Efficient Volume Raycasting for Interactive SPH Applications

Diploma Thesis in Computer Science

submitted by

Max A. Limper
born on November 17th, 1985 in Aachen

written at

Computer Graphics and Multimedia Systems Group
Department of Electrical Engineering and Computer Science
University of Siegen, Germany.

Advisors:

Prof. Dr. Andreas Kolb

Dipl.-Inf. Jens Orthmann

Started on: October 1st, 2011

Finished on: March 30th, 2012

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Siegen, den 29. März 2012

Abstract

Within this thesis, a new volume raycasting approach for particle-based SPH simulations is presented. The approach is combining two existing methods and can be seen as an alternative which is especially suited for interactive simulation environments. A perspective access structure enables several optimization mechanisms which can speed up the raycaster significantly. Besides that, the ray-based sampling enables a normalization of the samples, following the SPH model. Such a normalization can be used for a correct visualization as well as for a simple contour shading method. Finally, a precise method for gradient computation makes the integration of local volume illumination possible.

Zusammenfassung

Im Rahmen dieser Arbeit wird ein neuer Ansatz zum effizienten Volume-Raycasting für partikelbasierte Simulationen nach dem SPH-Modell vorgestellt. Der Ansatz verbindet zwei bekannte Verfahren und stellt eine Alternative dar, welche insbesondere für interaktive Simulationsumgebungen geeignet ist. Eine perspektivische Zugriffsstruktur ermöglicht verschiedene Optimierungsmechanismen welche die Geschwindigkeit des Raycasters erheblich verbessern können. Des Weiteren ermöglicht das strahlbasierte Vorgehen eine Normalisierung der Samples nach dem SPH-Modell, was neben einer korrekten Visualisierung auch für einen einfachen Ansatz zur Visualisierung von Konturen genutzt werden kann. Ein präzises Verfahren zur Gradientenbestimmung ermöglicht schließlich die Integration lokaler Beleuchtungseffekte.

Acknowledgements

First of all, I would like to thank my parents, brothers and friends, especially my girlfriend Marieke, for their emotional support and understanding during my work on this thesis.

I would also like to thank my supervisors Jens Orthmann and Andreas Kolb for their support and their supervision of my work.

Finally, special thanks go to my fellow student Jan Reckling, who has offered his time to proof-read this work.

Max Limper

Danksagung

Zunächst möchte ich mich bei meinen Eltern, Brüdern und Freunden, besonders bei meiner Freundin Marieke, für ihre emotionale Unterstützung und ihr Verständnis während der Zeit meiner Diplomarbeit bedanken.

Weiterhin gilt mein Dank meinen Betreuern Jens Orthmann und Andreas Kolb für ihre Unterstützung und Betreuung meiner Arbeit.

Schließlich möchte ich meinem Kommilitonen Jan Reckling dafür danken, dass er seine Zeit geopfert hat um diese Arbeit Korrektur zu lesen.

Max Limper

A Note on First-Person Perspective

A diploma thesis always involves a lot of contributions from the supervisors. For the programming part, the SVT framework of the Computer Graphics Group at Siegen University was used. I was able to build my experimental raycasting application on top of an existing SPH simulation that was created by Jens Orthmann. Also, Jens was always at hand when questions concerning the integration into the framework arised, as well as for lots of other questions. The main idea of using a perspective grid together with a ray-based approach, which was the starting point of this thesis, was also founded by Jens and Prof. Andreas Kolb. If this was a scientific paper, I would therefore be probably be named as one of three authors and write my part in the plural form (“We present ...“) instead of using the singular (“I present“) or avoiding any grammatical person at all by employing passive constructs all the time (“... is presented. “). Because of that, and because of personal taste, I decided to use the plural form throughout this thesis.

List of Figures

3.1 The SPH volume raycasting pipeline 10
3.2 Sampling Rate Reduction on User Interaction 12
3.3 Transfer Function Undersampling and Stochastic Jittering 13
3.4 A simplified example of our perspective grid in view space 17
3.5 Perspective grids 18
3.6 Uniform grid cells (2D situation) 20
3.7 Cell size computation in view space 20
3.8 A map from particles to relevant cells 22
3.9 A map from cells to relevant particles 23
3.10 The empty successors cache 24
3.11 The effect of unnormalized sampling 26
3.12 Unnormalized sampling without and with pseudo normalization 27
3.13 Ray bundles and slab caching 31
3.14 Sampling at contour regions 39
3.15 Surface enhancement techniques 40
3.16 Adaptive sampling without opacity correction 41

4.1 Relevant cells for a single particle 48
4.2 Ray vectors 49
4.3 Performance for different slab depths 56

List of Tables

4.1 Different sampling rates along z in our setup 46
4.2 Occupancy calculation for different visualization techniques 52
4.3 Memory consumption of the perspective grid's particle map 53
4.4 Performance of different visualization techniques 55
4.5 Timings for empty space leaping 57
4.6 Impact of the transfer function's opacity on performance 58
4.7 Timings for grid construction 59

Listings

2.1 Launch of a CUDA kernel 5
3.1 Barrier synchronization on shared memory usage 30
3.2 Slab-based sampling and compositing 32
3.3 Normalized sampling 37
4.1 Different CUDA kernels in our GPU program 46
4.2 Transformation of the particles' positions to view space 47
4.3 Mapping from view space to the perspective grid 48
4.4 A simplified raycasting main loop 50
4.5 Collaborative ray parameter computation 54
4.6 Opacity correction 54
4.7 Early ray termination with atomic operations 55

Contents

1 Introduction **1**

2 Background **2**

2.1 GPU-Based Volume Raycasting 2

2.2 Volume Raycasting with CUDA 5

2.3 SPH Volume Raycasting 7

2.3.1 The SPH Model 7

2.3.2 Approaches to SPH Volume Raycasting 8

3 A Highly Parallel SPH Volume Raycasting Solution **9**

3.1 The SPH Volume Raycasting Pipeline 9

3.2 Sampling Rate Reduction on User Interaction 11

3.2.1 Reduction Method 11

3.2.2 Stochastic Jittering 12

3.3 Adaptive Step Size 14

3.4 Perspective Access Structure 16

3.4.1 Conception 16

3.4.2 Grid Construction 18

3.4.3 Empty Successors Cache 22

3.4.4 Summary 24

3.5 Normalization 25

3.5.1 Unnormalized vs. Normalized Sampling 26

3.5.2 Pseudo Normalization 27

3.6 Cached Sampling 29

3.6.1 Ray Bundles 29

3.6.2 Slab-Caching 30

3.7 Shading 33

3.7.1 Gradient-Based Shading 33

3.7.2 Contour Shading 37

3.8 Compositing 41

3.8.1 Opacity Correction 41
3.8.2 Integration 42
3.9 Summary 44

4 Results **45**
4.1 Implementation 45
4.1.1 Framework Integration & Raycasting Setup 45
4.1.2 The CUDA Program 46
4.1.3 Performance Optimization Techniques 53
4.1.4 Measured Performance 55
4.2 Summary & Discussion 60

5 Conclusion **63**
5.1 Summary 63
5.2 Limitations and Future Work 65

Bibliography **70**

Chapter 1

Introduction

Within the last decades, volume raycasting has become a well-known and well-established technique in many fields of application, such as medical visualization, scientific visualization or computer games. A lot of different rendering algorithms, performance optimizations and techniques for several use cases have been presented. However, the application of volume rendering techniques to the visualization of interactive SPH simulations is a relatively young area of research.

In 2010, [FAW10] have proposed an efficient method to render very large SPH datasets that have been computed and pre-processed offline, mostly running at even interactive framerates. At the same time, [OKK10] have shown how an on-the-fly volume visualization for interactive SPH simulations can benefit from caching mechanisms during the raycasting process, which makes interactive framerates possible for small and medium particle sets in an interactive SPH simulation setup. While the first approach uses a scattering method which resamples the particle data onto a perspectively distorted grid, the second one uses a gathering approach which collects the particles contributions along the rays.

Within this thesis, a novel raycasting approach for interactive SPH simulations is presented. The method is based on CUDA and combining a perspective access structure, as proposed by [FAW10], with a ray-driven gathering approach, as used by [OKK10]. Using the example of our CUDA program, performance optimization methods are shown. In addition, several visualization techniques are evaluated, including gradient-based illumination and a simple approach to contour shading.

In chapter 2, a brief overview of the necessary background for SPH volume raycasting with CUDA will be given. Chapter 3 covers the conceptional part of this work. Different aspects of the raycasting process are examined by using a model entitled the *SPH Volume Raycasting Pipeline*. Within the following chapter 4, the basic structure of our CUDA program as well as performance measurements will be presented and discussed. Finally, in chapter 5, a conclusion is made. This includes the highlighting of the limitations of our approach and an outlook on possible future work.

Chapter 2

Background

2.1 GPU-Based Volume Raycasting

In the wide field of visualization, many rendering techniques have evolved within the last decades. The term *volume rendering* describes a range of different methods that can be used to visualize the data of a 3D scalar field ([HKRs$^+$06]). The most popular fields of applications are the visualization of medical datasets, like for CT (computerized tomography) or MRI (magnetic resonance imaging) data, geological data visualization and visualization for different kinds of fluid simulation, usually summarized as computational fluid dynamics (CFD).

An average volume rendering task can become much more time-consuming than the rendering of polygonal meshes and is still a challenge for today's high-performance graphics hardware. Before such hardware was available, volume rendering was performed entirely on the CPU, as described for example by [Lev90]. The basic idea is to sample the volume data along rays that are originating in the virtual eyepoint and travelling through the image plane into the scene. This process is well-known under the name *raycasting*. For each pixel of the resulting image, one ray is cast through the scene. With the limited power of processors those days, compared to today, it was not possible to achieve interactive framerates on a desktop machine, even for small datasets. Hence, a lot of performance optimization methods and variations of the original volume raycasting concept, such as the *shear-warp* algorithm ([LL94]), have been presented.

With the advent of programmable graphics hardware, a new generation of methods suitable for interactive volume visualization evolved. Besides texture-based approaches to volume rendering ([EKE01, RGW$^+$03, KW03]), GPU-based volume raycasting has become a popular approach due to its high flexibility ([HKRs$^+$06, MGS$^+$01]). This was made possible through a programmable fragment processing stage, where the GPU's pixel processing (originally designed for polygonal raster graphics) can be freely programmed. This was a first step towards general purpose computation on the GPU, especially since dynamic looping and branching instructions have been made possible for fragment programs (also called *fragment shaders*) running on the GPU ([SKB+06]). In contrast to texture based volume rendering, many well-known acceleration methods from CPU raycasting can therefore be easily adapted for GPU-based raycasting.

Several techniques exist for the conversion from the sampled scalar values along the rays to pixel colors of the final image. Examples for such methods are maximum intensity projection (MIP), first local maximum and integration according to the emission-absorption model ([HKRs+06]). The last method is the most popular one for many fields of application, since it delivers the most meaningful results. The basic idea is to assume that each point inside the volume is on the one hand emitting light and on the other hand also absorbing an amount of incident light. The volume rendering integral describes how to apply this lighting model during the raycasting process:

$$I(D) = I_0 e^{-\int_D^{s_0} \kappa(t)dt} + \int_D^{s_0} q(s) e^{-\int_D^{s} \kappa(t)dt} ds \tag{2.1}$$

Here, $I(D)$ is the final intensity that reaches the eye. The first part of the equation, to the left of the $+$, describes the lighting contribution of the scenes background I_0 which is at each point t attenuated by the corresponding absorption factor $\kappa(t)$ while travelling through the volume towards the eyepoint. The second part, to the right of the $+$, describes the same for light emitted from any point s inside the volume: the emission $q(s)$ is attenuated by the absorption $\kappa(t)$ of at each subsequent point t along the ray. This integral is usually approximated by using a Riemann sum, with a discrete set of points along each ray to sample the emission and absorption inside the volume. The process of computing the Riemann sum along a ray by iteratively applying the emission and absorption contributions at each sampling point is also called *compositing* ([HKRs+06]). Because of several reasons, it has become popular to traverse the rays from the eyepoint through the image plane (front-to-back), instead of back-to-front as described in equation 2.1. One reason is early ray termination, where the traversal of each ray may terminate as soon as the accumulated absorption from front to back is near to the maximal value. Since the result will change just minimally when such a point is reached, the ray traversal can be finished before each point along the ray has been processed, which can significantly speed up the raycasting application.

The emission and absorption values for each possible scalar value are usually stored as a 1D or 2D texture, the so-called transfer function. During each step along the ray, the scalar value at the current position is determined. The value is then mapped to emission and absorption coefficients via lookups in the transfer function's texture. This step is also called *classification*. Finally, in the compositing step, the result is combined with the emission and absorption that has already been accumulated along the ray. This happens by applying the following equation

$$C_{new} = C_{old} + (1 - \alpha_{old})\alpha_{contrib}C_{contrib}, \tag{2.2}$$

where C_{new} denotes the new color, C_{old} the old color and $C_{contrib}$ the color contribution that has been obtained via the transfer function. In the very most cases, this values will be represented as RGB triplets. In the same manner, the accumulated absorption and the absorption obtained from the transfer function are denoted as α_{old} and $\alpha_{contrib}$ respectively. The absorption for the next step α_{new} is then computed as

$$\alpha_{new} = \alpha_{old} + (1 - \alpha_{old})\alpha_{contrib}. \tag{2.3}$$

This basic ideas are already sufficient to understand the fundamental concept. Within the next section, we will address NVIDIA's CUDA technology, which can be used as an alternative to traditional fragment shader approaches ([SKB$^+$06]) to implement GPU-based volume raycasting.

2.2 Volume Raycasting with CUDA

With the advent of programmable graphics hardware, especially since GPU programs with dynamic loop and branch instructions are possible, it has become more and more popular to exploit the power of modern GPUs for other purposes than pure graphics programming. This concept is usually referred to as *General Purpose Computing on the GPU*, or short GPGPU ([OLG$^+$07]). With the release of the first CUDA (*Compute Unified Device Architecture*) SDK in 2007, NVIDIA presented a hardware and software concept for GPGPU. CUDA-capable GPUs provide a hardware interface for the CUDA API, which is based on C programming with minimal extensions specific for GPU computing. Using CUDA for a highly parallel task within an application makes it possible to solve this task in a fraction of the original time which would be needed for a sequentially running CPU program. This is due to the CUDA hardware architecture, which can process many thousand threads in parallel without much overhead for thread scheduling ([KH10]). The GPU program which is executed for all threads in parallel is called the *kernel*. The programmer will have to specify how many threads should be launched, and how those threads are grouped into thread blocks. Each thread block can accommodate a fixed number of threads at maximum. An example can be seen in listing 2.1, where the CUDA kernel function `myKernel(int x)` is launched using the operator `<<<...>>>`, one of CUDAs extensions to the C programming language ([NVI11]).

```
int x = 23;

//configure thread setup variables
dim3 threadsPerBlock(8, 8, 1);
dim3 blocks(12, 12, 1);

//launch the CUDA kernel, for all threads in parallel
myKernel<<<blocks, threadsPerBlock>>>(x);
```

Listing 2.1: Launch of a CUDA kernel.

Each block of threads can use a special barrier synchronization function `syncthreads()` to synchronize all of its threads at a certain point within the kernel function. This is an important feature which enables the threads to operate collaboratively on parts of the input data. Besides the main graphics memory, which is in the context of CUDA referred to as the *global memory*, there are also other kinds of memory within the CUDA model. One of those is the *shared memory*, which is allocated per thread block. This special on-chip memory has much less access latency than the global memory and is therefore well-suited for operations that are performed collaboratively by all threads within a block. It can this way furthermore reduce the amount of local memory needed by each thread, which may lead to more threads running in parallel. Since local memory is a special subset of the global memory, it has the same access latency. Besides the local memory, a fixed number of on-chip registers with very low access latency is assigned to each thread. As the amount of possible per-thread registers on the hardware is very limited, the CUDA compiler will usually decide to keep only the most frequently accessed variables within a kernel in registers.

Since it provides a flexible programming model and promises high performance for parallel tasks, CUDA has become popular for raycasting applications. It is especially an alternative to the fragment shader approach, which is still connected to the traditional graphics pipeline, originally intended to render polygonal data. [MHS08] and [MRH10] have tested CUDA implementations for volume raycasting against the traditional approach based on fragment programs. Of course, changing the API will not change the underlying hardware, therefore the difference in performance can not be expected to be very huge. However, they find that CUDA programs offer more chances to perform low-level optimizations by hand. This is due to the flexibility of CUDA, where the programmer has more possibilites to configure the execution on the hardware directly, e.g. by deciding over the number of threads per thread block and by managing different types of memory. Of course, this additional freedom comes at the cost of additional responsibility. In order to write an optimized CUDA program, an in-depth understanding of the CUDA architecture is a prerequisite ([KH10]).

Within the next section, we will explain the SPH model and furthermore describe existing approaches to SPH volume raycasting, with and without CUDA.

2.3 SPH Volume Raycasting

2.3.1 The SPH Model

In section 2.1, we have mentioned that one field of application for volume raycasting is computational fluids dynamics (CFD). The *Smoothed Particle Hydrodynamics* (SPH) model can be seen as a CFD method, although its origins are in astrophysical simulations ([Mon05]). It is a so-called *Lagrangian* method, which means that no explicit grid is used to discretize the values of the fluid within the simulation domain. Instead, the fluid is represented as a set of particles that move with the fluid. For each particle, physical properties like mass, density or the concentration of a substance within the fluid are stored and transported along with the particle. If we want to visualize such properties, we have to be able to obtain the scalar value of the property at a random point in space. Therefore, the question arises how the scalar field can be evaluated, using the particle data. This is brings us to the core idea of the SPH model, which is the application of a so-called *smoothing kernel.* The smoothing kernel is a function $W(d,h) : \mathbb{R}^2 \to \mathbb{R}$ that is used to determine the weight of each particle's contribution to the scalar value $Q(\vec{x})$ at point $\vec{x}$ by evaluating

$$Q(\vec{x}) = \frac{\sum_i Q_i \cdot V_i \cdot W(\|\vec{x} - \vec{x}_i\|, h)}{\sum_i V_i \cdot W(\|\vec{x} - \vec{x}_i\|, h)}, \tag{2.4}$$

where the positions, scalar quantites and volume of the particles are denoted as $\vec{x}_i$, Q_i and V_i respectively. As can be seen, the first parameter to the kernel function is the distance of the corresponding particle to the point of evaluation. The second parameter is the particle's smoothing length, which is constant within many applications. The smoothing length describes the limited influence radius of the particle. Hence, all particles further away than h will not contribute to the point of evaluation. An example for a kernel function is the *poly6* kernel, which was used within our application throughout and is defined as

$$W_{poly6}(d,h) = \begin{cases} \frac{315(h^2-d^2)^3}{64h^9\pi}, & \text{if } d \le h \\ 0, & \text{else.} \end{cases} \tag{2.5}$$

Usually, the kernel function is radially symmetric like the poly6 kernel, although other approaches have been proposed ([YT10]). A lot of work has been done to optimize the accuracy of SPH models as well as the performance of CPU-based SPH simulations ([Her94, CM99, APKG07, BK02]). With the appeareance of programmable graphics hardware, it has become possible to perform the simulation entirely on the GPU in a convenient way, enabling interactive SPH simulations by using the standard graphics pipeline ([ZSP08]) or CUDA ([OKK10]). Note that SPH is very well-suited for a parallelization via CUDA, since a lot of computations have to be performed particle-wise during each time step of the simulation.

2.3.2 Approaches to SPH Volume Raycasting

To visualize the scalar quantites of an SPH simulation, volume rendering techniques need to be employed. In this context, two basic approaches to volume raycasting can be identified. Scattering approaches ([SDG08, FAW10]) are resampling the particles scalar quantites onto a grid. The scalar values inside the grid can then be visualized using standard volume raycasting approaches, as used for e.g. medical datasets. In contrast, gathering approaches ([OKK10, ZSP08]) evaluate the scalar field directly at each sampling point along the rays. In order to be efficient, such approaches need a mechanism to obtain a subset of potentially relevant particles. Therefore, in contrast to scattering approaches, spatial access stuctures like octrees have to be employed.

A recent example of a scattering approach, using programmable fragment shaders, can be found in [FAW10]. The key idea is to resample the scalar quantities of the particles onto a perspective grid which discretizes the view frustum. This process involves a high amount of memory consumption, since each sampling point along each ray is represented by a point inside the perspective grid. Nevertheless, once the resampling step is finished, the actual raycasting process is performed in a pretty straightforward way. Since the points of the grid are already perspectively arranged to exactly match the samples along the rays, the raycasting procedure boils down to the classification and compositing steps. To make the expensive resampling process more efficient, they are organizing the particles inside an octree. This makes it possible to obtain a subset of the whole particle set, which is useful to include only particles that are potentially located inside the view frustum into the resampling process. Such an optimization is especially necessary because they are visualizing non-interactive, precomputed data with many millions of particles.

In contrast, [OKK10] have proposed a gathering solution for interactive SPH simulations, based on CUDA. As a spatial access structure, they are using a data-parallel octree that is built entirely on the GPU, following the proposal of [ZGHG11]. In their publication, three different caching mechanisms are proposed in order to make the gathering process more efficient. The node cache is the first caching mechanism and used to store the identifiers for the currently relevant octree nodes in shared memory for a block of threads, representing a bundle of rays. The second mechanism, called the influence cache, is realized by their octree structure. Each node includes also particles that do not have their center within the node, but are instead spreading into it from a neighboured one. This way, additional node fetches during the sampling process can be avoided. Finally, the slab cache is proposed as a third caching mechanism, which follows the proposal of [MRH10] for the raycasting of regular, grid-based volume datasets.

Chapter 3

A Highly Parallel SPH Volume Raycasting Solution

3.1 The SPH Volume Raycasting Pipeline

For our application, we have decided to use a gathering approach to SPH volume raycasting (see section 2.3.2). Within this chapter we will discuss the conception of our raycaster and see several reasons that are justifying such a decision. Using a gathering approach involves the process of following rays through the image plane and collecting the particles contributions in order to determine the values of the scalar field at each sampling point along each ray. The scalar values are then mapped to emission and absorption coefficients by a transfer function. Those coefficients are in turn used to evaluate the emission-absorption model along the ray. From a conceptual point of view, this is already the description of the whole raycasting process. However, if we want to implement this process in an interactive application under real-time demands, a couple of problems needs to be solved.

Following the concept of the volume rendering pipeline shown in [HKRs$^+$06], we can divide the raycasting process into several stages to obtain a more specific SPH volume raycasting pipeline (see figure 3.1) through which we are running each time we are generating a new raycasting result. Each stage of the pipeline corresponds to a problem which can be solved in various ways, basically independent from the rest of the pipeline.

At first, we have to set up the rays. We have to decide over the number of rays we want to use as well as over the starting point of each individual ray. After this very first step, we iterate through the following recurring stages of the SPH volume raycasting pipeline. During the ray traversal stage, we move forward along the ray to the next sample position. This can, for example, involve the skipping of emtpy space. The particle access stage describes the localization of particles that might be relevant for the current sampling point, which is made possible by a spatial access structure, like a uniform grid, for instance. During the sampling stage, we compute the value of the scalar field which we want to visualize. This is

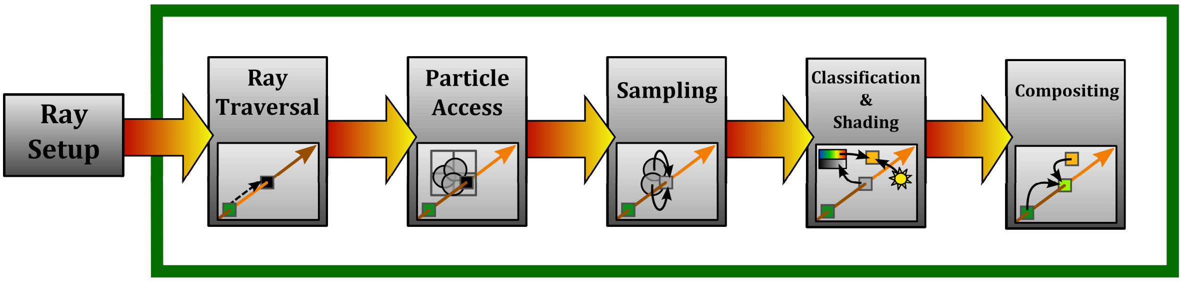
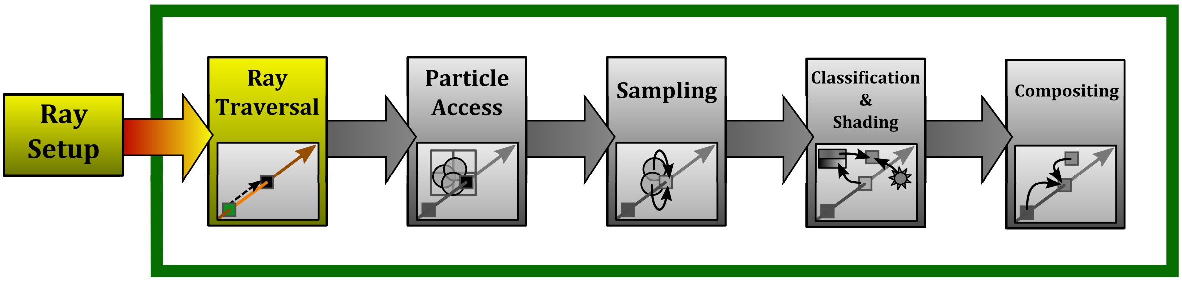


Figure 3.1: The SPH volume raycasting pipeline. The content of the green box shows the actual pipeline which is executed a lot of times while following a ray through the scene.

achieved by computing the weighted sum of all relevant particles contributions. The following classification stage maps the sampled scalar value to emission and absorption values, which are, for example, represented as an RGB color and an alpha value. Finally, the compositing stage applies those emission and absorption values for the current sample before we continue with the ray traversal for the next one.

Within the following sections, we will be discussing all important features of our raycaster at each stage of the SPH volume raycasting pipeline.

3.2 Sampling Rate Reduction on User Interaction



Because we are dealing with an interactive setup, allowing fluent user interaction is an important aim of our application. For the raycaster, this means that we have to provide methods to perform the raycasting process at different precision levels, since reducing the sampling rate will boost the performance of our application, which might be necessary for fluent user interaction. Within this section, we will present the methods we have applied to achieve this.

3.2.1 Reduction Method

In order to raise the framerate of the raycasting application to a number which is suitable for fluent user interaction, two basic ways of reducing the sampling rate can be identified. On the one hand, we can reduce the image plane's resolution. On the other hand, we can reduce the number of sampling steps along each ray. Both methods can be found in common volume rendering frameworks ([MSRMH09, KP$^+$11]). In our application, we are always using a combination of both. Since, following [FAW10], our choice for the sampling rate along the rays is directly linked to the resolution of the image plane (see section 3.3), we always reduce the number of sampling steps per ray if we choose to downscale the image plane for raycasting. We are furthermore following the *Interactive Speed*-approach used in [KP$^+$11], since we found it to be the best working solution. The idea is to decrease the sampling rate on the image plane and along the rays equally, until a desired speed of the raycasting application is reached. In our application, the reduction is expressed by a scalar factor $r \in \mathbb{R}, r \in [1, r_{max}]$ which is adjusted every frame after checking the current framerate. This happens according to the following formulation

$$r_{new} = \begin{cases} min(r_{old} + 0.25, r_{max}), & \text{if the camera moved} \\ r_{old}, & \text{else, if } t_{last} < 0.25 \\ 1, & \text{else} \end{cases} \tag{3.1}$$

, where t_{last} denotes the elapsed time since the last camera movement, r_{new} the reduction factor which will be used to render the next image and r_{old} the one used for the last image respectively. If the raycaster needed more time to generate the last image than what would be allowed according to our desired reference frame rate, e.g. 12 frames per second, we adjust the coarseness factor by 0.25. We found this to be a useful value in our particular setup, but of course this adjustment value may vary from case to case - the choice is a balance between a fast reaction on user interaction, leading to a higher adjustment value, and the avoidance of a fast over-reaction, limiting that adjustment value. Besides this adjustment, which follows the *Interactive Speed*-approach of [KP$^+$11], we add a cooldown period of a quarter-second in which the sampling rate reduction is kept before returning to 100% quality (see equation 3.1). This additional feature is useful because, in a typical case, the user will often move the camera, release the mouse button and continue with the next camera adjustment. If the user wants, for example, to see a special feature of the scene from the opposite direction, he will probably rotate the camera by 180 degrees and then release the mouse button. Then, using another button, he will zoom in, in order to see the feature of interest. Between those two phases of interaction, an instant slowdown due to the direct return to 100% quality rendering will be found disturbing, which is the reason for using the cooldown period.

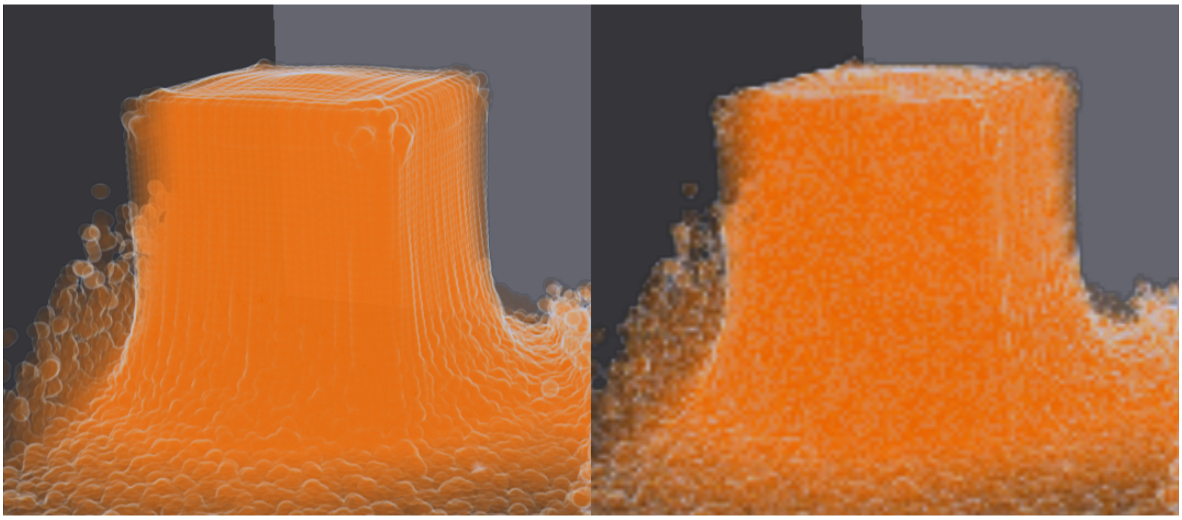


Figure 3.2: The left image was rendered with 100% quality. The right image was rendered with a reduction factor of 4, applied to the image plane resolution as well as the number of samples along the rays. The low-resolution rendering result is interpolated when being displayed inside the viewport, using bilinear interpolation. The undersampling along the ray direction is causing image noise instead of wood-grain artifacts due to stochastic jittering (see 3.2.2).

3.2.2 Stochastic Jittering

We have also implemented the concept of stochastic jittering, which is not only useful when we are using a lowered sampling rate along the rays, as shown in figure 3.2. It can also help to hide artifacts caused by an undersampling of the transfer function, which might occur even if we are performing a high-resolution sampling of the scalar field. Thinking about the sampling of the transfer function, we find that the sampling rate we need for a satisfying result is independent from the sampling rate we need to sample the scalar field. This problem has already been described in [HKRs$^+$06]. A good solution is to use pre-integrated transfer functions as proposed by [EKE01]. The main idea of the approach is to use a pre-computed high-quality sampling of the transfer function, assuming a linear progression between two subsequent sampled values. However, in our application, we are not really running into problems with possible classification artifacts. This is due to the fact that the transfer function, and the scalar field as well, will usually not contain very high frequencies. The only case where such a situation will certainly occur is when we are performing an unnormalized sampling, which introduces high frequencies into the sampled scalar field. This topic will be addressed in section 3.5. A resulting image from our application using unnormalized sampling with stochastic jittering is shown in figure 3.3.

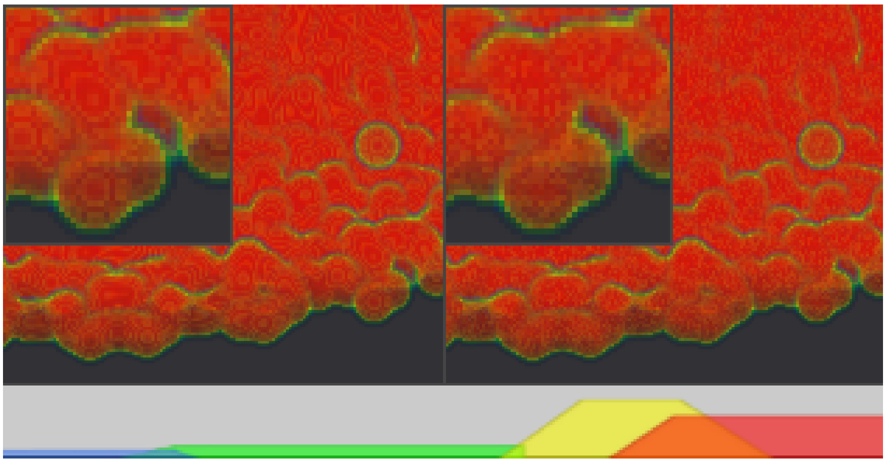
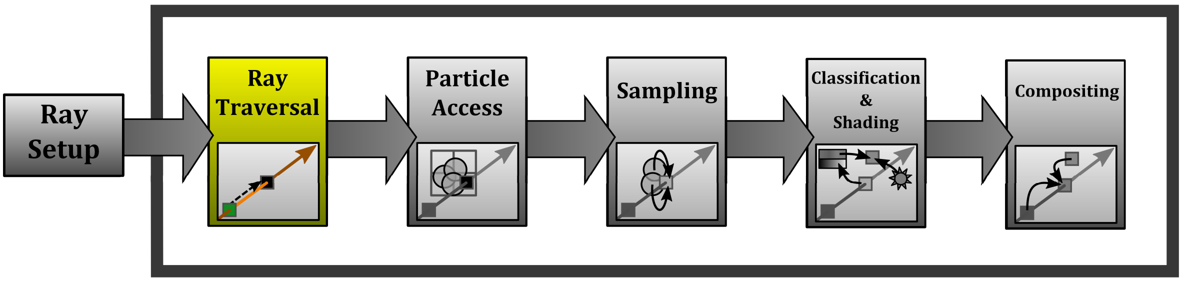


Figure 3.3: Transfer Function Undersampling and Stochastic Jittering. In the bottom, the transfer function is depicted. The left image shows a regular rendering without stochastic jittering, revealing the typical wood-grain artifacts, caused by an undersampling of the high-frequency emission and absorption values when using an unnormalized sampling (see section 3.5). The right image shows the result of a rendering with the same number of samples and stochastic jittering.

3.3 Adaptive Step Size



Instead of a uniform sampling rate, we are using an adaptive sampling mechanism as proposed by [FAW10]. The number of samples per ray $m \in \mathbb{N}$ is computed by taking the image plane resolution in y-direction (res_y, which is usually lower than the resolution in x-direction) as well as the vertical opening angle (fov_y) of the view frustum into account:

$$m = \frac{\ln\left(\frac{f}{n}\right)}{\sigma} \tag{3.2}$$

, where f denotes the distance of the far clipping plane from the viewer, n the distance of the near clipping plane respectively and σ denotes the size of a single pixel on the image plane at a distance of 1, with

$$\sigma = 2 \cdot \frac{\tan(\frac{fov_y}{2})}{res_y} \tag{3.3}$$

, as already derived by [FAW10]. If the reader is further interested in the computation of σ, we would like to redirect to section 3.4, especially figure 3.7, where the computation of the cell size on the image plane is performed analogous. The resulting value of m is then adjusted by an additional factor λ, which scales m with respect to the frustum boundaries, where the z-axis aligned sampling along the rays will lead to an increasing stepsize compared to rays at the center of the frustum (see [FAW10]):

$$\lambda = \sqrt{\left(\frac{res_x \cdot \sigma}{2}\right)^2 + \left(\frac{res_y \cdot \sigma}{2}\right)^2 + 1}. \tag{3.4}$$

It is easy to see in equation 3.2 that our number of samples per ray is linearly dependent on the resolution of our viewport. As also can be seen in the equation, it is especially crucial for the number of sampling steps in this approach how close we choose our near and far clipping plane. The opening angle of the camera is another factor. The example results shown by [FAW10] are generated using a wide opening angle and far and near clipping distances with a small ratio. This enables them to use a number of sampling steps along the rays which equals almost the resolution in x and y. In order to implement the adaptive stepping, we need a function which maps each step to the corresponding point along the ray. Following [FAW10], we are defining a mapping between the index of the sample point, here called the *Ray Coordinate* $\lfloor u \rfloor$, to the corresponding distance along the ray t and vice versa. Using the number of steps m which was derived above, the relation between $t \in \mathbb{R}$ and $u \in \mathbb{R}$ is defined as:

$$t = n \cdot \left(\frac{f}{n}\right)^{\frac{u}{m}} \tag{3.5}$$

This equation can, obviously, directly be used to compute t for a given u. To receive an equation which describes the inverse mapping, we first solve for u:

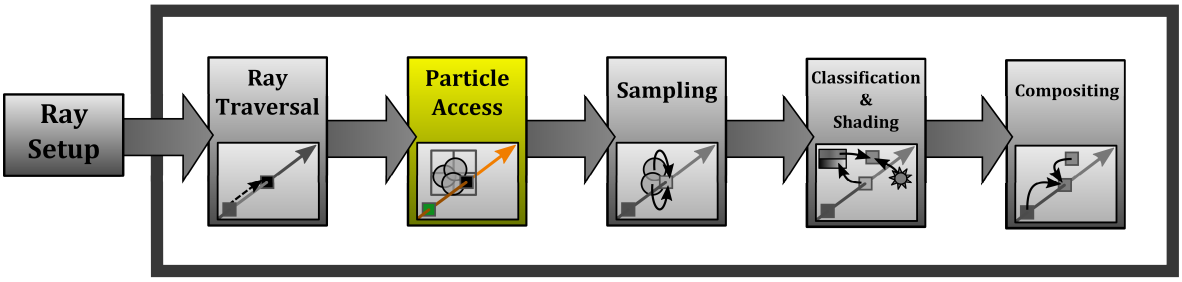
$$t = n \cdot \left(\frac{f}{n}\right)^{\frac{u}{m}} \Leftrightarrow \frac{t}{n} = \left(\frac{f}{n}\right)^{\frac{u}{m}} \Leftrightarrow \frac{u}{m} = \frac{\ln\left(\frac{t}{n}\right)}{\ln\left(\frac{f}{n}\right)} \Leftrightarrow u = m \cdot \frac{\ln\left(\frac{t}{n}\right)}{\ln\left(\frac{f}{n}\right)} \tag{3.6}$$

Then, since $\lfloor u \rfloor$ is an integer coordinate, we still have to compute the floor to receive the final result:

$$\lfloor u \rfloor = \left\lfloor m \cdot \frac{\ln\left(\frac{t}{n}\right)}{\ln\left(\frac{f}{n}\right)} \right\rfloor \tag{3.7}$$

Since we are using a different stepsize for each ray segment, the weight of those segments during the compositing step should not be equal. Using the standard compositing way as for a uniform stepsize would lead to a change in opacity if we move our virtual eyepoint closer towards the volume or farther away from it: due to the adaptive sampling rate there will be more or less steps which sample the scalar field inside the volume, depending on the distance to the eyepoint. We will show how to deal with this problem in section 3.8.

3.4 Perspective Access Structure



Since it would be incredibly inefficient to walk along a ray and consider all particles at each sampling point, we need a spatial access structure to limit the amount of particles taken into account during the sampling to a compact set of particles close to the current sampling point. This can be achieved by various access structures. An example is the data-parallel Octree used by [OKK10], mainly following the proposal of [ZGHG11], which can be efficiently constructed in parallel by computing the relevant cells for each particle on every frame. As already noted by [Gre10], the size of the simulation domain has a direct impact on the size of every data structure which is aiming to discretize this domain, like a uniform grid. [IABT11] are presenting a method to deal with infinite domains, at the cost of potential hash collisions. The scattering method proposed by [FAW10], finally, uses a view-aligned perspective grid of fixed size.

3.4.1 Conception

Within this work, we are proposing a novel spatial access structure which is used to access the particle set in an efficient way during the sampling process. In contrast to existing gathering approaches, we are using a perspective grid which discretizes our view frustum (see figure 3.4). The basic structure of the grid is following the main idea of [FAW10], which is to construct a perspective grid in view space with the same sampling rate along x, y and z. This means that our number of cells along the ray direction, which corresponds to the direction of the z-axis in view space, is computed by taking the resolution of the image plane into account. In contrast to [FAW10], however, we are using a much lower resolution. This is due to the fact that we do not want to scatter the particle values directly onto our grid, instead our aim is to store references to all relevant particles for each cell of the grid, like we also would for a uniform grid or the finest level of an octree. As already implied, this approach has the advantage that we do not have to care for the size of our simulation domain, because we are discretizing our view frustum instead.

Still, the major aim of this approach is to build an access structure which can easily be used to obtain a candidate subset of the whole particle set as potentially relevant particles for a given sampling point along the ray. The traversal of such a view-aligned structure

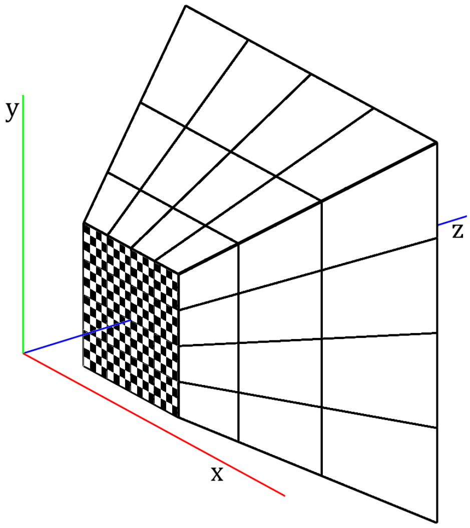


Figure 3.4: A simplified example of our perspective grid in view space. Note how the cells are aligned to equally sized pixel tiles on the image plane, as well as to the paths of the rays through those pixels.

is pretty straightforward, because the cell search in x and y direction has to be performed only once, while the z-index of the current cell is just depending on the current value of the ray parameter. This is an advantage over octrees, because we do not have to perform any ray-cell intersection computations. Our cell structure is subdividing the image plane into equally sized tiles, so we can assume that each cell is traversed by the same number of rays, for example 16 × 16. This can be advantageous when performing the raycasting on a highly parallel computing device, as we will see in section 3.6.

Furthermore, we can easily handle an adaptive sampling rate along the rays. Using the same adaptive sampling mechanism for cell size and sampling stepsize, we can ensure that each cell contains a fixed number of sampling steps along z, which is just the relation between the sampling resolution and the grid resolution, expressed through the ratio between the size of a pixel σ (see section 3.3) and the size of a cell $s \in \mathbb{R}$, both measured on the image plane. In our application, for instance, we are preferring a relation of $1 : 16$, which means that each grid cell contains 16 adaptive steps along the ray. Of course, this whole concept can also be used with a fixed sampling rate, in that case we will just have to use a uniform cell size in z-direction. Figure 3.5 shows the two possibilities. Nevertheless, we are preferring an adaptive sampling along the rays, as described in section 3.3, so we are also sticking to the adaptive cell size along z.

Since we want the number of sampling steps in each cell to be constant, we have to adjust the original number of steps along the ray m (see section 3.3) when using our perspective grid. The final value of the number of steps, $m_{adjusted}$, can then be obtained by computing the next number which is divisible by the number of samples per cell $\left(\frac{s}{\sigma}\right)$:

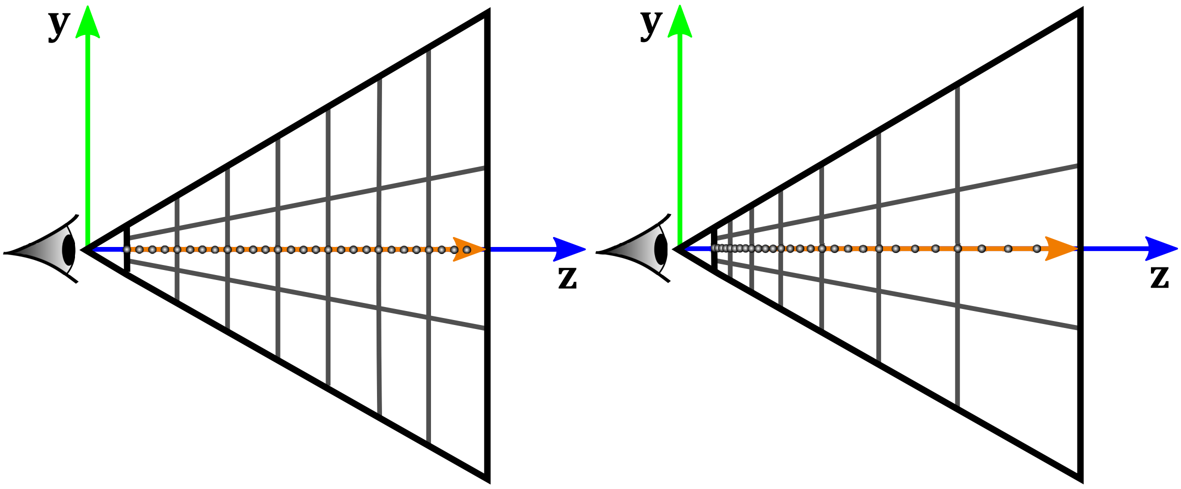


Figure 3.5: Perspective grids. The left image shows a perspective grid with a uniform cellsize along the z-axis, which is used together with a uniform stepsize along the rays. The right image shows a perspective grid like ours that has an an adaptive cell size along the z-axis, used with an adaptive stepsize along the rays. Both grids have the special property that each cell contains a fixed number of sampling points (in this example: 4).

$$m_{adjusted} = \left(\frac{s}{\sigma}\right) \left\lceil \frac{m}{\left(\frac{s}{\sigma}\right)} \right\rceil . \tag{3.8}$$

The number of cells along z $cells_z$ is then defined as

$$cells_z = \frac{m_{adjusted}}{\left(\frac{s}{\sigma}\right)}. \tag{3.9}$$

As already implied, the special properties of our access structure, which is aligned with the rays inside the view frustum on the one hand and the sampling points along the rays on the other hand, are very well-suited for a fast and highly parallel raycasting implementation. We will discuss this application of our perspective grid in section 3.6. In the rest of this section, we will now cover the process of grid construction.

3.4.2 Grid Construction

In our approach, as usual for grid-based methods (see [Gre10]), an identifier is assigned to each cell of the grid. For a grid of $N_x \times N_y \times N_z$ cells and a cell with the 3D grid coordinate c, this value is just the linear index of the cell in memory, computed the following straightforward way:

$$IDX(c) = c_z \cdot N_x \cdot N_y + c_y \cdot N_x + c_x \tag{3.10}$$

In the first step of grid construction, for each particle, we need to find all cells to which the particle is contributing and store the indices of those cells. It is also possible to store, for each particle, just the single cell it is located in, using the particles center (see [IABT11]). This is a useful approach for grids that are used for the actual SPH simulation, since such a simulation is particle-driven, i.e. each particle can check a fixed number of cells (e.g. 27) to obtain its neighbourhood. However, in our case, this shifts workload from the grid constuction to the critical point of cell traversal during the raycasting, which we want to avoid since we may not evaluate too many irrelevant particles during the sampling. Instead, we are following [OKK10] and computing all relevant cells of each particle during the grid construction for fast access during the cell traversal. A common way to achive this for uniform grids or octrees is to allocate a fixed number of slots in memory for each particle, since, for uniform cells, we can assume that each particle can just contribute to a fixed number of $n_{max} \in \mathbb{N}$ cells at maximum. Usually ([Gre10, OKK10, IABT11]) the cell size is chosen to be equal to the particle diameter. This way a particle can contribute to eight cells at maxium, i.e. $n_{max} = 8$ (see figure 3.6). For a bigger cell size, a particle might still contribute to eight cells at maximum, since it might still spread across the cell's border into the next cell in each dimension if it is closer to the cell's border than its radius, so eight cells is always the minimal value of n_{max}. For a smaller cell size, we can compute n_{max} by finding the smallest integer number of cells that is able to enclose the particle in each dimension. Because the particle might still reach into a neighboured cell, as already mentioned, we have to add one to this value. The result has then to be taken by the power of three to account for all dimensions. Since we are assuming cubical cells, we can divide the particles diameter by the cell size and round up, which equals the number of cells that is able to enclose the particle in any of the three dimensions. With a cell size of s and a particle diameter $h \in \mathbb{R}$, this means that the maximum number of cells $n_{max}(s, h) : \mathbb{R}^2 \to \mathbb{N}$ is computed according to the following equation:

$$n_{max}(s, h) = \left(\left\lceil \frac{h}{s} \right\rceil + 1 \right)^3 . \tag{3.11}$$

After allocating the memory that is potentially needed to store all particle-cell connections, the relevant cells are determined in parallel. Each thread computes the relevant cells for a single particle and writes the result to a section of $n \leq n_{max}$ entries in the previously allocated memory, corresponding to the particle's relevant cells. For slots that remain unused, because the particle is contributing to less than n_{max} cells, a special index `INVALID_CELL_INDEX` is written.

In our approach, however, it is a little bit more difficult to determine n than for a uniform grid. Since the size of a cell varies among the view frustum, we have to consider the finest possible cell size, which is the subset of cells with a zero z-index, located directly in front of the near plane, as can also be seen in figure 3.5. This cell size depends on the size of a cell on the image plane. In our application, we have chosen a size on the image plane of 16×16 pixels (using a resolution of 512×512 pixels) for those cells, which leads to 32 cells in x-direction

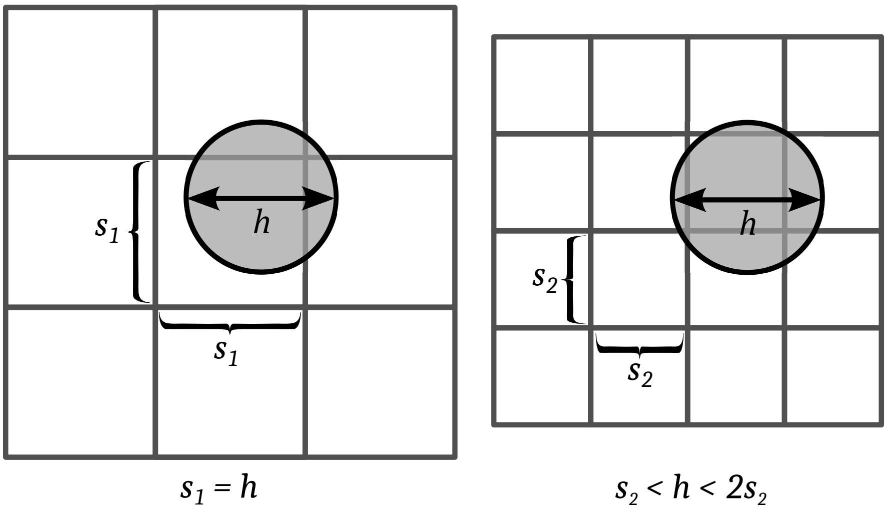


Figure 3.6: Uniform grid cells (2D situation): in the left image, the cell size equals the particle diameter. It is easy to see that each particle can contribute to two cells at maximum in each dimension. In the right image, the size of a single cell is smaller than the particles diameter, but the size of two cells is still bigger. One can see that a particle might contribute to three cells at maximum in each dimension.

and 32 cells in y respectively. Knowing that the sampling rate along z is the same as along x and y, and knowing that the cells just get larger while moving along z inside the frustum, we can assume a uniform cell size with the size of a cell on the image plane in view space to compute n_{max}. This size is depending on distance of the near clipping plane n as well as on the vertical opening angle fov_y, as can also be seen in figure 3.7.

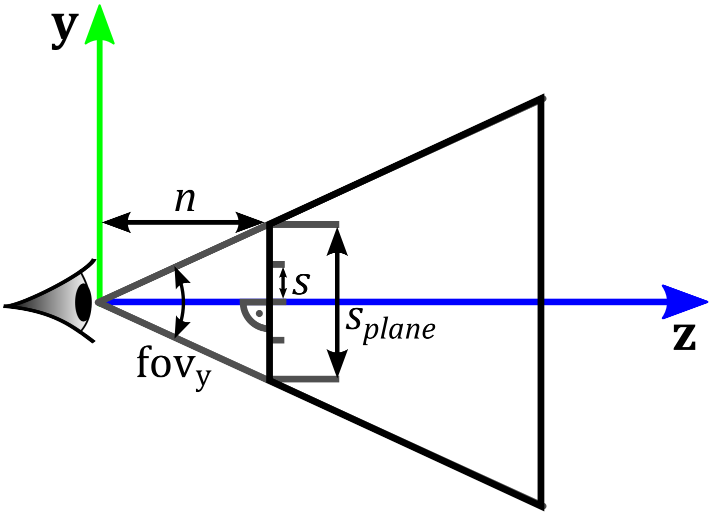


Figure 3.7: Cell size computation in view space. Parameters that are important to compute the size of the image plane in view space are the distance from the virtual eyepoint to the near clipping plane n and the vertical opening angle of the frustum fov_y.

Looking at the right triangle depicted in figure 3.7, we can see that half of the image planes size $s_{plane} \in \mathbb{R}$ equals the length of the rightmost side of the triangle in the image. The upper side of the triangle has a length of n, so we can compute the size of half of the image plane by using the product of n with the tangent of half of fov_y:

$$\frac{s_{plane}}{2} = n \cdot \tan\left(\frac{fov_y}{2}\right) \tag{3.12}$$

If we multiply the result by two, we obtain the size of the image plane s_{plane}. After dividing this by the number of cells in y, we will finally get what we we are looking for, the size s of a cell on the image plane:

$$s = n \cdot \left(\frac{2 \cdot \tan\left(\frac{fov_y}{2}\right)}{cells_y}\right) \tag{3.13}$$

Note that this corresponds exactly to the computation of the pixel size in the approach of [FAW10], with the sole difference that we use $cells_y$ instead of the vertical viewport resolution res_y. In our example setup, the near plane is located 40 units away from the viewer, the vertical opening angle equals 60 degrees. According to the above equation, s equals to approximately 1.44 units for those parameters. Having a global particle radius of 1 unit in our simulation, we find that a particle might contribute to more than eight cells, since our cell size is smaller than the particle diameter. Nevertheless, the size of two cells is still bigger than the particle diameter, so that a particle might cover at maximum three cells in each dimension. This leads us to the conclusion that $n_{max} = (2+1)^3 = 27$, i.e. a particle might contribute to 27 cells at maximum.

Compared to the eight relevant cells that are usually used with uniform grids, this is clearly a drawback of our approach, especially because most of the 27 slots are not really used, since the cell radius gets much bigger than the original 1,44 units while moving along the z-axis inside the frustum (see figure 3.5). Besides that, we also have to perform more checks, which are in addition also using a more complex geometry, since, in view space, each cell is represented by a small frustum, instead of a simple cube.

At this point, one might wonder if we are consuming too much memory when allocating the slots for each particles relevant cells, with $n_{max} = 27$ slots for each particle. Using $N_{particles}$ particles and assuming that the cells indices will be stored as 4 byte integer values, we can compute the space which is necessary to store all relevant cells for each particle by computing $n_{max} \cdot N_{particles} \cdot 4$. For an example case $n_{max} = 27$ and $N_{particles} = 524,288$ we will need to reserve $27 \times 524,288 \times 4$ byte $= 54$ MiB. As can be seen, for our choice of n=27, the memory consumption stays moderate, even for an amount of particles that is relatively large for an interactive SPH setup, so we are accepting the memory consumption at this point and referring to a more detailed analysis in section 4.1.

In order to be able to use the perspective grid, the mapping from particles to cell indices which we have constructed needs to be reversed, as described by [Gre10]. In order to do so, we need another array in memory that is used along with the array of relevant cells. This doubles the memory consumption which has been described above, but, as already implied, this is no problem for our interactive medium-scale setup. The second array holds the particle indices, i.e. each particle writes its index in each of its n_{max} slots in that array. After the parallel computation of relevant cells, this leads to a map consisting of two arrays, as depicted in figure 3.8.

Cell Indices:	**...**	14	7	*INV*	*INV*	14	79	42	7	**...**
Particle Indices:	**...**	**23**	23	23	23	**24**	24	24	24	**...**

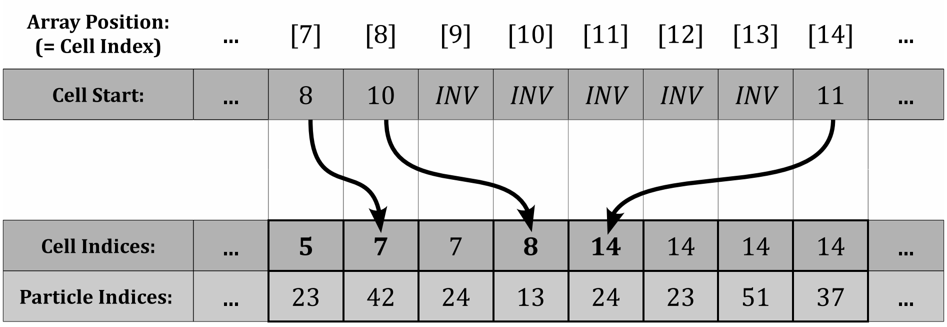
Figure 3.8: A map from particles to relevant cells, represented by two arrays. If a particle is contributing to less than n (here: n = 4) cells, a special identifier `INVALID_CELL_INDEX` is written (here abbreviated as INV).

In order to reverse the mapping to obtain a mapping from cells to particles, we need to sort all entries of the map by the content of the cell index array. This can, for example, be efficiently done by using a radix sort. This widely-used sort method can be executed with several threads running in parallel, each processing a section of the input array, which can lead to a significant speedup compared to non-parallel sorting algorithms, as shown by [SHZO07].

Once sorted, we already have the basic access structure we are looking for. However, we do not know at this point at which position in the array the list of relevant particles for a certain cell starts and where it ends. Following [Gre10], we are computing those array indices for each cell and storing them in two additional arrays, called *cell start* and *cell end* (see figure 3.9). Each of those arrays has the same size as the number of cells. Assuming 4 byte indices again, this leads to 252 KiB per array for our example setup of 32 x 32 x 63 cells, so it is easy to see that the memory consumption caused by this additional structure is neglectable. In order to obtain the indices, we are using as many threads as there are entries in the map, i.e. $N_{particles} \times n_{max}$. Each thread checks if the corresponding map entry has a succeeding entry with a different cell index. If so, the thread has detected the last particles index for the corresponding cell, as well as the first particle index for the succeeding entries cell. In this case, the thread writes those two values to the cell start and cell end arrays.

3.4.3 Empty Successors Cache

During the ray traversal in our interactive raycasting setup, we will encounter many samples with a zero value. This is due to the fact that the distribution of particles inside the simulation domain and thus also inside our view frustum will be, in contrast to other setups like for the large non-interactive datasets used by [FAW10], relatively sparse. Therefore, one might think



Array Position: (= Cell Index)	...	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	...
Cell Start:	...	8	10	*INV*	*INV*	*INV*	*INV*	*INV*	11	...
Cell Indices:	...	**5**	**7**	7	**8**	**14**	14	14	14	...
Particle Indices:	...	23	42	24	13	24	23	51	37	...

Figure 3.9: A map from cells to relevant particles. The start index and end index of each cells section inside the map are stored in two separate arrays, one for cells start index and one for the cells end index. Here, the cell start array is shown. Each array position correspond to a cell with the same index. As can be seen, cell 7 starts at index 8, cell 8 at 10 and cell 14 at 11, while cells 8-13 are containing no particles, so their start and end indices are set to the special identifier INV.

about approaches to skip empty areas during the sampling. This will significantly reduce the number of samples taken into account and therefore improve the performance of the raycasting application. This idea is usually refered to as *empty space skipping* or *empty space leaping*, and has become a well-known concept in the area of volume rendering (see [HKRs$^+$06, Lev90, OKK10, MGS$^+$01]). Clearly, skipping empty regions requires prior knowledge about those regions. In our case, this means that we have to store information about empty regions in our access structure, which is our perspective grid. This is where another useful property of the grid comes into play. We build our grid aligned to our rays through the image plane. Therefore we know that, during the ray traversal, when the information about empty space is needed, the skipping of empty space will always happen along the positive z-axis of our grid. With this in mind, we can build a helper structure which stores the number of coherent cells that can be skipped in z-direction $z_{empty} \in \mathbb{R}$, including the cell itself, for each empty cell. We will call this structure the *empty successors cache* in the following. The construction of this cache can be done in parallel for each cell after the other grid construction steps, when we can already determine which cells are empty by checking against `INVALID_CELL_INDEX` in the cell's entry of the cell start array. Each cell that is empty initializes its value of z_{empty} with 1. Then, the next cell in z direction is checked. If this cell is empty, z_{empty} is incremented by one. This step is repeated as long as an empty cell was found during the last step. If a non-empty cell is found, the thread writes the resulting value of z_{empty} to the empty successors cache and terminates. For non-empty cells, the value z_{empty} in the empty successor cache will be set to zero (see figure 3.10).

This structure can then be used during the raycasting process to skip large empty regions within a single step. If we, for example, have a situation where all cells in z-direction are empty for a certain x and y coordinate, we will realize this directly during raycasting when checking the empty successors cache for the first cell. The entry will then equal N_z, so we

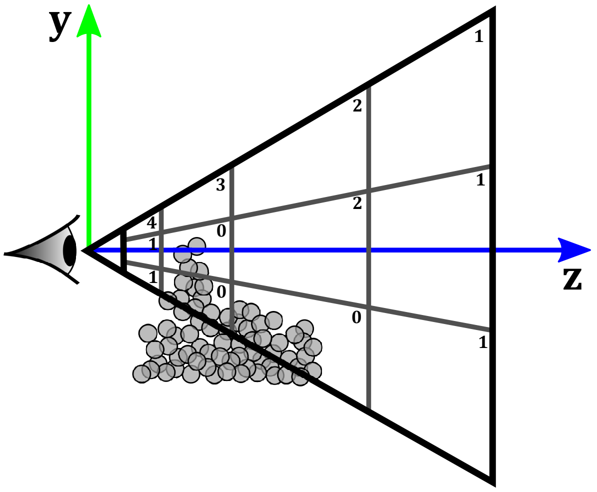


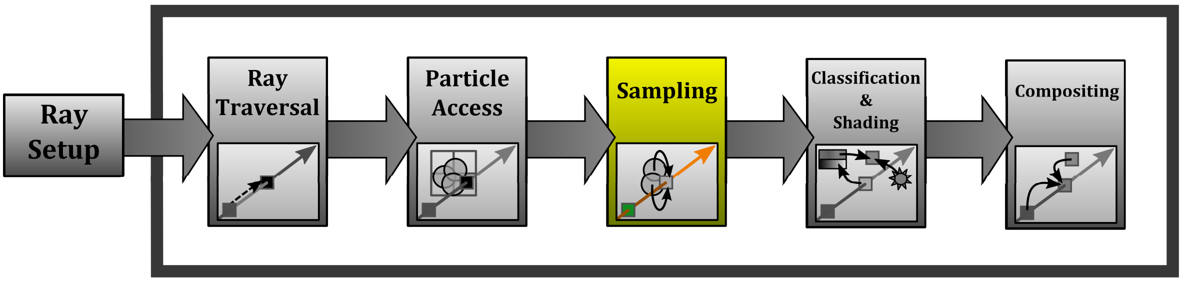
Figure 3.10: The empty successors cache. For each empty cell, the number of coherent empty cells in z-direction, including the cell itself, is written. For non-empty cells, zero is stored.

skip N_z cells in a single step and realize that we are already finished with the whole grid traversal.

3.4.4 Summary

So far, we have discussed all steps that are necessary to construct a perspective grid which contains references to all relevant particles for each cell. The perspective grid is aligned to the view frustum, which can be very useful during the sampling process. The cellsize is furthermore adaptive, analogous to the increasing sampling stepsize along the rays, as shown in section 3.3. This leads to the special property that each cell contains a fixed number of samples. Finally, storing the number of succeeding empty cells in z-direction for each cell enables us to perform empty space skipping. Within section 3.6.2, we will see how those major advantages of the perspective grid can be exploited during the raycasting process.

3.5 Normalization



Following the SPH model, we are obtaining the value of the scalar field at a given point by computing a weighted sum of all particles' contributions. In order to preserve constant functions in our final result, we should then divide this weighted sum by the sum of all weights ([BK02]), according to the following equation:

$$Q(\vec{x}) = \frac{\sum_i Q_i \cdot V_i \cdot W(\|\vec{x} - \vec{x}_i\|, r)}{\sum_i V_i \cdot W(\|\vec{x} - \vec{x}_i\|, r)} \tag{3.14}$$

This division is called normalization. It's one advantage of gathering approaches over scattering approaches that we are able to perform such a normalization during our main sampling pass. In the approach of [FAW10], for example, a normalization is not possible because the scalar field is directly reconstructed from the scattered values which are just the sum of each particles weighted contribution:

$$Q(\vec{x}) = \sum_i Q_i \cdot V_i \cdot W(\|\vec{x} - \vec{x}_i\|, r) \tag{3.15}$$

Obviously, one cannot reconstruct the weights out of the result of this computation afterwards. The only imaginable way to provide a normalized result would be to use a second scattering pass in which only the weights of the particles (not multiplied by their scalar values) are resampled onto the grid. Since the resampling, as already mentioned by [FAW10], is much more time-consuming than the raycasting through the pre-sampled perspective grid, this is not a good option because it causes an immoderate performance decrease.

In our approach, we can easily gather the sum of weights along with the contributions and then apply normalization after all contributions have been summed. Of course, this also causes a decrease of performance, because our CUDA program will need additional memory to save the sum of weights along with the contribution for each sample point. This additional memory potentially decreases the number of threads that can execute in parallel and therefore also performance. However, the performance drop is rather small and still acceptable. With this possibility in mind, we will compare the use of unnormalized and normalized sampling for our application within the next section.

3.5.1 Unnormalized vs. Normalized Sampling

Raycasting with normalized sampling results in vastly different images, which represent the SPH model much more precisely. This is due to the fact that, in the case of a simple unnormalized sampling, we are always running through the lower spectrum of our scalar field when intersecting a particle with our ray. When hitting the surface of the particle set, this leads to our transfer function being applied to that lower part of the spectrum for a few samples. Note that, in the extreme case of a border particle with the maximum scalar value, being intersected through its center by a ray, we will encounter all possible scalar values from zero to the maximum value on the rays way from the particles border to its center. Hence, all possible different emission and absorption values inside the transfer function will need to be applied. Figure 3.11 illustrates this situation. In addition to the fact that this is unwanted in most cases, the danger of visibly undersampling the transfer function this way is much higher than for normalized sampling. Figure 3.3 shows the result of such an undersampling.

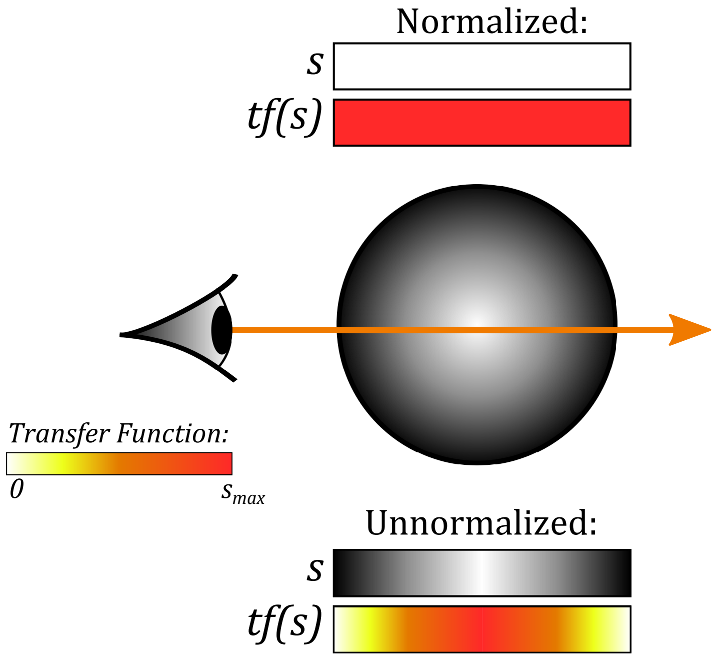


Figure 3.11: The effect of unnormalized sampling. In the lower part of the figure, two color bands show the scalar values (s) that are gathered along the ray using an unnormalized sampling of a particle without any neighbours, as well as the corresponding emission values $tf(s)$ that are obtained from an example transfer function. In the upper part of the figure, the same information is shown for normalized sampling. The gradient inside the particle shows the particles weight according to its kernel function, with white indicating the highest weight.

Still, if we use a different color for the lower spectrum of the scalar values than for the rest, we can this way visualize the surface of the fluid without explicitely employing any surface shading technique. The result somehow reminds of a lighted surface, as can be seen in figure

3.12. This feature can, under certain circumstances, give us more visual information with less effort, since we are able to recognize depth relations much better, compared to unshaded and normalized rendering.

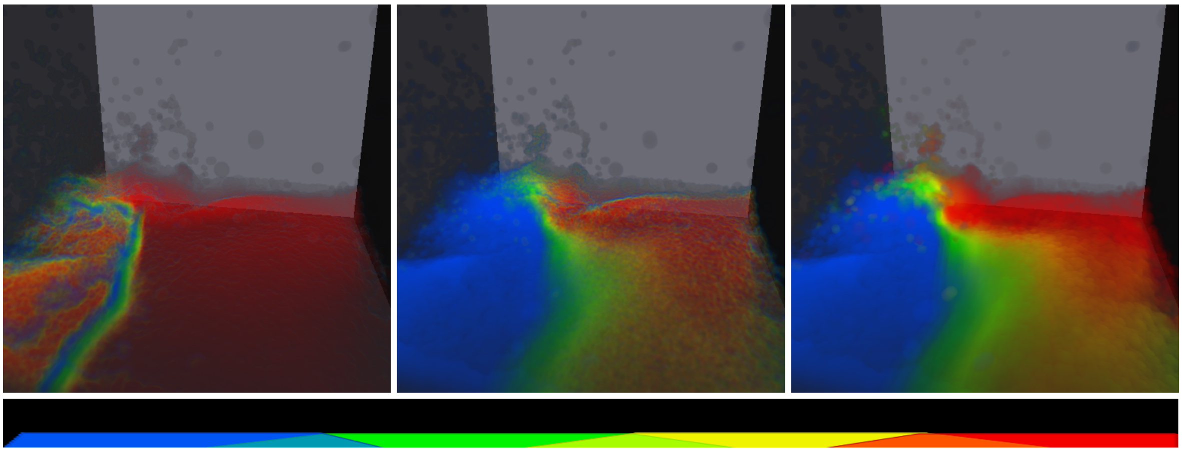


Figure 3.12: Unnormalized sampling without (left) and with (center) pseudo normalization (see section 3.5.2). At the bottom, the transfer function is depicted. A pseudo normalization resembles much more the correct, normalized visualization of the several concentration levels, shown on the right. Note how the application of the lower spectrum of the transfer function in the two leftmost figures, although wrong, enhances surface visibility.

3.5.2 Pseudo Normalization

In addition to the “surface highlighting“-effect which occurs when using an unnormalized sampling (see 3.5.1), we will also obtain wrong values inside the fluid, because the lack of any normalization means that the division of each sampled value by the sum of kernel weights is not applied. Usually, sampled values will be too large because of that. This effect is visible in figure 3.12, where the red area is much larger for the unnormalized rendering.

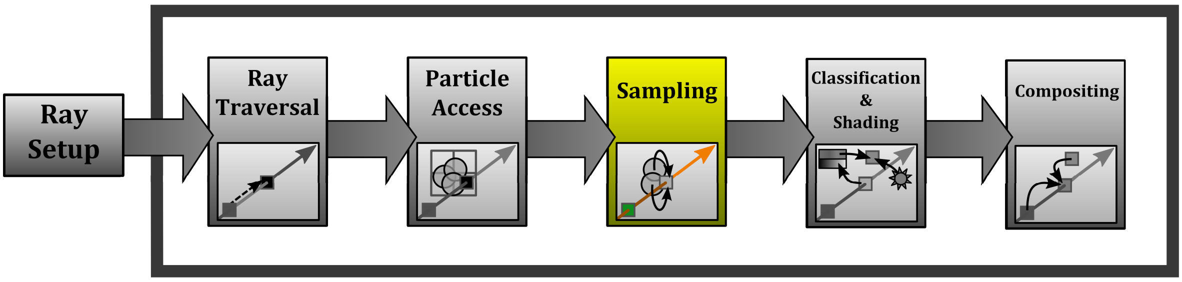
An approximate solution to this, in the following called *Pseudo Normalization*, is very easy to realize and can also be applied in scattering solutions like [FAW10]. All we have to do is to divide all sampled values by a representative maximum sum of weights w_{max} which assumes a fully occupied particle neighbourhood of $n_{neighbours} \in \mathbb{N}$ neighbours. Inside dense regions of the volume this will probably be a good solution, since we can assume the number of neighbours to be approximately constant inside incompressible fluids. All we need is an assumption about the structure of the densely packed particles. As a pure educated guess, which we derived from looking at the debug rendering of particles as spheres inside our application, we set the assumed number of neighboured particles inside the fluid to $n_{neighbours} = 12$. For a better foundation of $n_{neighbours}$, this topic might need further investigation. Also, we have assumed that the distance to each of those neighbours $d_{inner} \in \mathbb{R}$ equals half of the particle's support radius h. However, we found that this provided us with satisfying results, as shown in figure 3.12. The representative sum of weights w_{max} is then computed as

$$w_{max} = n_{neighbours} \cdot (v_{cnst} \cdot W(d_{inner}, r_{cnst})). \tag{3.16}$$

Note that this formulation uses the additional assumption of a fixed particle volume v_{cnst} and a fixed smoothing length r_{cnst}. Nevertheless, this is always the case in our particular setup.

Because it is faster and has the advantage over unshaded normalized raycasting that we can easily recognize surfaces, we decided to keep pseudo-normalized as an optional method in our raycaster. Nevertheless, our first choice is still the correct way of visualizing the scalar field, which includes normalized sampling. In section 3.7, we will explain how we can solve the problem of missing visual information about surfaces by employing surface shading techniques.

3.6 Cached Sampling



When implementing our gathering approach to SPH volume raycasting, we are reliant on programmable high performance graphics hardware in order to achieve interactive framerates. We can benefit especially from the ability of such hardware to execute many thousands of threads in parallel. In order to exploit this parallelism for volume raycasting, a lot of optimization mechanisms have been proposed ([MGS$^+$01, MRH10, OKK10, MHS08]). For the special case of SPH volume raycasting, the approach of [OKK10] has shown how a highly parallel raycasting program can benefit from three different caching mechanisms. All of those three mechanisms, the node cache, the influence cache and the slab cache, are already implied by our perspective grid (see section 3.4). Within this section, we will show how the special properties of this grid can be exploited during the sampling process.

3.6.1 Ray Bundles

Since our perspective access structure is view-aligned, we can assume that a ray's position in x- and y-direction inside the perspective grid will never change during the ray traversal. Furthermore, the perspective grid has the special property that the cell structure is subdividing the image plane into equally sized tiles of $N_{tile} \times N_{tile}$ pixels (see section 3.4.1). This way, we are able to share information about the grid traversal during the raycasting process between bundles of up to $N_{tile} \times N_{tile}$ rays. In our particular setup, with a grid resolution that leads to cells with a size of 16×16 pixels on the image plane, this corresponded at the same time to the organization of CUDA threads in each block: each block manages $16 \times 16 = 256$ threads. Note that this is not an unusal number of threads per block but instead recommended for several CUDA applications [KH10]. What makes it attractive to organize threads this way is that they can access a shared low-latency on-chip memory which is allocated block-wise, simply called *shared memory*. Shifting shared resources to shared memory can reduce the amount of local memory and registers needed by each thread. This lowered memory consumption can, under certain circumstances, lead to more threads running in parallel ([KH10]). In our case, the information that can be shared between threads in the same bundle is everything related to the traversal of our perspective grid, in particular

- the 3D index c of the current cell,
- the linear index $IDX(c)$ of the current cell,
- the number of empty cells in z-direction z_{empty} that can be skipped,
- the integer ray coordinate $\lfloor u \rfloor$ (see 3.3) and finally
- the ray parameter t, corresponding to the current ray coordinate.

All of those points are caused by the fact that all rays of a bundle are always sampling positions within the same cell. For x- and y-direction, we have identified this as a useful property of our perspective grid. For the z-direction, this is due to our adaptive sampling mechanism, following [FAW10], where all rays perform the same sampling in view space.

Before writing to the shared memory, we have to use a synchronization barrier (which is already provided by CUDA) to ensure that all threads have executed the program until this point. The first thread of the block is then used to compute the shared values and to write them to shared memory. After such a write transaction, another synchronization is necessary before all threads may continue their work. A simplified example of such a process is shown in listing 3.1:

```
//... do something, using a shared memory variable s_rayCoord
synchronizeThreads();
//first thread modifies s_rayCoord
if (isLeader){
    s_rayCoord = s_rayCoord + 1;
}
synchronizeThreads();
//... continue, using the new value of s_rayCoord
```

Listing 3.1: Barrier synchronization on shared memory usage.

Note that keeping the grid traversal information in shared memory corresponds to the node cache proposed by [OKK10]. In our setup, this method enables the hardware to use less memory for each thread, and therefore to accommodate more threads at the same time.

3.6.2 Slab-Caching

With the concept of a slab cache, which stores a fixed number of samples along the ray, [OKK10] are following the proposal of [MRH10]. However, in their case of SPH volume rendering, the slab cache is not only utilized for gradient computation, like in the case of [MRH10], but also to reduce the amount of global memory read operations. As described in [KH10], such operations can become a bottleneck for the performance of the GPU programm, since they have a long latency. The solution mentioned by [KH10], which is to transfer shared information from global memory to shared memory, is adapted by [OKK10] in form of the

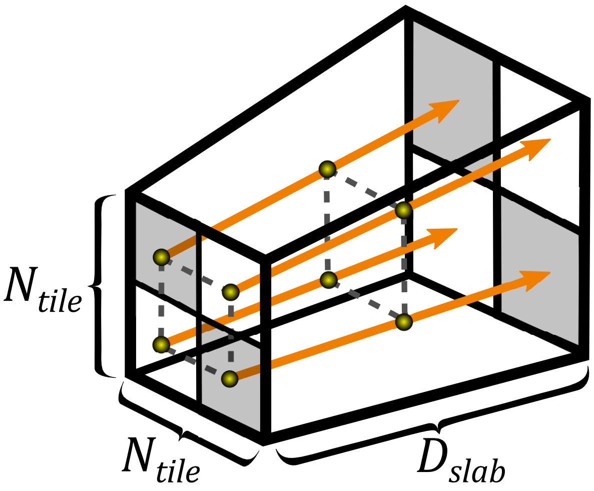


Figure 3.13: Ray bundles and slab caching. Each bundle of ray consists of $N_{tile} \times N_{tile}$ rays (here: 2×2). Along the rays, a fixed number of D_{slab} (here: 2) samples is processed together during each gathering step.

slab cache. In the case of [OKK10] and in our case as well, the major part of global memory traffic consists of particle read operations. In an approach without any slab cache, as assumed in the previous section, the positions and scalar values of all particles of the current cell have to be fetched from global memory at each sampling point. But if we can manage to fetch the particles' values only once for three succeeding sampling points, we will have reduced the global memory traffic for this task to $\frac{1}{3}$. Our cell structure is ideally suited for such an approach because we can always assume that only one cell is relevant for a cached ray segment, in contrast to the octree used by [OKK10], and that furthermore the number of samples per cell is constant (see section 3.4.1). With this constant number of N_{tile} sampling points per cell, which should be a power of two, we can then define the size of the slab cache as a fixed number $D_{slab} \in \mathbb{N}$ with

$$D_{slab} = \frac{N_{tile}}{2^k} \tag{3.17}$$

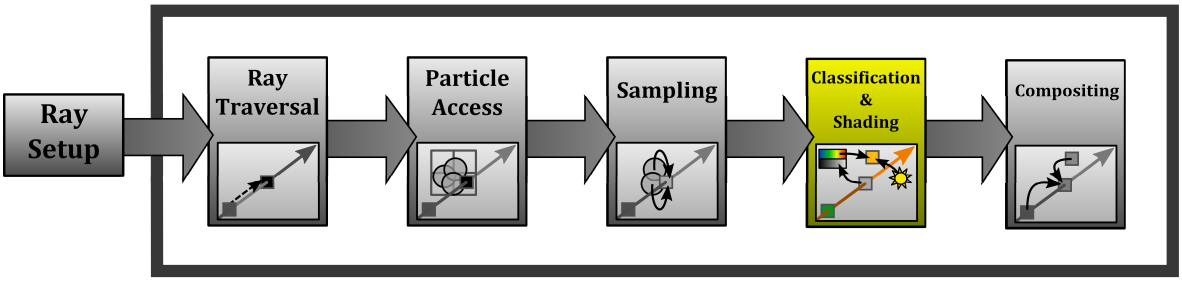
, where $k \in \mathbb{N}$ is a constant out of $\{0, ..., \log_2(N_{tile})\}$. As an example, with $N_{tile} = 16$ we could decide to use a cache for each ray that stores the sampled values for 16, 8, 4, 2 or 1 point(s). D_{slab} can also be referred to as the *slab depth* ([OKK10, MRH10]). An example of a slab cache with a slab depth of $D_{slab} = 2$ is depicted in figure. The size of the cache will be a tradeoff between the benefit of reducing global memory access and the additional local memory and register consumption of each thread. Note that we also have to store all additional information that is gathered during the sampling for each sampling point in the cache, like the accumulated sum of weights (see section 3.5) or the gradient vector (see section 3.7.1). Because of this, it is good to allow a flexible size of the slab cache, as described by equation 3.17. We will discuss the tradeoff between memory access speed and memory consumption more detailed in section 4.1.4 and give recommendations concerning a good choice for D_{slab}.

Having performed the sampling for all sampling points in the cache, we can then run through each of the following pipeline stages (which are classification, shading and compositing) from front to back in order to obtain the correct result. The concept is shown in listing 3.2 for a simple unnormalized sampling.

```
for (int i = 0; i < num_relevant_particles; ++i){
  particle p_i = relevant_particles[i];
  float w = weight(sample_position, p_i);
  //sample particles contribution to all cache entries
  for (int s = 0; s < slab_depth; ++s){
    sampled_value[s]  += p_i.value * w;
  }
}
//perform compositing for all cache entries
for (int s = 0; s < slab_depth; ++s){
  color = integrate_sample(sampled_value[s], color);
}
```

Listing 3.2: Slab-based sampling and compositing.

3.7 Shading



The prior aim of our volume raycaster is to visualize a scalar quantity of interest. In section 3.5, we have discussed the topic of normalizing the sampled values. We have found that, although sample normalization is the correct way of visualizing the scalar field, its major drawback over unnormalized sampling is the loss of surface visibility. This is where shading techniques come into play, since we are able to extract information about the shape of an object by looking at an image where the object is illuminated. Because of this aspect, which is widely known as *shape from shading* ([Hor70]), computing the illumination caused by an external light source in addition to the volumes own emission can add a lot of realism and additional information to our visualization, especially for surface regions.

Within the past decades, a lot of illumination techniques have been proposed, reaching from simple local illumination models to complex global illumination techniques which take even scattering effects inside the volume into account (see [HKRs$^+$06]). In our interactive setup, however, we are just interested in a fast, basic way of illuminating the volume, primarily in order to make the location and orientation of surfaces visible. Because of that, we are limiting ourselves to such a local illumination method within this section.

3.7.1 Gradient-Based Shading

During the whole history of computer graphics, a lot of research has been done on computing the lighting of surfaces. Especially the phenomenological Blinn-Phong lighting model ([Bli77, Pho75]) has become popular as a basic shading technique for surfaces. This model has become the lighting method which is implemented in standard graphics hardware, therefore it has also become part of the early OpenGL API. Our aim is to apply this model to our volume. However, we don't have any surface definitions directly at hand, so we need to think about a fast and reliable way to extract surfaces from our particle set. Knowing the surface, we can then use the surface normal to compute the lighting of the surface according to the Blinn-Phong model. We are interested in shading only boundary surfaces between regions with different scalar values, which is funded by the assumption that light will only be reflected on such surfaces while travelling through homogenous regions without any reflection or refraction ([HKRs$^+$06]). Therefore, we need a way to obtain those so-called isosurfaces.

Like isolines (also: *contour lines*) in 2D, isosurfaces describe surfaces where all points on the surface have a constant value in three-dimensional space. In addition to that, such surfaces occur at boundary regions, which means that the value of the scalar field differs on both sides of the surface. A classical method of extracting such a surface from a volumetric dataset is known as the marching cubes algorithm ([LC87]). The algorithm is using a uniform cell grid, where the scalar values at each cells corner points are checked. Finding that the isosurface of interest must be running through a part of the cell, a polygon is inserted, covering the points of the approximated isosurface inside the cell. Extracting a high-resolution surface mesh for SPH simulations via the marching cubes algorithm is a standard approach, used to generate the resulting images in [YT10, YTWJ12, APKG07]. All of those publications cover methods to enhance the appeareance fluid-air interface, aiming at a smooth surface reconstruction. While, in this context, explicit triangulation via marching cubes is a well-established method to extract a single surface, which can then be lighted, it is not well-suited for our application since it causes too much computational overhead for our interactive real-time raycasting setup and requires a uniform discretization of the simulation domain. As an alternative to explicitely extracting the isosurface, another, gradient-based method has become more popular in volume raycasting, as proposed by [Lev88] (see also [HKRs$^+$06]). The aim of this method is to find any intersection of our ray with any isosurface during the ray traversal, which enables us to directly compute the lighting at each sampling point and apply it along with the emission term. To do so, the gradient of the scalar field is estimated along the three principal directions in 3D space, which leads to the so-called gradient vector of the scalar field

$$\nabla s(x) = \begin{pmatrix} \frac{\partial s(x)}{\partial x} \\ \frac{\partial s(y)}{\partial y} \\ \frac{\partial s(z)}{\partial z} \end{pmatrix}. \tag{3.18}$$

This vector has the special property that it always points to the direction of the steepest ascent ([HKRs$^+$06]). Such information is very useful for our purpose of computing the isosurface lighting: on the one hand, we find that the vector is always perpendicular to the isosurface, so it can be normalized and used as a surface normal during the lighting computation. On the other hand, the vector will be of zero length inside regions with a constant scalar value, since the gradient in such regions will be zero. We can use this information to find the boundary surfaces in we are looking for. As can be seen, the problem of computing the local illumination during the raycasting process can pretty much be reduced to finding the gradient vector at each sampling point.

For the rendering of voxel datasets, a lot of different techniques exist to obtain the gradient during the raycasting process ([HKRs$^+$06]). Gradients can be estimated with various methods, like finite differences or more complex filter kernels. Also, the gradient vectors might be computed for the whole voxel dataset in a pre-processing step, or computed on-the-fly during raycasting. Such on-the-fly methods, using central differences, have been applied by [OKK10] and [FAW10] for the rendering of isosurfaces during SPH volume raycasting. [OKK10] are using additional border rays around their ray bundle, with the only purpose of using them to compute the central differences. Since the computation takes place between samples on neighbouring rays, with a distance increasing along the ray direction, the step size of the central differences has to be adjusted accordingly. [OKK10] have adapted this concept of gradient computation, utilizing the slab-cache, from [MRH10]. [FAW10] are firstly mapping the sampling points' positions from view space back to texture space. Finding that, in contrast to texture space, the distance along the z-axis is not uniform in view space, they are mapping the samples to view space, computing the neighbour positions along the z-axis needed for central differences, and mapping those positions back to texture space. To obtain the scalar values at that sampling points (which will probably not lie exactly on one of the grid points) in texture space, hardware-accelerated trilinear interpolation is used. This is actually a mistake, because such an interpolation does not take the non-uniform distance along the z-axis in view space into account, but as already noted by [FAW10], the error caused by this method is nearly invisible. Taking a closer look at the SPH model, we find that the gradient vector of the scalar field can be computed in a way that pretty much resembles the evaluation of the scalar field itself (see section 3.5):

$$\nabla Q(\vec{x}) = \sum_i Q_i \cdot V_i \cdot \nabla W(\|\vec{x} - \vec{x}_i\|, r). \tag{3.19}$$

According to equation 3.18, we have to compute the partial derivatives of the function which is used to evaluate the scalar field in order to obtain the gradient vector. In the context of the SPH model, as can be seen in equation 3.19, this means that we have to compute the partial derivatives of $W(\|\vec{d}\|, r)$ along our coordinate axes. Knowing that the derivative of the SPH kernel $W'(\|\vec{d}\|, r)$ describes the derivation by the distance parameter $\|\vec{d}\|$, we can argue that the directional derivative $\nabla W(\|\vec{d}\|, r)$ can be obtained from the derivative by taking the direction of the distance vector $\vec{d}$ from our point of evaluation to the particles center into account. This direction is the axis along which the distance parameter $\|\vec{d}\|$ is varied in $W(\|\vec{d}\|, r)$. Note how this also corresponds to the direction of steepest ascent mentioned above. The directional derivative of the SPH kernel function can this way be expressed as

$$\nabla W(\|\vec{d}\|, r \;=\; W'(\|\vec{d}\|, r) \cdot \frac{\vec{d}}{\|\vec{d}\|}. \tag{3.20}$$

We can show this formally by applying the del operator (∇) to the SPH kernel and solving via the chain rule

$$\nabla W(\|\vec{d}\|,r) \;=\; \frac{d}{dr}\cdot W(\|\vec{d}\|,r)\cdot\nabla\|\vec{d}\| \;=\; W'(\|\vec{d}\|,r)\cdot\frac{\vec{d}}{\|\vec{d}\|} \tag{3.21}$$

, knowing that the application of the del operator to the length of a vector results in the normalized vector, as can be seen by applying the chain rule another time:

$$\begin{aligned}
\nabla\|\vec{d}\| &= \nabla\sqrt{d_x^2+d_y^2+d_z^2} \\
&= \nabla(d_x^2+d_y^2+d_z^2)^{\frac{1}{2}} \\
&= (d_x^2+d_y^2+d_z^2)^{-\frac{1}{2}}\cdot\nabla(d_x^2+d_y^2+d_z^2) \\
&= \frac{1}{2}\cdot\frac{1}{\sqrt{d_x^2+d_y^2+d_z^2}}\cdot\nabla(d_x^2+d_y^2+d_z^2) \\
&= \frac{1}{2}\cdot\frac{1}{\sqrt{d_x^2+d_y^2+d_z^2}}\cdot 2\vec{d} \\
&= \frac{\vec{d}}{\|\vec{d}\|}.
\end{aligned}$$

Hence, in contrast to simple gradient estimation schemes such as finite differences, we can compute the gradient vector exactly by inserting equation 3.20 into 3.19:

$$\nabla Q(\vec{x}) = \sum_i Q_i\cdot V_i\cdot W'(\|\vec{d}\|,r)\cdot\frac{\vec{d}}{\|\vec{d}\|}. \tag{3.22}$$

This evaluation can take place in the same step as the regular sampling of the scalar field, i.e. we can accumulate the gradient vectors the same way as we already accumulate the scalar values. This way of gradient computation is much more reliable than any gradient estimation method, since it provides us with the exact solution. It is furthermore easy to integrate in our application, because we can compute the gradient iteratively along with the scalar values at each sample point. Having computed the gradient vector, we can normalize and invert it to obtain the surface normal. This step is necessary because the gradients always point into the direction of steepest ascent. That means, for the principal surface of the fluid, that they are pointing inwards, into the volume. Since the surface normal is expected to point outwards, we have to invert the gradient vector. This leads to the following definition of the surface normal at a sampling point $\vec{x}$:

$$n(\vec{x}) = -\frac{\nabla Q(\vec{x})}{\|\nabla Q(\vec{x})\|}. \tag{3.23}$$

This normal vector can be used to compute the shading at a point along the ray according to the Blinn-Phong model, where the gradient magnitude is not zero. The result is then, together with the emissive contribution, blended with the stored values according to the emission-absorption model (see section 3.8.2). A resulting image of a normalized sampling with a shaded surface can be seen in figure 3.15.

Note that we have not employed any technique in order to obtain a smooth surface, i.e. our lighted surfaces will usually appear uneven because of the particles' isotropic spherical smoothing kernels. Although very recently a new method has been proposed to overcome this issue using anisotropic kernels ([YTWJ12]), we do not take into account such approaches since they are usually applied to a high-quality offline simulation which is not comparable to our interactive setup.

3.7.2 Contour Shading

In the previous section, we have shown how to overcome the lack of surface visibility for a normalized sampling by employing local volume illumination. We have also found that an unnormalized or a pseudo-normalized sampling can highlight the surface of the fluid at the cost of a wrong visualization (see section 3.5). This visualization is especially wrong because the lower spectrum of the transfer function is applied at the boundary regions due to the unnormalized application of the SPH smoothing kernels. Hence, the enhancement of the surface comes at the cost of wrong colors at the surface boundaries (3.12) in our final image. Within this section, we will show how to overcome this limitation using normalized samples and how we can still be able to exploit the surface highlighting mechanism present in unnormalized or pseudo-normalized sampling approaches.

During a normalized sampling approach, we are gathering the particles' weights along with the unnormalized sampled value at each sampling point. When all contributions have been gathered, we can then divide the unnormalized sampled value by the sum of weights in order to obtain the normalized value 3.14. This basic method is shown in listing 3.3.

```
//1. gather contributions and sum of weights
float sampled_value  = 0;
float sum_of_weights = 0;

for (int i = 0; i < num_relevant_particles; ++i)
{
  particle p_i = relevant_particles[i];

  float w = weight(sample_position, p_i);

  sampled_value  += p_i.value * w;
  sum_of_weights += w;
}

//2. normalize sampled value
sampled_value /= sum_of_weights;
```

Listing 3.3: Normalized sampling.

The visual surface enhancement we will encounter when using unnormalized sampling methods is caused by the lack of this normalization step: Instead of visualizing only values of the scalar field, we are also visualizing the sum of weight of the smoothing kernels (see also figure 3.11). Because this sum is smaller at the boundary surface than elsewhere, we are able to highlight the surface this way.

Our idea is to re-use the sum of weights we have gathered to perform the normalization to exploit exactly this effect. In contrast to unnormalized methods, we can choose any random color we want to visualize the surface regions instead of taking the same transfer function which we already use to visualize our scalar field. We will refer to this color as the *contour color* for the rest of this section. The amount of contour color contribution $C_{contour}$ at a sampling point $\vec{x}$ is determined by the sum of weights, which we have already gathered in order to normalize the samples, as shown in listing 3.3. We can add this contribution to the emissive term, which is combined with the stored color and opacity values during the compositing stage (see section 3.8.2). To obtain $C_{contour}(\vec{x})$, we have to compute a contour intensity factor $I_{contour}(\vec{x}) : \mathbb{R}^3 \rightarrow \mathbb{R}$ which is then multiplied with the contours base color C_{base} (e.g. black) in order to receive the contour contribution:

$$C_{contour}(\vec{x}) = I_{contour}(\vec{x}) \cdot C_{base} \tag{3.24}$$

As for the color contribution, we have to compute a contribution of the contour to the current alpha value (in the following denoted as $\alpha_{contour}(\vec{x})$) which is achieved in a similar manner, using a base alpha value α_{base}:

$$\alpha_{contour}(\vec{x}) = I_{contour}(\vec{x}) \cdot \alpha_{base} \tag{3.25}$$

Since we know that the sum of kernel weights will be small for boundary regions and large for regions inside the volume, we can compute $I_{contour}(\vec{x})$ by inverting this property, which is done by subtracting the sum of weights from the constant maximal sum of weights of the SPH kernels w_{max} (see section 3.5.2):

$$I_{contour}(\vec{x}) = w_{max} - \sum_i V_i \cdot W(\|\vec{x} - \vec{x}_i\|, r) \tag{3.26}$$

In common contour shading approaches for volume rendering, the gradient as well as its magnitude are used to determine a contour factor (see [RE01, CMH$^+$01, HKRs$^+$06]). The gradient direction can be used to determine a surface normal, which can then be taken into account in order to decrease the contour intensity factor of surfaces we are looking at in a perpendicular direction. This way, contours at the volumes borders in image space can be enhanced. The gradient magnitude can be used to determine the “surfaceness“ of the volume (see [RE01]) and taken into account as another factor for the contour intensity. However, this involves an explicit gradient computation, which does not come as zero cost. As shown in section 3.7.1, we have to provide additional memory resources to accumulate the gradient vector during the sampling. Also, we have found that the method described by equation 3.26 provides acceptable results that do not involve any explicit gradient computation (see figure 3.15), even if this means that we are not taking the surface direction explicitely into account. We are achieving a similar effect because the length of the section where a ray intersects the surface region will be proportional to the angle of incidence between the ray and the surface, as shown in figure 3.14.

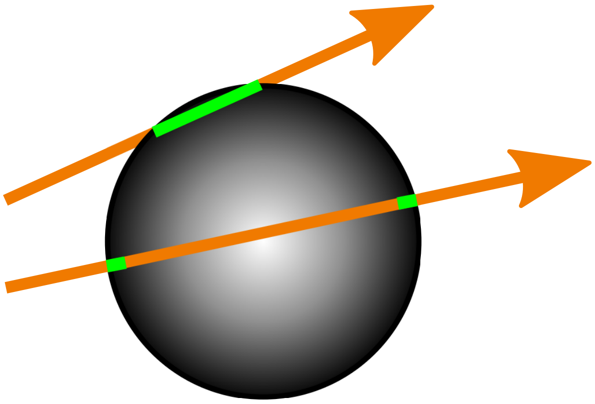


Figure 3.14: Sampling at contour regions. The upper ray will gather a larger contour intensity, since it samples a longer section inside the black region with low weight than the lower ray. This is caused by the different angles of incidence of both rays.

Since our formulation of the contour intensity (see 3.26) does not account for the scalar field's values Q_i carried by the particles, we can only visualize the principal surface of the fluid this way. In order to visualize also contours of isosurfaces inside the fluid, we would have to compute the gradient as shown in section 3.7.1. Still, in contrast to unshaded normalized sampling, we find that our contour enhancement method provides acceptable results which allow the user to recognize the fluid-air interface at almost no computational overhead. Therefore, this method can be seen as a cheap alternative to gradient-based shading.

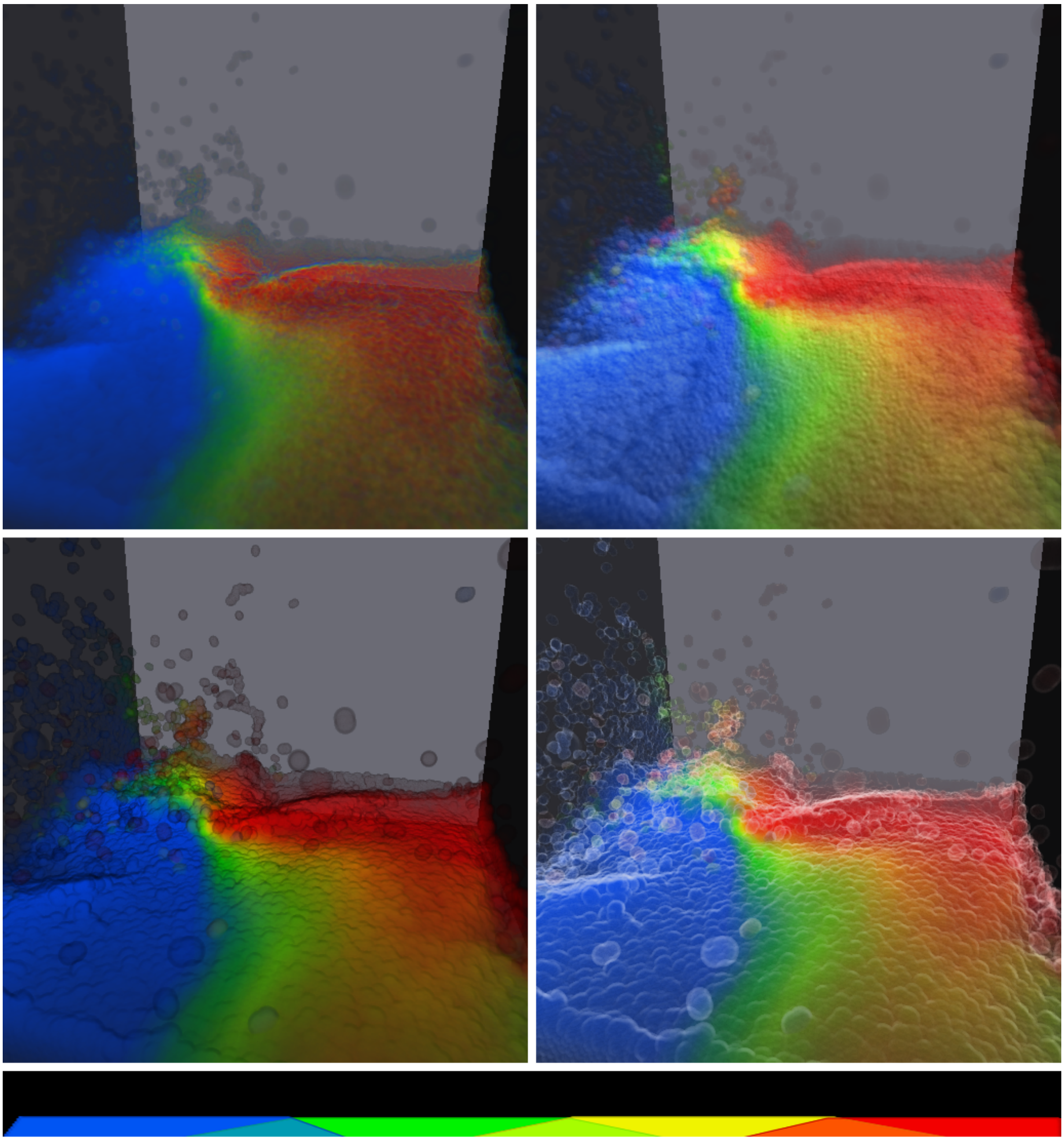
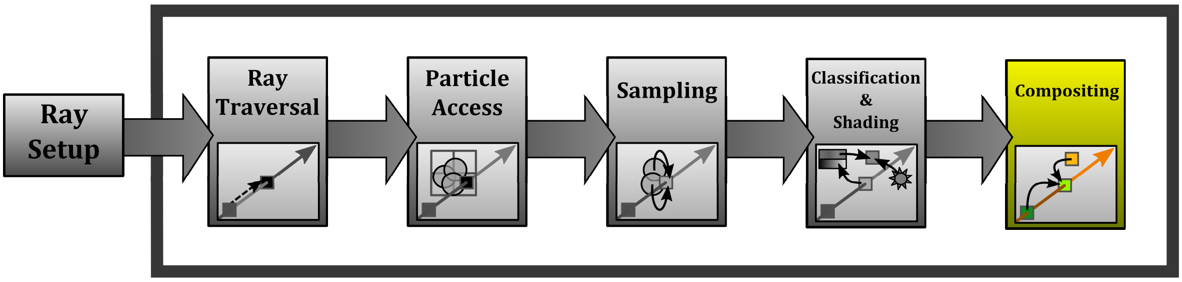


Figure 3.15: Surface enhancement techniques (from upper left to lower right): pseudo-normalized sampling (see section 3.5.2), gradient-based shading, contour shading with black contours, contour shading with white contours. The bottom of the figure shows the transfer function which was used.

3.8 Compositing



After obtaining the sampled value of the scalar field at a point along the ray, a transfer function, represented by a 1D texture, is used to convert that value to emission and absorption coefficients in the form of a four-element `rgba` float vector. The first three components define the red, green and blue color color component, i.e. the emissive value at the sampling point, while the last component of the vector represents the alpha component, i.e. the opacity at the sampling point. Those values have to be combined with the color and alpha values that have already been gathered along the ray. Within this section, we will highlight additional challenges of this process which arise from the key features of our raycasting setup.

3.8.1 Opacity Correction

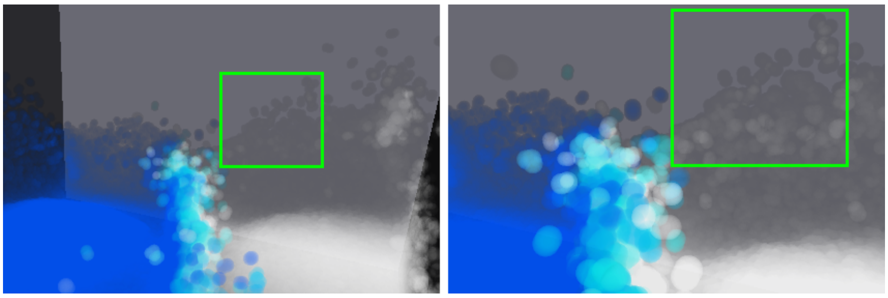


Figure 3.16: Adaptive sampling without opacity correction. The two images were rendered from different points of view. Note how the distance to the volume affects the opacity of the raycasting result. Here, this is especially visible in the corrsponding regions of the images that are marked with green rectangles.

Since, with our adaptive sampling mechanism, we are using a different stepsize for each ray segment, the weight of the those segments when computing the resulting color of the corresponding pixel cannot be equal, unlike for a uniform stepsize. The problem becomes visible when we move our virtual eyepoint closer towards the volume or farther away from it, which leads to a change of the number of samples within the volume (see figure 3.16). Therefore, we need to perform a so-called *opacity correction* ([HKRs^{+}06]). Assuming that the sampled opacity value remains unchanged along the ray within the segment, this leads to the following correction term for an uncorrected opacity α and a corrected opacity $\alpha_{corrected}$ respectively (see [HKRs$^+$06]):

$$\alpha_{corrected} = 1 - (1-\alpha)^{(\frac{l}{l_0})} \tag{3.27}$$

, where l denotes the length of the current segment and l_0 is a constant which describes the length of a reference segment. For the color contribution, the correction term is much simpler since, in contrast to the opacity value, the color contribution does not change over the length of the segment. Having an uncorrected color C, the corrected color $C_{corrected}$ can be computed as

$$C_{corrected} = C(\frac{l}{l_0}) \tag{3.28}$$

3.8.2 Integration

We are storing pre-multiplied colors as described by [Bli94] inside the transfer function, which means that the color values are already weighted by the corresponding opacity values. During the compositing, we can save an extra multiplication this way.

The integration of the sampled emission and absorption values along the ray from front to back, using associated colors, has been described in [HKRs$^+$06]. The new color C_{new} is computed by adding the color contribution of the current segment $C_{contrib}$ to the current color C_{old}, attenuated by the current absorption α_{old}:

$$C_{new} = C_{old} + (1-\alpha_{old})C_{contrib}. \tag{3.29}$$

For the new absorption, the equation is similar:

$$\alpha_{new} = \alpha_{old} + (1-\alpha_{old})\alpha_{contrib}. \tag{3.30}$$

We obtain the absorption of the current ray segment by sampling our transfer function, where $\alpha_{contrib} = \alpha_{absorption}$, the alpha value of the sampled color from our transfer function. For the emission contribution $C_{contrib}$, we might have to take our local illumination into account. If we decide to use a shading technique, an additional color contribution $C_{shading}$ will have to be added to the emissive value $C_{emission}$ to obtain $C_{contrib}$ ([HKRs$^+$06]). Since we are using pre-multiplied associated colors, we are not multiplying the emission term again by $\alpha_{contrib}$ during the compositing (see equation 3.29). Therefore, in order to receive a correct result for the shading contribution, we also have to pre-multiply it by $\alpha_{contrib}$:

$$C_{contrib} = C_{emission} + \alpha_{contrib}C_{shading} \tag{3.31}$$

We use this equation to combine the contribution of our gradient-based shading with the emission value obtained from our transfer function. As can be seen in equation 3.31, the final color of any pixel can just get brighter when using local volume illumination, since we are not allowing for negative light sources. This is just a natural assumption and not very suprising. Nevertheless, we have to change this model if we want to use our contour shading method (see section 3.7.2), where the contours might also appear black, for instance. In this case we will have to use the contours opacity $\alpha_{contour}$ (see equation 3.25) to pre-multiply the contour color, instead of using the local absorption ($\alpha_{contrib}$) as above:

$$C_{contrib} = C_{emission} + \alpha_{contour} C_{contour} \tag{3.32}$$

Additionally, the contours absorption $\alpha_{contour}$ has to be added to the local absorption value:

$$\alpha_{contrib} = \alpha_{absorption} + \alpha_{contour} \tag{3.33}$$

Note that the contours optical properties are defined as a pair of a base color value C_{base} and a base opacity value α_{base}. This enables us, in contrast to regular lighting, to achieve the effect of black contours, where C_{base} is zero in each component but α_{base} is greater than zero.

3.9 Summary

Within this chapter, we have shown all key features of our SPH volume raycaster. To divide the raycasting process into several stages that can be analyzed separately, we have introduced the SPH volume raycasting pipeline. The main task during the ray setup stage is the configuration of the resolution of our raycaster, which includes the sampling rate along the rays as well as the number of rays shot through the image plane. This resolution can be made dependent on the movement of the virtual camera, so that a fluent user interaction is made possible. For the ray traversal, we are using an adaptive stepsize, which is derived from the resolution of the image plane, as proposed by [FAW10]. During the particle access stage, we are using a perspective grid which is aligned to the view frustum. The perspective grid allows us to use basically infinite simulation domains and has some other advantages which make the grid traversal very efficient. One advantage is that the cell search becomes trivial, another one is the alignment of the cell structure with the adaptive samples along the rays. In the sampling stage, we exploit those advantages by following equally sized bundles of rays through the image plane and by employing a slab cache, as proposed by [OKK10, MRH10]. We are also normalizing our samples by dividing each sample's value by the gathered sum of weights, which provides us with a precise visualization of the scalar field. Within the following classification and shading stage, we are applying a transfer function to obtain emission and absorption coefficients from our sampled values. Furthermore, we are adding contributions from our shading techniques, which are gradient-based local volume illumination and a simple yet efficient contour shading approach. During the compositing stage, we combine the emission and absorption values along the ray after applying opacity correction for each sample. This step is necessary because of the adaptive sampling mechanism, which leads to non-uniform ray segment lengths.

In chapter 4, we will analyze the implementation and results of our raycasting concept.

Chapter 4

Results

4.1 Implementation

In chapter 3, we have presented our approach to a highly parallel SPH volume raycasting solution which is designed for interactive setups. This section covers a more detailed discussion on implementation and resource demands of our concept. The different visualization techniques that we have proposed are compared in terms of memory consumption and performance within our experimental setup. Finally, the results are discussed and recommendations on performance tuning are given.

4.1.1 Framework Integration & Raycasting Setup

Our raycaster was implemented as a module for the *Simulation and Visualization Toolkit* (SVT), developed at the computer graphics and multimedia systems group at Siegen University. The toolkit is based on *OpenSceneGraph* (OSG), an open-source scenegraph API. Withing the OSG-based application, the implementation of a new component can then be realized in the form on a new node that is added as a child to an already exising node in the graph. For our raycaster, this node was realized as a computation node, using the `osgCompute` extension to OSG. This extension provides mechanisms to integrate GPGPU functionality, as provided by the CUDA API, into scenegraphs ([OKK09]). On each frame, the computation node executes a CUDA program which is implementing our raycasting concept and writing the result to graphics memory which is bound to a texture. The texture is then displayed on a screen-aligned quad. This way, realizing the upscaling for raycasting settings with lower resolution, as described in section 3.2, can be easily done without additional efforts: for raycasting resolutions that are lower than the textures resolution, we simply map the texture coordinates from their original range $[0, 1] \times [0, 1]$ to $[0, r^{-1}] \times [0, r^{-1}]$, where r is the reduction factor which was proposed in section 3.2.1.

For our view frustum, we were using a near plane at a distance of 20 units and a far plane at 400 units respectively. The vertical opening angle of the frustum (fov_y) was set to 60°. We could also have used the existing settings provided by the application, which were using a much smaller opening angle and a closer near plane. Nevertheless, since we want to keep the number of steps along the rays m as small as possible (see section 3.3), we are changing the settings when our raycaster is invoked. This corresponds to the setup used by [FAW10], where a large opening angle and a small relation between near and far plane lead to a small value for $m_{adjusted}$. Result for $m_{adjusted}$ in our application, using a cellsize on the image plane of 16 × 16 pixels, are shown in table 4.1 for different resolutions. Note that this means a slight change of perspective inside the application when the user is activating the raycasting module. However, we found this acceptable against the background of increasing performance.

Resolution	$m_{adjusted}$	$cells_z$
1024 × 1024	3440	215
512 × 512	1728	108
256 × 256	864	54

Table 4.1: Different sampling rates along z in our setup.

4.1.2 The CUDA Program

Besides the main raycasting kernel, our CUDA program consists of several parallelized kernel functions. A conceptual overview is given in listing 4.1.

```
dim3 blocks, threads;
//... setup blocks / threads needed to execute one thread for each particle

kTransformParticlePositionsToViewSpace<<<blocks, threads>>>(...);

kComputeRelevantCells<<<blocks, threads>>>(...);

//... sort the map of particles to cells by the cell indices

//... setup blocks / threads needed to execute one thread for map entry

kFindCellStartCellEnd<<<blocks, threads>>>(...);

//... setup blocks / threads needed to execute one thread for each cell

kCountEmptySuccessorsAlongZ<<<blocks, threads>>>(...);

//... setup blocks / threads needed to execute one thread for each pixel

 kRaycast<<<blocks, threads>>>(...);
```

Listing 4.1: Different CUDA kernels in our GPU program.

The first kernel (`kTransformParticlePositionsToViewSpace`) in listing 4.1 transforms each particle's position from world space to view space by simply multiplying its world space position by the view matrix. However, in our setup, the particles carry their influence radius as an additional information in their w component, so we have to extract this information. After this, we can replace it by 1 (since the particle's position is a point in homogenous coordinates) and insert the influence radius again as the w component after the multiplication with the view matrix has happened. Also, the standard view matrix, as also used by OSG, is assuming a right-handed coordinate system with the viewer looking along the negative z-axis. In contrast, our perspective grid is using a left-handed coordinate system where the viewer is looking along the positive z-axis. Therefore, we also have to invert the z-component in this step. This leads to the code shown in listing 4.2.

```
void kTransformParticlePositionsToViewSpace(const ParticleGrid grid,
              const float4 * particleWorldPositions, const Mat4x4 viewMatrix){
    const unsigned int ptclIdx = blockIdx.x * blockDim.x + threadIdx.x;

    if (ptclIdx < grid.numPtcls)
    {
        float4 posW  = particleWorldPositions[ptclIdx];
        float4 posWH = make_float4(posW.x, posW.y, posW.z, 1.0f);

        float3 posView = make_float3(viewMatrix.mult(posWH));
        posView.z *= -1.0f;

        grid.ptclPositions[ptclIdx] = make_float4(posView, posW.w);
    }
}
```

Listing 4.2: Transformation of the particle's positions to view space.

The next kernel function in listing 4.1, `kComputeRelevantCells`, computes the corresponding entries of the map from cells to particles for each particle. Each cell is a small asymmetric frustum in view space, which makes it more expensive to check whether a particle intersects a cell. Because of that, we determine a candidate cell range by using the particle's bounding box in a first step. To obtain this range, we need to find the minimum and maximum coordinates given by the bounding box and map those coordinates to cell coordinates inside our perspective grid. Listing 4.3 shows the code of the function which provides such a mapping.

The function computes the z coordinate of the grid cell by applying exactly the same function which is used to compute the current ray coordinate $\lfloor u \rfloor$ from a given distance t to the eyepoint along the z-axis (see section 3.3). Instead of the pixel size σ, the size of a cell on the image plane s is used. In listing 4.3, this is done in line 3 via the function `dMapDistanceToCoord_Grid`. Since the cell size in x and y rises linearly along z, the cell size $s'(z) : \mathbb{R} \to \mathbb{R}$ at the given z coordinate in view space can be obtained by multiplying the z coordinate by s : $s'(z) = s \cdot z$ (see line 5 in listing 4.3). Since the perspective grid is centered

```
uint3 dGet3DGridCoordinate(const float3 viewSpacePosition)
{
    unsigned int zGrid = dMapDistanceToCoord_Grid(viewSpacePosition.z);

    float cellSizeXY   = viewSpacePosition.z * gCnst.CellSize;

    unsigned int xGrid = viewSpacePosition.x / cellSizeXY + gCnst.CellsX * 0.5f;
    unsigned int yGrid = viewSpacePosition.y / cellSizeXY + gCnst.CellsY * 0.5f;

    return make_uint3(xGrid, yGrid, zGrid);
}
```

Listing 4.3: Mapping from view space to the perspective grid.

around the origin in x and y, the current grid cell in x and y is obtained by dividing x and y in view space by the cell size at the given z position $s'(z)$ and adding half of the number of cells in each dimension, as shown in listing 4.3. This way, we ensure that the 3D grid coordinates' range equals $[0, cells_x - 1] \times [0, cells_y - 1] \times [0, cells_z - 1]$.

Knowing the cell range covered by the particle's bounding box, we check each cell in that range precisely. This is done by a check for each of the six planes of the faces of a frustum-shaped cell. If the particle has a positive oriented distance to one of the planes that is larger than its radius, this plane is a separating plane and the particle lies outside the cell. If no separating plane was found, the particle intersects the cell. Figure 4.1 shows an example situation where a set of cell candidates has been determined by using the particle's bounding box and where all relevant cells within the candidate set have been detected.

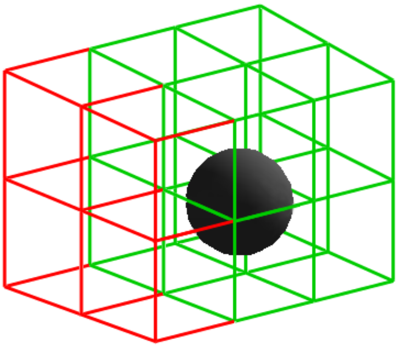


Figure 4.1: Relevant cells for a single particle. Out of 27 possible checks, given as the maximum number of relevant cells per particle, only 12 checks are performed. This is due to the computation of a candidate set of cells by using the particles bounding box. Among the 12 candidates, 4 candidates (shown in red) were discarded after performing a check against each cell's separating planes.

After the map of particles to relevant cells has been computed, we have to sort the map by the cell indices, as described in section 3.4.2. This step is also implied in listing 4.1 at line 9. For the radix sort, we are using the free CUDPP library, which has been developed along with the publication of [SHZO07] by the authors. The following kernel functions 3.4.2 `kFindCellStartCellEnd` and `kCountEmptySuccessorsAlongZ` are executed in parallel in order to build the cell start / cell end arrays and the empty successors cache respectively, as desribed in section 3.4.2.

The last kernel function shown in listing 4.1, `kRaycast`, is our main raycasting function and hence the most complex and important one. It is implementing the traversal of the whole volume raycasting pipeline, starting with the ray setup stage. Within this stage, the ray vector $\vec{r}$, reaching from the eyepoint to the ray's position on the image plane, is computed. We are not normalizing $\vec{r}$, instead we are just dividing it by the distance D_{near} to the near plane, leading to ray vectors with increasing length towards the frustum boundaries. This ensures that we can use the same ray parameter t for all rays to get the same position along the z axis for all rays in each sampling step, as shown in figure 4.2.

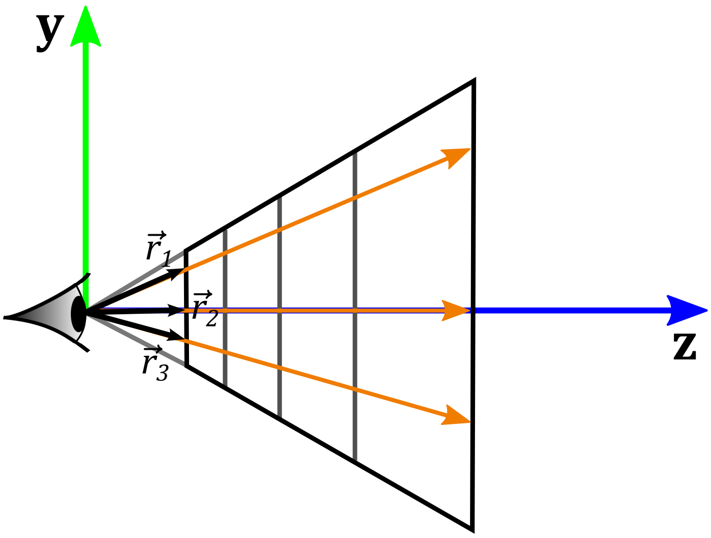


Figure 4.2: Ray vectors. The vectors from the eye to the image plane are shown as black arrows. The same ray parameter t can be used to obtain sampling points that have the same position along the z-axis for all rays, since the vectors will just be divided by the distance to the near plane instead of being normalized.

For stochastic jittering, the local offset for each ray's parameter is read from a 2D array with 32 × 32 entries. This offset has to be added to the shared ray parameter t each time the sampling position is computed. After the ray setup stage, the main loop of the volume raycasting pipeline is entered by each ray. The basic structure of the main loop is shown in listing 4.4.

Note that this is a very simplified version of our raycaster, where some code optimizations are missing as well as many key features, such as flexible slab size, sample normalization, opacity correction and shading. Nevertheless, it is well-suited to explain the basic strategy

```
while (sGridIdx3D.z < gCnst.CellsZ)
{
  syncthreads();
  if (isLeader){
      sCellIdx    = dGetLinearIdxFrom3DIdx(sGridIdx3D);
      sEmptyCells = grid->cellsToEmptyCellSpaceAlongZ[sCellIdx];
      sCellStart  = grid->firstCellArrayIndices[sCellIdx];
      sCellEnd    = grid->lastCellArrayIndices[sCellIdx];
      sGridIdx3D.z  += sEmptyCells;
      sCurrentCoord += sEmptyCells * SAMPLES_PER_CELL;
  }
  syncthreads();

  if (sEmptyCells)  //empty space leaping
    continue;

  #pragma unroll SLAB_DEPTH
  for (int s = 0; s < SLAB_DEPTH; ++s)
    samples[s] = 0.0f;

  //add contribution of all particles in this cell to all sampling points
  int ptclListPos = sCellStart;
  do {
    int pctlArrayIndex = grid->ptclsToCells_ptcls[particleListPos];
    float ptclValue    = grid->ptclValues[pctlArrayIndex];
    float ptclVolume   = grid->ptclVolumes[pctlArrayIndex];
    float4 ptclPos     = grid->ptclPositions[pctlArrayIndex];

    #pragma unroll SLAB_DEPTH
    for (int s = 0; s < SLAB_DEPTH; ++s){
      float rayParameter = mapCoordToDistance(sCurrentCoord + s);
      float3 samplePos   = (rayParameter + jitterOffset) * rayVec;
      float3 distVec     = samplePosView - make_float3(particleViewPos);

      samples[s] += ptclValue * ptclVolume * dComputeWeight(distVec, ptclPos.w);
    }
  } while (++particleListPos <= sCellEnd);

  //front-to-back compositing for all sampling points
  #pragma unroll SLAB_DEPTH
  for (int s = 0; s < SLAB_DEPTH; ++s){
    //apply transfer function
    float4 colorContrib = make_float4(0.0f);

    if (sampledValues[s] > 0.0f)
      colorContrib = tex1D(gTransferFuncTex, sampledValues[s]);

    //compositing using the "over" operator (assuming pre-multiplied colors)
    color += colorContrib * (1.0f - color.w);
  }

  //cell iteration, performed by the leading ray
  syncthreads();
  if (isLeader){
      sGridIdx3D.z  += 1;
      sCurrentCoord += SLAB_DEPTH;
  }
}
```

Listing 4.4: A simplified raycasting main loop.

that we are using. The first ray of each ray bundle, identified as the first CUDA thread of each block, is starting the loop with the initialization of shared values that are related to cell traversal. This includes cell start and end indices in the map from cells to particle indices as well as the number of empty cells that can be skipped and the current ray coordinate. In line 14 and 15 of listing 4.4, empty cells are skipped, while lines from line 52 until the end describe the advance of the rays for non-empty cells. The rest of the code shown in listing 4.4 can be divided into two parts: lines 17 to 37 describe the sampling stage, lines 39 to 50 describe the classification and compositing stages. During the sampling stage, we add the contribution of each particle in the current cell to the sampling points of our slab cache. Note that we use the simplified assumption here that the slab cache has the exact size of the number of samples per cell. In our final program, this number can also be a fraction of the number of samples per cell, which involves multiple passes to process the aforementioned two stages in lines 17 to 50. Since the number of samples per slab is fixed, we can tell the CUDA compiler to unroll the loops in lines 18, 30 and 41 `SLAB_DEPTH` times, using the `unroll` pragma ([NVI11]). This way, the incrementation of the counter variable `s` is saved, as well as the expensive `if`-clause. This method can help us to improve the performance of our CUDA kernel ([KH10]). As the sampling has been done for all points inside our slab cache, the compositing step processes all of these points subsequently from front to back.

As can be seen, a lot of variables are necessary for our main raycasting kernel. We have already stored some of them in shared memory, so that they do not affect the memory consumption of each single thread. Still, each thread will need a certain amount of local memory and registers. In our application, those registers can be identified as the limiting factor, using NVIDIAs interactive excel sheet known as the *CUDA Occupancy Calculator*. This sheet enables the programmer to explore the occupancy of the streaming multiprocessors, taking the possible limiting factors into account. Those are the number of threads per block, the amount of shared memory consumed by each block and finally the number of registers per thread. We can know the first two of those three factors by checking the source code of our CUDA program. The third one, the number of registers per thread, is controlled by the CUDA compiler. Nevertheless, we can determine the number of registers assigned to each thread by using NVIDIAs *Visual Profiler*, a free tool that comes with the *CUDA Toolkit 4.0* which we have used. For our raycaster, the number of shared memory and per-thread memory needed by the raycasting kernel depends on the visualization technique we want to use. For normalized sampling, we need additional memory per thread to accumulate the sum of weights for each sample inside our slab cache. In addition to that, we also need to accumulate the gradient vector for each of those samples, if we want to use gradient-based illumination (as also mentioned in sections 3.6.2 and 3.7.2). The number of threads does not depend on the visualization technique we are using. In our application, we were always using bundles of 16×16 rays, leading to 256 threads per block. The results of occupancy calculation for different techniques are shown in table 4.2, assuming a slab depth of $D_{slab} = 16$:

Technique	Shared Memory / Block	Registers / Thread	Occupancy
Pseudo-Normalized Sampling	104 bytes	40	50,0%
Contour Shading	104 bytes	37	50,0%
Gradient-Based Shading	104 bytes	55	33,0%

Table 4.2: Occupancy calculation for different visualization techniques.

The Occupancy Calculator sheet reveals the number of registers as the limiting factor for occupancy in all cases. Note that the occupancy depends on the specific CUDA architecture which the graphics hardware is implementing, denoted as the *Compute Capability*. The CUDA compiler was left the freedom to decide about the number of registers that should be used. The compiler's optimization is dependent on many factors, one of those is the Compute Capability. For all our performance measurements, a *Geforce GTX 550 Ti* device with Compute Capability 2.1 was used.

Furthermore, we have analyzed the memory consumption of our perspective grid. As explained in section 3.4.2, the perspective grid consists of four structures:

- The cell start array
- The cell end array
- The empty successors cache
- The map from cells to relevant particles

The first three data structures have a neglectable size, since they are storing one value for each cell: using a grid with a cell size on the image plane of 16 × 16 pixels, with a resolution of 512 × 512 pixels, we have $32 \times 32 \times 108 = 110592$ cells (see 4.1). With four bytes for each entry, the memory consumption of each of the first three data structures equals $110592 \text{ byte} * 4 = 432$ KiB.

The map from cells to relevant particles, on the other hand, is much larger. For each particle out of $N_{particles}$ particles in total, we need to reserve a fixed number of n_{max} slots for each particle in this map when determining its relevant cells (see figure 4.1). Each entry of the map needs eight byte, four for the particle index and four for the cell index. This leads to a total memory consumption of $N_{particles} \times n_{max} \times 8$ bytes for the map. The amount of memory needed for the map for different values of $N_{particles}$ and n_{max} is summarized in table 4.3.

$N_{particles}$	n_{max}	Memory Consumption
131,072	8	8 MiB
131,072	27	27 MiB
524,288	27	108 MiB
524,288	64	256 MiB

Table 4.3: Memory consumption of the perspective grids particle map.

As we can see, for relatively large particle sets (for interactive simulations), the memory consumption stays still moderate. The GPU which we have used for our experiments for instance (a *Geforce GTX 550 Ti*), has a total amount of 1024 MiB of graphics memory, so there is no danger that we are running out of memory because of the perspective grid within our interactive setup. Note that even for an uncommonly high value for n_{max} of 64, the amount of memory requested is still manageable. For simulations with more particles, note that it is sufficient to reserve this space for all particles inside the view frustum. If we, for instance, have 10 million particles inside our simulation domain, but we know we will see one million of them at maximum at a time, the approach might still work out. However, we have not been investigating such large scale particle sets, which can nowadays not be simulated in real-time using standard consumer hardware and are therefore not possible for interactive simulations. For such cases, we are referring to [FAW10], who have shown how their approach can be used to interactively explore large, pre-computed cosmological SPH datasets with a pre-computed access structure.

4.1.3 Performance Optimization Techniques

Our sampling points along the rays are organized in slabs of equal size. Within each of those slabs, the location of the sampling points is determined by the adaptive sampling mechanism we are using. During each traversal of the points inside the slab cache, we have to know the current ray parameter t for each of those points in order to determine its position in view space. Since this parameter is shared along all rays of the bundle for a given position inside the slab cache (see section 4.1.2), we can store it in shared memory. Furthermore, this counts for all positions inside the slab cache, so we have an array `sRayParams` stored in shared memory which stores the ray parameters for all D_{slab} samples in our slab cache. We need to compute the ray parameter for each of those samples and store it in the corresponding entry of this array. Instead of employing another loop, which could be executed by the leading ray, we can compute each entry of this array by using the first D_{slab} arrays of a block in parallel, as shown in listing 4.5.

The reader might have realized that we are actually using $D_{slab} + 1$ pre-computed values

```
if (rayIdx <= SLAB_DEPTH)
{
  sRayParams[rayIdx] = mapCoordToDistance(sCurrentCoord + rayIdx);
}
syncthreads();
```

Listing 4.5: Collaborative ray parameter computation.

in this example. The additional ray parameter at position `sRayParams[SLAB_DEPTH]`, which might correspond to a point beyond the cells border, is only used for opacity correction. Since we want to obtain the length of each ray segment during this process, we have to provide an extra entry in order to be able to also compute the length of the last sample in the slab cache, as shown in listing 4.6.

```
for (int s = 0; s < SLAB_DEPTH; ++s)
{
  //... classification and shading: determine colorContrib

  //opacity and color correction
  const float segmentLength = (sRayParams[s+1] - sRayParams[s]) * rayVectorLength;

  colorContrib.w = 1.0f - pow((1.0f - colorContrib.w), segmentLength / referenceLength
      );
  colorContrib.x = colorContrib.x * (segmentLength / referenceLength);
  colorContrib.y = colorContrib.y * (segmentLength / referenceLength);
  colorContrib.z = colorContrib.z * (segmentLength / referenceLength);

  //... compositing: apply colorContrib
}
```

Listing 4.6: Opacity correction.

We have already introduced empty space leaping, provided by our perspective grid, as a well-known performance optimization technique. Another very important optimization technique for volume raycasting is known as *Early Ray Termination* (ERT). The basic idea is that the pixel color which is gathered along a ray will change just very slightly after its opacity component (α) has become greater than a limiting value $\alpha_{terminate}$ which is near to 100%. Therefore, one may terminate the ray traversal earlier, which will avoid further sampling and hence increase performance at the cost of a nearly invisible change in pixel color, compared to a full sampling along the ray. Since our raycasting process is performed by bundles of rays, it is not possible to terminate single threads without terminating the whole block. This is especially due to our shared computation of the ray parameters (see listing 4.5). Therefore, like [MRH10] and [OKK10], we are using a block-wise ERT mechanism. There are several possilities to implement such a mechanism within a CUDA kernel. The problem that needs to be solved is that the threads need to share information about the current value of α that they have gathered. If all rays have an opacity of $\alpha \geq \alpha_{terminate}$, the whole block may terminate.

The most simple possibility is to store a flag for each thread in shared memory. Each ray can then set its corresponding flag as soon as its gathered opacity is great enough to terminate. The leading ray sets its flag only if all other flags are already set. Each thread can then, after synchronizing, check the flag of the leading ray and terminate if it is set. For our bundles of 256 rays, this approach involves additional shared memory of $256 \times 4 = 1024$ byte. We can reduce this amount of memory needed by using CUDA's warp voting functions. Since threads are always processed as so-called warps of 32 threads, special vote functions can be used to evaluate a condition warp-wise. Nevertheless, we are using a third method which uses CUDA's atomic operations ([NVI11]), needing just a single variable in shared memory. The idea is to initialize the termination flag with 1, which is done by the leading ray, and letting each thread combine it with its evaluated ERT condition by using an atomic AND operation. The concept is shown in listing 4.7.

```
if (isLeader)
{
  sBlockTerminationSignal = 1;
}
syncthreads();

atomicAnd(&sBlockTerminationSignal, unsigned int(color.w >= 0.95f));
syncthreads();

if (sBlockTerminationSignal)
  return color;
```

Listing 4.7: Early ray termination with atomic operations.

4.1.4 Measured Performance

We have measured the performance of all visualization techniques that were proposed within our experimental setup, using 131,072 particles. Table 4.4 shows timings for the actual ray-casting process, without perspective grid construction, for different techniques and different viewport sizes.

Resolution	Pseudo-Normalized Sampling$_{16}$	Contour Shading$_4$	Gradient-Based Shading$_2$
1024 × 1024	219 (4.57)	421 (2.36)	801 (1.25)
512 × 512	100 (10.0)	184 (5.43)	361 (2.77)
256 × 256	79 (12.66)	140 (7.14)	287 (3.48)

Table 4.4: Performance of different visualization techniques. All timings are given in milliseconds. In brackets, timings are expressed as frames per second. Grid construction is not included.

Those measurements were performed with different settings for the slab depth D_{slab}, indicated by the small subscripted numbers next to the name of each technique. In section 3.6.2, we have shown that D_{slab} can be configured using an integral factor k:

$$D_{slab} = \frac{N_{tile}}{2^k} \tag{4.1}$$

For each technique, we have determined the ideal value of k by testing all valid values from $k = 0$ to $k = 4$ for our configuration with $N_{tile} = 16$. The results for a resolution of 256×256 pixels can be seen in figure 4.3.

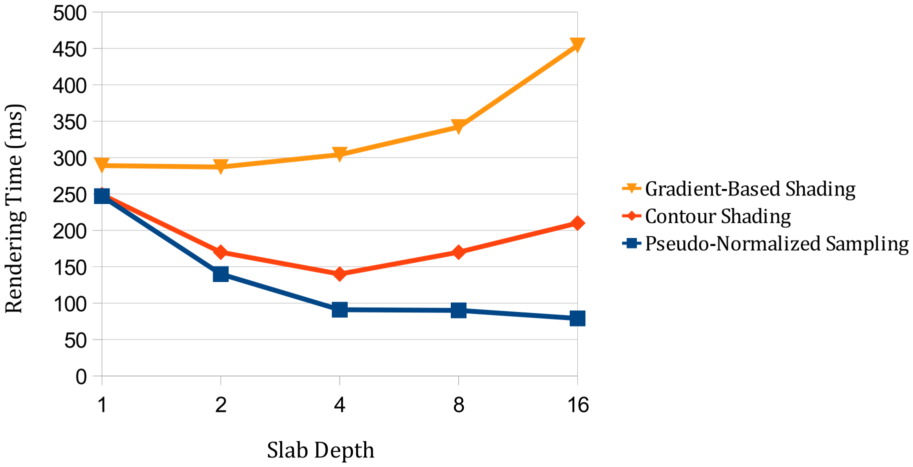


Figure 4.3: Performance for different slab depths.

Obviously, the slab depth should not be equal for all techniques. This is due to the memory consumption of the different CUDA kernels for different techniques, as described in section 4.1.2. If a CUDA kernel is consuming more resources, the hardware can potentially accommodate less threads. We can counteract this behaviour by reducing the slab depth, which will lead to less values in our slab cache and therefore to a smaller memory consumption of each thread. This will, on the other hand, make our slab cache less efficient. Reducing the size of the cache by half will lead to a doubling of global memory traffic for particle access operations, since we will need to access the particle set in twice as many passes to process all of the N_{tile} sampling points within the cell. Therefore, we find that the ideal size of D_{slab} is always a tradeoff between global memory traffic and per-thread memory consumption. As can be seen in figure 4.3, the ideal size of D_{slab} for pseudo-normalized sampling is $D_{slab} = N_{tile} = 16$. The contour shading technique uses normalized samples, leading to higher per-thread memory consumption, since we have to accumulate the sum of weights along with the samples. This makes a smaller slab depth potentially more efficient, in this case the optimal slab depth is $D_{slab} = 4$. Finally, when using gradient-based shading, the gradient vector has to be accumulated along with the samples and the sum of weights. Since the gradient vector consists of three floating point values, it causes three times as much additional memory consumption compared to the sum of weights. In our setup, this leads to the slab cache having no measurable effect on performance for gradient-based shading, since the performance for $D_{slab} = 2$ equals approximately the performance measured for $D_{slab} = 1$, where no slab-caching mechanism is employed at all.

To measure the effect of our empty space leaping (ESL) mechanism, we have compared the the performance of two raycasters, one with empty space leaping enabled and one without empty space leaping. The raycaster without empty space leaping was skipping empty cells individually, while the raycaster with empty space leaping was building and using our empty successors cache. The technique used was contour shading, with a slab depth of $D_{slab} = 4$, the number of particles was 131,072. Table 4.5 shows the total timings needed for raycasting and grid construction (including the construction of the empty successor cache, if any) for different viewport resolutions.

Resolution	Raycasting / Grid Construction, ESL	Raycasting / Grid Construction, no ESL
1024×1024	421 / 52	475 / 43
512×512	184 / 38	178 / 37
256×256	140 / 36	128 / 36

Table 4.5: Timings for empty space leaping. All timings are given in milliseconds.

Both using a slab depth of $D_{slab} = 4$, the raycaster with ESL enabled gets 43 registers per thread and 52 byte of shared memory, while the raycaster without ESL was assigned 45 registers per thread and needs 56 bytes of shared memory. As can be seen, for lower resolutions the number of cells in z-direction that can be skipped on average is too small to compensate the overhead which is caused by the higher amount memory consumption. This way, the raycaster without ESL outperforms the one with ESL enabled. However, this changes when the resolution increases, because the number of cells in z-direction gets larger (see table 4.1). This makes skipping empty regions in a single step more efficient compared to cell-wise skipping, so that ESL finally pays off.

We should note at this point that the raycasting performance results for the different techniques as shown in table 4.4 are roughly average timings, since there are other factors besides the slab depth that have a strong influence on performance. One of those factors is the amount of opacity which we use in our transfer function. Since we are using early ray termination (ERT), as shown in section 4.1.3, transfer functions with higher opacity values will lead to an earlier termination of the ray bundles. For the results shown in table 4.4, we have used a transfer function T with average opacity values. We have also measured the performance with and without opacity correction for the same example setup, using the contour shading technique with a 1024 × 1024 resolution of the image plane, for two additional transfer functions. T_{low} is containing smaller opacity values than those used in T, while T_{high} contains higher opacity values respectively. The ray termination threshold for the opacity value was set to $\alpha_{terminate} = 0.95$. The results are summarized in table 4.6.

Using ERT?	T_{low}	T	T_{high}
no	671	671	671
yes	483	421	239

Table 4.6: Impact of the transfer functions opacity on performance. All timings are given in milliseconds.

Furthermore, the distance from the viewer to the volume has an impact on the performance. Usually, one would expect that the performance decreases when we are getting closer to the volume, since more of our adaptively distributed sampling points will lie inside the volume. This is also the case in our application. However, in our setup this behaviour was not as strong as for the raycasting module of [OKK10]. We have compared both modules with ours configured to use pseudo-normalized sampling, which was closest to the unnormalized sampling used in their module. As a result, we have observed that our raycaster outperforms their raycaster when the eyepoint is close to the volume. On the other hand, when moving farther away from the volume, the method of [OKK10] is faster than our method. We believe that there are two reasons for that. As a first reason, we find that our perspective grid is very well-adapted to the particles in regions that are close to the image plane, which makes our raycaster fast in those regions. This changes when moving away from the image plane along the z-axis in view space, since the cellsize increases along z. Therefore, cells that are farther away are potentially containing a lot more particles and therefore also more non-relevant particles for each sampling point. This will cause more unnecessary global memory traffic and slow down the performance of our application. In contrast, [OKK10] are using an octree that is built in world space, with a uniform cell size at each tree level. Therefore, in their raycasting module the cells are always equally well adapted to the particles. The second reason is that they are employing an additional particle hierarchy to speed up the raycasting process in far regions. In a pre-processing step for the raycaster, coarser representations of the particle set are built. Those coarser variants are then used instead of the original particle set when sampling far regions. [FAW10] are employing a similar mechanism, where a particle hierarchy is used along with an octree.

So far, we have evaluated the performance of our raycaster. All timings were given without the time needed to construct the perspective grid. This was due to the fact that the construction of the grid is not the limiting factor in our application, so we have omitted to mention it until this point. The part of the grid construction which has the strongest impact on the performance is the sorting of the map from particles to cells, in order to obtain a mapping from cells to particles. We have measured the time needed for the whole grid construction and the time needed to sort the particle set, for two different particle sets and three different resolutions, and summarized the results in table 4.7.

Resolution	Total (Sorting), 2^{16} Particles	Total (Sorting), 2^{17} Particles
1024×1024	32 (16)	52 (31)
512×512	20 (16)	38 (31)
256×256	19 (16)	36 (31)

Table 4.7: Timings for grid construction. All timings are given in milliseconds.

As can be seen, the time needed for the sorting step is constant and just depending on the number of particles. Since radix sort has a linear runtime ([SHZO07]), we can assume that the sorting time is linearly depending on the number particles. The measurements shown in table 4.7 are approving this assumption. The time which is needed for e.g. the construction of the empty successors cache scales with the number of cells, depending on the resolution of the viewport. As can be seen in table 4.7, there is a threshold for the number of cells where the hardware cannot accommodate enough threads and hence needs multiple passes, which also causes a threshold for the time needed for grid contruction.

4.2 Summary & Discussion

In section 4.1, we have presented the implementation of our approach to SPH volume raycasting. It has been shown that the performance of our raycaster strongly depends on the configuration which we are using. An important parameter in this case is the slab depth D_{slab}, which is adjusted depending on the technique which we are using. This adjustment is made due to the tradeoff between global memory traffic and local memory consumption. As for most CUDA kernels, the ideal configuration of our raycasting kernel is furthermore dependent on the particular hardware platform. We have seen that a huge gap in performance exists between the pseudo-normalized raycaster and the contour-shading raycaster on the one hand and the raycaster with gradient-based shading on the other hand. Occupancy calculations support this observation and can therefore serve as a first clue towards performance limiting factors.

Our raycaster is reducing the amount of per-thread memory a lot, compared to other approaches, by sharing the whole data related to cell traversal data per thread block. This is made possible by our perspective grid as well as the sampling rate along the z-axis in view space, which is the same for all rays. The ray parameters within a slab can therefore be computed in a single step in parallel, which is another advantage of this approach. Nevertheless, the main limiting factor for our kernel is still the amount of local memory consumption. For our empty space leaping mechanism, this means that the small overhead caused by its additional memory requirements pays off only at high resolutions (see table 4.5).

As another performance optimization technique, we have employed early ray termination (ERT). Besides the configuration of the slab depth, this identifies the most important factor for the performance of our application, which is the opacity used within the transfer function. Using ERT, changing the opacity of the transfer function can lead to a performance change by a factor of two for our raycaster (see table 4.6).

In order to build our perspective grid, we have to find all relevant cells for each particle, which can be done in parallel for all particles. One might think that this is an expensive operation, since we have to check a set of frustum-shaped cell candidates for intersection with the particle. However, this is clearly not the most expensive part of the grid construction. Instead, the sorting of the map from particles to cells in order to obtain the map from cells to particles consumes the most amount of time (see table 4.7). Luckily, we have found that this sorting operation scales linearly with the size of the particle set, so that the time needed for constructing the perspective grid will very likely always be small, compared to the time which is needed to perform the actual raycasting.

We have also evaluated the overall global memory consumption of our perspective grid. The most important part of the grid structure is the map from cells to relevant particles. This map is, nevertheless, consuming still a moderate amount of global memory modern graphics hardware can handle. Compared to e.g. the perspective grid of [FAW10] this is an advantage, since they need multiple passes to process the grid as it is too big to fit into GPU memory at all. Certainly we should note that their raycaster is much faster than ours, and that is has been designed to explore very huge non-interactive datasets and therefore follows a completely different approach. For sets of more than two million particles, as used in their example datasets, the memory consumption of our perspective grid will also be too large to fit into common GPU memory. Put in a nutshell, we are benefitting from the fact that our particle dataset are relatively sparse compared to the size of our view frustum, so it is more sensible to make our memory consumption dependent on the number of particles than on the viewports resolution, in contrast to the setup of [FAW10].

We have also found that our raycaster outperforms the one of [OKK10] at distances close to the volume, while the opposite is true at larger distances between the virtual eyepoint and the volume. This caused by our adaptive cellsize in view space and their hierachical particle representation. Nevertheless, having a better performance at close regions is still an acceptable result.

Besides those performance aspects, we find that our implementation provides visually convincing results. Providing three visualization techniques which run at different speed levels and provide different levels of visual quality, the user may always choose the best-suited approach for the particular purpose. The pseudo-normalized sampling provides the fastest visualization, at the cost of visual quality. The contour shading approach is providing visually correct results at acceptable speed by employing normalized sampling. The gradient-based shading approach is the slowest visualization method. However, it takes the direction of an external light source and the surface normal into account, therefore it is able to visualize more information than the contour shading approach. It is furthermore configurable by all common parameters for blinn-phong shading, such as the range and intensity of specular highlights, while the contour shading approach is just configurable by the contour color and contour opacity. Also, the gradient-based method can visualize any isosurface within the volume, while the contour shading approach is limited to the principal surface. Nevertheless, during our experimentation, we found the contour shading approach to be the best solution, since it visualizes this most important surface in a convincing way and is furthermore much faster than gradient-based shading.

It depends on the definition of the term whether we can call our approach truly interactive. As shown in table 4.4, the raycaster that realizes our preferred visualization technique, which is contour shading, is able to perform the raycasting process at approximately 5 frames per second, excluding grid construction. In terms of games or movies, this is for sure not interactive. However, in our simulation setup, the most important task of a raycaster is to instantly provide a high-quality snapshot of the current situation. Unlike for games, the user does not have to react instantly on events inside the virtual environment. Instead, the possibilty of navigating through the scene to inspect the data is important. At 3 - 5 frames per second, we found that this can still be done relatively conveniently. Since we have the option to speed up navigation by instantly reducing the resolution for our raycaster, and also the option to switch back as soon as the interaction for navigation is over, we are providing a convenient environment for an ad-hoc inspection of the data. Because of those reasons, we can state that our result is an interactive SPH volume raycasting solution.

Chapter 5

Conclusion

5.1 Summary

Within this thesis, a novel approach to SPH volume raycasting, which is able to provide a high quality volume visualization, has been proposed. Using a gathering approach, implemented in CUDA as a parallel raycasting algorithm, the method is better suited than other approaches ([FAW10]) to visualize the exact values of the scalar field. This is due to the fact that it provides the possibility of normalizing the sampled values in the same pass where the regular sampling takes place, unlike for scattering approaches. Our approach uses a discretization of the view frustum, as proposed by [FAW10], in order to build a perspective grid which stores references to all relevant particles for each cell. The adaptive cell size of the perspective grid is aligned with our adaptive steps along the rays, leading to a fixed number of samples per cell. This property, along with the alignment of the cells to the view frustum, can be exploited by using several caching mechanisms. Rays are bundled into units of equal size, which correspond exactly to the thread blocks used in our CUDA program. Therefore, information about the cell traversal can be shared among all rays in a block, reducing the memory consumption per thread. Also, a slab cache, following [MRH10, OKK10], is used to process a certain number of sampling points in a single pass when sampling the scalar field.

Furthermore, in order to provide the user with visual information about surfaces for the normalized sampling result, local volume illumination can be added. In contrast to other approaches like [OKK10, MRH10], we are directly obtaining the exact gradient in order to compute a normal vector which is needed for the lighting computations. This method is more stable than any gradient estimation technique, such as finite differences. In addition, we have presented a simple yet visually satisfying contour shading approach for SPH volume data, which can be seen as a cheaper alternative to gradient-based shading. For an even cheaper visualization at the cost of visualizing false information in some regions, we have proposed pseudo-normalization. The simple idea behind this technique, which can also be used for scattering approaches, is to divide each sample by a constant factor, assuming a fully occupied particle neighbourhood at each point. Measuring the performance of different visualization techniques, we have found that gradient-based shading is much slower than contour shading, which is in turn slower than pseudo-normalized sampling. The performance of each technique can be optimized by using the ideal slab depth, which is tied to the memory consumption of the threads and therefore to the visualization technique we choose.

As additional optimization mechanisms, empty space skipping and early ray termination have been implemented. While empty space skipping pays off only at high resolutions, early ray termination is the most important optimization method in our application. Since this method is dependent on the opacity which is used for the transfer function, the choice of the transfer function can also have a huge impact on the performance of the whole application.

The proposed optimization and visualization techniques, along with the chance to change the quality instantly on user interaction, provide us with a high-quality interactive raycasting solution that is especially suited for small and medium scale particle sets, as used in interactive SPH simulation environments.

5.2 Limitations and Future Work

Throughout this work, we have assumed fixed cell size of 16 × 16 pixels on the image plane for our perspective grid. This leads to 256 pixels in each cell and therefore to 256 threads in each thread block of our CUDA program, which is a good choice. However, this ties the cell size that is used for the perspective grid to the size of CUDA's thread block and vice versa, which is not desireable. If, on a future device, the ideal number of threads should be 1024, our approach will encourage us to use a cell size of 32 × 32 pixels on the image plane. Otherwise, we would have the change the raycasting kernel, since have been assuming that each ray in a ray bundle will always traverse the same cells as the others. Such a change will come along with a more complex CUDA kernel, introducing additional memory needs. On the other hand, using cells with a size of 32 × 32 pixels on the image plane will, in our particular setup, also slow down the raycaster. This is due to the fact that, on average, more non-relevant particles will be fetched from global memory at each sampling point. To summarize this aspect, we can state that the configuration which we used to implement our well-performing raycaster is a very sensitive construct which depends on the particles' influence radius as well as on the ideal number of threads per block for our CUDA program.

In general, a major disadvantage of a CUDA-based approach like ours is the difficult process of performance optimization. Since we have to optimize each visualization technique of the raycaster individually on a particular hardware platform, changing that platform means a time-consuming performance tuning process, including re-compilation of the program each time a different slab depth is tested. However, CUDA is able to handle shared memory of flexible size. Using this feature, we could adjust the slab depth dynamically in our program, making a re-compilation unneccessary. Still, the CUDA compiler will always optimize our raycasting kernel for a particular compute capability. Optimizing for our particular platform means that we loose backwards compatibility if we do not tell the compiler to ensure compatibility with older compute capabilities. Since, in this case, the amount of resources like registers might be limited by the compiler, we might loose performance again. Therefore, it will probably be the best choice to re-compile the program when porting to another hardware platform. This is, of course, a general problem and does not apply specifically to our raycaster. Nevertheless, since a few years, a lot of work is invested in making the creation of efficient CUDA programs easier ([KH10]). Within a few years, simple optimization tasks like loop unrolling should be performed fully automatically by the CUDA compiler, and profiling tools like NVIDIA's Visual Profiler will be developed further to make performance optimization for CUDA kernels even easier than it already is.

Besides that, using CUDA means, of course, that our raycaster will only run on NVIDIA hardware. This could change within the next years, when OpenCL as a platform-independent standard for parallel computing has become more mature. Since our concept is not relying explicitely on special features provided by CUDA, it is probably possible to port the implementation to OpenCL in the future.

Our gradient computation scheme is another possible limitation of our CUDA program. In contrast to [MRH10, OKK10], we are not using finite differences that have been computed among neighbouring samples. Instead we are accumulating the gradient vector for each sample in our slab cache, which either limits the slab cache to a very small size, like only two samples, or makes it even completely useless. It is worth further investigation if we can improve the performance of our approach, since we have not compared it to any other method yet. A similar mechanism like the one used by [MRH10, OKK10] is not suited for our approach because of several reasons: Estimating the gradient by taking neighbouring samples into account would mean that we have to use a slab cache of at least three samples per ray in order to be able to compute the finite differences. This slab cache for all rays has then to be stored in shared memory , which will potentially limit its size. Furthermore, we will need the ray bundles to overlap with each other, the rays at the border of each bundle will then just be used for gradient computation. This will make our cell structure much more complicated and less efficient. Still, it might be interesting to have a direct comparison of our method of accumulated gradients with another method which consumes less memory per thread. Such a method could involve a small slab depth of e.g. 4 samples, which would lead to $4 \times 16 \times 16 \times 3 \times 4$ bytes = 12KiB shared memory to store all accumulated gradients. This is a huge amount of additional shared memory consumption, but still manageable and therefore worth a comparison against our current method of storing the accumulated gradients in local thread memory.

Another limitation is the amount of particles which our approach can handle. Since we are keeping a map from cells to relevant particles in graphics memory, we can only handle up to approximately 1 - 4 Million particles, depending on the amount of graphics memory which is provided by the hardware. However, this is not a big problem at the moment, since the original scope of our approach are interactive setups, where such particle sets are rather uncommon at the moment.

Finally, we find that a hierarchical particle representation, as used by [OKK10, FAW10], could also help us to speed up our raycaster. The creation of the coarser representations of the particle set could be performed adaptively within our perspective grid: since we have already aligned our access structure to the view frustum, we could limit the upsampling to far regions inside the frustum.

Bibliography

[APKG07] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively sampled particle fluids. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[BK02] Javier Bonet and Sivakumar Kulasegaram. A simplified approach to enhance the performance of smooth particle hydrodynamics methods. *Appl. Math. Comput.*, 126(2-3):133–155, March 2002.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 192–198, New York, NY, USA, 1977. ACM.

[Bli94] James F. Blinn. Compositing, part 1: Theory. *IEEE Comput. Graph. Appl.*, 14(5):83–87, September 1994.

[CM99] Paul W Cleary and Joseph J Monaghan. Conduction modelling using smoothed particle hydrodynamics. *Journal of Computational Physics*, 148(1):227 – 264, 1999.

[CMH$^+$01] Balázs Csebfalvi, Lukas Mroz, Helwig Hauser, Andreas König, and Meister Eduard Gröller. Fast visualization of object contours by non-photorealistic volume rendering. Technical Report TR-186-2-01-09, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, April 2001.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 9–16, New York, NY, USA, 2001. ACM.

[FAW10] Roland Fraedrich, Stefan Auer, and Rudiger Westermann. Efficient high-quality volume rendering of sph data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1533–1540, November 2010.

[Gre10] Simon Green. Particle simulation using cuda. Whitepaper, NVIDIA Corporation, December 2010.

[Her94] M. Herant. Dirty tricks for sph. *Memorie della Societa Astronomica Italiana*, 65:1013+, 1994.

[HKRs+06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[Hor70] Berthold K.P. Horn. Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1970.

[IABT11] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. *Computer Graphics Forum*, 30(1):99–112, 2011.

[KH10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[KP+11] Ron Kikinis, Steve Pieper, et al. *3D Slicer 4.0*, November 2011.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, July 1990.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 451–458, New York, NY, USA, 1994. ACM.

[MGS$^+$01] M. Meißner, S. Grimm, W. Straßer, J. Packer, and D. Latimer. Parallel volume rendering on a single-chip simd architecture. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, PVG '01, pages 107–113, Piscataway, NJ, USA, 2001. IEEE Press.

[MHS08] L. Marsalek, A. Hauber, and P. Slusallek. High-speed volume ray casting with cuda. In *Proceedings of IEEE Symposium on Interactive Ray Tracing (2008)*, page 185. IEEE, August 2008.

[Mon05] J J Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703, 2005.

[MRH10] Jörg Mensmann, Timo Ropinski, and Klaus H. Hinrichs. An advanced volume raycasting technique using gpu stream processing. In *GRAPP: International Conference on Computer Graphics Theory and Applications*, pages 190–198, Angers, 2010. INSTICC Press.

[MSRMH09] Jennis Meyer-Spradow, Timo Ropinski, Jörg Mensmann, and Klaus Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13, 2009.

[NVI11] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2011. Version 4.1.

[OKK09] Jens Orthmann, Maik Keller, and Andreas Kolb. Integrating gpgpu functionality into scene graphs. In *VMV*, pages 233–244. DNB, 2009.

[OKK10] Jens Orthmann, Maik Keller, and Andreas Kolb. Topology-caching for dynamic particle volume raycasting. In *Proceedings of Vision, Modeling and Visualization*, pages 147–154, Siegen, Germany, 2010.

[OLG$^+$07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[RE01] Penny Rheingans and David Ebert. Volume illustration: Nonphotorealistic rendering of volume models. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):253–264, July 2001.

[RGW$^+$03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the* *symposium on Data visualisation 2003*, VISSYM '03, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[SDG08] George Stantchev, William Dorland, and Nail Gumerov. Fast parallel particle-to-grid interpolation for plasma pic simulations on the gpu. *J. Parallel Distrib. Comput.*, 68(10):1339–1349, October 2008.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[SKB$^+$06] Magnus Strengert, Thomas Klein, Ralf P. Botchen, Simon Stegmaier, Min Chen, and Thomas Ertl. Spectral volume rendering using gpu-based raycasting. *The Visual Computer*, pages 550–561, 2006.

[YT10] Jihun Yu and Greg Turk. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 217–225, Aire-la-Ville, Switzerland, 2010. Eurographics Association.

[YTWJ12] Jihun Yu, Greg Turk, Chris Wojtan, and Chee Jap. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 33rd Annual Conference of the European Association for Computer Graphics (Eurographics) (to be published*, Aire-la-Ville, Switzerland, 2012. Eurographics Association.

[ZGHG11] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, May 2011.

[ZSP08] Yanci Zhang, Barbara Solenthaler, and Renato Pajarola. Adaptive sampling and rendering of fluids on the GPU. In *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 137–146, 2008.