya simulations of the input data using the Maya API. This object performs the simulation instantiation and splatting procedure to specify simulation data to Maya, and the returned result is saved to file.

5.2 Rendering environment

As described in Section 3.2.2 Pixar's rendering engine RenderMan®[2], or more specific Photo-Realistic RenderMan (PRMan), uses the Reyes algorithm to evaluate lighting equations. The micropolygon estimations have been used since [Cook et al., 1987] for surface shading, and later the concept was extended to

[1] http://www.autodesk.com/maya
[2] https://renderman.pixar.com/

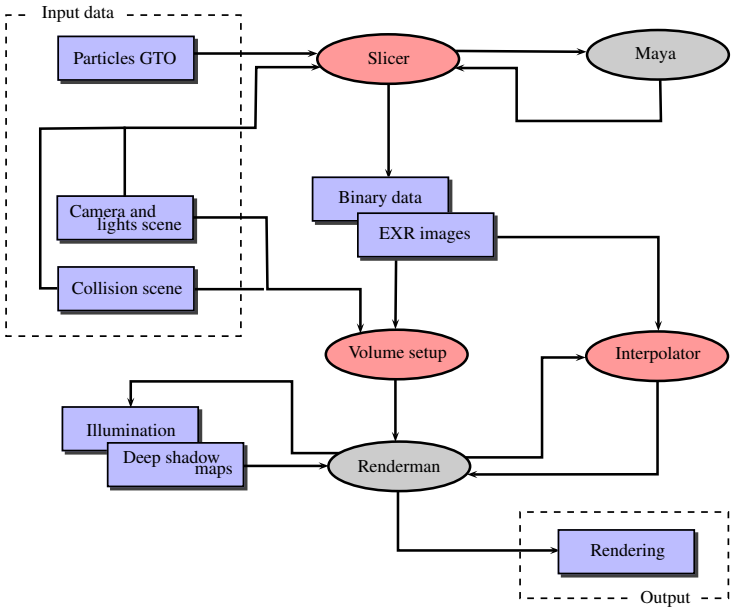


Figure 5.1: Collaboration diagram for the implementation.

include volume rendering. In RenderMan 15, the concept of *blobbies* was introduced for simple and efficient volume rendering instantiation, where a blobby is nothing more than a metaball specifying a volume rendering region.

RenderMan is the rendering environment used in the implementation, and with the concept of blobbies a rendering volume can be specified according to the description in Section 4.4.1 to enable RenderMan's volume rendering features on this region.

Volume rendering with RenderMan is handled programmatically in the same way as surface shading, using a regular surface shader written in the RenderMan Shading Language (RSL). However, to enable volume rendering behavior in a straight-forward manner, the shading parameters have different meanings than when dealing with surfaces.

Using the concept of a surface shader the illumination can be baked – saved that is – to file using the RSL `bake3d` function. In a preprocessing rendering pass the calculated single scattering illumination can be baked for external processing. This processing can then be done by RenderMan's filtering feature, `ptfilter`, which calculates a volume color bleeding diffusion that approximates a multiple isotropic scattering process for the illumination. In a subsequent rendering pass this multiple scattering illumination is read and accounted for to incorporate an isotropic multiple scattering effect.

The approximation of multiple scattering does not include the effects of shadowing and it is therefore dealt with using deep shadow maps, as described

in Section 3.2.3. In a preprocessing rendering pass, deep shadow maps are calculated from each of the light sources in the scene. These maps are used in the final rendering pass to lookup for occlusion when performing the single scattering evaluations.

5.3 Data handling

There are a couple of different data representations involved in the simulation refinement and rendering. First, there is an input particle simulation for which the refinements are calculated. The format of this point cloud is the open source GTO[3], which can hold different geometries and properties binary in a structured manner. Additionally, there are input Maya scene files to specify collision and camera/light scenes for the simulation.

The simulated slices need to be saved to file for later use in the rendering stage of the method. The simplest and fastest way of doing this is by storing the simulations as binary raw data. This is, however, not an appropriate solution when it comes to rendering of many high-resolution slices since all data for one frame have to be loaded into memory for use in the density interpolation lookups. If the hardware at hand has capabilities to handle this though, it enables the fastest lookup times once the data have been loaded into memory.

To overcome the problem with memory usage, the implementation also has support for the OpenImageIO[4] library. OpenImageIO supports the reading and writing of chunks of data, instead of having to deal with all data at once. When it comes to rendering of the slice data, this leaves a very small runtime memory footprint at the cost of slightly longer lookup times. Additionally, using the OpenImageIO library, data is read and written as high dynamic range (HDR) image data. In the implementation this is done using the OpenEXR[5] format, supporting a loss-less compression to spare disk space. Using an image format to store the simulations the outputs can be directly viewed, for fast turnaround times when calibrating the simulation, instead of having to visualize the data with rendering methods such as for binary data.

5.4 Pipeline

The general pipeline objects, central for the implemented method, are written in C++. This includes the splatting procedure, the Maya interface, the RenderMan interface for volume rendering setup, the density interpolations etc. The collaborations of the different parts in the pipeline are illustrated in Figure 5.1.

In the simulation and rendering a number of different features can be calibrated. For a through-out description of all possible parameter settings that can be used, they are given in the listings in Appendix B.

[3]http://www.tweaksoftware.com/products/open-source-software/gto/
[4]http://www.openimageio.org/
[5]http://www.openexr.com/

Chapter 6

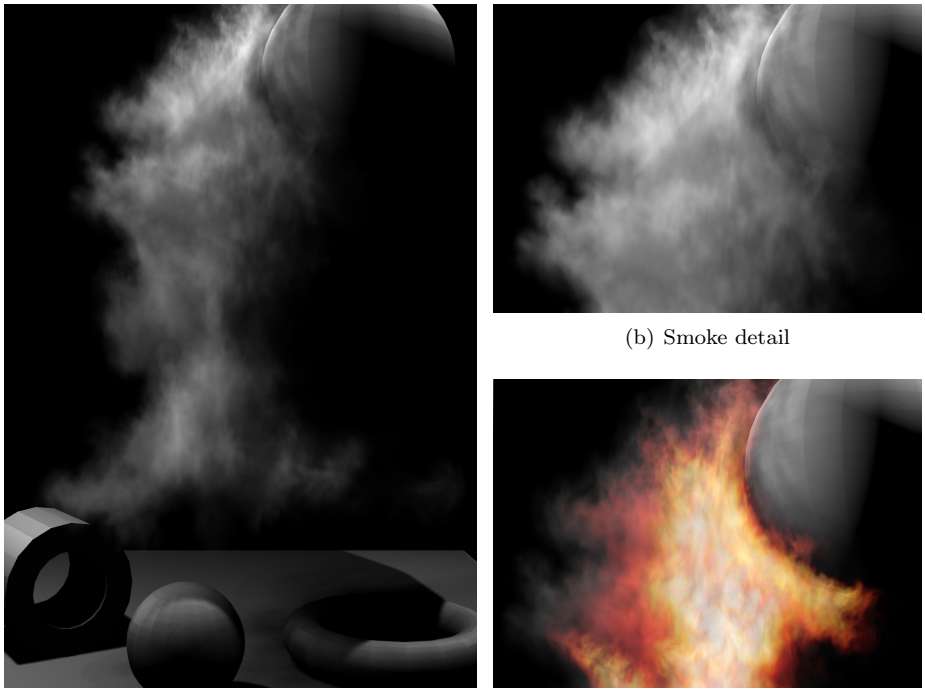
Results

The resulting simulation and rendering framework can be used in many ways. A simulation can for example be made from a very few number of particles, where the slice simulations can be set to govern much of the fluids global behavior. The opposite is another possibility, where most of the simulation comes from the use of many particles, whereas the slice simulations only add the very finest local details.

In Figure 6.1, the final outcome of a simulation and rendering using the developed method is illustrated. The input to this simulation is a particle simulation containing approximately 400K particles, but only every second particle is actually used. The result is a high-resolution rendering of a large-scale effect, where the small-scale details are effectively simulated as shown in Figure 6.1(b). As seen, the refinement simulations can interact with the environment and the rendering includes volumetric lighting and shadowing. In Figure 6.1(c) a simple blackbody radiation function has been applied, where the temperature of the media is based on the simulated density, to achieve a fire simulation. The emissive media does not cast light onto its surroundings; it is given more to illustrate that the method applies to other phenomena than just smoke.

6.1 The slice technique

The principle using the slice simulation approach was visualized in Figure 4.5, but to further motivate the technique consider Figure 6.2. Figure 6.2(a) is simulated in three dimensions, but with the z-dimension reduced to the number of slices used in the two-dimensional slice simulation in Figure 6.2(b). Thus, the same calculations are involved and with the same resolutions, but one is in 3D using a lot of memory, and one in 2D where the simulation can be separated into a set of parallelize-able slice simulations using considerably less memory. As seen in the figure the results are very similar in detail. This simulation was created using a three-dimensional turbulence texture as input to the simulation, so the movement of the smoke is similar in all directions. However, if the movement had



(a) Smoke in environment

(b) Smoke detail

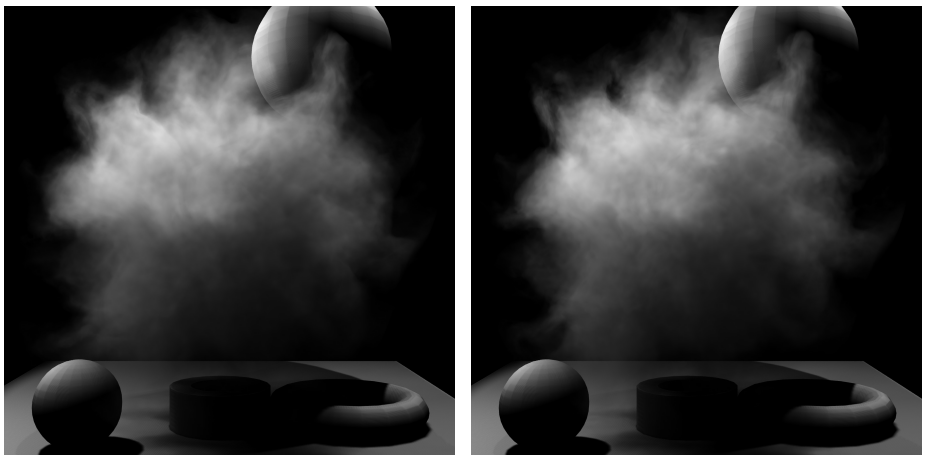
(c) Emission

Figure 6.1: Rendering results.

been mostly along the viewing direction of the camera it would not have been as easy to get away with the slice approximation when creating an animation.

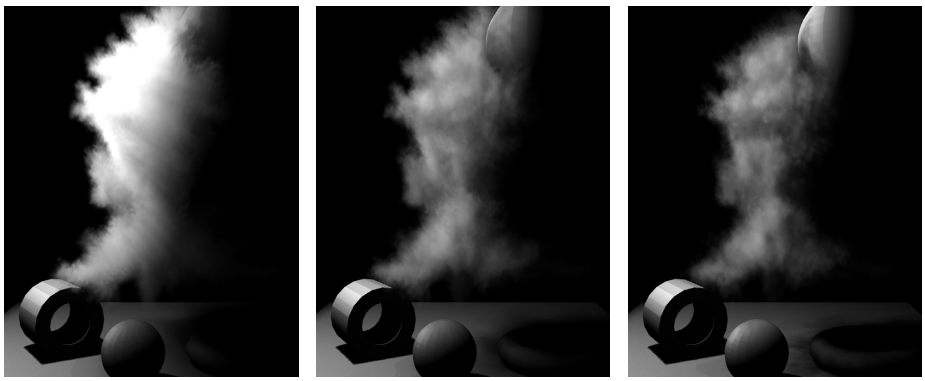
The number of slices used for simulation refinement is central to the technique, which is illustrated in Figure 6.3, and can affect the final result by a great deal. An appropriate number depends on the properties of the simulation, where the complexity and movement along the camera viewing direction is essential, but something in between 64 and 128 slices generally produces good results.

One important advantage using the slice simulation technique is the independent simulations possibility of the different simulation planes. As described in [Horvath and Geiger, 2009], a GPU implementation can be used for efficient simulations, but the developed implementation is targeted at CPU parallelization on multiple cores by separately handling the different slices. This is a motivated approach given the possibility to do this parallelization on a render farm in a VFX pipeline.



(a) 3D simulation (b) 2D simulation

Figure 6.2: Difference between 3D and 2D simulations.

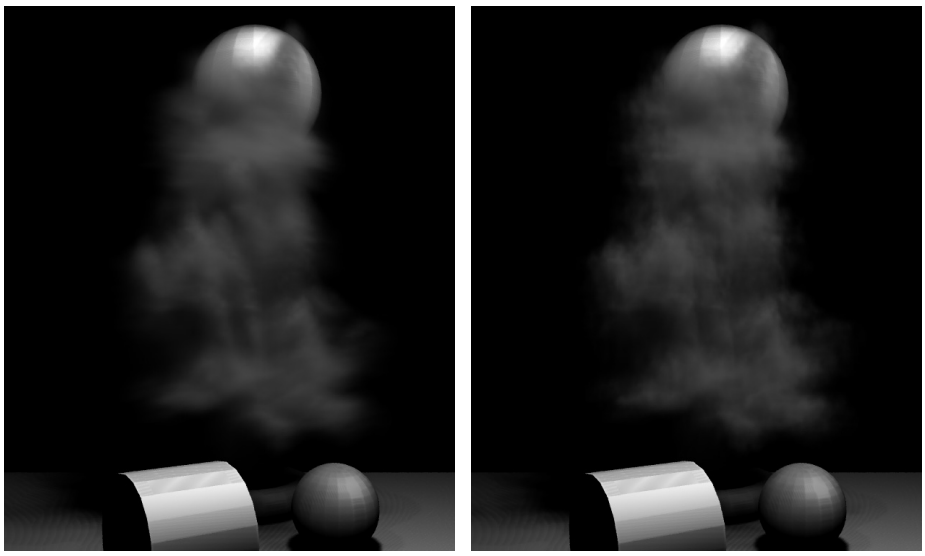


(a) 8 slices (b) 32 slices (c) 256 slices

Figure 6.3: Simulation and rendering result using different number of slices.

6.1.1 Turbulence textures

As described in Section 4.3.1 a fractal turbulence texture, advected through the simulation, is used to break up some of the interpolation artifacts when transforming the two-dimensional slices to a three-dimensional density function. To illustrate the effect of such added interpolation complexity, Figure 6.4(a) shows a rendering where the camera is placed in the perpendicular direction to the camera used for simulation refinement. Since the technique is based on a rendering where the camera direction is supposed to be close to the simulation



(a) Without noise (b) With noise

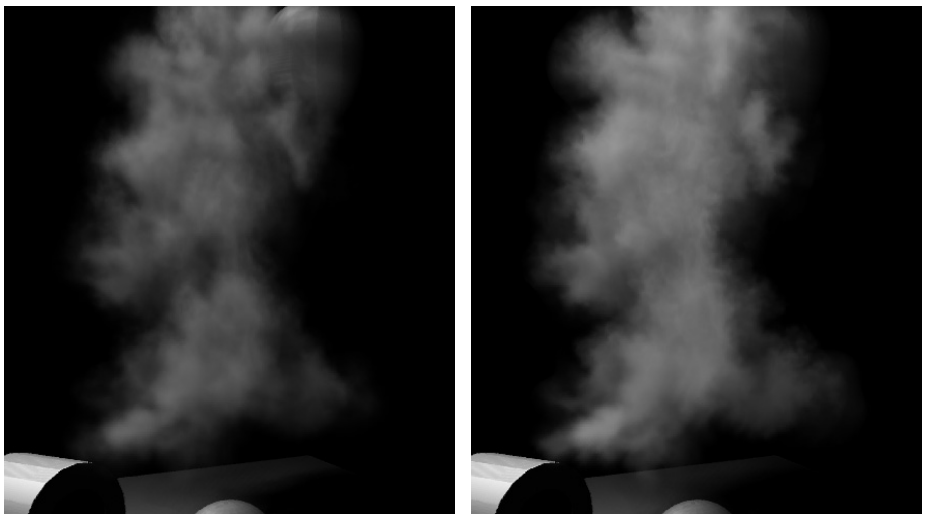
Figure 6.4: Rendering from perpendicular to the simulation direction, without and with the use of advected turbulence texture in the density interpolation.

direction, this shows obvious artifacts with substantial loss of detail. In Figure 6.4(b) the interpolation z-positions have been perturbed with a turbulence texture and the result is artificially recreated details that helps in suppressing the artifacts.

Since a turbulence function based on simplex noise is very fast to evaluate, the added interpolation complexity is of low computational cost. However, the texture coordinates used for noise lookup have to be advected through the simulations and stored to disk. Additionally, interpolations have to be done to find the simulated texture coordinates for a continuous space. But all in all, this technique of suppressing artifacts adds substantial detail at low cost.

6.1.2 Camera movement

One of the difficulties in using the slice simulation technique is how to handle a camera moving within the scene, for which a possible solution was proposed in Section 4.3.2. The result of applying such interpolation between two different simulations, solved in different slice directions, is shown in Figure 6.5. In Figure 6.5(a) the simulated slices are viewed from a direction deviating from the simulation camera by about 30 degrees, and the interpolation between the slices starts to come clearly visible, resulting in a blurred volume with lost fine details. To overcome this problem, Figure 6.5(b) – which shows the same simulation –



(a) Without interpolation (b) With interpolation

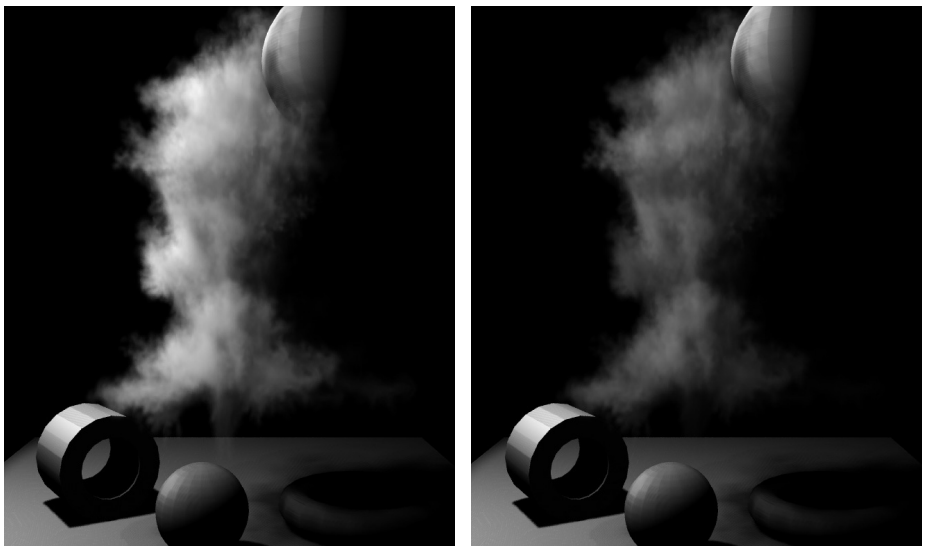
Figure 6.5: Simulation direction interpolation.

uses a second simulation direction interpolated into the rendering, thus adding back the details lost.

In the limit of number of simulation directions, this interpolation method resolves the problem of view-dependency, but since this is not practical – neither in regards to memory nor computations – the slice technique clearly has a shortcoming in this context. There may also be artifacts introduced in the interpolation between different simulations if their directions deviate to much and the number of slices used is low.

One important observation though, that enables successful use of this technique without having to deal with unmanageable amounts of different simulations, is the very movement itself of the camera. Since the scene is moving with respect to the camera, a motion blurred effect is actually expected, and it also makes detail harder to distinguish.

Comparing the proposed technique to the one in [Horvath and Geiger, 2009], the introduced interpolation of slices to a three-dimensional density function makes directional artifacts more appealing. It is the interpolation that makes the artifacts appear as blurring of the volume. This fact makes it possible to get away with many of the artifacts, and it also enables the use of rendering directions quite different from the simulation directions without having to use multiple simulation refinements. However, if the camera is moving very slowly it may be better to consider a dynamic simulation direction instead, where the simulation direction is kept the same as the rendering direction.



(a) $k = -0.7$ (b) $k = 0.8$

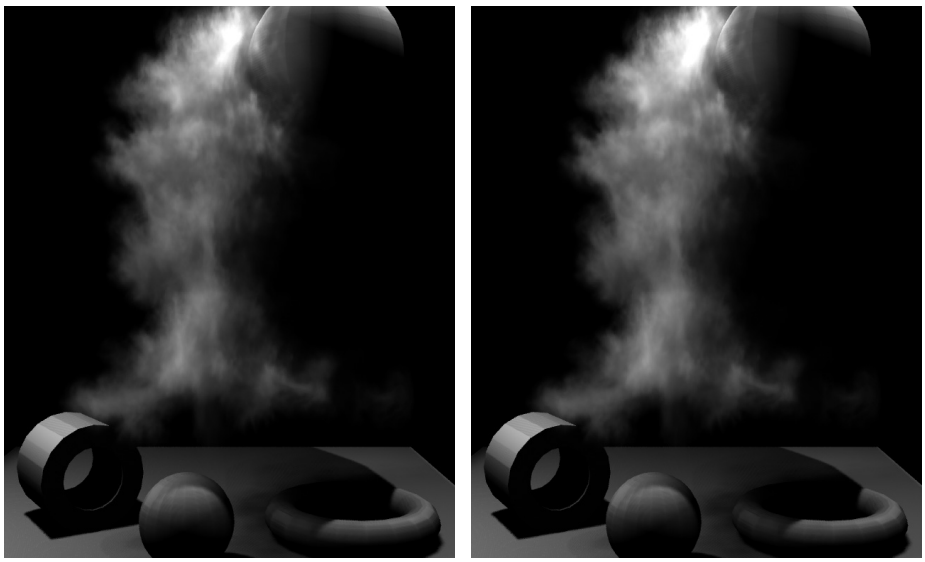
Figure 6.6: Rendering with Schlick phase function for the single scattering, and with different scattering properties.

6.2 Rendering

Since the rendering of the participating media does not account for shadows, those are approximated using deep shadow maps as explained in Section 3.2.3. Deep shadow maps enables fast estimation of otherwise heavy calculations, and the result improves visual quality greatly as seen in the different renderings in this chapter. However, artifacts are easily introduced, as for example can be seen in the patterns on the shadows from the volume in Figure 6.1(a) onto the plane.

The choice of phase function in the volume rendering equation, Equation (3.18), can affect the rendering outcome to achieve different light scattering behaviors. As example, in Figure 6.6 the single scattering evaluations are done using the Schlick phase function in Equation (4.6) with different values of the scattering coefficient k. For $k < 0$ this creates a backward scattering behavior and for $k > 0$ forward scattering, resulting in the different renderings in the figure.

Furthermore, multiple scattering can be accounted for, as described in Section 5.2. Using that technique the multiple scattering is limited to isotropic scattering events, and it is still costly to incorporate into the final result since a preprocessing rendering pass is needed to bake the illumination. Furthermore, the multiple scattering – or diffusion – calculations on the baked illumination may take considerable time. The result of incorporating the multiple scattering



(a) 0 bounces (b) 4 bounces

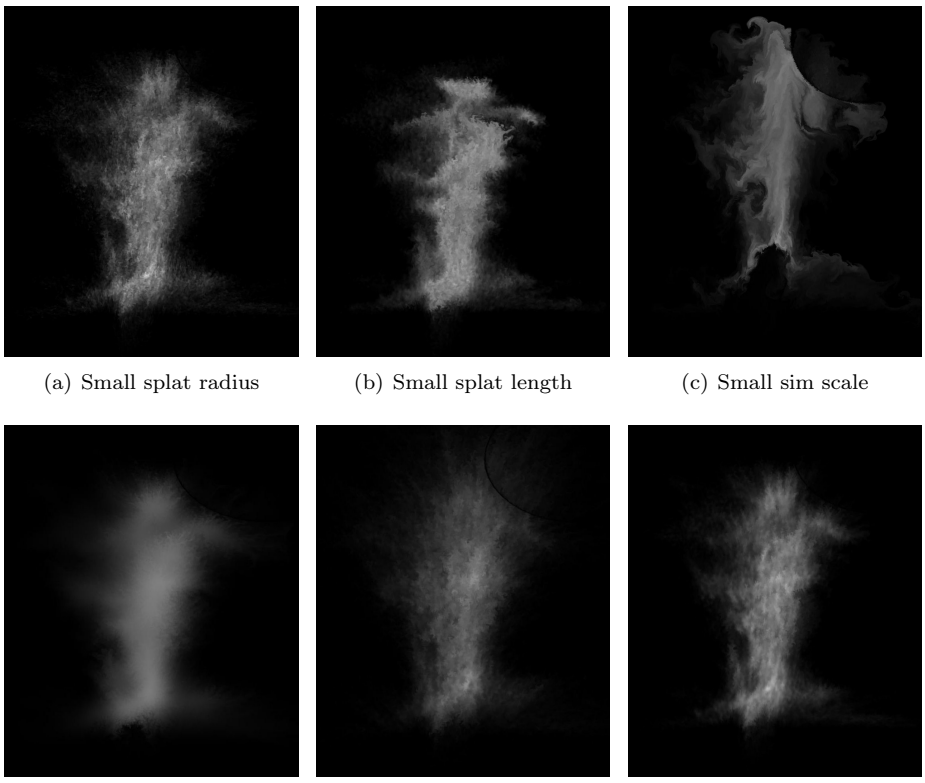
Figure 6.7: The effect of multiple isotropic scattering.

effect is shown in Figure 6.7(b) for four bounces, and compared to rendering with only single scattering in Figure 6.7(a). The little visual improvement in the multiple scattering rendering, compared to the increased rendering time, makes the use of this effect questionable.

For other effects multiple scattering may be of greater importance, and anisotropic scattering events can effect the outcome in relatively large extent. In those cases, the implemented method falls short, but the restriction to single scattering and possibly multiple isotropic scattering holds as approximation for many different media and effects.

6.3 Parameter specifications

There is a large number of parameters that can be used to control the behavior of the simulations and the renderings. At a first glance this can give the impression of a tool hard to calibrate and manage, but for most of the parameters the default values should hold for many situations. The use of an HDR image format for storage also enables direct visualization of the simulation results, and to calibrate the simulation parameters generally only one slice is needed to get a feeling for how the outcome will be, so calibration is relatively fast and straight-forward. However, a better approach to this calibration would be preferable, e.g. using some kind of GUI for interactive calibration, instead of having to change parameters in a text file as is the case now.

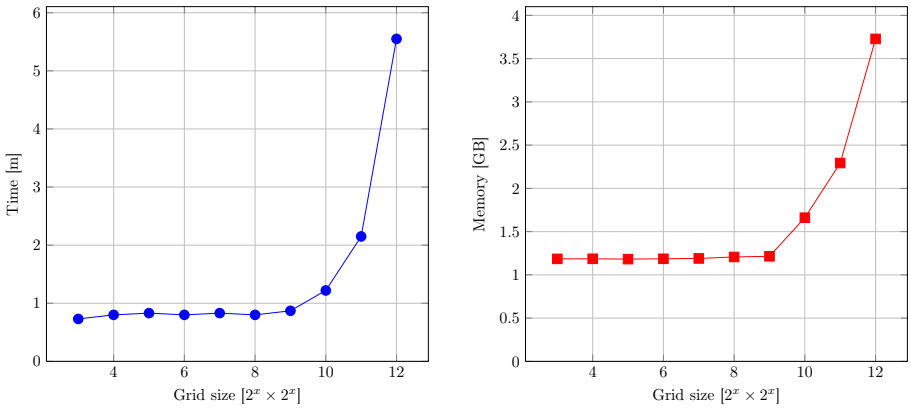


(a) Small splat radius (b) Small splat length (c) Small sim scale

(d) Large splat radius (e) Large splat length (f) Large sim scale

Figure 6.8: Direct visualization of HDR simulation results for some different parameter setting extremes, for one view-aligned splatting slice.

The large amount of different parameters gives the user the ability to achieve many different results, both by modifying the simulation and by controlling the rendering characteristics. This flexibility should enable the use of the method for achieving many different computer graphics effects. In Appendix B all the different parameters are listed and explained shortly, but to give a visualization of how e.g. some of the parameters used in the fluid simulation refinements can affect the outcome, a set of simulated slices are given in Figure 6.8. These slices show a couple of parameter setting extremes used in the solver, and the resulting simulations are saved and directly visualized as HDR images, using the OpenImageIO library.



(a) Time (b) Memory

Figure 6.9: Simulation time and memory performance. The initial memory overhead in the right plot is caused by the simulation environment in Maya.

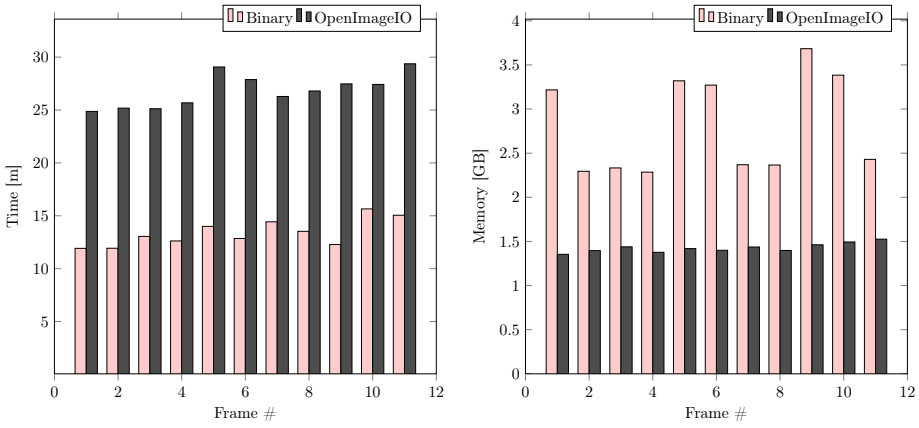
6.4 Performance

To evaluate the performance of the implementation is difficult. The simulation/rendering times and the memory usage are highly dependent on the parameter calibrations, the camera placement and the scene characteristics. This section tries to give a general overview of the frameworks effectiveness using a test simulation and rendering setup, which is the one given in Figure 6.1.

6.4.1 Simulation performance

First, Figure 6.9 illustrates the time and memory involved in the simulation stage for one simulation slice at different grid sizes, where the resolution is $2^x \times 2^x$ for the different values on the horizontal axis; that is, the plot ranges from resolution 8×8 to 4096×4096 grid cells. The benchmark calculations include both the splatting procedure and the fluid simulation. In this test case, every second particle of approximately 400K particles are used in the splatting, and the result is relatively long splatting times as compared to the simulation times. For example, with a simulation grid of resolution 1024×1024, as in Figure 6.1, this gives splatting times in the vicinity of the rendering times for the settings used in the figure. Since the method is capable of creating detailed simulations from considerably fewer particles it is thus possible to reduce the simulation times greatly for other effects.

The memory usage involved in the simulations through Maya, in Figure 6.9(b), does not really reflect the true memory usage of the method itself since the use of Maya's features for fluid solving adds a considerable memory overhead. However, the figure clearly illustrates the large amounts of memory needed when



(a) Time (b) Memory

Figure 6.10: Rendering time and memory performance, compared between binary raw data and OpenImageIO. In the right plot a memory overhead is caused by the rendering environment.

increasing the simulation refinements to high resolutions; it becomes rather obvious that such resolutions would be impractical, or even impossible, to simulate in three dimensions.

6.4.2 Rendering performance

Figure 6.10 shows the time used and memory consumption when rendering a range of 11 frames, for the scene in Figure 6.1. The simulation refinements used for these benchmark measures are done with 64 slices at a simulation grid resolution of 1024×1024 grid cells. And the renderings are calculated at a resolution of 2324×1766 pixels, using 2 light sources with separate 512×512 large deep shadow maps.

In the same way as for the simulation memory plot, the memory usage in Figure 6.10(b) does not correspond to the actual memory of the method itself; that is, loading slice data and performing an interpolation. Instead, a lot of the memory is due to the rendering environment, deep shadow maps etc. The purpose of the plot is more to show the distinct difference in memory usage in using the different data formats. To calculate just the size of all slices – that is the memory needed to load them all for one frame – it is given as:

$$5 \cdot N \cdot S_x \cdot S_y \cdot \texttt{sizeof(float)} + 28 \cdot \texttt{sizeof(float)} + 2 \cdot \texttt{sizeof(int)}$$

Here, the first term is multiplied by 5 since there need to be stored 2 floats for velocity, 2 floats for density and 1 float for density. The two last terms accounts for meta data, such as transformation matrix and similar. For the

calibration used in the measurements this means that the memory used only for the density function in the binary rendering, is approximately 1.34GB, for a float size of 4 bytes.

Comparing the rendering benchmarks using binary raw data and the OpenImageIO library, it is clear that they present different compromises between rendering time and memory usage. Thus, the decision of data format should be taken with respect to available memory; if enough memory is at hand in the rendering environment the binary format is the best choice, but increasing the number of slices and/or the simulation resolution to even higher quality, makes OpenImageIO a more practical choice. Another aspect in the comparison is the disk space. The space needed for a frame of binary raw data is the same as the runtime memory needed, but when using an HDR image format the memory needed depends on the nature of the data and a compression of a factor 0.5 comparing to raw data is not unusual.

To summarize, a rendering time of about 10-15 minutes – at 2324×1766 pixels resolution – is achieved using binary raw data. In combination with the simulations of the 64 slices for one frame, the final time is not much more than one hour, which is a reasonable time considering the relatively large number of particles used. Furthermore, given the possibility to do the slice simulation refinements in parallel on multiple CPUs in a VFX pipeline, the total time is decreased to under 20 minutes for this case.

Chapter 7

Discussion

A new framework for simulation and rendering of gaseous phenomena has been presented in this thesis. By combining a number of the different existing concepts for such phenomena, it overcomes many of their separate issues. The result is a flexible and potent method, but further work can surely improve it for better control and use in VFX, and examples are included in the next section.

The developed implementation is essentially an extension of the slice simulation method proposed for fire in [Horvath and Geiger, 2009]. Many extra features have been incorporated though, to enable the same concept for other phenomena than fire. These extensions are mainly to get a more general rendering approach, since the rendering technique in the earlier work is rather limited. The result is the possibility to use the method for rendering of a broad range of gaseous phenomena, incorporating the theory behind light interaction with participating media in the rendering.

One of the difficulties in using the proposed method, is that it is relatively sensitive to artifacts, which occurs due to the independent slice simulations; artifacts one can get away with when rendering fire, cause of the emissive nature. Solutions to most of the arising problems have been considered and used to suppress these artifacts, and the final result is capable of generating many high-quality effects at low cost.

7.1 Further Work

There are a number of different areas for which improvements can be made in the implementation. The most important ones are probably those dealing with the possible artifacts inherent in the slice simulation technique. Improvements may for example include:

- Splines in particle splatting, as described in [Horvath and Geiger, 2009]; that is, splat along a spline formed between a particles positions in two consecutive frames, instead of only splatting along the velocity direction.

- GPU implementations for parallel calculations, where suitable.
- More sophisticated – and possible physically based – noise generation, such as the one in [Neyret, 2003].
- Better view interpolation, or investigation of other methods to deal with camera movements to avoid costly extra simulations.
- Multiple anisotropic scattering, for example by using volumetric photon mapping as described in [Jensen, 2001].
- Level of detail in simulations; that is, a simulation bounding box aligned with the view frustum. For simplicity the sizes of the slices currently are the same.
- Investigation of other fluid simulation implementations than the one in Maya currently used, and find one optimized for memory, to enable larger grid sizes. Ultimately implement a customized fluid solver.
- Introduce some kind of dependency between the slices in the simulations, so that they does not deviate to much from each other.
- More sophisticated emission for fire, and extension to other effects, such as sand, dust etc., including their different light interaction properties in the rendering.
- Improve usability of the current implementation, to enable easier calibration of the simulations and renderings.

Appendix A

Outline

To illustrate the simulation and rendering procedures in the developed method in an organized way, this appendix outlines the different steps and calculations involved. The details of these steps are described in Chapter 4.

A.1 Simulation

In Algorithm 1 the main steps in the simulation refinement are given in a pseudo-code style. The slice initialization expression includes specifying a scene with collision objects and setting the fluid solver attributes. Additionally, texture coordinates are initialized, which are defined to be $s \in [0, 1]$ and $t \in [0, 1]$ from left to right and from bottom to top in the slice, respectively. And in the simulation step the coordinates are moved with the simulation. A slice initialization can also be specified to be loaded from file, to enable a continuation of an earlier simulation. The output of the simulations is a set of simulation refinement slices, which includes densities, velocities and texture coordinates.

Algorithm 1 View-aligned simulation refinement

```
1: Find bounding box around input particles in frame range f_0 ... f_F
2: for all cameras do
3:    Calculate transformation matrix, world coordinates → simulation space
4:    for slice s = 1 to numberOfSlices do
5:       Initialize a slice s_sim for simulation, with texture coordinates
6:       for frame f = f_0 to f_F do
7:          s ← Splat particles from frame f
8:          Interpolate splatted slice data s into simulation s_sim
9:          Simulate slice s_sim
10:         Save simulated data s_sim to file
11:      end for
12:   end for
13: end for
```

The particle splatting step in Algorithm 1, for slice s and frame f, is outlined in Algorithm 2. The result of this step is a two-dimensional grid with splatted densities and one with splatted velocities, which then can be used for interpolation into the current simulation.

Algorithm 2 Particle splatting procedure

```
1: Input is slice s
2: for all particle p do
3:    x ← Position of particle p
4:    v ← Velocity of particle p
5:    for all splats do
6:       x_R ← Random position along v from x
7:       d_z ← Projection distance between x_R and slice s
8:       g_z ← Gaussian kernel weight using d_z
9:       if g_z below some threshold then
10:         Continue to next particle
11:      end if
12:      for all grid positions x_g within radius of Gaussian splatting kernel
         around x_R do
13:         d_xy ← Distance between x_g and x_R
14:         g_xy ← Gaussian kernel weight using d_xy
15:         Accumulate density g_xy g_z to grid cell x_g
16:         if d_xy < stored velocity distance v_d at x_g then
17:            Accumulate particle velocity to grid cell x_g
18:            v_d ← d_xy
19:         end if
20:      end for
21:   end for
22: end for
23: return s
```

A.2 Rendering

The main outline of the different renderings needed to form a final result is as stated in Algorithm 3. This result includes effects such as shadowing and multiple scattering, which are achieved using the calculated deep shadow maps and the baked and processed illumination.

Finally , how the actual renderings are performed is outlined in Algorithm 4, which includes the steps needed for interpolation of density from the slice simulation refinements.

Algorithm 3 Renderings overview

1: **for** light source $l = 1$ to $numberOfLightSources$ **do**
2: Deep shadow map $d_i \leftarrow$ Render from light source l
3: **end for**
4: Baked illumination $I_b \leftarrow$ Render from camera position with all shadow maps d_i
5: $I_m \leftarrow$ Calculate multiple scattering of I_b
6: Rendering output $I_o \leftarrow$ Render from camera position with all shadow maps d_i and multiple scattering I_m

Algorithm 4 Interpolation and shading

1: $(\Omega_1, \Omega_2) \leftarrow$ Calculate view interpolation weights
2: Initialize rendering region with metaballs
3: **for all** shading samples s **do**
4: **for** view $v = 1, 2$ **do**
5: Density $D \leftarrow 0$
6: $\mathbf{x} \leftarrow$ Position (x, y, z) of s
7: $(s, t) \leftarrow$ Interpolate texture coordinates at $\mathbf{x}$
8: $\mathbf{v} \leftarrow$ Interpolate velocity at $\mathbf{x}$
9: $T \leftarrow$ Turbulence lookup using (s, t, z)
10: $\mathbf{x}_T \leftarrow (x, y, z + T)$
11: **for all** multiple interpolations **do**
12: $\mathbf{x_R} \leftarrow$ Random position along $\mathbf{v}$ from $\mathbf{x_T}$
13: $D_t \leftarrow$ Interpolate density at $\mathbf{x_R}$
14: $D \leftarrow$ Accumulate with D_t
15: **end for**
16: **end for**
17: Use D, I_m and d_i in volume rendering calculations
18: **end for**
19: **return** Shaded samples I

Appendix B

Parameters

A large number of behaviors can be simulated and rendered using the implemented method, and this appendix tries to give an explanation of how the calibrations is done. The parameters are divided into five sets, controlling different parts of the framework. These are parameters for *general* settings(Table B.1), *simulation*(Table B.2), *rendering visualization/initialization*(Table B.3), *slice interpolation*(Table B.4) and *shading*(Table B.5).

For those parameters that have been used in equations and similar throughout the thesis, the corresponding denotation is also given in parenthesis.

Parameter	**Description**
Frame range ($f_0 \dots f_F$)	Which frames should be simulated/rendered.
Frame increment	To specify if not every frame should be used.
Data format	To select between 3 different ways of storing simulation data: 1. Texture format. 2. Binary format. 3. Both formats.
Tile size	Size of tiles when using OpenImageIO
Number of slices	How many view-aligned planes should be used for discretization of the volumes z-axis.
Slice	Which slice to simulate/render, if only one should be used. If option is set to 0 all slices are simulated/rendered. This is used to simulate specific slices for parallelization purposes.

Table B.1: General parameters

Parameter	**Description**
Release frame	To select a frame where the simulation should be let free, i.e. without any more particle splattings interpolated into the simulation.
Cameras	Which cameras should be used from the specified Maya setup.
Simulation resolution	Resolution of simulation slices.
Use last frame	Flag to specify if the simulation of the frame preceding the frame range should be used as initial state, loaded from file. This only works if the bounding box is manually specified, otherwise it may differ between the simulations.
Evaluate bounding box	Flag to specify if the bounding box should be manually specified or automatically evaluated from the input simulation.
Bounding box	Manually specified simulation bounding box.
Bounding box margins	Margins around automatically evaluated bounding box.
Splat radius (r_{xy})	Radius of the particle splatting on the planes.
Splat radius (r_z)	Approximate number of slices covered in the splatting, i.e. the splatting radius in the viewing direction.
Density interpolation weight (ω_a)	Additive frame interpolation weight for density.
Velocity interpolation weight (ω_a)	Additive frame interpolation weight for velocity.
Interpolation multiplication weight (ω_m)	Multiplicative frame interpolation weight for density and velocity.
Density scale	Density scaling in splatting.
Velocity scale	Velocity scaling in splatting.
Density dissipation	Density dissipation rate (The frame interpolation weights might as well be used).
Simulation time	Total simulation time for each frame's slice simulations.
Time step	Time step.
Solver quality	Quality of the Maya fluid solver.

Velocity swirl	The amount of swirling motion used in the fluid solver.
Number of splats (N_s)	The number of times each particle should be randomly splatted along its velocity vector.
Splat length	The length scaling of the random splatting along a particles velocity vector.
Coordinate speed	Speed of the advected texture coordinates moving with the simulation.
Simulation scale	Grid scaling, that is the spatial occupancy of the simulation plane. Can be used to achieve large scale or small scale simulations.
Particle skip	To specify if not every particle should be used in the splatting.

Table B.2: Simulation parameters

Parameter	**Description**
Visualization mode	To select between 8 different ways of visualizing the simulation data: 1. Volume rendering. 2. One slice visualized on a plane. 3. All slices visualized on planes. 4. All slices visualized with a cut-through (to show more of the occluded slices). 5. One slice visualized with bounding box. 6. All slices visualized with bounding box. 7. All slices visualized with bounding box and a cut-through. 8. Blobbies visualized as surfaces.
Coordinate system	In which coordinate system slice visualizations should be drawn: 1. Simulation coordinate system. 2. Rendering camera coordinate system.

Blobby drawing method	How blobby regions should be instantiated: 1. One large blobby encompassing the bounding box. 2. Blobbies at original input particle positions. 3. Blobbies where volume density is above a certain threshold.
Blobby scale	Scale of blobby sizes.
Blobby particle skip	To specify if not every particle positions should be used for blobby instantiation. That is if blobbies are drawn at original input particle positions.
Level-set dimensions	3D grid dimensions for blobby "level-set" initialization. For use when blobby are drawn by thresholding density function.
Level-set threshold	Blobby "level-set" threshold.

Table B.3: Rendering visualization/initialization parameters

Parameter	**Description**
Motion points	Number of interpolation points in slice interpolation.
Motion length	Max interpolation distance along velocity vectors.
Interpolation method	How to interpolate binary raw slice data: 1. Nearest neighbor lookup. 2. Linear interpolation. 3. Interpolation with Gaussian kernel.
Interpolation radius	Radius, in voxels, of Gaussian interpolation kernel.
View interpolation exponent	Exponent for interpolation of different views. That is if multiple cameras are used, and thus multiple simulations.
Frequency (F)	Frequency of interpolation turbulence function.
Amplitude	Amplitude of interpolation turbulence function.
Octaves (O)	Number of octaves in interpolation turbulence function.

Lacunarity (l)	The lacunarity of the interpolation turbulence function.
Gain (g)	The gain of the interpolation turbulence function.

Table B.4: Slice interpolation parameters

Parameter	**Description**
Density scaling	Volume density scaling.
Density exponent	Volume density exponent (for contrast).
Intensity scaling	Volume intensity scaling.
Slice intensity	Intensity scaling in slice visualizations.
Phase function ($P(\mathbf{x}, \Theta \leftrightarrow \Psi)$)	For specifying the scattering phase function: 1. Schlick 2. Rayleigh 3. Henyey-Greenstein
Scattering coefficient (g, k)	Scattering coefficient to control the scattering behavior, i.e. the amount of forward or backward scattering. Used in Schlick and Henyey-Greenstein phase functions since the Rayleigh phase function describes an isotropic scattering.
Temperature scaling	Temperature scaling for a simple black body radiation shading, where the temperature is based on the volume density.

Table B.5: Shading parameters

Bibliography

Aitken, M., Butler, G., Lemmon, D., Saindon, E., Peters, D., and Williams, G. 2004. The lord of the rings: the visual effects that brought middle earth to the screen. *In* SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, p. 11, New York, NY, USA. ACM.

Blinn, J. F. 1982. A generalization of algebraic surface drawing. *ACM Trans. Graph.* 1:235–256.

Clinton, A. and Elendt, M. 2009. Rendering volumes with microvoxels. *In* SIGGRAPH '09: SIGGRAPH 2009: Talks, pp. 1–1, New York, NY, USA. ACM.

Cook, R. L., Carpenter, L., and Catmull, E. 1987. The reyes image rendering architecture. *In* SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pp. 95–102, New York, NY, USA. ACM.

Csuri, C., Hackathorn, R., Parent, R., Carlson, W., and Howard, M. 1979. Towards an interactive high visual complexity animation system. *In* SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques, pp. 289–299, New York, NY, USA. ACM.

Desbrun, M. and Cani, M.-P. 1996. Smoothed particles: A new paradigm for animating highly deformable bodies. *In* R. Boulic and G. Hegron (eds.), Eurographics Workshop on Computer Animation and Simulation (EGCAS), pp. 61–76. Springer-Verlag. Published under the name Marie-Paule Gascuel.

Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual simulation of smoke. *In* SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 15–22, New York, NY, USA. ACM.

Foster, N. and Metaxas, D. 1996. Realistic animation of liquids. *In* GI '96: Proceedings of the conference on Graphics interface '96, pp. 204–212, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

Foster, N. and Metaxas, D. 1997. Modeling the motion of a hot, turbulent gas. *In* SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 181–188, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Gingold, R. A. and Monaghan, J. J. 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society* 181:375–389.

Gustavson, S. 2005. Simplex noise demystified. Available at: `http://www.itn.liu.se/~stegu/simplexnoise/` [29/09/10].

Horvath, C. and Geiger, W. 2009. Directable, high-resolution simulation of fire on the gpu. *ACM Trans. Graph.* 28:1–8.

Jensen, H. W. 2001. Realistic image synthesis using photon mapping. A. K. Peters, Ltd., Natick, MA, USA.

Kajiya, J. T. and Von Herzen, B. P. 1984. Ray tracing volume densities. *In* SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 165–174, New York, NY, USA. ACM.

Lokovic, T. and Veach, E. 2000. Deep shadow maps. *In* SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 385–392, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Max, N. 1995. Optical models for direct volume rendering. *Visualization and Computer Graphics, IEEE Transactions on* 1:99 –108.

Neyret, F. 2003. Advected textures. *In* SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 147–153, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Perlin, K. 1985. An image synthesizer. *In* SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 287–296, New York, NY, USA. ACM.

Perlin, K. 2001. Noise hardware. *In* M. Olano (ed.), Real-Time Shading SIGGRAPH Course Notes.

Perlin, K. and Hoffert, E. M. 1989. Hypertexture. *In* SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques, pp. 253–262, New York, NY, USA. ACM.

Rasmussen, N., Nguyen, D. Q., Geiger, W., and Fedkiw, R. 2003. Smoke simulation for large scale phenomena. *In* SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, pp. 703–707, New York, NY, USA. ACM.

Reeves, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *In* SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques, pp. 359–375, New York, NY, USA. ACM.

Schechter, H. and Bridson, R. 2008. Evolving sub-grid turbulence for smoke animation. *In* SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 1–7, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Selle, A., Rasmussen, N., and Fedkiw, R. 2005. A vortex particle method for smoke, water and explosions. *In* SIGGRAPH '05: ACM SIGGRAPH 2005 Papers, pp. 910–914, New York, NY, USA. ACM.

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

Stam, J. and Fiume, E. 1993. Turbulent wind fields for gaseous phenomena. *In* SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, pp. 369–376, New York, NY, USA. ACM.

Williams, L. 1978. Casting curved shadows on curved surfaces. *In* SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques, pp. 270–274, New York, NY, USA. ACM.

Wrenninge, M., Bin Zafar, N., Clifford, J., Graham, G., Penney, D., Kontkanen, J., Tessendorf, J., and Clinton, A. 2010. Volumetric methods in visual effects. *In* SIGGRAPH '10: ACM SIGGRAPH 2010 Courses. Available at: `http://magnuswrenninge.com/volumetricmethods` [29/09/10].

Wyvill, G., McPheeters, C., and Wyvill, B. 1986. Data structure for *soft* objects. *The Visual Computer* 2:227–234.

Zhou, K., Hou, Q., Ren, Z., Gong, M., Sun, X., and Guo, B. 2009. Renderants: interactive reyes rendering on gpus. *In* SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers, pp. 1–11, New York, NY, USA. ACM.