Jörg Mensmann

Exploiting Spatial and Temporal Coherence in GPU-Based Volume Rendering

Münster • 2010

WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

Informatik

Exploiting Spatial and Temporal Coherence in GPU-Based Volume Rendering

Inauguraldissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften durch den Fachbereich Mathematik und Informatik der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Jörg Mensmann
aus Haltern

2010

Typeset with LaTeX2e and KOMA-Script using the fonts Palatino, Bera Sans, and Bera Mono. Text editing was performed using GNU Emacs/AUCTeX. Figures were created with Inkscape, GIMP, gnuplot, and OpenOffice.org.

Dekan:	Prof. Dr. Christopher Deninger
Erster Gutacher:	Prof. Dr. Klaus H. Hinrichs
Zweiter Gutacher:	Prof. Dr. Sergei Gorlatch
Tag der mündlichen Prüfung:	16. Juli 2010
Tag der Promotion:	21. Juli 2010

Abstract

Efficiency is a key aspect in volume rendering, even if powerful graphics hardware is employed, since increasing data set sizes and growing demands on visualization techniques outweigh improvements in graphics processor performance. This dissertation examines how spatial and temporal coherence in volume data can be used to optimize volume rendering. Several new approaches for static as well as for time-varying data sets are introduced, which exploit different types of coherence in different stages of the volume rendering pipeline. The presented acceleration algorithms include empty space skipping using occlusion frustums, a slab-based cache structure for raycasting, and a lossless compression scheme for time-varying data. The algorithms were designed for use with GPU-based volume raycasting and to efficiently exploit the features of modern graphics processors, especially stream processing. The improvements in rendering performance achieved by the optimizations allow interactive rendering of volumetric data on common workstations and open up new vistas for more complex visualization techniques.

Contents

Preface

So long, and thanks
for all the fish.

(Douglas Adams)

THIS DISSERTATION represents the result of work carried out from November 2006 to June 2010 at the Department of Computer Science, University of Münster under the kind guidance of Prof. Dr. Klaus Hinrichs, whom I wish to thank for giving me the opportunity for this interesting research and for his encouragement throughout. Furthermore, I would especially like to thank PD Dr. Timo Ropinski, who is the initiator and project leader of the VOREEN software project, for his constant support and fresh ideas.

I would like to express my thanks to my past and present colleagues from the Visualization and Computer Graphics Research Group, including Dr. Frank Steinicke, Dr. Jennis Meyer-Spradow, Jörg-Stefan Praßni, Stefan Diepenbrock, and all others, for all the fruitful on-topic as well as off-topic discussions and for a very nice working environment.

Stefan Diepenbrock also supported me with additional implementation work on the occlusion frustum approach presented in Chapter 5. I would like to thank Johannes Lülff and Michael Wilczek from the Institute for Theoretical Physics for valuable discussion about time-varying volume data and for supplying their data sets. Finally, thanks go to everyone who participated in the development of the VOREEN framework. The project allowed me to concentrate on relevant topics when implementing the techniques discussed in this thesis, as well as giving me the possibility to take part in the construction of such a complex and powerful visualization system.

Parts of this work were supported by grants from Deutsche Forschungsgemeinschaft through the Sonderforschungsbereich 656 "Molekulare kardiovaskuläre Bildgebung" (MoBil), project Z1. Acknowledgments for the volume data sets used throughout the thesis can be found in Appendix B.

Münster, June 2010 *Jörg Mensmann*

Chapter 1

Introduction

With the emergence of volume data in medicine, but also in many other fields such as fluid dynamics and seismology, and with the ever increasing amount and complexity of such data, volume visualization has become an essential tool for data analysis. This was made possible by improvements in graphics hardware, which today allow interactive rendering of large volume data sets even on standard workstations—although current graphics processors (GPUs) were designed primarily for rendering triangles and not intended for rendering volume data. Volume rendering is a computationally expensive task, and efficiency is a key aspect to be considered in any application dealing with volume data for visualization purposes.

We have approached the goal of improving volume rendering performance by developing techniques that exploit *spatial* and *temporal coherence* in volume data in order to accelerate different stages of the rendering process and thus increase the rendering frame rate. The presented techniques efficiently utilize new features in graphics hardware such as geometry shaders and stream processing.

In the context of volume rendering, *data coherence* is an important aspect to achieve good performance results. A volumetric data set is defined to be coherent if its data elements exhibit a non-random distribution. Hence, coherence means that the data has a certain structure, which can be exploited by appropriate algorithms. Two main types of coherence can be identified: spatial and temporal coherence. Both types of coherence can describe the data, but they can also characterize the behavior of algorithms interacting with the data, especially how an algorithm accesses the data elements.

Spatial data coherence refers to data that exhibits an intrinsically coherent structure in the spatial domain, i. e., the values of spatial data elements are not distributed randomly and are not independent of the values of neighboring elements. For a spatially coherent algorithm the same properties apply to the way in which the data elements are processed, i. e., the memory access pattern. An example for such a spatially coherent algorithm is an image-order rendering technique, which exploits the tendency of neighboring fragments to use the same data during rendering. In the special case of image-order volume rendering using the raycasting technique, this translates to the tendency of neighboring rays to access the same voxels during ray traversal. Such spatially coherent algorithms must be seen in the context of caching mechanisms, as the algorithms ensure data locality and therefore efficient cache utilization. But spatial coherence can also be exploited directly. For example, volume rendering can be accelerated by skipping over parts of the volume that do not contribute to the final image, i. e., parts in which all voxels are empty.

Temporal data coherence can be found in data that models time-varying phenomena at multiple time steps. For this kind of data the state of an element at a certain time step typically depends on its state in the previous step, hence exhibiting coherence in the time domain. This coherence can be exploited for volume data compression by storing only those parts of the volume that change between two time steps. Even if the data itself is not time-varying, temporal coherence can be introduced by running multiple iterations of an algorithm, when the results are associated with different time steps. A temporal component is often introduced into rendering algorithms by continuously changing the viewing parameters, e. g., the camera position, and rendering multiple output frames while these changes are applied. With small view changes between rendering the frames, the resulting images exhibit coherence, as each image only differs slightly from the previous image.

Even when exploiting coherence for optimization, volume rendering is still computationally expensive and can benefit significantly from the use of graphics hardware. Graphics processors are not only getting faster with each new generation, but they also add new features, so that today they support many functions previously reserved to CPUs. Hence, while new hardware will typically speed up volume visualization simply by means of higher clock rate or by integrating more compute cores, exploiting the new features is crucial to fully utilize the hardware resources and achieve optimal performance results. Even though interactive volume rendering is already possible with current graphics processors, improving the rendering performance further is an important challenge. Faster volume rendering will allow using more complex visualization techniques or larger data sets without switching to specialized hardware solutions or reducing the image quality. After all, volume rendering will not become “fast enough” in the foreseeable future, because of rising data set sizes as well as growing demands on visualization.

Volume rendering is not supported natively by common graphics hardware. Instead, implementations have to utilize hardware resources in unconventional ways.

To fully exploit these resources, features available in GPUs as well as the overall system architecture must be examined and volume rendering algorithms must be adapted. This holds true especially for raycasting, a volume rendering technique that yields optimal image quality, but is also computationally expensive. Raycasting is akin to raytracing and therefore the approach differs significantly from the triangle rasterization performed by common graphics processors. Nonetheless, it can be implemented to exploit the hardware resources, albeit not directly.

In this dissertation, we introduce three GPU-based algorithms for accelerating volume rendering by exploiting coherence. They target different parts of the volume rendering pipeline and support both static and time-varying data. First, we present a new approach to empty space skipping that exploits the temporal coherence between sequential output images. It utilizes geometry shaders, which were previously not used in volume rendering. Second, we examine the applicability of the new stream programming model to volume rendering in comparison to the classical shader programming approach, and we present a novel caching technique exploiting previously unavailable hardware features. The described algorithm makes use of spatial ray coherence and relies on data locality in volume raycasting. Third, we develop a lossless compression scheme for time-varying volume data, using spatial as well as temporal data coherence. It utilizes both the CPU and the GPU to increase overall rendering performance and to achieve interactive frame rates.

This dissertation is structured as follows. Before we describe the acceleration techniques in more detail, we review fundamental concepts and related work in GPU programming (Chapter 2) and volume rendering (Chapter 3), and present a short introduction to the Voreen volume rendering framework (Chapter 4), which we used for implementing our techniques. These chapters may be skipped by readers already familiar with stream processing and volume rendering or having prior experience with Voreen. In Chapter 5 we present the empty space skipping approach, which is based on the novel concept of *occlusion frustums*. We discuss the applicability of the stream programming model for volume rendering in Chapter 6 and present the *slab-based* raycasting algorithm, which was specially adapted to the hardware architecture. After having discussed static data in the previous chapters, we transfer our findings to time-varying volume data in Chapter 7. This type of data introduces individual performance challenges, especially regarding data set size and bandwidth bottlenecks, which we approach using a *hybrid compression scheme*. In the final Chapter 8 we summarize the results of our acceleration approaches and discuss the lessons learned, especially those concerning the use of novel hardware features. We also describe which role further advances in graphics hardware might play in the future.

The contributions presented in this dissertation are based on the following publications: The occlusion frustum approach for empty space skipping described in Chapter 5 was presented at the Eurographics/IEEE Symposium on Volume and Point-Based Graphics (Mensmann et al., 2008a). Initial findings of the slab-based volume raycasting approach from Chapter 6 were presented as a poster at the ACM Conference on High Performance Graphics (Mensmann et al., 2009), while the final results were presented at the International Conference on Computer Graphics Theory and Applications (GRAPP) (Mensmann et al., 2010a). Finally, the time-varying compression scheme described in Chapter 7 was presented at the IEEE/Eurographics Symposium on Volume Graphics (Mensmann et al., 2010b). Further work related to volume rendering has been conducted by contributing to the following publications: The basic idea for the occlusion frustum proxy geometry introduced in Chapter 5 came from an evaluation of complex proxy geometries for volume raycasting, which has also been used for implementing interactive volume deformation (Mensmann et al., 2008b). As examples for complex visualization techniques that would benefit from performance improvements in the raycasting component, contributions have been made to work on dynamic ambient occlusion (Ropinski et al., 2008) and shape-based transfer functions (Praßni et al., 2010). Finally, contributions have been made to the development of the Voreen framework, which has been used for developing and integrating the acceleration techniques, and to the related publications (Meyer-Spradow et al., 2009, 2010).

Chapter 2

General-Purpose Programming on Graphics Processors

Graphics processors do not support volume rendering directly and therefore the same methods as for implementing general-purpose tasks need to be applied to realize volume visualization techniques on GPUs. Hence, we give a brief introduction to the relevant topics of GPU programming, stream processing, and the CUDA architecture.

Graphics processors have evolved from providing access to a simple frame buffer into fully programmable and massively parallel stream processors. Today's GPUs often outperform CPUs for tasks that can be adapted to their architecture, even for applications outside of graphics programming. Using GPUs for tasks not directly related to graphics and traditionally performed by CPUs is called *general-purpose computing on graphics processing units* (GPGPU).

In this chapter, we examine classical GPGPU programming as well as the more recent stream processing approach. One implementation of this approach is the CUDA architecture, which we describe in more detail, as we will use it extensively in Chapters 6 and 7. We assume that the reader is already familiar with the basic principles of computer graphics and GPU programming, as described, e. g., in the OpenGL Programming Guide ("The Red Book") by Shreiner (2009).

2.1 GPGPU Approaches

Graphics processors have evolved significantly in recent years. While early 3D accelerators could only draw textured triangles, over time more and more functionality was relocated from the CPU to the GPU, such as geometry and lighting calculations. With the introduction of programmable graphics processors fully supporting branching and looping in shader programs, the hardware became capable of performing tasks not directly related to graphics. At the same time the graphics hardware performance was—and still is today—increasing more rapidly than that of CPUs (Owens et al., 2007). CPUs are optimized for executing sequential code, and much of their previous performance increase was based on increasing clock rate. As this is limited by thermal issues and power requirements, recent developments focused on multi-core approaches rather than increasing the performance of a single processor. Modern CPUs support concurrent execution both by utilizing multiple independent cores as well as on the instruction level of single cores through extensions such as Streaming SIMD Extensions (SSE) for the Intel x86 instruction set. Still, the much simpler and inherently parallel architecture of GPUs allows to use additional transistors more efficiently, achieving higher performance with the same transistor count. Another advantage of graphics processors for certain applications is their specialized hardware for operations such as texturing. These are intended for use in graphics programming and are highly optimized for this task, but they can also be exploited for applications unrelated to graphics.

The parallel architecture of graphics processors is naturally suited for many data-parallel tasks found in computer graphics. Still, while fully programmable, this specialized architecture is only applicable to problems that can be mapped to the hardware efficiently. CPUs spend a considerable amount of their transistors on branch prediction and cache management to optimally utilize the hardware when executing sequential code. These features are missing from GPU architectures, hence they will not perform well with sequential algorithms containing lots of conditional branches and loops. Therefore, one must carefully verify whether the problem is suited for a solution with this type of architecture before starting with an implementation. A survey of GPGPU techniques by Owens et al. (2007) describes basic operations such as map/reduce or scatter/gather, and more complex algorithms for sorting, searching, or solving differential equations. These are used in fields like physically-based simulations, signal processing, or computer vision.

2.2 Stream Processing

In contrast to the sequential programming model of CPUs, the *stream processing* model (Kapasi et al., 2003; Owens, 2005) structures programs in a way so that they can be mapped efficiently to the highly parallel structure of GPU architectures. It is based on *streams* of structured data that are processed by *kernels*. The most important property of kernels is that they operate on entire streams instead of on individual elements. At the same time only a simple control flow is supported, with the goal of achieving a coherent branch behavior for adjacent elements in the stream. Hence, stream elements can be processed in parallel by the data-parallel hardware of graphics processors. It is interesting to note that this approach fits well with the traditional model of a graphics pipeline, in which streams of primitive data such as vertices, triangles, or fragments are sent through the individual processing stages, i. e., kernels.[*] Examples of stream processors are programmable GPUs, but also the Cell Broadband Engine Architecture (Kahle et al., 2005) can be used as a stream processor, although its architecture is more complex and also supports other programming models.

2.2.1 History of stream processing in graphics

Initially, programming of graphics processors for GPGPU tasks was performed by using standard graphics APIs such as OpenGL. Simple tasks can be implemented by basic graphics operations such as storing data in textures and blending multiple textures using the fixed function pipeline. For more complex problems, shader programs must be used. Typically fragment shaders are used as kernels, for which the input streams are made available as textures and the output stream is the resulting image of rendering a screen-aligned quad while the fragment shader is active. Fragment shaders were originally intended for implementing illumination models for scene geometry, but they are flexible enough to support implementing quite different algorithms.

When the suitability of using graphics processors for general-purpose computations became apparent, several approaches for accessing GPU hardware without going through graphics APIs were made available by hardware vendors as well as third-party developers. AMD/ATI supported direct GPU programming through their Stream SDK (AMD, 2009) and the Brook+ stream processing language (Buck et al., 2004b). NVIDIA introduced the Compute Unified Device Architecture (CUDA) as both a parallel architecture for their GPUs and a programming model (Nickolls et al., 2008). It allows implementing stream processing kernels in the C programming language while permitting full access to the hardware. More recently, OpenCL (Munshi, 2009) was introduced as an industry standard in order to provide a vendor-neutral solution. Also based on the stream processing concept, it shares many similarities with CUDA. Therefore, most of the results obtained for CUDA can be directly mapped to OpenCL. We believe that, due to the support by major hardware vendors and availability for several operating systems, OpenCL will become the standard solution

[*] Although related, we do not consider shader programming and other graphics-based GPGPU techniques as belonging to the stream processing model. While a fragment shader can be interpreted as a kernel processing a stream of input fragments, we believe shader programming to be too specific to graphics to fit into the general stream model. Hence, with the term *stream processing*, we refer to CUDA or OpenCL, but not to shader programming.

GPU	SP	MP	regs	bandwidth
GeForce 8800 GT	112	14	8,192	57.6 GB/s
GeForce GTX 280	240	30	16,384	141.7 GB/s

Table 2.1: Specifications of the CUDA-capable graphics processors we used for testing our techniques, showing the number of available scalar processors (SP) and multiprocessors (MP), the number of hardware registers per multiprocessor (regs), and the maximum memory bandwidth.

for implementing stream processing programs in the foreseeable future. However, OpenCL implementations only became available recently and have not yet reached a level of stability and optimization comparable to the more mature CUDA implementations. Consequently, we focused on using CUDA for implementing stream processing algorithms on the GPU, but we see no problems in mapping our results and porting our algorithms to OpenCL.

2.3 CUDA Basics

An in-depth understanding of the underlying hardware architecture is essential to get good performance results with CUDA. Some basic information about the architecture and its limitations are given in this section, while for a deeper discussion we refer to external resources (NVIDIA, 2010). Our discussion is specific to CUDA devices supporting compute capability 1.x, as GPUs implementing the new FERMI architecture (GeForce 400 Series) and supporting compute capability 2.x were not yet available at the time of writing this thesis.

A CUDA-capable GPU can apply a *computation kernel* to a large number of parallel threads. Up to 512 threads are organized in thread *blocks*, which have access to an on-chip shared memory. A CUDA *device*, i. e., a GPU, consists of multiple *streaming multiprocessors* (MP), onto which thread blocks are distributed (see Figure 2.1). Specifications of some CUDA devices are listed in Table 2.1. Thread blocks are further partitioned into *warps* of 32 threads that are executed by the *scalar processors* (SP) on the multiprocessor. All threads in a warp are executed in parallel with different code paths leading to serialization of execution.[*] It is therefore important to have a high degree of branch coherence inside a warp.

The stream processing model made available through CUDA can be characterized as SIMT: single instruction, multiple threads. It is related to the common SIMD model

[*] Since each streaming multiprocessor contains 8 scalar processors, only 8 of the 32 threads in a warp can actually be executed in parallel, but this is not visible to the programmer.

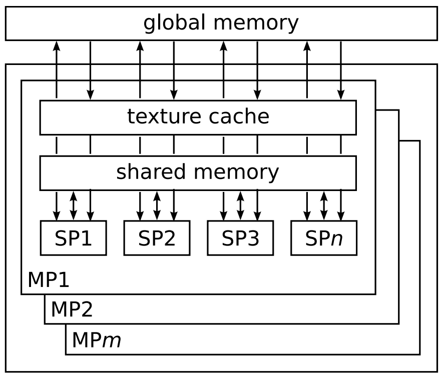


Figure 2.1: High-level overview of the CUDA architecture.

(single instruction, multiple data), but sets the focus on threads that process the individual data elements in parallel. While the hardware supports a large number of threads, they are *oversubscribed*, i. e., many more logical threads are started than physical threads are available. In combination with fast switching between active threads this allows efficient handling of stalling operations without a deep and complex pipeline as in CPUs: When a thread block waits for the result of a memory transaction, a multiprocessor can simply switch to a different thread block that performs arithmetic operations or for which the memory transaction has already finished. In the optimal case this approach achieves full utilization of the computing resources by hiding memory latency through thread switching.

2.3.1 Differences between CUDA and graphics programming

While using the same hardware as shader programs, CUDA makes available certain features that are not accessible by applications through graphics APIs. In contrast to shader programs, a CUDA kernel can read and write arbitrary positions in GPU *global memory*. The global memory space is located in the device memory and therefore has a higher latency and lower bandwidth than on-chip memory such as registers or caches. To achieve maximum bandwidth from global memory, suitable access patterns have to be chosen to *coalesce* simultaneous memory accesses into a single memory transaction. The coalescing rules (NVIDIA, 2010, p. 144) can be observed easily for block-wise loading operations, while more complex—effectively random—access patterns will not achieve full coalescing. As an example for a coalescing rule, the memory access from all threads of a half-warp (i. e., 16 threads) will be coalesced into a single memory transaction if all the 8-bit words accessed by the threads lie in the same 32-byte memory segment.

Each multiprocessor on a CUDA device contains a small amount of on-chip memory that can be accessed by all threads in a thread block and can be as fast as a hardware register. This *shared memory* is not available to shader programs. The total amount of shared memory in each multiprocessor—and therefore the maximum amount available to each thread block—is limited to 16 kB with compute capability 1.x hardware. Similar to the coalescing issues, shared memory can only be as fast as a hardware register as long as the access pattern introduces no memory bank conflicts. While the penalty is less grave than with non-coalesced global memory access, bank conflicts can massively decrease the effectiveness of the shared memory.

For fragment shaders the processing order of the fragments is undefined and cannot be controlled, although the order can have great influence due to coherence aspects. On the contrary, the size and distribution of CUDA thread blocks must be controlled manually. The block size is limited by the available hardware registers and shared memory: Each thread block can use a maximum of 16,384 registers, depending on the hardware (compare Table 2.1). With a block size of 256 this would allow 64 registers per thread, while with a smaller block size of 64 the number of available registers increases to 256. At most half of these should be used per block to allow running multiple thread blocks on a multiprocessor at the same time. This means that a complex kernel must be run with a smaller block size than a simple one. Likewise, making full use of all available shared memory, e. g., to share information between adjacent threads, restricts a multiprocessor to work on only one thread block at the same time. Therefore the amount of shared memory allocated per block should be at most half the total amount available in the multiprocessor, in order to be able to hide memory latency by switching between active threads.

Choosing an appropriate block size is essential to get optimal performance from a CUDA kernel. However, the optimal block size depends on several boundary conditions and no general value suitable for all applications can be given. Fortunately, knowing the capabilities and limitations of a CUDA device allows to assess the effect of block size on performance for a specific kernel configuration. The *warp occupancy*, which is the ratio of active warps to the hardware-dependent maximum number of warps on a multiprocessor, can be retrieved through the occupancy calculator tool (NVIDIA, 2008a) and gives a rough estimate on the degree of device capacity utilization. Making sure the hardware is fully utilized by a kernel is an important step for getting optimal CUDA performance. However, when a kernel is not bandwidth-bound but compute-bound, a higher warp occupancy will have no influence on the kernel performance. Unlike as with the asynchronous memory transactions, switching to another thread is not feasible for a compute-bound kernel, as the arithmetic units cannot perform asynchronous computations.

Chapter 3

Concepts of Volume Rendering

This chapter recapitulates volumetric data representations and volume rendering, with a focus on the GPU-based techniques that we will use in the following chapters. A more elaborate discussion of many different aspects of volume rendering may be found, for example, in the book by Engel et al. (2006).

In computer graphics, three-dimensional scenes are traditionally modeled geometrically, using surface representations such as polygonal meshes. While rendering of such geometric data is directly supported by specialized graphics processors, a surface-based representation cannot display the interior structure of an object. Volume rendering uses 3D scalar data to model objects without relying on surfaces. A volumetric data set is typically stored as a three-dimensional array, where each element, called *voxel* (volumetric pixel), describes a property of the corresponding position in space.

The major source of volumetric data are tomographic scanners used in medical as well as technical applications. Computed tomography (CT) is the most prevalent acquisition technology, but there exists a multitude of other imaging modalities such as magnetic resonance imaging (MRI), positron emission tomography (PET), or 3D ultrasonography (US). While great technical effort is required to retrieve volumetric data from real-world objects, volume data can also be constructed synthetically. Simulations of natural phenomena, e. g., in meteorology, often use a discrete volumetric representation of space. This representation is used primarily for storing the current state of the simulation, but it can be directly used for visualization as well. Hence, simulations are another important area generating volumetric data.

Acquisition devices and simulations generate volumetric data of different resolutions and with different data types, typically arranged in a uniform grid. More complex structures, e. g., tetrahedral grids, are less common in practical applications and therefore we are focusing on uniform grids in this thesis. Scalar values are commonly stored as 16-bit unsigned integers, of which especially CT data often only uses 12 bits. For certain applications 8-bit integer data is used as well, and simulations often return 32-bit floating point data. Volume data where each voxel is associated with a vector value is handled by the separate research field of flow visualization. Some scanner types can acquire not only a single volume but entire series of volumes, where each volume corresponds to a time step. These time-varying data sets allow for analyzing temporal behavior in addition to spatial properties.

Regardless whether constructed by scanning physical objects or coming from a synthetic model, a volumetric data representation poses great demands on data processing as well as visualization because of its high memory requirements. At the same time, the data can exhibit a high level of complexity, which requires interactive visualization techniques to make interior structures and data correlations visible to the user. The resolution of volumetric data sets can easily reach 512^3 voxels or even more, and each of these voxels must be potentially visited to generate a correct rendering. Hence, efficiency is an important aspect of any system working with volume data, especially for visualization, where interactive performance with at least 10 frames per second (FPS) is desired.

In the following, we first discuss the theoretical background of volume graphics before describing common volume rendering approaches. Afterwards, we focus on the GPU-based volume raycasting technique and existing acceleration methods.

3.1 Theoretical Background

The most common optical model for visualizing volumetric data is an emission-absorption model (Max, 1995), which assumes the volume to consist of gas that can emit light and absorb incident light. Volume rendering is typically described by the *volume-rendering integral*, simulating light flow through a volume from a start point $s = s_0$ to the end point $s = D$:

$$I(D) = I_0 \, e^{-\int_0^D \kappa(t)dt} + \int_0^D q(s) \, e^{-\int_s^D \kappa(t)dt} \, ds.$$

Here, using the notation by Engel et al. (2006), $I(D)$ describes the radiance when leaving the volume at $s = D$, I_0 represents the background light at the position $s = s_0$, κ is the absorption coefficient, and q describes the emission. The absorption can be alternatively modeled using the transparency for the material between s_1 and s_2 defined by

$$T(s_1, s_2) = e^{-\tau(s_1,s_2)} = e^{-\int_{s_1}^{s_2} \kappa(t)dt}.$$

Typically the volume-rendering integral cannot be evaluated analytically, so a numerical approach is used, which approximates the integral by a Riemann sum over n equidistant segments $s_0 < s_1 < \cdots < s_n = D$. This gives the discretized volume-rendering integral:

$$\begin{aligned} I(D) &= \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \quad c_0 = I(s_0), \\ T_i &= T(s_{i-1}, s_i), \\ c_i &= \int_{s_i - 1}^{s_i} q(s)\, T(s, s_i)\, ds. \end{aligned}$$

Now only the transparency T_i and the color contribution c_i must be approximated for each segment:

$$T_i \approx e^{-\kappa(s_i)\Delta x}, \quad c_i \approx q(s_i)\Delta x, \quad \Delta x = (D - s_0)/n.$$

For implementation, an iterative computation of the discretized volume-rendering integral is used. In the common front-to-back compositing scheme, which describes traversing viewing rays from the viewpoint into the volume, the radiance at the current position C_{src} is added to the previous radiance C_{dst}, attenuated by the current opacity $(1 - \alpha_{\mathrm{dst}})$:

$$\begin{aligned} C_{\mathrm{dst}} &\leftarrow C_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) C_{\mathrm{src}}, \\ \alpha_{\mathrm{dst}} &\leftarrow \alpha_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) \alpha_{\mathrm{src}}. \end{aligned}$$

Different compositing schemes can also be useful. For example, in medical imaging a maximum intensity projection (MIP) is often applied:

$$C_{\mathrm{dst}} \leftarrow \max(C_{\mathrm{dst}}, C_{\mathrm{src}}).$$

3.2 Volume Rendering Approaches

A simple solution for visualizing volumetric data is to extract an isosurface, e. g., using the *Marching Cubes* algorithm (Lorensen and Cline, 1987), and to render the resulting mesh geometry. However, such an *indirect* volume rendering ignores the main advantage of volumetric over geometric representations: the ability to visualize the interior structure of the data. This requires a *direct volume rendering* (DVR) approach (Levoy, 1988), rendering the data without an intermediate geometric representation

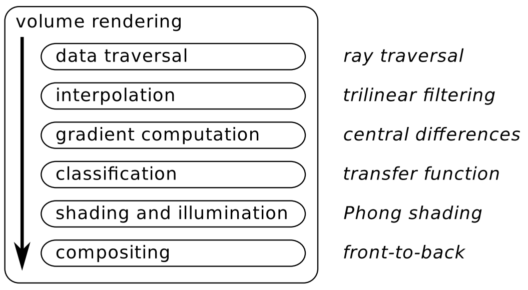


Figure 3.1: The generic volume rendering pipeline. For each stage of the pipeline a common implementation is listed on right.

and taking all voxels into account for visualization. It implements the generic volume rendering pipeline shown in Figure 3.1. The expressiveness of volume renderings can be increased significantly by enhancing DVR with transfer functions and illumination models. However, the tradeoff between image quality and rendering performance needs to be taken into account. We will discuss these topics in the remainder of this section.

3.2.1 Direct volume rendering

Volume rendering algorithms can be classified as *image-order* or *object-order* techniques. An image-order algorithm works on each pixel in the output image and analyzes the volume data to determine the resulting pixel color. In contrast, object-order techniques work on the voxels and determine the effect each voxel has on the resulting image.

A simple approach for object-order rendering is *splatting* (Westover, 1990), where the voxels are considered as particles and painted onto the screen with a size depending on their distance to the view plane. The technique has two mayor drawbacks, namely a suboptimal rendering quality and a low rendering performance, since the number of considered points is equal to the number of voxels, independent of whether they are visible or not. Instead of starting with individual voxels, *slice rendering* (Cullip and Neumann, 1994) uses 2D slices to sample the volume, which are projected onto the view plane and blended either in front-to-back or back-to-front order. This can exploit the texturing hardware and can therefore be implemented efficiently on the GPU. However, the technique suffers from visualization artifacts in the case of perspective projection and is inflexible when it comes to incorporating more complex visualization algorithms.

The most common image-based rendering approach in computer graphics is raytracing (Whitted, 1980), where rays are sent from each pixel of the image plane into

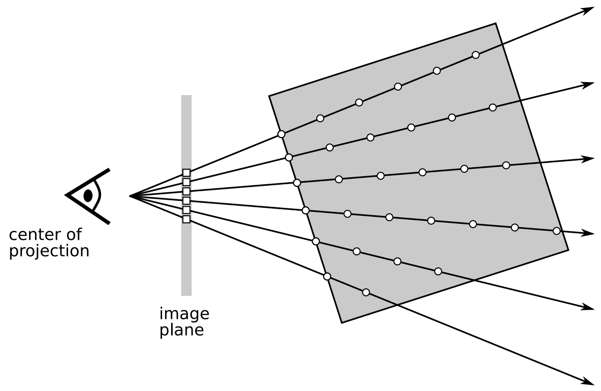


Figure 3.2: Principle of volume raycasting: Rays are cast from the center of projection through pixels in the image plane into the volume. The pixel colors are computed by sampling the volume at discrete positions on the associated rays and accumulating the results according to the volume-rendering integral

the scene and analyzed to determine the color values of the pixels. Volume raycasting, first described by Levoy (1990), uses the same approach for volumetric data, but without secondary rays and is therefore not capable of simulating effects such as shadows and reflections. As illustrated in Figure 3.2, the volume data is sampled at equidistant intervals on each ray using trilinear interpolation. The sampled intensity values are accumulated according to the front-to-back compositing scheme described in the previous section to get the resulting color for each pixel. This approach partly resembles the physical light transport, resulting in the best image quality of the common volume rendering techniques, according to an evaluation performed by Smelyanskiy et al. (2009). The straight-forward structure of the algorithm simplifies integrating extensions to basic raycasting. But unlike the texture slicing approach, raycasting is not directly supported by graphics hardware, and the algorithm is the computationally most expensive of the discussed volume rendering approaches.

3.2.2 Transfer functions

The scalar values stored in a volume can relate to different physical or simulated measurements, depending on the acquisition technology: radiodensity measured in the Hounsfield scale for CT data, emission of a radionuclide with PET, or temperature distribution in a fluid dynamics simulation. To map the various voxel intensities to visually sensible opacity and color values, a transfer function is typically applied. It can either be specified analytically, e. g., as a simple ramp function, or using a discrete lookup table (LUT). For GPU-based volume rendering, usually the lookup table is stored in a 1D texture, and linear filtering is applied for smooth transition between its entries. Multidimensional transfer functions take additional data into account. For example, they combine voxel intensities and gradient magnitude, for which the lookup table can be stored in a 2D texture.

3.2.3 Illumination models

Local illumination models such as Phong lighting (Phong, 1975) are often applied when rendering geometric data to increase realism and spatial comprehension. The same would be useful for direct volume rendering. However, Phong lighting and most other local illumination models require a normal vector for each illuminated point, which is not directly available for volumetric data, as they contain no surface information. Fortunately, a gradient can be computed for each voxel to estimate a normal vector that is sufficient for lighting. This computation typically uses forward of central differences, increasing the cost of rendering, as three or six additional neighbor voxels need to be sampled.

Global illumination takes light interaction between objects into account to realize effects such as reflection and caustics. As each object can potentially influence every other object in the scene, global illumination techniques such as radiosity (Goral et al., 1984) are computationally expensive even for geometrically defined scenes. With volumetric data each voxel can be assigned different optical properties while still influencing all other voxels. Hence, interactive global illumination for volume data is limited to simple effects such as direct reflection or shadows (Ropinski et al., 2010), or approximative approaches (Ropinski et al., 2008).

3.2.4 Performance and image quality

The performance of a volume rendering technique is typically inversely-proportional to the resulting image quality. For example, increasing the resolution of the output image leads to more rays being traversed when using raycasting. Another configurable parameter is the sampling rate, which controls the step size between sampling points on a ray. As a corollary of the Shannon-Nyquist sampling theorem (Shannon, 1949), each voxel would have to be sampled twice to be correctly reconstructed by the volume rendering. However, this only holds as long as there are no high frequencies in the transfer function, which can require to further increase the sampling rate for visually correct results. Both increasing the viewport size and increasing the sampling rate increases the overall number of volume sampling operations in the relatively slow texture memory, and therefore raises the costs of rendering. Hence, choosing these values is always a tradeoff between rendering performance and image quality.

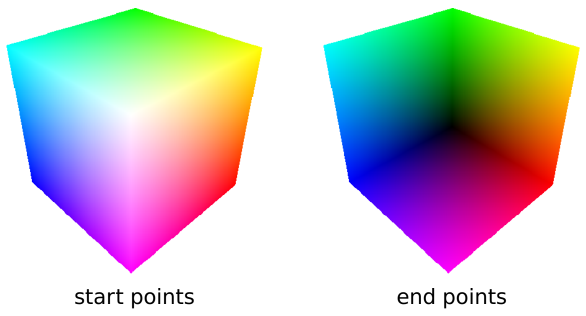


Figure 3.3: Ray parameter textures generated by rendering a cube proxy geometry, for use by GPU-based raycasting. Each pixel encodes the ray start or end point of the respective ray.

3.3 GPU-based Raycasting

GPU-based raycasting was first described by Röttger et al. (2003) and extended by Krüger and Westermann (2003). With modern programmable graphics hardware it allows to run the entire raycasting process in a single pass of a fragment shader. The volume is stored in a 3D texture through which the shader casts a ray that is traversed in a single loop. The 3D texture is sampled in equidistant intervals, making use of the trilinear filtering supported by the texturing hardware. If necessary, a transfer function and a local illumination model can be applied to the sampled intensity values. The color value accumulated during raycasting is returned as the resulting fragment color of the shader.

The Krüger-Westermann approach does not compute the *ray parameters*, i. e., the ray start and end points, analytically but utilizes the geometry pipeline to generate them. It uses a proxy geometry of the volume—typically the data set's bounding box—to generate two textures containing the start and end points for each ray. The proxy geometry is rendered using a fragment shader that maps the (x, y, z) position of each fragment to (r, g, b) color components, resulting in the ray start point texture. Rendering the back faces instead of the front faces provides the ray end point textures. The two resulting textures (Figure 3.3) are made available to the raycasting shader, which only has to perform a lookup in both textures to determine the ray parameters for each ray. The ray parameter textures are sometimes also called *entry-exit point* (EEP) textures, referring to the points where the rays enter and exit the volume.

Although initially seeming more complex than direct calculation of the intersection points between a ray and the volume bounding box in the shader, the Krüger-Westermann approach has two major advantages over an analytical solution: First, it is independent of the view and projection transformations, meaning that the shader needs no information about camera parameters. The projection can even be switched between perspective and orthogonal without any modification of the shader, as the transformation is already applied when rendering the proxy geometry with the current viewing parameters. Second, modifying the proxy geometry can be a simple but powerful tool for optimization as well as for realizing complex visualizations. For example, we have implemented interactive deformation of volumes by just deforming the proxy geometry (Mensmann et al., 2008b). In Chapter 5 we will describe how an optimized proxy geometry can be used for accelerating volume raycasting.

3.4 Accelerating Volume Raycasting

Even when implemented on the GPU, volume raycasting leaves room for optimization to achieve interactive performance even with high viewport resolutions, large data sets, or expensive visualization techniques. We will shortly describe some of the most common optimizations in this section. The main cost factor for raycasting on the GPU is sampling in the volume texture. Even though the bandwidth for accessing graphics memory from the GPU is significantly higher than the memory bandwidth of standard CPUs, it is still orders of magnitude slower than on-chip memory. In addition, there is a memory latency introduced with each texture access, and the GPU possibly has to wait for the results of the memory transaction to become available before performing further operations. Therefore, optimizations that reduce the number of sampling operations promise to increase the overall rendering performance. Two common techniques implementing this approach are *empty space skipping* and *early ray termination*. Additional effort is required for large volume data sets that do not fit into GPU memory. This problem can be addressed by *bricking* techniques.

3.4.1 Empty space skipping

Many volume data sets contain a considerable amount of empty space, i. e., voxels that are fully transparent and do not contribute to the final image. The most obvious example is air around an object in a CT volume, but any voxels that are assigned zero opacity by the transfer function can be interpreted as empty space. When the distribution of empty space in a volume is known beforehand, sampling operations in empty space can be avoided by simply skipping over empty regions during ray traversal. For detecting these empty regions, spatial data structures such as octrees should be used. Applying this optimization can have a tremendous effect on rendering performance, depending on both the data set and the applied transfer function. We will discuss existing approaches for empty space skipping and introduce a novel technique that incorporates temporal coherence in Chapter 5.

3.4.2 Early ray termination

In the basic raycasting approach rays are always traversed completely from start to end point. This is necessary for compositing techniques like maximum intensity projection, as even the last voxel sampled on the ray may contain the maximum intensity value. However, for direct volume rendering the opacity is accumulated monotonically, meaning that after the accumulated opacity has reached a value of 1.0, it will not be changed by further sampling operations on the ray. In this case the ray traversal can be terminated, as all the following voxels on the ray will have no influence on the result. For practical applications it can be useful to choose a threshold slightly below 1.0, because when the accumulated value is close to full opacity, further changes will not be visually perceivable. Early ray termination is most effective for dense data sets, where rays can terminate shortly after hitting the opaque object. However, when the transfer function is configured to show the object as semi-transparent, the accumulated opacity might never reach the threshold and consequently rays will never be terminated before reaching their end point.

3.4.3 Bricking and multi-resolution approaches

Basic volume raycasting requires the entire volume texture to be available in GPU memory to fully exploit the high graphics memory bandwidth. For larger volumes, commonly a divide-and-conquer approach that subdivides the volume into several sub-blocks or *bricks* is used. Each of these bricks is chosen small enough to fit into graphics memory. The bricks are uploaded to the GPU one after another and volume rendering is applied to each brick individually. Hence, this *out-of-core* technique can render the volume without having the entire data available in graphics memory at the same time. However, it requires an additional step to combine the renderings of the individual bricks to form the rendering of the complete volume. Bricking can be implemented with GPU-based raycasting by subdividing the standard cube proxy geometry into bricks and applying the raycasting fragment shader to each brick, after sorting the bricks by their distance to the camera. As managing of bricks and especially the compositing of intermediate results add some overhead compared to basic raycasting, the technique exhibits performance disadvantages when the volume fits into memory, and should not be used in this case.

Bricking can also be used to implement empty space skipping simply by ignoring bricks that only contain fully transparent voxels. The bricking approach is also suitable for multi-resolution rendering, which changes the level-of-detail (LOD) of each brick, i. e., its resolution, based on some weighting function such as the distance to the camera. While the basic approach is simple, a significant effort is required to prevent filtering artifacts from appearing between bricks having different resolutions. However, such a multi-resolution rendering can also be implemented without bricking (Ljung et al., 2006).

Chapter 4

Voreen – The Volume Rendering Engine

The Voreen framework provides the basis for implementing the rendering techniques presented in the following chapters. Therefore, we review the fundamental concepts and features of the framework, as well as possibilities for extending the system by integrating additional functionality.

Visualizations of volumetric data are often created by combining multiple rendering and image processing techniques. The complexity of volumetric data requires that users are able to interactively change rendering parameters such as camera position or transfer function, but potentially also to modify more complex settings, such as replacing entire rendering modules or adding additional image filters. Traditional programming of visualization techniques is not suitable for supporting such a dynamic workflow; hence, a more interactive approach that supports rapid-prototyping of visualizations is necessary.

The idea of the Voreen (*Vo*lume *Re*ndering *En*gine) project (Meyer-Spradow, 2009; Meyer-Spradow et al., 2009) is to support the development of such complex volume visualizations by using a visual programming paradigm and thus achieve both a high level of flexibility and rendering performance. The component-based architecture makes the system highly flexible and extensible to support easy integration of new rendering techniques or data processing algorithms. Therefore, we realized all the acceleration techniques for volume rendering presented in the following chapters using Voreen.

4.1 Data-flow Concept

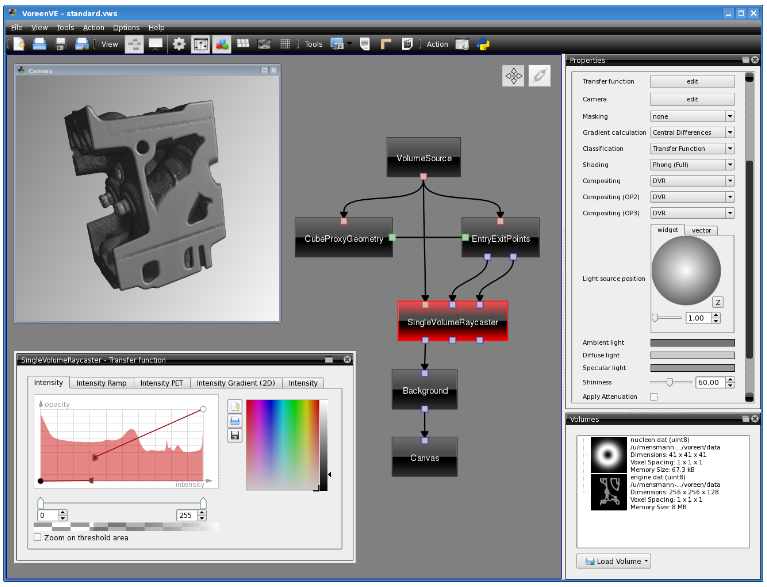
Visualizations in Voreen are constructed based on the concept of *data-flow networks*. Such a network consists of *processors*, which are autonomous functional building blocks that perform a specific task. The processors can work on different types of input data such a 2D images, 3D volumes, or mesh data, and can also output such data. Processors can be connected through their ports: *inports* for data input and *outports* for output. From each connected processor outport the data flows to the corresponding inport. In addition to in- and outports, processors can have *coprocessor ports*. A coprocessor in VOREEN is a processor that supports certain functionality and may be called by a connected processor, similar to a method call in object-oriented programming. This can be useful for data that is not suitable for transfer by the data-flow concept, e. g., because it is stored in an implementation-specific format and is too large to be converted on demand.

The generic but flexible data-flow concept supports implementing complex visualizations by combining multiple processors. Examples for processors include a volume processor that downsamples an input volume, an image processor that applies a Gaussian blur to an input image, or a more complex rendering processor that visualizes an input volume using raycasting. It is notable that the data-flow concept does not limit the performance of techniques implemented using this scheme. In practice, the approach only adds a small overhead compared to a direct implementation.

Since information about the entire network structure is available to a central *network evaluator* before rendering is started, the evaluator can apply some optimizations that would not be possible with local knowledge only. For example, network branches not connected to an output window can be ignored. In complex networks the order of evaluation can be modified to minimize the resource requirements, e. g., the number of OpenGL textures used for storing intermediate rendering results. Hence, the logical flow of information in the data-flow network can differ from the actual order of execution. A caching mechanism in the evaluator determines whether results of sub-networks can be reused instead of executing the processors again, which is important especially for interactive applications. The concept of a global evaluator also simplifies the implementation of processors, as they need no information about a network's topology, but receive all data through the connected ports.

4.2 User Interface

Although data-flow networks in VOREEN can also be created and configured by method calls in program code, a user will typically start the VOREENVE (visualization environment) application (shown in Figure 4.1) and use its graphical network editor to create rendering networks through visual programming. The user can influence the visual result of a rendering network by adding processors or modifying connections between existing ones, i. e., changing the network topology. Individual processors can



Transfer function: edit
Camera: edit
Masking: none
Gradient calculation: Central Differences
Classification: Transfer Function
Shading: Phong (Full)
Compositing: DVR
Compositing (OP2): DVR
Compositing (OP3): DVR
Light source position: 1,00
Ambient light
Diffuse light
Specular light
Shininess: 60,00
Apply Attenuation

nucleon.dat (uint8)
/u/mensmann-.../voreen/data
Dimensions: 41 x 41 x 41
Voxel Spacing: 1 x 1 x 1
Memory Size: 67.3 kB
engine.dat (uint8)
/u/mensmann-.../voreen/data
Dimensions: 256 x 256 x 128
Voxel Spacing: 1 x 1 x 1
Memory Size: 8 MB

Figure 4.1: Screenshot of the VoreenVE application for visual programming of data-flow networks. In addition to the central network editor, the windows for the rendering result and for modifying the transfer function are visible on the left. The properties of the selected processor are listed on the right.

be modified through their *properties*. Processor properties are represented by class member variables and can be accessed via an automatically constructed GUI. They allow to model basic processor settings such as raycasting sampling rate, but also complex properties such as transfer functions are supported.

4.3 Integration of GPU-based Raycasting

Because the focus of VOREEN lies on volume visualization, the data-flow implementation of GPU-based raycasting is of major importance. Figure 4.2 shows a data-flow network implementing volume raycasting. In this network, the volume data of the current data set is made available by a `VolumeSource` processor and sent to the three connected processors. The actual ray traversal is implemented in `SingleVolumeRaycaster`. Since we use the Krüger-Westermann approach, the processor also needs two ray parameter textures in addition to the volume data. They are delivered by `CubeProxyGeometry` and `EntryExitPoints`, which work together for this task. `CubeProxyGeometry` represents the bounding box geometry of the connected volume, and `EntryExitPoints` triggers rendering of this proxy geometry by calling a

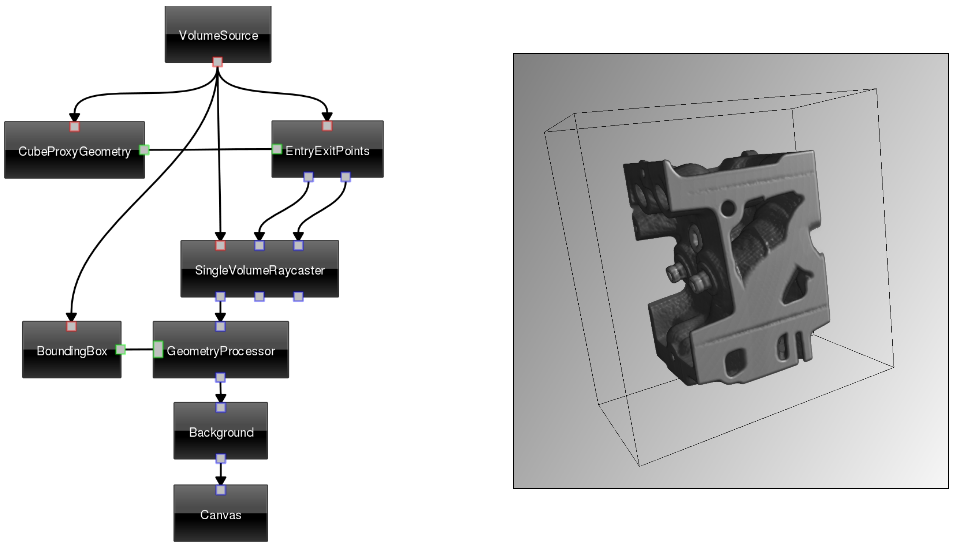


Figure 4.2: An example data-flow network implementing GPU-based raycasting in Voreen and the resulting output image.

method through the coprocessor connection between the two processors. Afterwards, `SingleVolumeRaycaster` binds the resulting ray parameters as OpenGL textures and executes a fragment shader to perform the actual ray traversal through the volume. A `GeometryProcessor` adds a wireframe of the data set bounding box to the raycasting result by calling `BoundingBox` as a coprocessor. The `Background` processor combines the resulting image with a color gradient background and directs the result to the final `Canvas` processor for display in the output window.

Other visualization systems typically encapsulate the entire volume rendering in one large monolithic block (compare Meyer-Spradow et al., 2009, p. 7). However, splitting up raycasting into several processors, as performed in Voreen, is more flexible. For example, the basic `CubeProxyGeometry` can be exchanged easily by a more complex proxy geometry, a feature we will use in the following chapter.

4.4 Adding New Components

A new rendering technique can be integrated into Voreen by implementing new processors. A processor is implemented by a class inheriting from `Processor` or one of its more specialized subclasses. The actual functionality of a render processor is located in its `process()` method. It can access the data sent to the processor's inports and renders it, making the output available through one or more outports.

The functionality of the `process()` method can of course differ entirely between processors used for rendering and data processing, but the overall structure is the same in either case. There is no limitation on the kind of operations a processor can perform in `process()`, e. g., all of OpenGL can be used directly.

4.5 Performance Considerations

Despite the high level of abstraction introduced by the visual programming approach in Voreen, the abstraction does not cause a performance penalty for rendering. Overhead in the data-flow architecture comes from two aspects: communication between processors and managing the execution of processors. Most interaction between processors is limited to transfer of volume and image data via ports. As this data is typically located on the GPU as a 3D or 2D texture, it does not have to be moved, but only references to the data need to be passed to processors. Evaluating a render network introduces some overhead, for example, when a generic processor spends resources on initializing a feature that is not actually utilized in a specific network configuration. The total amount of this overhead depends on the number of processors in the network, which is small for most applications. There is no noticeable overhead in practice, even for data-flow networks consisting of a hundred processors. The reason for this low overhead comes from the large difference between the computational costs for the management overhead and the costs for expensive visualization techniques such as volume raycasting. In a network containing a raycasting processor, the overwhelming part of the total runtime will be spent in the fragment shader performing the ray traversal. In a network implementing GPU-based volume raycasting, less than 5% of the total runtime is spent on rendering the proxy geometry and further initialization, while according to our measurements more than 95% of the time is spent in the `process()` method of the raycasting processor waiting for the GPU to finish running the fragment shader.

By designing the architecture to make use of OpenGL resources directly and minimizing initialization overhead and communication between processors, visualizations developed with Voreen achieve performance results comparable to a direct, but more laborious, manual implementation. Therefore Voreen is also suitable for applications requiring interactive frame rates and even for implementing and testing time-critical rendering algorithms. The performance aspects of volume rendering with Voreen were also verified by other groups: Eisenmann et al. (2009) evaluated volume rendering for pre-operative planning in neurosurgery. Of the three volume rendering libraries tested (VTK, Voreen, and VGL), only Voreen was able to achieve interactive frame rates and high quality output.

Chapter 5

Empty Space Skipping using Occlusion Frustums

In this chapter we introduce a novel approach to empty space skipping in order to reduce the number of costly volume texture fetches during ray traversal. We generate an optimized proxy geometry for raycasting, which is based on occlusion frustums obtained from previous frames. The technique does not rely on any preprocessing, introduces no image artifacts, and—in contrast to previous point-based methods—works also for non-continuous view changes. Besides the technical realization and the performance results, we also discuss the potential problems of ray coherence in relation to our approach and restrictions in current GPU architectures.

Many volumetric data sets contain a considerable amount of empty space, i. e., voxels that do not contribute to the final image. What makes up an "empty" voxel depends on the transfer function, but certain intensity ranges, such as air around an object in a CT scan, are mapped to zero opacity in most applications. Ignoring voxels that do not contribute to the final image is an obvious way to increase rendering performance, but it is important that this optimization introduces no serious overhead. Therefore we have developed an empty space skipping technique that is realized by exploiting the geometry processing capabilities of programmable graphics hardware, which are normally not utilized during raycasting. The vertex and fragment processing units of earlier graphics hardware were independent, resulting in the vertex units being mostly idle when raycasting was implemented within a fragment shader. More recent GPUs switched to a unified architecture where processing units are dynamically assigned to process either vertices or fragments, and thus ensure better utilization of the available computing resources. Nonetheless, raycasting still does not make use of the geometry processing capabilities of the hardware. Therefore we have investigated how a more complex proxy geometry can be used for supporting the fragment processing unit, to save costly computations and reduce the number of memory transactions.

Data exploration is an important application for volume rendering. For this task it is especially important for the user to be able to change all rendering parameters interactively. Changes in some parameters like the transfer function can have global effects on the empty space, depending on which source data values are mapped to zero opacity. These parameter changes can therefore invalidate any prior knowledge about the distribution of empty space and nullify any data structures which rely on this information. Hence, optimization techniques that rely on empty space information and require expensive preprocessing to adapt the underlying data structures to changes in the empty space are unsuitable for interactive data exploration. We introduce *occlusion frustums* to improve the rendering performance of GPU-based raycasting, which require no preprocessing as they are constantly regenerated. Our approach relies on the observation that for many applications only small viewpoint changes are applied between consecutive frames. Using occlusion frustums for empty space skipping introduces no rendering artifacts and also works for non-continuous viewpoint changes.

This chapter is structured as follows. Section 5.1 examines related work regarding volume raycasting optimizations. In Section 5.2 we discuss the potential effects of hardware restrictions on general raycasting optimizations. Our occlusion frustum approach is introduced in Section 5.3, and its implementation on the GPU and integration into Voreen are presented in Sections 5.4 and 5.5. We present and discuss results in Section 5.6, and give a summary in Section 5.7 .

5.1 Related Work

GPU-based raycasting as introduced by Röttger et al. (2003) and enhanced by Krüger and Westermann (2003) uses a proxy geometry most often resembling the data set bounding box to specify ray parameters, as shown in Figure 5.1. Many acceleration techniques have been proposed for raycasting, often trying to reduce the large number of sampling operations in the volumetric data. Avila et al. (1992) introduced polygon assisted raycasting (PARC), which approximates the volume object by a polygon mesh and restricts raycasting to those parts of the rays lying inside the geometry. This was implemented by Leung et al. (2006) using the *Marching Cubes* algorithm (Lorensen and Cline, 1987) for extracting the object surface. Similarly, Westermann and Sevenich (2001) utilized hardware-based texture slicing to speed up software-based raycasting.

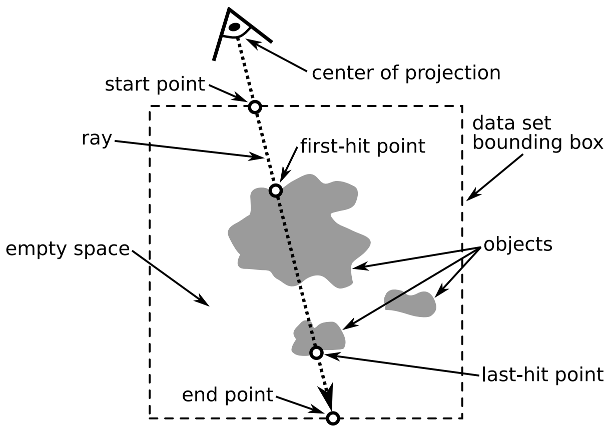


Figure 5.1: Casting a ray through a volume data set.

They render the data set with a fast but low-quality slice-based approach and use the resulting depth image to get an optimal ray setup for the high-quality raycasting performed in a second rendering pass. Another approach using distance transforms was introduced by Šrámek and Kaufman (2000).

A faster but also coarser approximation of the volume object can be generated by partitioning the volume into uniform blocks and not rendering those consisting only of empty voxels. Many authors have implemented this *bricking* approach, for example, Hadwiger et al. (2005) and Scharsach et al. (2006). Li et al. (2003) used adaptively partitioned subvolumes to add empty space skipping to slice rendering, grouping similar voxels into subvolumes. These object-order techniques can adapt to limited changes in the transfer function by storing minimum and maximum voxel intensities for each block. But for larger changes in the transfer function these methods generally require to rebuild the data structure, for which every voxel in the data set has to be considered. This makes these techniques rather unsuitable for interactive data exploration where the opacity mapping is changed frequently. Also, it is often difficult—if not impossible—to adapt the more complex approaches efficiently to the GPU programming paradigm and to integrate them into existing rendering frameworks.

Temporal coherence is often exploited for geometry-based rendering techniques. For example, Havran et al. (2003) reuse ray/object intersections computed in the last frame for acceleration of raycasting in the current frame. The idea of skipping empty voxels around a volume object by exploiting temporal and spatial coherence between consecutive frames was introduced by Gudmundsson and Randén (1990) for parallel projection and later generalized by Yagel and Shi (1993) under the name *space leaping*. They approximate optimal ray start points by extracting first-hit points from the depth images of previous frames and *reprojecting* them to the current view using point splatting. Due to discretization of screen-space positions to integer pixel locations, some pixels will not be covered by the reprojection and therefore a *hole filling* is required, triggering a full raycasting for such pixels. The reprojection is only possible for small view changes; for larger changes a full raycasting of virtually all pixels is necessary. The approach is illustrated in Figure 5.2.

Several extensions of the reprojection approach have been presented. For example, Yoon et al. (1997) transformed rays instead of points to accelerate isosurface rendering. Wan et al. (2002) presented a cell-based reprojection scheme, which they combined with a spatial data structure based on distance fields for hole filling. Besides high memory requirements for the data structure, their technique will fail to detect objects becoming visible when large viewpoint changes are performed. Instead of frame-to-frame coherence, Lakare and Kaufman (2004) exploit coherence between rays by casting *detector rays* to get empty space information for multiple adjacent rays at the same time. While independent of the transfer function, their technique can only give accurate results when single voxels are projected over multiple pixels on the screen, as it is the case in virtual endoscopy applications where the camera is placed close to the object surface.

All of the previously described space leaping methods that make use of temporal and spatial coherence were implemented on the CPU. Many of them cannot be ported directly to the GPU and are therefore not useful in a GPU-based raycasting system. Only few solutions were presented that directly use graphics hardware for acceleration. The approach by Westermann and Sevenich (2001) utilizes hardware-based texture slicing to speed up software-based raycasting. Klein et al. (2005) implemented Yagel and Shi's space leaping with point reprojection on programmable graphics hardware using vertex shaders. For larger view changes they described artifacts as "unavoidable" with their technique. The reprojection technique was also applied in combination with a method to prevent unnecessary raycasting when using time-varying data by Grau and Tost (2007).

5.2 Impact of Hardware Restrictions on GPU-based Raycasting

Although the highly parallel architecture of modern graphics processors makes volume raycasting usable for interactive applications, it must be kept in mind that this architecture has some restrictions. Detailed information about these restrictions with

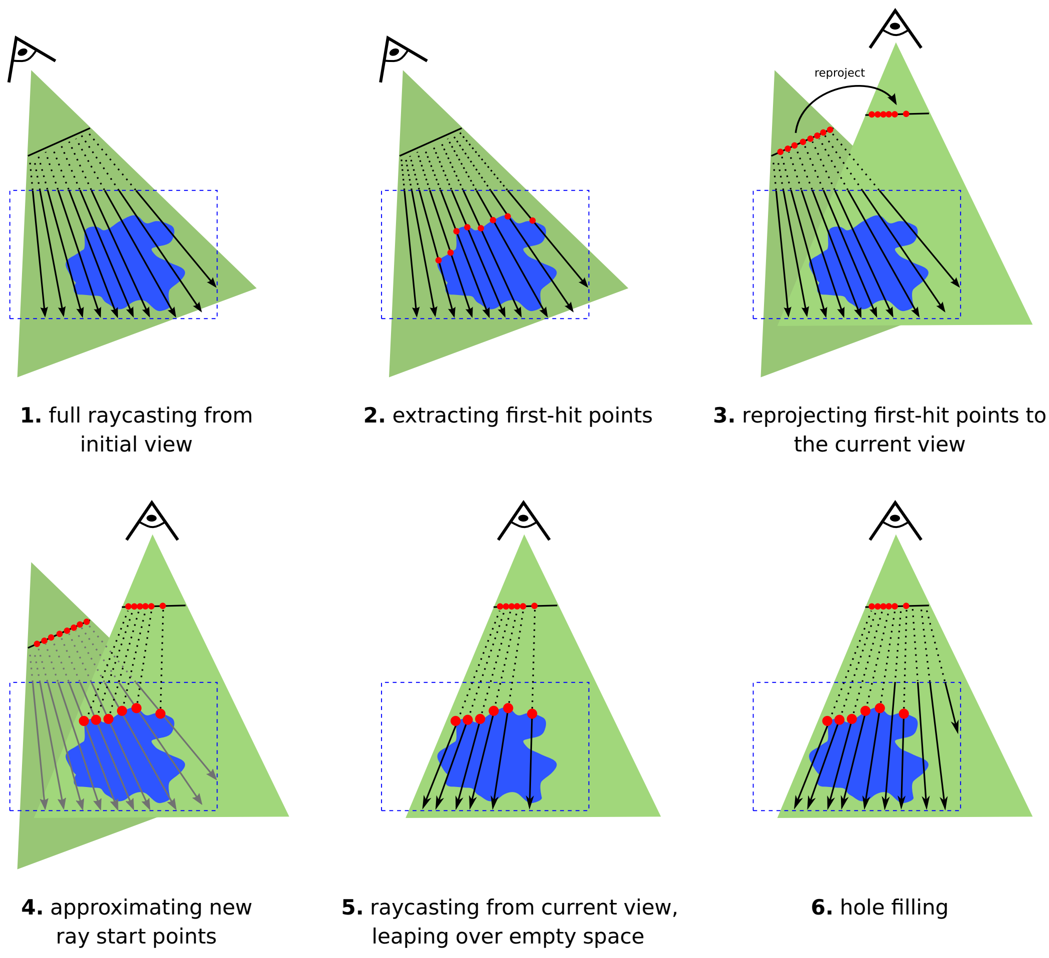
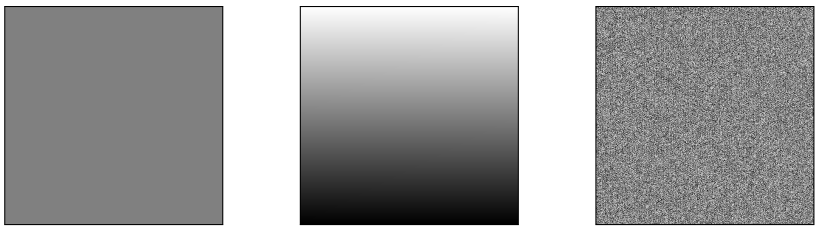

reproject
1. full raycasting from initial view
2. extracting first-hit points
3. reprojecting first-hit points to the current view
4. approximating new ray start points
5. raycasting from current view, leaping over empty space
6. hole filling

Figure 5.2: Illustration of the space leaping approach by Yagel and Shi (1993). Optimal ray start points for the current view are approximated by reprojecting first-hit points of the previous view. As the reprojection is not lossless, a final hole filling step is necessary.



(a) constant **(b)** linear **(c)** random

Figure 5.3: Grayscale textures for controlling the distribution of ray lengths on the screen for measuring its influence on raycasting performance. Each pixel corresponds to a ray and the ray length is proportional to the pixel luminance. The sum of all ray lengths is the same in either case.

distribution	fps	overhead
constant	251.3	—
linear	249.2	0.8%
random	149.7	67.9%

Table 5.1: Influence of ray length distribution on casting 512^2 rays with an average ray length of 128 samples. The difference between a constant and a linear distribution is insignificant, but a random distribution of ray lengths introduces a rendering overhead of about 68%.

regard to shader programming is not available from vendors, hence it is necessary to gather data experimentally, e. g., with the GPUBench tool (Buck et al., 2004a; Houston, 2007).* One hardware restriction significant for raycasting is *branch coherence*: Fragment processors on modern GPUs process fragments in groups rather than individually, and the fragment with the most time-consuming calculations limits the progress of the entire group, as the group can only finish when all of its fragments are completed. Houston (2007) demonstrated this for different GPUs by distributing pixels that cause "slow" or "fast" calculations either randomly or in groups on the screen. Coherence regions with the same calculation time for 4×4 to 16×16 pixels gave nearly optimal results, depending on the graphics hardware, while a random distribution resulted in the worst performance.

* With the introduction of new programming interfaces such as NVIDIA's CUDA, vendors have become more open about the inner workings of their GPUs, making more hardware documentation available than when graphics processors could only be used through shader programming. However, the exact mapping of shaders to the hardware resources is still undocumented and often experimental testing is required to get optimal performance results.

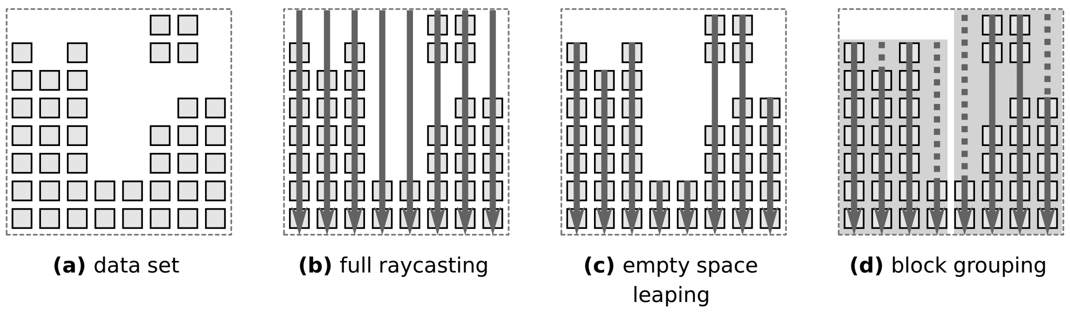


Figure 5.4: Effect of block grouping on raycasting and empty space leaping. Starting at the optimal ray start points (c) instead of the bounding box (b) prevents most sampling operations in empty space. Block grouping (with a block size of 4) can potentially undo much of these savings (d).

For GPU-based raycasting, computation time for each fragment is mainly influenced by the ray length, as this controls the number of texture fetches. Empty space leaping and early ray termination can disturb coherence of computation times for adjacent rays, as they modify the initially equal ray lengths depending on the volume data. To examine this matter, we have implemented a simplified GPU raycaster that can set the lengths of the rays to be either constant, linearly increasing, or randomly distributed over the screen. It uses the textures in Figure 5.3 to control the distribution of ray lengths. The sum of all ray lengths (and therefore the total number of texture fetches) is the same in either case. As shown in Table 5.1, there is no significant performance difference between constant and linear increasing lengths, but a random distribution of lengths increases rendering time by about 68%.

However, experimental results suggest that ray length distribution is not that random for non-synthetic data. This can be quantified by comparing the original unmodified volume raycasting to one where we group adjacent rays in blocks, and apply the longest ray length within each block to all rays in the block. Hence, we simulate the behavior of a fragment processor, which also processes fragments in this fashion. As illustrated in Figure 5.4, block grouping could undo the effect of empty space leaping when ray lengths are distributed randomly. Implementing the block grouping scheme in a full volume raycaster, we do not directly consider the lengths of the rays but instead count the number of texture fetches, which also covers gradient calculation with six additional texture fetches for each non-empty voxel. Since the computation time for a block is determined by its longest ray, the shorter rays in the block result in idling fragment units. We implemented the ray length analysis into Voreen by extending the standard raycaster to count the number of

data set	block size		
	4^2	8^2	16^2
head	+2.3%	+4.9%	+10.2%
engine	+4.7%	+10.3%	+19.9%
aneurysm	+3.4%	+7.1%	+13.8%

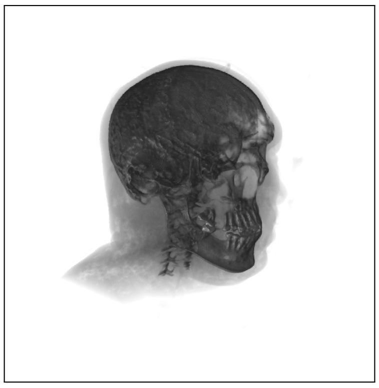
Table 5.2: Overhead when all rays in each block have to wait for the longest ray inside the block to finish. The overhead is lower than what would be expected for a purely random distribution of ray lengths.

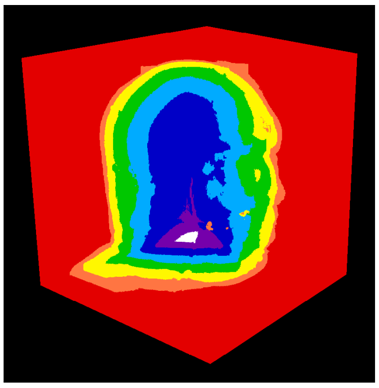
texture fetches it performs and outputting the results to an additional render target. Table 5.2 shows the overhead introduced by the block grouping, i. e., the total idle time, for semi-transparent, dense, and sparse data sets. The overhead ranges from 2% to 20% compared to when no block grouping is applied. This is much less than what would be expected for a purely random distribution. Visualizing ray lengths as in Figure 5.5 shows that they are distributed quite uniformly, so we expect the penalty to pay for a block having to wait for the calculation of its longest ray to be comparatively small.

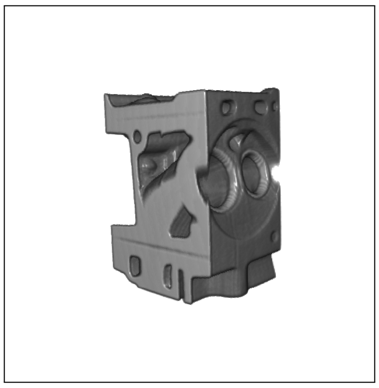
Thus we could show that although in theory branch coherence could be a potential problem for raycasting optimizations, the effects are most visible with synthetic worst-case data, while with real-world data the resulting ray lengths are much more coherent, which will not be changed dramatically by optimizations of the proxy geometry.

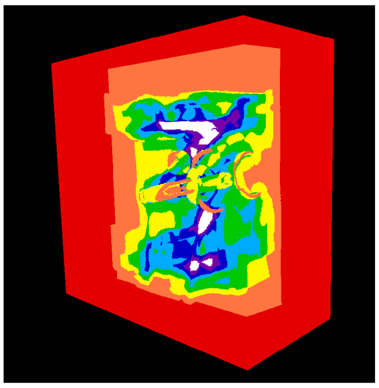
5.3 Optimizing the Proxy Geometry for Space Leaping

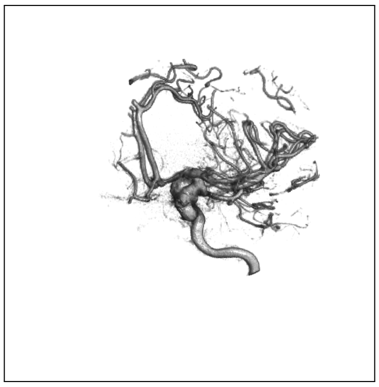
GPU-based raycasting usually utilizes a cube as its proxy geometry for generating the ray start and end points, illustrated in Figure 5.1. As the colors on the cube surface encode the exact position in space (compare Figure 5.6a), any other geometry can be used instead, as long as it encloses all relevant voxels. A straight-forward approach to minimize the sampling of empty voxels would therefore be to enclose all non-empty voxels inside a closely-fitted proxy geometry, as proposed with the PARC algorithm by Avila et al. (1992). The complexity of this geometry is data-dependent, and some kind of simplification would be needed to prevent the generation of excessively complex geometries. While giving optimal results with regard to preventing unnecessary sampling, generating such a geometry would be quite costly and, even worse, it would become invalid as soon as the transfer function is changed (Figure 5.7), a common operation for data exploration. Hence, an on-the-fly process with fast

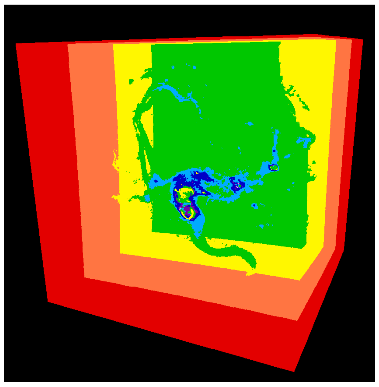

(a) vmhead


(b)


(c) engine


(d)


(e) aneurysm


(f)

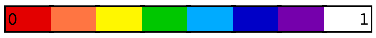


Figure 5.5: Analyzing the distribution of ray calculation costs in a semi-transparent, dense, and sparse data set. The number of texture fetches per ray (including gradient calculations) is normalized and quantized for each data set.

adaptation to changed viewing parameters would be preferred, even when giving slightly less optimal results. Therefore we refrain from any preprocessing and choose to do a less costly proxy geometry construction for each frame, using information obtained during the rendering of the previous frame.

5.3.1 Occlusion frustums as proxy geometry

The concept of occlusion volumes is well-known for geometric visibility calculation and occlusion culling, e. g., see Schaufler et al. (2000). Objects directly visible from the camera are considered as occluders that cast an *occlusion volume* into the scene, similar to casting a shadow. All objects inside this occlusion volume are known to be invisible from the camera and can be removed during occlusion culling. Our approach is based on visibility information retrieved from a *first-hit image* (Figure 5.6b). In this image, each pixel corresponds to a ray, and the pixel color specifies the position of the first non-empty voxel on this ray. The alpha channel is used to mark rays that only hit empty voxels. When we consider the first-hit voxels as occluders, we can extract regions where no non-empty voxels are located. These are marked as *definite miss* in Figure 5.6c. When using the naming scheme by Yoon et al. (1997), the different types of regions can be classified as follows:

Definite hit. Regions where non-empty voxels are definitely located. This is only known for the first-hit voxels.

Definite miss. Regions where only empty voxels are located. This includes all empty voxels lying between the start and the first-hit point of a ray, or between the start and the end point when the ray hits no non-empty voxels.

Possible hit. Nothing is known for all other voxels, namely those lying behind first-hit point voxels. This can be interpreted as the first-hit voxels casting a shadow or occlusion volume, with no information available about the voxels occluded by this shadow.

To prevent the raycaster from sampling in regions that are known to contain only empty voxels, we can build a proxy geometry consisting of the first-hit voxels and their occlusion volume. Not an exact geometry is necessary, but a simplification is sufficient as long as it contains all possibly non-empty voxels. For each first-hit voxel a quadrilateral pyramidal *occlusion frustum* is constructed, with its top base located at the voxel position. The occlusion frustums are constructed by extruding the quad forming the top base along the projectors extending from the center of projection through the vertices of the quad (illustrated in Figure 5.8). The depth extent of the

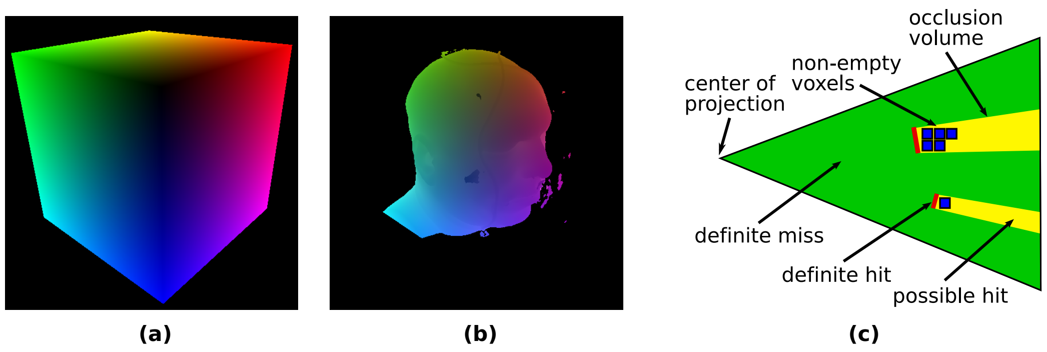


Figure 5.6: (a) Proxy geometry and (b) first-hit point image for the *vmhead* data set. (c) Visibility information retrieved from a first-hit image can be used for constructing the occlusion frustums.

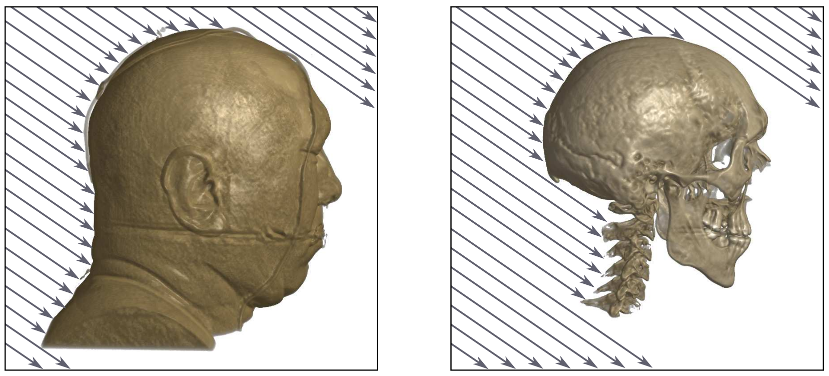


Figure 5.7: Changing the transfer function invalidates empty space information.

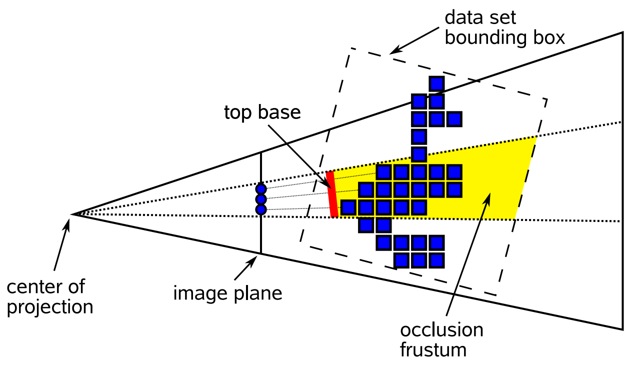


Figure 5.8: Construction of a two-dimensional occlusion frustum with a block size of three. The voxel closest to the center of projection determines the position of the occlusion frustum's top base

frustum is chosen large enough so that its bottom base lies outside the data set bounding box. The union of all these constructed occlusion frustums is the complete occlusion volume. Unlike point-based space leaping, no reprojection is necessary to adapt the generated proxy geometry to changed viewing parameters. As the geometry is created in object space, it is sufficient to apply the desired view transformation and simply render the geometry from the new viewpoint (see Figure 5.9).

Special consideration is necessary for objects that are initially located outside the view frustum. As the generation of occlusion frustums is image-based, they can only contain voxels lying inside the view frustum. Consequently, when the view is changed so that previously hidden objects become visible, they will be skipped by raycasting as they do not lie within the proxy geometry that consists of all occlusion frustums (see Figure 5.10). To handle this case, the proxy geometry must be enlarged to enclose all regions outside the view frustum where non-empty voxels may be located. The additional geometry can be generated by subtracting the view frustum of the previous frame from the initial cube proxy geometry, which is guaranteed to contain all voxels.

5.3.2 Clipping the occlusion volume

The occlusion frustum approach can be compared to the shadow volume algorithm (Crow, 1977) for adding shadows to a geometrically constructed scene. There the silhouette of occluders is typically extruded to infinity. This cannot be directly translated to our case, since additional information is carried by the geometry. All proxy geometry has to be placed inside a bounding box of $[0,1]^2$ because of the way ray parameters are encoded as colors. Those are clamped to $[0,1]$ by the graphics hardware and therefore geometry outside this unit cube would lead to incorrect ray parameters. Additionally the frustum must be closed from all sides to give correct results for all possible viewing directions. Therefore the initial occlusion frustum has to be clipped against the data set bounding box to construct the final frustum.

5.3.3 Possible extensions to the occlusion frustum approach

The basic occlusion frustum approach is flexible and offers several options for further optimization. However, it must be examined carefully whether extensions result in an overall performance advantage.

Incremental geometry refinement. As presented here, the generated geometry is recomputed for every frame. However, the approach could be changed to be incremental by calculating the intersection between the newly generated frustums

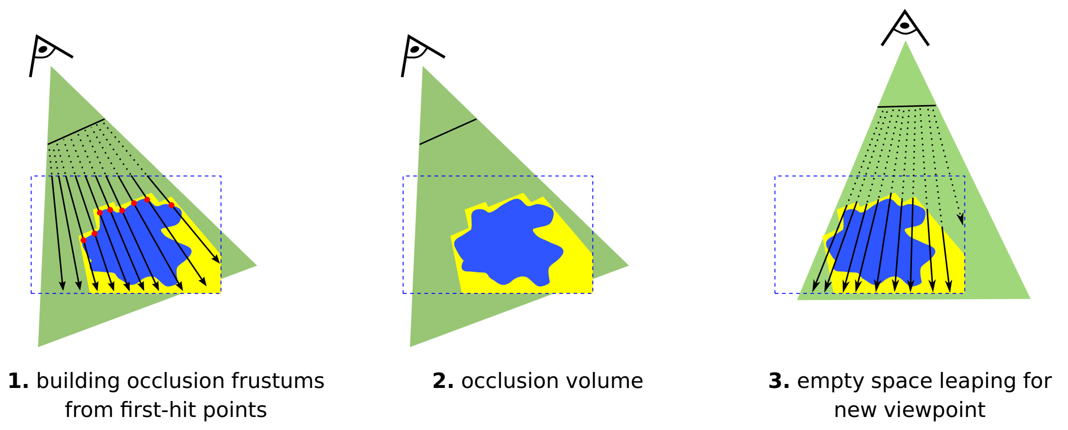

1. building occlusion frustums from first-hit points
2. occlusion volume
3. empty space leaping for new viewpoint

Figure 5.9: An occlusion volume (yellow) is constructed as the union of all occlusion frustums generated from the first-hit points (red). To achieve empty space leaping for a new viewpoint, it is sufficient to apply the new view transformation and use the occlusion volume as the proxy geometry.

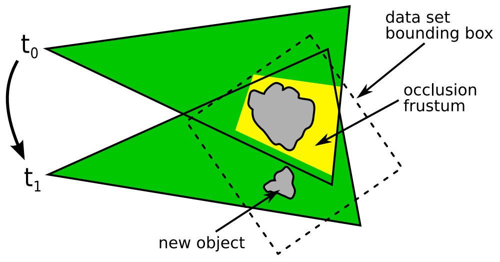


Figure 5.10: New incoming object problem: The new object is outside the occlusion frustum constructed at t_0 and therefore not considered for raycasting at t_1.

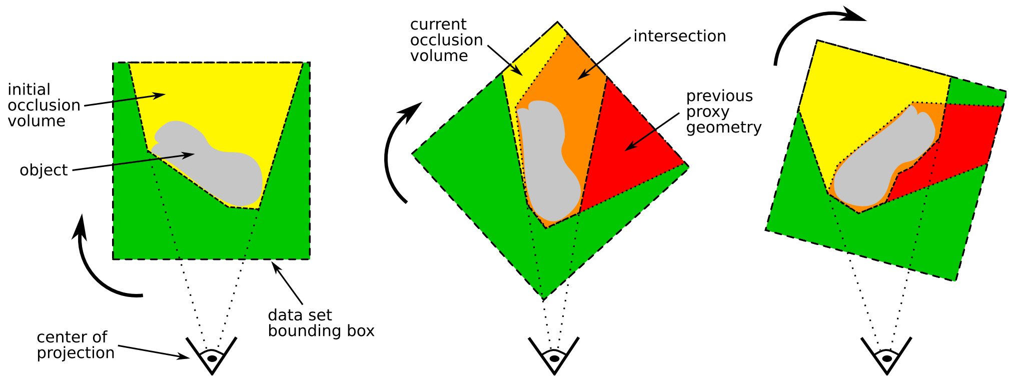


Figure 5.11: Incremental refinement of the proxy geometry over several frames. For each frame the new proxy geometry (orange) is computed as the intersection of the current occlusion volume (yellow) and the previous proxy geometry (red).

and the previous proxy geometry: With every view change the geometry would get closer to the optimal proxy geometry (Figure 5.11). Unfortunately this would substantially increase the complexity of the geometry calculations and the amount of created geometry. Also, direct implementation on graphics hardware would become difficult. An additional problem that might arise is that any voxels missed in a single frame due to undersampling could lead to incorrectly identifying parts of the volume as empty and permanently removing them from the proxy geometry. The problem is less serious with the non-incremental approach, as there the geometry is constantly refreshed and errors will be removed immediately. For small view changes undersampling is not a significant problem, as every voxel that is ignored for proxy geometry calculations due to undersampling would not be visible in the final image anyway. In summary, an incremental approach is costly and may accumulate image artifacts.

Empty space behind the object. Most space leaping techniques only consider the empty space between the ray start and the first-hit point. Especially for sparse data sets, however, much more empty space can remain between the first-hit point and the ray end. The empty space *inside* an object can get too complex to be handled efficiently, but the empty space *behind* it is much simpler. Instead of first-hit points, now information about *last*-hit points is necessary, i. e., the position of the last non-empty voxel between first-hit point and ray end (compare Figure 5.1). The last-hit image could be used to construct the back of the occlusion frustums, instead of building them by clipping against the bounding box. Though initially reasonable, we have not implemented this extension for two reasons: First, it is not compatible with early ray termination, as finding the last-hit points requires a full traversal of all rays. Second, while about doubling the costs for creating the occlusion frustum, for the typical use case of a rotating object the amount of traversed empty voxels that could be saved with this extension is relatively small.

Multiple views. For some use cases the volume object needs to be displayed from multiple viewpoints at the same time, e. g., to increase spatial comprehension in a medical application. Here it might be advantageous to generate the occlusion frustum proxy geometry for just one viewpoint and reuse it for the others. Obviously, the efficiency of this optimization depends on how the viewpoints are arranged. When two views show the same object from opposite sides, even the previously dismissed method to include the empty space behind an object might be advantageous.

In case the viewport sizes of the different views are not equal, it must be decided whether to generate the occlusion frustums from the larger viewport and reuse them for raycasting on the smaller one or vice versa. Although the optimized proxy geometry can be created faster from the smaller viewport as fewer pixels need to be analyzed, it can nonetheless be beneficial to use the larger viewport instead. The generated proxy geometry is not only used for rendering the current frame on the smaller viewport, but also for rendering the next frame on the larger viewport. Since the overall performance relies most on the rendering speed on the larger viewport, it is important that the proxy geometry removes unnecessary sampling operations in empty space with the highest efficiency when rendering to this viewport. Hence, the proxy geometry generated from the larger viewport should also be used for raycasting on the smaller one to get best performance. This also resolves the problem of screen space undersampling, which could lead to rendering artifacts when the occlusion frustums are generated from a small first-hit point image and are then enlarged for raycasting on a much larger viewport. As in this case not all rays on the larger viewport have been previously consulted while generating the occlusion frustums, the geometry could be too small, i. e., voxels would be incorrectly classified as empty space. This issue does not occur when the viewport size used for the raycasting is smaller than that of the first-hit point image.

Delayed geometry updates. With a direct implementation of our approach, the occlusion volume geometry is regenerated for every frame. This ensures an optimal geometry as long as there is enough frame-to-frame coherence, i. e., the viewpoint only changes slightly in between frames. For very small viewpoint changes, reusing the previous occlusion volume proxy geometry instead of regenerating it only introduces a small overhead, which may be smaller than the cost for regenerating the occlusion frustums. Hence, a practical optimization would be to only update the proxy geometry when viewing parameters have changed by more than a certain threshold, e. g., when the camera was rotated by more than 5 degrees relative to when the occlusion geometry was generated. The actual thresholds would need to be determined experimentally, as the break-even point between using an outdated occlusion volume proxy geometry and the costs of generating a new one depends on several parameters such as complexity of the data set. The expected unsteadiness of the frame rate caused by the delayed updates of the geometry might be of concern for continuous animation. The effectiveness of optimization also highly depends on the application and, more specifically, on the usage pattern regarding viewpoint changes. Therefore, we have not implemented the optimization for our tests of the occlusion frustum approach, as we are interested in the behavior of the approach in the general case and not only for a specific usage pattern.

5.4 GPU Implementation

As described before, the starting point of our approach is the first-hit image. It can be generated in the raycasting fragment shader by detecting the first-hit voxel and writing its position into an additional rendering buffer. By exploiting the OpenGL multiple render target extension, this can be done during normal ray traversal without the need for a second pass, so that the first-hit image is extracted with minimal overhead. The steps described in the following subsections are inserted into the volume rendering process just before the actual raycasting operation, where normally a cube proxy geometry is rendered to generate the ray start and end point textures used by the raycasting. The only additional change is instructing the raycaster to also output the first-hit image, as described above. Thus, the proposed optimization can be easily integrated into existing volume raycasting frameworks that use the Krüger-Westermann approach, what we demonstrate by integrating it into Voreen in the next section.

5.4.1 Analyzing first-hit points

Generating an occlusion frustum for each pixel in the first-hit image would result in a prohibitively large amount of geometry for high viewport resolutions. The complexity can be reduced by downsampling the first-hit image, so that an occlusion frustum corresponds to multiple pixels. However, special care must be taken to ensure that no important information is lost during the downsampling, which would lead to an incorrect occlusion frustum and rendering artifacts. To get correct results, we first group adjacent pixels as square *occlusion blocks*. We then analyze all voxels corresponding to pixels in each block to find the voxel with the minimum distance to the viewpoint. This voxel's position is later used for constructing a frustum that encloses all non-empty voxels hit by rays associated with the occlusion block. Figure 5.8 on Page 37 illustrates the two-dimensional case with a block size of three, showing that the position of the occlusion frustum's top base is determined by the voxel closest to the center of projection. The proxy geometry is enlarged by the simplification, which leads to sampling of some empty voxels and thus reduces efficiency of space leaping. But as discussed in Section 5.2, the hardware processes fragments block-wise and the slowest fragment limits the processing speed for all fragments in a block. Hence, the block simplification suits the hardware limitations, and an unsimplified solution would not result in significantly better performance results. The simplification can be implemented efficiently as a fragment shader and return the *occlusion block texture*.

We implemented the grouping of adjacent pixels in the first-hit image to generate the occlusion block texture as a fragment shader running in a ping-pong scheme with the first-hit texture as input. In each pass the shader replaces every 2×2 pixel block by the one pixel inside the block that is closest to the camera. The process therefore halves the side length of the input texture in each pass. Hence, when a block size of 4×4 is requested, two passes are needed. This could also be performed in a single pass of a slightly more complex shader, but the ping-pong scheme is a standard approach for this type of problem and runs efficiently. In addition, the runtime of this simple image-based operation is insignificant in comparison to the more complex generation of the occlusion frustum geometry or the raycasting.

5.4.2 Generating occlusion frustums

The algorithm for generating the occlusion frustum geometry is especially suitable for implementation on the GPU using geometry shaders. They allow to generate new graphics primitives (e. g., points or triangles) from input primitives sent by previous stages of the graphics pipeline. While they can theoretically generate arbitrary amounts of output primitives from one input primitive, current implementations force the programmer to specify the maximum number of output primitives in advance and are most efficient when this value is not set too high.

In our algorithm, an occlusion frustum that consists of twelve triangles has to be generated for each non-empty occlusion block. These input blocks can be modeled easily as point primitives, with their x- and y-coordinates set to the texture coordinates of the corresponding texel in the occlusion block texture. The application sends each point primitive and the occlusion block texture to the geometry shader, which either outputs the clipped frustum or zero triangles, depending on whether the block corresponding to the input point is empty or not. In the geometry shader the frustum is constructed out of triangle primitives and clipped against the data set bounding box. To ensure that it contains all relevant voxels, the frustum is enlarged slightly and moved towards the camera. The resulting geometry is rendered using a fragment shader that assigns the vertex position as fragment color, while the z-buffer handles overlapping frustums.

The entire occlusion frustum shader consists of three components: First, a vertex shader that reads from the first-hit point texture and calculates the basic properties of the corresponding occlusion frustum, such as distance to the camera. Second, the geometry shader that uses this information to generate, clip, and output the triangles that make up the frustum. Third, a simple fragment shader that sets the interpolated vertex positions generated by the geometry shader as the output fragment color, which is then written to the ray start point texture. Since the geometry shader combines both generating the occlusion frustum geometry and sending it to the rasterization stage, it needs to take different viewing parameters into account. For generating the occlusion frustums the same camera parameters as used for rendering the originating first-hit image must be applied, i. e., the viewpoint of the previous frame. To get the ray parameters for the current frame, the proxy geometry must then be transformed according to the new camera parameters, which is performed in a single pass of the geometry shader.

For supporting older hardware, the algorithm could also be adapted to vertex shaders since the maximum number of generated vertices for each occlusion block is known beforehand. However, this might result in a significant performance penalty, as the texture fetch would have to be made per-vertex instead of per-frustum. For empty occlusion blocks a degenerate geometry with all vertices of the corresponding frustum set to zero would be built, effectively removing it from the output.

After the occlusion frustums have been constructed, we finally add the bounding box cube, from which the view frustum of the previous frame is subtracted, to the proxy geometry for detecting objects that become visible (see Subsection 5.3.1). The resulting geometry can be used for ray setup by rendering the front faces to get ray start points, while the ray end points are still retrieved by rendering the back faces of the data set bounding box. When the empty space information is invalidated by changing rendering parameters like the transfer function, a single frame has to be rendered using the data set bounding box as the proxy geometry, but subsequent frames can again use the occlusion frustums.

5.5 Integration into Voreen

We have implemented the presented occlusion frustum approach using OpenGL and GLSL shaders. It was integrated into the Voreen volume rendering framework for combination with the standard raycasting processors already available in the system.

5.5.1 Data-flow network

Integrating the occlusion frustum approach into Voreen is quite simple, as it only covers analyzing the previous first-hit image and constructing the optimized proxy geometry. A simple Voreen network that uses an occlusion frustum proxy geometry is shown in Figure 5.12b. Compared to a standard network (Figure 5.12a) there are several changes: Two processors were replaced, an additional outport of the raycasting processor is used, and an additional `RenderStore` processor with accom-

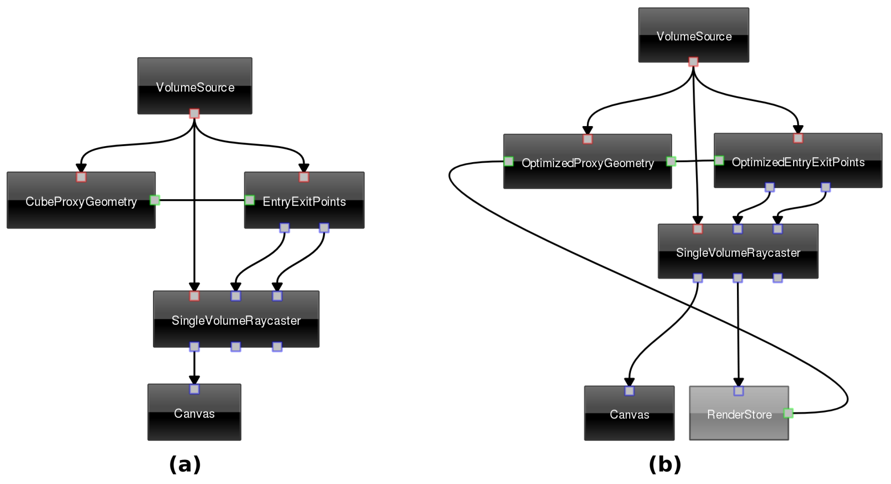


Figure 5.12: (a) Standard Voreen network for volume raycasting, (b) extended network using the optimized proxy geometry based on occlusion frustums. The second outport of `SingleVolumeRaycaster` outputs a first-hit image that is stored in `RenderStore`, from which it is accessed by `OptimizedProxyGeometry` through the coprocessor connection.

panying connections was added. The two new processors `OptimizedProxyGeometry` and `OptimizedEntryExitPoints` look very similar to their standard counterparts `CubeProxyGeometry` and `EntryExitPoints`, with the only exception being an additional coprocessor port for `OptimizedProxyGeometry`, which is used for accessing the first-hit image. Instead of relying on a simple cube, the new processors generate ray start and end point textures using an occlusion frustum proxy geometry. The standard `SingleVolumeRaycaster` can be configured to output results of different compositing modes at the same time, so we select DVR for the first outport and first-hit points for the second, previously unused, outport. Thanks to OpenGL's multiple render target extension, this can be performed with little overhead in a single ray traversal.

The first-hit image is needed by the `OptimizedProxyGeometry`, but a direct connection from `SingleVolumeRaycaster` is not possible for transferring the image, as this would introduce a cycle into the network. However, the `OptimizedProxyGeometry` processor only needs the first-hit image of the *previous* frame to generate the optimized proxy geometry and ray parameter textures for the *current* frame, so no cycles are required. Hence, the standard `RenderStore` processor is used, which stores a copy of its input image and makes it available to other processors via a coprocessor port. Hence, the first-hit image generated by the `SingleVolumeRaycaster` is directed into the `RenderStore`, which shares a coprocessor connection with `OptimizedProxyGeometry`. When the `OptimizedProxyGeometry` is processed the next time, i. e., when the next frame is to be rendered, it can then access the first-hit image of the previous frame through the coprocessor connection.

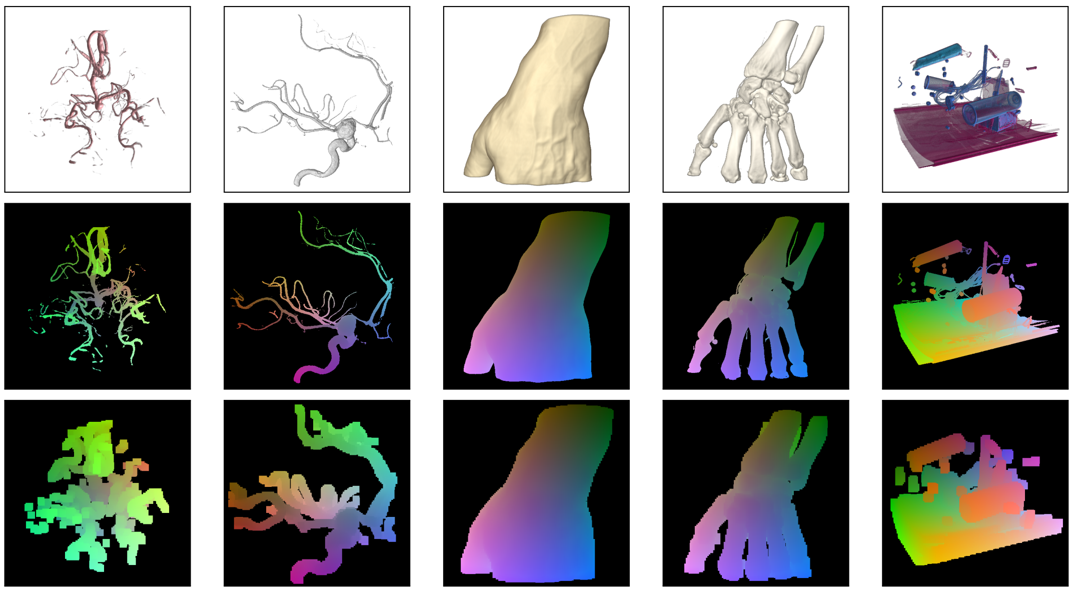
5.5.2 Implementing the optimized proxy geometry processor

We integrated the functionality of analyzing the previous first-hit image and generating the occlusion frustum geometry completely into the `OptimizedProxyGeometry` processor. It first runs the block simplification on the first-hit image retrieved from `RenderStore` using multiple passes of a simple fragment shader to get the occlusion block texture. Afterwards it executes the shaders for generating and rendering the occlusion frustum geometry with the occlusion block texture and the predefined point primitives as input. The generated proxy geometry is used by `OptimizedEntryExitPoints` to output the ray start point texture. When no valid first-hit image is available, i. e., for the first frame or when the transfer function has been modified, the simple proxy geometry as inherited from `CubeProxyGeometry` is used instead. For the ray end points the back faces of the default geometry from `CubeProxyGeometry` are used.

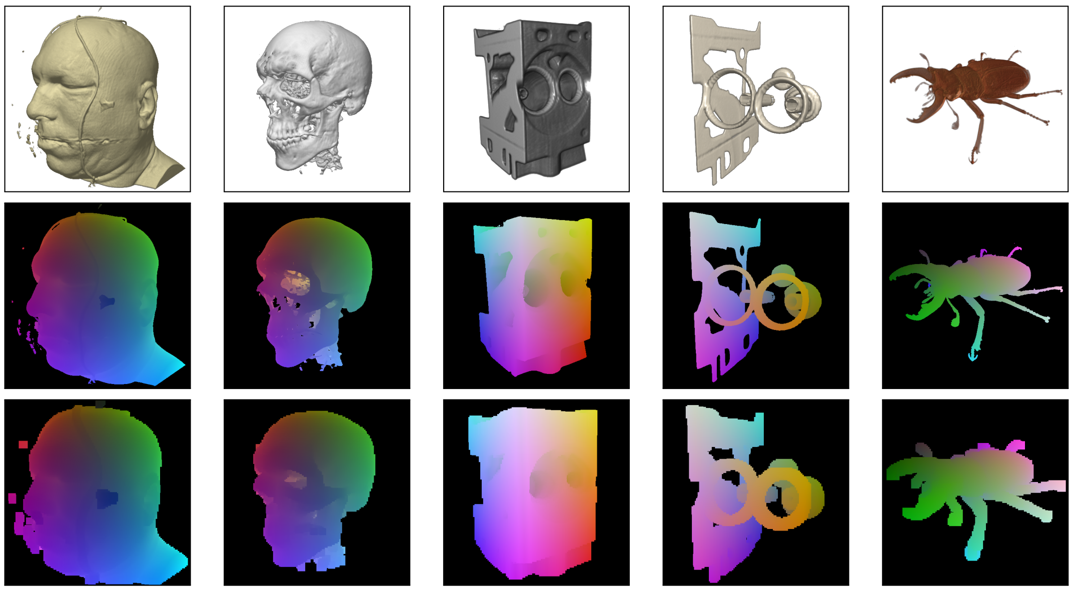
5.6 Results

5.6.1 Performance evaluation

All tests were conducted with an Intel Core 2 Duo E6300 CPU and an NVIDIA GeForce 8800 GT graphics board with 512 MB of onboard memory. We have tested our algorithm with different sparse and dense data sets; the results are shown in Figure 5.13 and Table 5.3. Rendering was performed using on-the-fly gradient calculation, Phong lighting, and early ray termination. The objects were constantly rotated and visualized using direct volume rendering. Occlusion frustum optimization was applied with a block size of 4×4, a compromise between accuracy and complexity of the generated proxy geometry. In our tests we did not observe artifacts caused by the optimization. As it is an image-based technique, undersampling could be problematic, since voxels missed due to undersampling might lead to incorrectly identifying parts of the volume as empty and removing them from the proxy geometry. In practice this poses no problem because of the constant refreshing of the occlusion frustum geometry. Also, the occlusion blocks lower the chance that this happens, as all voxels in a block would have to be missed due to undersampling to remove the associated frustum.



(a) vertebra **(b)** aneurysm **(c)** hand (skin) **(d)** hand (bone) **(e)** backpack



(f) vmhead (skin) **(g)** vmhead (bone) **(h)** engine **(i)** engine (interior) **(j)** stagbeetle

Figure 5.13: Results of applying our space leaping technique to different dense and sparse data sets. The resulting final image, first-hit image, and the constructed occlusion frustum geometry, which is an approximation of the first-hit image, are shown.

5.6.2 Discussion

As expected, the highest speedups were found with large but sparse data sets like *vertebra*[*]. For dense objects with little empty space like *vmhead (skin)*, hardly any optimization is possible. The amount of empty space depends on the transfer function, and so does the speedup, as shown with the *hand* and *vmhead* data sets, where different transfer functions for showing skin and bone structures are applied. Since the optimization introduces an additional overhead, our approach can only result in a significant speedup if the costs of sampling empty voxels are higher than those for generating the occlusion frustums. Fortunately, the optimization is most useful for high-resolution data sets and high-quality rendering, where the costs for sampling empty voxels will also be high. As the runtime of our approach is mainly dependent on the resulting 2D image resolution, but not 3D data set resolution, it will produce good results in this case. The costs for optimization increase with the viewport size, but the amount of saved sampling operations increases proportionally. Therefore the technique scales well even to the quite large viewport size of 1024^2, for some data sets even producing greater speedups than for 512^2.

Besides with rotations, we also tested the performance when the point of view is moved to a random position after each frame. While reprojection methods cannot give valid results in this case, our approach can even then produce an acceleration compared to the cube proxy geometry, as indicated in Table 5.4. Random viewpoint changes destroy most of the frame-to-frame coherence and therefore reduce the effect of the optimized proxy geometry on rendering performance. For some data sets like *engine* the optimization overhead then gets larger than the costs of sampling empty voxels, leading to a performance decrease. But nonetheless a correct image is rendered in any case, and an optimization is still achieved for some data sets.

Comparing our results to those of similar methods, Klein et al. (2005) report a speedup of up to 1.7x for semi-transparent volume rendering with small view changes. They report an even larger speedup of up to 2.7x for isosurface rendering, as this rendering mode benefits even more from space leaping, but it cannot be compared to our direct volume rendering results. Our results on similar data sets are comparable to—if not better than—their semi-transparent rendering, possibly because no hole-filling is required. Hence, our technique is competitive with point-based

[*] While named "vertebra", the data set actually is a rotational angiography scan of a human head with an aneurysm. Vertebrae are the individual bones that build the spinal column of vertebrate animals, but these are not visible in the data set. We would like to thank Prof. Dr. Timothy S. Newman from the University of Alabama in Huntsville for pointing out this discrepancy to us. To be consistent with other uses of the data set and because we were already using a different data set called "aneurysm", we stayed with the original name.

		viewport 512^2				viewport 1024^2			
data set	size	cube	opt.	s	triangles	cube	opt.	s	triangles
vertebra	512^3	19.1	43.1	2.26	33,876	7.6	18.9	2.49	167,904
aneurysm	256^3	29.7	55.1	1.86	47,688	8.5	15.6	1.84	182,208
hand (skin)	256^3	19.4	25.1	1.29	65,772	5.6	7.1	1.27	250,056
hand (bone)	256^3	25.7	48.8	1.90	53,676	7.6	12.9	1.70	217,524
backpack	$512^2 \times 373$	24.0	44.6	1.86	31,608	8.4	16.8	2.00	128,724
vmhead (skin)	256^3	36.9	39.1	1.06	81,576	4.0	3.9	0.98	327,624
vmhead (bone)	256^3	27.0	36.7	1.36	54,888	7.1	9.2	1.30	216,060
engine	$256^2 \times 128$	24.2	27.7	1.14	47,664	7.4	8.3	1.12	189,672
engine (int.)	$256^2 \times 128$	37.5	64.6	1.72	24,300	11.3	21.9	1.94	93,132
stagbeetle	$416^2 \times 247$	8.5	15.3	1.80	37,464	3.1	6.6	2.13	135,108

Table 5.3: Results for different data sets, with average frame rates for a full rotation of the object using a cube proxy geometry and our occlusion frustum optimization, resulting speedup factor s, and triangle count of the optimized proxy geometry.

data set	cube	opt.	s
vertebra	16.8	22.9	1.36
vmhead (bone)	19.1	19.5	1.02
engine	25.6	19.8	0.77

Table 5.4: Resulting frame rates and speedup factors for random viewpoint selection on a 512^2 viewport. The point of view was changed to a random position after each frame, destroying most of the frame-to-frame coherence.

reprojection, with the additional benefit that it allows also large viewpoint changes without introducing artifacts. More recently, Liu et al. (2009) proposed to use *proxy spheres* instead of occlusion frustums to construct an optimized proxy geometry, and conducted a detailed comparison of their technique to ours. However, their general approach is different in that they perform a preprocessing step on the GPU to detect which blocks in the volume are active, i. e., visible, and construct a proxy geometry consisting of discs oriented orthogonal to the view direction. Hence, their technique is a descendant of the PARC algorithm (Avila et al., 1992). In contrast, our approach does not require any preprocessing but instead makes use of frame-to-frame coherence. Also, the occlusion frustums give correct results for any volume, while the proxy spheres approach fails for data sets with anisotropic voxels.

5.7 Summary

After examining architectural limitations of graphics hardware with respect to raycasting optimization, we have presented a purely GPU-based method for accelerating volume raycasting using empty space leaping. By exploiting the temporal coherence between consecutive frames we achieved a speedup of up to a factor of two. Artifacts could be caused by undersampling, but as the created geometry is constantly refreshed, they will be removed immediately. As it requires no preprocessing, the technique is also suitable for data exploration applications. Best results can be expected when large parts of the data set have been removed by the transfer function, a typical operation for data exploration. Compared to point-based reprojection methods, our approach results in a proxy geometry that can be used for any point of view, although optimal space leaping is expected for small view changes. Holes can never appear and consequently no hole filling is necessary, which would trigger full raycasting for unknown regions and reduce optimization efficiency.

The technique can be integrated easily into an existing GPU-based raycasting infrastructure, as it requires only minimal changes within the raycaster and has no external dependencies. To further improve optimization, it should be examined how to perform incremental optimization of the proxy geometry by refining the previous occlusion frustums after each view change, instead of recreating them from scratch. This would require implementing the refinement step on the GPU, as well as an approach to prevent visual artifacts caused by the incremental construction of the geometry.

Chapter 6

Applying GPU Stream Processing to Volume Raycasting

The ray traversal in GPU-based raycasting is usually implemented in a fragment shader, utilizing the hardware in a way that was not originally intended. New programming interfaces for stream processing, such as CUDA or OpenCL, support a more general programming model and the use of additional device features, which are not accessible through traditional shader programming. In this chapter we first compare fragment shader implementations of basic raycasting to implementations directly translated to CUDA kernels. Then we propose a new slab-based raycasting technique that is modeled specifically to use the additional device features to accelerate volume rendering. We conclude that new stream processing models can only gain a small performance advantage when directly porting the basic raycasting algorithm. However, they can be advantageous through novel acceleration methods that use the hardware features not available to shader implementations.

Implementations of GPU-based volume raycasting usually apply general-purpose GPU programming techniques (GPGPU), which skip most of the geometry functionality of the hardware and use fragment shaders to perform raycasting through the volume data set. The unified shading architecture of modern GPUs supports dynamic allocation of computing resources to different shader types, so that applications that rely heavily on fragment operations achieve a high overall utilization of all hardware resources. Still, it is not obvious whether this not originally intended use of the hardware gives optimal performance results, or if the processing overhead from the graphics system has a negative impact on the overall performance of such a raycasting implementation.

Modern GPUs support stream processing as an alternative programming model to classical graphics APIs such as OpenGL. These stream processing models, e. g., NVIDIA's CUDA or OpenCL, allow a more general access to the hardware and also support certain hardware features not available via graphics APIs, such as on-chip shared memory.

As described in Chapter 3, rendering approaches for volumetric data can be classified as object-order and image-order traversal techniques. Object-order techniques like slice rendering simplify parallelization by accessing the volume data in a regular manner, but cannot easily generate high quality images and are rather inflexible with regard to acceleration techniques. Image-order techniques such as raycasting, on the other hand, can generate good visual results and can be accelerated easily, e. g., with early ray termination (ERT) or empty space skipping. However, their ray traversal through the volume leads to highly irregular memory accesses. This can undermine caching and also complicate efforts towards a parallel implementation.

In contrast to many other applications of stream processing that often achieve high speedup factors, such as CT reconstruction, volume raycasting already uses the graphics hardware instead of the CPU. Hence, no major speedups are expected simply by porting a raycasting shader to CUDA. However, fragment shader implementations do not allow sharing of data or intermediate results between different threads, i. e., rays, and therefore this data needs to be fetched or recalculated over and over again. More general programming models exploiting fast on-chip memory could allow a massive reduction in the number of memory transactions and therefore make more advanced visualization techniques available for interactive use.

The remainder of this chapter is structured as follows. In Section 6.1 we review previous work related to raycasting and stream processing. Section 6.2 discusses features and limitations of the CUDA programming model with regard to raycasting. In Section 6.3 we first examine the general suitability of stream processing for direct volume rendering (DVR) by comparing CUDA- and shader-based raycasting implementations. Afterwards, in Section 6.4, we discuss acceleration techniques that utilize the additional device features accessible through CUDA and introduce a novel slab-based approach, going beyond what is possible with shader programming. The integration of the CUDA-based techniques into Voreen is described in Section 6.5. We present the performance results in Section 6.6 and conclusions in Section 6.7.

6.1 Related Work

6.1.1 GPU-based volume raycasting

To speed up rendering, or to support data sets not fitting in GPU memory, the volume data can be subdivided into bricks (Scharsach et al., 2006) through which rays are cast independently, while compositing the results afterwards. Law and Yagel (1996) presented a bricked volume layout for distributed parallel processing systems that minimizes cache thrashing by preventing multiple transfers of the same volume data to the same processor in order to improve rendering performance. Grimm et al. (2004) used this approach to get optimal cache coherence on a single processor with hyper-threading. However, the method is only applicable to orthographic projection but not to perspective projection, and therefore unsuitable for general volume visualization.

Many acceleration techniques have been developed for the more general problem of raytracing, often incorporating efficient spatial data structures. These can be utilized when raycasting geometric objects or isosurfaces from volumetric data, e. g., implicit acceleration structures (Wald et al., 2005), but they are not applicable to direct volume rendering and therefore beyond the scope of this thesis.

6.1.2 GPU stream processing

New programming interfaces for stream processing allow to bypass the graphics pipeline and directly use the GPU as a massively parallel computing platform. The stream processing model is limited in functionality compared to, e. g., a multi-CPU architecture, but can be mapped very efficiently to the graphics hardware, which is akin to a SIMD system.

Intel's upcoming Larrabee architecture (Seiler et al., 2008) can be considered a hybrid between a multi-core CPU and a GPU, which will allow a software-based implementation of rasterized graphics APIs such as OpenGL. The available information indicates that the Larrabee hardware will also efficiently support tasks such as raytracing and volume raycasting (Smelyanskiy et al., 2009). In the context of stream processing also the Cell Broadband Engine Architecture (Kahle et al., 2005) is often mentioned. However, it is more akin to a MIMD architecture and thus the underlying programming model is more complex compared to pure stream processors. It is also not as widely available as GPUs and we therefore do not consider it as suitable for visualization tasks.

We have chosen CUDA as our platform for evaluating raycasting techniques because it is the most mature and stable of the current programming models and it is available for multiple operating systems. At the time of writing this thesis the publicly available implementations of OpenCL had not yet reached a level of stability and efficiency that is comparable to CUDA. For example, CUDA implementations of sample programs from NVIDIA's GPU computing SDK typically outperformed feature-identical programs implemented in OpenCL. But as CUDA shares many similarities with the OpenCL programming model, porting results to this new industry standard can be performed easily, when implementations finally improve in stability and performance.

Besides for numerical computations, CUDA has been used for some rendering techniques, including raytracing (Luebke and Parker, 2008). A simple volume raycasting example is included in the CUDA SDK (NVIDIA, 2008c). Maršálek et al. (2008) also demonstrated proof of concept of a simple CUDA raycaster and did a performance comparison to a shader implementation. Their results showed a slight performance advantage for the CUDA implementation, but they did not incorporate lighting or other advanced rendering techniques, which would be required for practical applicability. Grauer et al. (2008) have started work on a CUDA-based volume rendering module for the 3D Slicer visualization system, but the project has since been abandoned. Kim implemented bricked raycasting on the Cell Architecture (Kim and JaJa, 2008) and on CUDA (Kim, 2008), distributing some of the data management work to the CPU. He focused on streaming volume data not fitting in GPU memory and did not use all available hardware features for optimization, such as the texture filtering hardware. Smelyanskiy et al. (2009) compared raycasting implementations running on a simulation of the Larrabee architecture and running on CUDA hardware, focusing on volume compression and not including texture filtering. Kainz et al. (2009) recently introduced a new approach for raycasting multiple volume data sets using CUDA. It is based on implementing rasterization of the proxy geometry manually instead of relying on the usual graphics pipeline.

6.2 Raycasting with CUDA

6.2.1 Using the CUDA architecture for raycasting

Implementing raycasting using CUDA allows for more flexibility compared to a fragment shader implementation. However, for an efficient implementation it is essential to consider the strengths and limitations of the hardware architecture. In addition to the basic information discussed in Section 2.3, we have examined the individual properties of graphics processors with regard to a raycasting implementation to decide how to make best use of the hardware's capabilities.

The inherent parallelism in the raycasting algorithm is based on the fact that rays are independent of each other and therefore can be traversed in parallel. Hence, it is obvious to model rays as individual threads for implementing a raycasting kernel. CUDA threads are organized into thread blocks, and the block size can have a significant influence on the kernel performance, i. e., it must neither be chosen too large nor too small. For arranging rays in thread blocks, there are basically two approaches: arranging the rays in a linear fashion (one-dimensional) or building rectangular blocks of rays (two-dimensional). Both approaches are reasonable in that they will return correct results, but from a performance perspective rectangular blocks are to be preferred. One important precondition for optimal performance when executing a thread block is a high degree of branch coherence in the threads, or, more specifically, in those threads that are executed in the same warp. As threads with different code paths inside a warp lead to serialization of execution, an insufficient coherence between threads can foil any potential speedup. Taking our experiments about ray coherence in shader-based raycasting from Section 5.2 into account, we chose rectangular thread blocks. In contrast to one-dimensional blocks they maintain spatial coherence, i. e., rays in the same block correspond to neighboring pixels on the screen, while the rays in a one-dimensional block may be located far away from each other in screen space. The optimal dimensions for the rectangular thread blocks cannot be predicted but should be determined experimentally, as we will show in Section 6.6. The available documentation resources indicate that the presented approach of mapping rays to threads and grouping rectangular regions of the screen for CUDA raycasting is actually quite similar to how a fragment shader implementation would operate. This is not surprising, as graphics processors were initially designed specifically for supporting fragment shaders, so the basic usage pattern for CUDA must be similar to attain optimal utilization of the hardware resources.

A major reason for the high performance that CUDA kernels can achieve is that they can make use of the high memory bandwidth of modern graphics processors. This, however, requires that the used memory access pattern complies with the coalescing rules (compare Section 2.3) to consume the minimum number of memory transactions. Hence, the achievable throughput for random access in global memory will be much lower than for a regular access pattern that satisfies the coalescing rules. In a raycasting kernel the majority of all memory accesses will be in the volume data. Unfortunately, as raycasting is an image-order technique, the access pattern is not regular, and it will only achieve a low data throughput. However, this can be improved by using the texturing hardware, which does not require coalescing, as discussed in the next section. A raycasting kernel can also contain regular memory access, e. g., when reading information for ray setup from a 2D array. But as this only affects a small part of all memory reads, which are dominated by sampling the volume data, the influence on the overall performance by achieving coalescing will be limited.

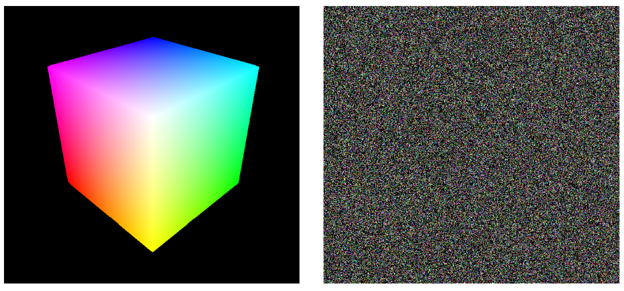
Limitations similar to the coalescing rules for global memory also exist for shared memory. In order to achieve the same performance as for accessing a hardware register, the access pattern in shared memory must chosen so that it does not cause bank conflicts. In a raycasting kernel the shared memory could, for example, be used as a fast cache for parts of the volume data that are accessed more than once. However, because of its irregular access pattern, a ray traversal through shared memory will cause bank conflicts, just like raycasting in global memory, and thus will not be able to achieve optimal performance.

The Krüger-Westermann approach for GPU-based raycasting is based on rendering a proxy geometry to generate ray parameters. When using CUDA for raycasting, there are several alternatives: using ray parameter textures from OpenGL, implementing rasterization in CUDA, and computing ray parameters analytically. While the analytical approach is easy to implement, much flexibility is lost, such as direct support for simple clipping, empty space skipping as discussed in Chapter 5, or the ability of deform the volume object (Mensmann et al., 2008b). Hence, we consider generating the ray parameter textures by rendering a proxy geometry with OpenGL as the most flexible and also most efficient solution. Rasterization can also be implemented with CUDA (Kainz et al., 2009), but the complexity of a full-featured and efficient rasterization implementation should not be underestimated. The existing OpenGL rasterization will certainly be more optimal, as it can potentially make better use of internal and possibly undocumented features of the hardware.

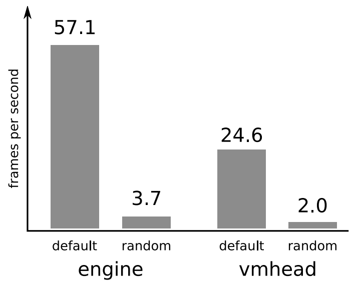
6.2.2 3D texture caching

Just like a shader, a CUDA kernel can also access 1D, 2D, and 3D textures, and thus use the highly optimized texturing hardware for filtering and border handling. For the CUDA hardware, caching of the texture memory is largely undocumented beyond the fact that it is optimized for 2D spatial locality and constant latency (NVIDIA, 2010, p. 89).[*] Knowledge about the caching behavior is important for judging the suitability of techniques that may influence the coherence of texture access. In Section 5.2 we already examined the effect of varying ray lengths on the performance of GPU-based raycasting. We used a similar test case to determine the influence of the texture cache on 3D texture fetches. To measure this effect, we applied the same random permutation to the pixels of the start and end point textures (Figure 6.1a) to deliberately destroy coherence while keeping the amount of volume texture fetches unchanged. We also modified all rays to have equal length, in order to remove the influence of ray length coherence issues. With these modified ray parameters we experienced a more than tenfold increase in the runtime of raycasting (Figure 6.1b), even when not including the time needed for generating the ray parameter textures. This demonstrates that raycasting relies heavily on the

[*] It is notable that before the introduction of CUDA, even less information was available from vendors about low-level features of their graphics processors. To understand the inner workings of GPUs, one had to rely mainly on the results of benchmarking tools such as GPUBench (Buck et al., 2004a).



(a)



(b)

Figure 6.1: (a) Ray start point texture before and after applying a random permutation. (b) Frame rates for raycasting two data sets using the default as well as the randomized start points.

texture caching hardware and that the usual traversal scheme with an advancing ray front, i. e., adjacent rays sampling adjacent voxels, offers a significant amount of locality. Therefore, any reasonable acceleration scheme must make sure that in addition to ray length coherence also the locality of texture access is maintained.

6.2.3 Accelerating raycasting

While easy to implement, the basic raycasting algorithm leaves room for optimization. Many techniques have been proposed for DVR, from skipping over known empty voxels (Levoy, 1990) to adaptively changing the sampling rate (Röttger et al., 2003). Most of these techniques are also applicable to a CUDA implementation. In this chapter, we rather focus on techniques that can use the additional capabilities of CUDA to get a performance advantage over a shader implementation.

Many volume visualization techniques take a voxel's neighborhood into account for calculating its visual characteristics, starting with linear interpolation, to gradient calculations of differing complexity, to techniques for ambient occlusion (Hernell et al., 2007). As the neighborhoods of the voxels sampled by adjacent rays do overlap, many voxels are fetched multiple times, thus wasting memory bandwidth. Moving entire parts of the volume into a fast cache memory could remove much of the superfluous memory transfers.

As noted in Section 6.2.1, each multiprocessor has available 16 kB of shared memory, but less than half of this should be used by each thread block to get optimal performance. Using the memory for caching of volume data would allow for a subvolume of 16^3 voxels with 16 bit intensity values. In practice slightly less is available, since kernel parameters are also stored in shared memory. While accessing volume data cached in shared memory is faster than transferring the data from global memory, this has some disadvantages compared to using the texturing hardware. First, the texturing hardware directly supports trilinear filtering, which would have to be performed manually with multiple shared memory accesses. Second, the texturing hardware automatically handles out-of-range texture coordinates by clamping or wrapping, and removes the need for costly addressing and range checking. Finally, the caching mechanism of the texturing hardware can achieve read performance similar to that of shared memory, as long as the access pattern exhibits enough locality. While typically having some locality, the access pattern is still irregular and therefore it will be difficult to prevent bank conflicts in shared memory, which also is not an issue for texture access.

When a volume is divided into subvolumes that are moved into cache memory, accessing neighboring voxels becomes an issue. Many per-voxel operations like filtering or gradient calculation require access to neighboring voxels. For voxels on the border of the subvolumes much of their neighborhood is not directly accessible any more, since the surrounding voxels are not included in the cache. The neighborhood can either be accessed directly through global memory, or included into the subvolume as border voxels, thus reducing the usable size of the subvolume cache. With a relatively small subvolume size of 16^3 voxels the number of border voxels outside the subvolume is about 42% of the number of voxels inside the subvolume. Moving border voxels into the cache reduces the usable subvolume size to 14^3, with 33% of the cache occupied by border data. This would substantially reduce the efficiency of the subvolume cache.

Bricking implementations for shader-based volume raycasting often split the proxy geometry into many smaller bricks corresponding to the subvolumes and render them in front-to-back order. This requires special border handling inside the subvolumes and can introduce overhead due to the multitude of shader calls. A CUDA kernel would have to use a less flexible analytical approach for ray setup, instead of utilizing the rasterization hardware as proposed by Krüger and Westermann (2003), or implement its own rasterization method (Kainz et al., 2009). As described above, due to the scarce amount of shared memory the total number of bricks would also be quite high, increasing the overhead for managing bricks and compositing of intermediate results. The bricking technique described by Law and Yagel (1996) is specially designed for orthographic projection, for which the depth-sorting of the bricks can be simplified significantly, compared to perspective projection. Their technique also relies on per-brick lists, where rays are added after their first hit of the brick and removed after leaving it. This list handling can be efficiently implemented on the CPU, but such data structures do not map well to GPU hardware. As graphics processors feature a massively parallel architecture, synchronization would be crucial to prevent two

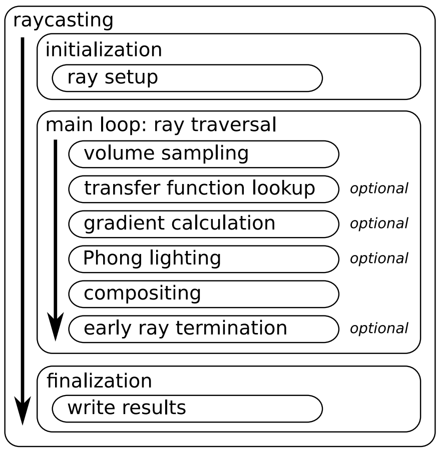


Figure 6.2: Building blocks for raycasting algorithms.

threads from writing to the same brick list at the same time, creating a race condition. However, efficient synchronization is not supported by the CUDA architecture, where data can be shared among thread blocks only between consecutive kernel calls. Kim (2008) works around this problem by handling the data structures on the CPU. As his aim is streaming of data not fitting into GPU memory, the additional overhead is of no concern, in contrast to when looking for a general approach for volume rendering.

To summarize, a direct bricking implementation with CUDA is problematic because only a small amount of shared memory is available and the ray setup and compositing for individual bricks is difficult. Therefore we will introduce an acceleration technique that is better adapted to the features and limitations of the CUDA architecture in Section 6.4.

6.3 Implementing Basic Raycasting

As illustrated in Figure 6.2, the basic raycasting algorithm (Krüger and Westermann, 2003) can be divided into three parts: initialization and ray setup, ray traversal, and writing the results. Raycasters implemented as fragment shaders or using CUDA can make different choices for the implementation of some of these building blocks, which we will discuss in this section.

6.3.1 Fragment shader implementation

Texture fetches can be used for retrieving ray start and end points, applying transfer functions, as well as for the volume sampling. All these operations can utilize the texturing hardware to get linear filtering. Results are typically written into one or more OpenGL render targets, hence also multiple outputs are possible, e. g., generating DVR and first-hit point rendering in one pass.

The source code of our fragment shader implementation is listed in Appendix A.1 (p. 107). Features marked as optional in Figure 6.2 can be enabled or disabled using preprocessor defines. As the shader is compiled by the OpenGL driver, this allows dynamic configuration of the raycasting process at runtime, while not introducing any overhead to the compiled shader program.

6.3.2 CUDA implementation

For each of the memory read operations in a CUDA kernel the question arises whether to use the texturing hardware or to read from global memory. Using textures for the ray start and end points does not have an advantage over memory reads in a CUDA implementation, as no filtering is necessary and coalescing can be achieved easily. Performance differences will have the biggest effect inside the raycasting loop, where both voxel sampling and transfer function lookup require filtering, so using textures is the natural choice. Also, textures have the additional advantage that for random access the latency of address calculations is better hidden (NVIDIA, 2008b, p. 68), potentially further improving performance compared to using global memory.

Our implementation first renders the proxy geometry into OpenGL textures to get the ray start and end points. Although at that point the ray parameter textures are already located in GPU memory, the raycasting kernel cannot access them directly, but an intermediate conversion is necessary. A texture first needs to be transferred into an OpenGL pixel buffer object (PBO), which the function `cudaGLMapBufferObject()` can then copy into the address space of CUDA, making its contents available to the raycasting kernel. This additional copy—even though an on-device operation—introduces an overhead that does not exist for a shader implementation of raycasting. In addition, several revisions of the CUDA runtime contained errors that severely limited the performance of copying from a PBO into CUDA memory, i. e., they would perform the copy through host memory instead of directly on the device (NVIDIA, 2009), causing a massive slowdown. The new OpenGL *interop* mechanism in CUDA 3.0 (NVIDIA, 2010, p. 38) supports binding an OpenGL texture directly to a CUDA array using `cudaGraphicsGLRegisterImage()`, removing the issues. However, for our tests this functionality was not yet available, so we had to use the intermediate copy. To prevent the issue from distorting our measurements, we did not include the time needed for the copy operation when timing CUDA-based raycasting, but only measured the runtime of the kernel. This allows a sensible comparison to a shader implementation, especially since the issue is resolved in newer versions of CUDA.

After creating and converting the ray parameter textures, the raycasting kernel is started with the chosen thread block size, with each thread in the block corresponding to a single ray. Following the scheme illustrated in Figure 6.2, the kernel first performs ray setup using the ray parameter buffers before entering the main loop. Inside the loop the texture fetches are performed and lighting calculation is applied before compositing the intermediate result and advancing the current position on the ray. When the end of a ray is reached, the fragment color is written to an output buffer mapped to a PBO. It is copied to the screen when processing of all thread blocks has completed.

If early ray termination is active, the main loop is terminated before reaching the ray end when the compositing results in an opacity value above a certain threshold. Since all threads in a warp operate in lock step, the thread has to wait for all the other rays to terminate by either reaching their end or through ERT. This is a hardware limitation, and therefore it also applies to the fragment shader implementation. In practice, however, this is of no concern, as neighboring rays usually exhibit a coherent behavior with regard to both ray length and ERT.

The CUDA implementation shown in Appendix A.2 (p. 110) is quite similar to the previously discussed shader program. It also uses the preprocessor to disable or enable optional features in the raycasting process. However, as the CUDA code is preprocessed at application compile time, this simple approach does not allow dynamic reconfiguration at runtime, unlike with the shader implementation. Besides code differences caused by the different APIs, some optimizations are specific to the CUDA kernel. When loading the ray parameters from global memory in lines 80 and 81, four floats are read, although only three are actually needed. This is done to comply with the coalescing rules and increases the overall throughput even though more data is transferred—because fewer memory transactions are required. Hence, it becomes clear that a CUDA implementation has more potential for low-level optimizations compared to a fragment shader.

6.4 Slab-based Raycasting

6.4.1 Slab-based approach

Since the bricking described in Section 6.2.3 is an object-order technique that is not well suited for a CUDA implementation because of the irregular memory pattern, we introduce an alternative caching mechanism that can be used in image-order by dividing the volume into *slabs*. In contrast to bricking, rays instead of voxels

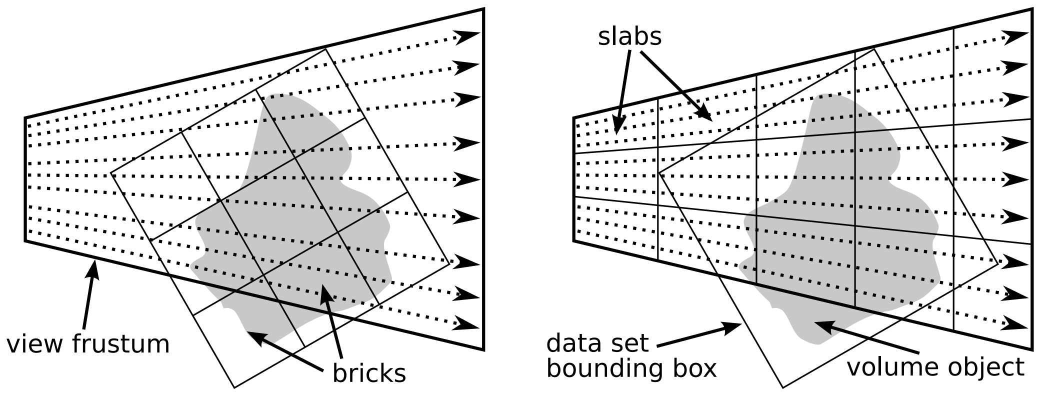


Figure 6.3: Bricking (object-order) and slab-based (image-order) approach for volume raycasting.

are grouped to build a slab. The screen is subdivided into rectangular regions and stacked slabs reaching into the scene are created, as shown in Figure 6.3. While for orthogonal projection the structure of a slab is a simple cuboid, it has the form of a frustum for perspective projection.

It would be optimal to move all voxels contained in a slab into shared memory. But unlike bricks, slabs are neither axis-aligned in texture space nor do they have a simple cuboid structure. Therefore either a costly addressing scheme would be required, or large amounts of memory would be wasted when caching the smallest axis-aligned cuboid enclosing the slab. As described in Section 6.2.3, both alternatives are not suitable for a CUDA implementation. However, a more regular structure can be found after voxel sampling. All rays inside a slab have approximately the same length and therefore the same number of sample points. Saving the voxel sampling results for all rays in a slab leads to a three-dimensional array which can easily be stored in shared memory.

Caching these data does not give a performance advantage per se, when samples are accessed only once. But several lighting techniques, e. g., ambient occlusion or even basic gradient calculation, need to access neighborhood voxels regularly. When these techniques access the same sample position multiple times, memory bandwidth and latency are saved. Unfortunately, the relation between adjacent samples in the cache is somewhat irregular, as rays are not parallel when applying perspective projection, and therefore the distance between sample points differs. However, often not the exact neighborhood of a voxel is needed by visualization techniques but an approximation is sufficient. For large viewport resolutions adjacent rays are close to parallel even with perspective projection, hence for approximation purposes one can consider them as parallel. Gradient calculation can then use the same simple addressing scheme as known from conventional raycasting to access neighboring voxels, although in this case the resulting gradients are relative to the eye coordinate system instead of the object coordinate system. While relying on an explicitly managed cache in shared memory, this method also makes use of the implicit cache of the texturing hardware as discussed in Section 6.2.2 when sampling the voxels that get written into shared memory. Hence, these two cache levels complement each other. As the access pattern for filling the cache is the same as for normal raycasting, the technique also maintains the locality of texture access, which was found to be an important factor for optimal performance in Section 6.2.2.

The slab-based approach is somewhat similar to packet-based traversal schemes for raycasting and raytracing of geometric data (Wald et al., 2001), where rays are grouped into packets to take advantage of coherence and allow the use of SIMD instructions and packet-based culling. But unlike for geometric data, the simple fragment shader implementation of volume raycasting already offers a high degree of coherence and uses the SIMT hardware efficiently, while culling is not applicable for volumetric data in general. The advantage introduced by the slab-based approach is an explicit voxel cache that can improve coherence for data intensive calculations such as gradient computation.

6.4.2 CUDA implementation

Just as with the implementation of the basic raycasting algorithm, also for the slab-based raycasting each thread corresponds to a ray and ray setup is performed through the ray parameter textures. However, the start points must be adapted for the slab structure, as described below. The main loop traverses the rays through the slabs, calculating the gradients using the cache in shared memory. Special handling is necessary for border voxels and for early ray termination.

Start point preprocessing. The slab algorithm relies on the fact that voxels are sampled by an advancing ray front and that sample points that are adjacent in texture space also lie close together in the cache. This only holds true as long as the view plane is parallel to one side of the proxy geometry cube, as otherwise ray start positions have different distances to the camera. This would result in incorrect gradients, since voxels adjacent in the volume may lie on different slices in the slab cache. A solution to the problem is modifying all ray start points in a slab to have the same distance to the camera as the one closest to the camera, as illustrated in Figure 6.4a. We use shared memory and thread synchronization to find the minimum

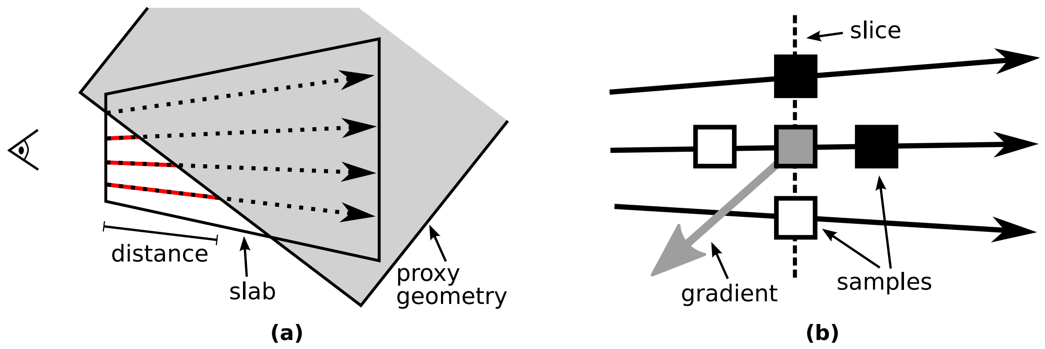


Figure 6.4: (a) Start point preprocessing modifies the ray start points in a slab so that all have the same distance to the camera. (b) Illustration of slab-based gradient calculation in eye-space, which uses adjacent samples on the current slice as well as the next and previous sample on the ray.

camera distance over all rays in a block and then move the start point of each ray to have this minimum distance to the camera. The CUDA API offers the `atomicMin()` function, which would seem ideal for this application. However, in our tests it sometimes gave incorrect results, it is not available on all CUDA platforms, and it also did not give a performance advantage compared to manual synchronization for our use case. Moving the start points does not lead to additional texture fetches, as the texture coordinates will lie outside of the interval $[0,1]^3$, which is checked before each 3D texture fetch. This check is not only needed for performance reasons but also for correctness, as CUDA only supports standard clamp-to-edge and wrapping as texture wrap modes, but no clamping to a predefined border color. Hence, without the check artifacts could appear.

Main loop. The main rendering loop consists of two parts. In the first part, the slab cache is filled with samples by traversing the ray. As a ray typically creates too many samples to fit in the slab cache completely, the slab depth *sd* controls the number of samples to write into the cache per ray at the same time. Samples with the same distance to the camera lie on the same *slice* in the slab. After thread synchronization the second part of the main loop uses the recently acquired samples to apply lighting and compositing. The ray traversal is started from the beginning for the slab, but now the samples are read directly from shared memory instead of the texture. Due to the regular access pattern this can be done without getting bank conflicts. The main loop is executed for all slabs corresponding to the current thread block.

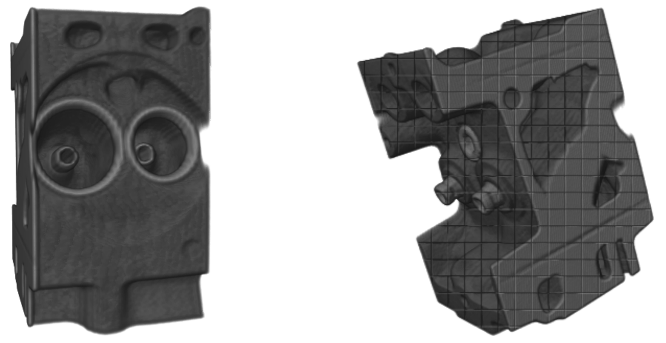


Figure 6.5: (a) Phong shading applied to the engine data set with gradient calculation. (b) The grid pattern of the thread blocks becomes visible through incorrect gradients when border handling is not performed.

Gradient calculation. A gradient is calculated by taking into account adjacent samples on the same slice from the top, left, bottom, and right rays (seen from the view point), and the next and previous samples on the current ray, as illustrated for the 2D case in Figure 6.4b. The gradients are therefore calculated in eye space and need to be transformed to object space for the lighting calculation. Since the adjacent samples are read directly from the slab cache, this gradient calculation requires no additional texture fetches, in contrast to the standard approach. When comparing the results to that of the standard gradient calculation, slight differences can be noticed, because different neighboring voxels are used. Still, the gradients of the slab-based calculation lead to correct rendering results when applying Phong lighting, as shown in Figure 6.5a. We have not conducted a formal study comparing the results of both approaches, as for most visualizations the accuracy of gradients is not of concern. The lighting models incorporating gradients are used mainly to support spatial comprehension of the data, for which our gradients proved to be sufficient.

Border handling. As with bricking, accessing the neighborhood of samples on the border of a slab requires special handling. This is necessary for gradient calculation, because ignoring voxels not accessible through the cache for gradient calculation leads to discontinuities in the gradients, which get visible as a grid pattern in the final image (Figure 6.5b). Directly accessing surrounding voxels would require retrieving additional ray parameters for rays outside the slab to calculate the relevant voxel positions. Hence, including the voxels into the cache is more reasonable, even if this reduces the usable cache size. We also tested accessing surrounding voxels directly, but this did not lead to a better performance. Instead, to include surrounding voxels into the slab, we added all adjacent rays. For these border rays only the first part of the main loop needs to be executed, in order to write the corresponding samples into the cache for access by the gradient calculation of the inner rays in the second part.

Early ray termination. As data sampled by one ray is also used for the gradient calculation in adjacent rays, early ray termination cannot stop the traversal of a single ray without taking its neighbors into account. Therefore it must be determined whether all rays in a slab have reached the required opacity threshold before further ray traversal can be terminated. The warp voting functionality made available with CUDA cannot be used, as an entire block has to be taken into account, not only a warp. However, the necessary synchronization can be easily performed using a flag in shared memory.

6.5 Integration into Voreen

We integrated the different CUDA-based raycasting implementations into Voreen, where they can use the existing proxy geometry and corresponding ray start and end point textures generated with OpenGL. As Voreen did not yet include support for CUDA, some adaptations of the volume handling were necessary in addition to implementing new raycasting processors.

6.5.1 Volume handling

One limitation of CUDA is that it can not directly use OpenGL textures but requires the volume data to be available as a CUDA array instead. As the volume handling in Voreen previously relied exclusively on an OpenGL representation implemented in the class `VolumeGL`, we had to extend the volume handling to support CUDA arrays as well. We utilized a flexible approach that allows simple addition of further hardware representations of volume data.

Volumetric data in Voreen is managed by the classes `Volume` and `VolumeHandle`. Using the *decorator pattern* (Gamma et al., 1995), we extended the basic `Volume` class to support several different hardware-specific representations of the volume data (Figure 6.6). The representation is either generated explicitly by calling `generate-HardwareVolumes()` with the appropriate parameter or implicitly when an accessor method such as `getVolumeGL()` is called. The implicit creation of the hardware-specific representation when it is accessed for the first time allows the volume handling to refrain from requiring any knowledge about the data-flow network. The

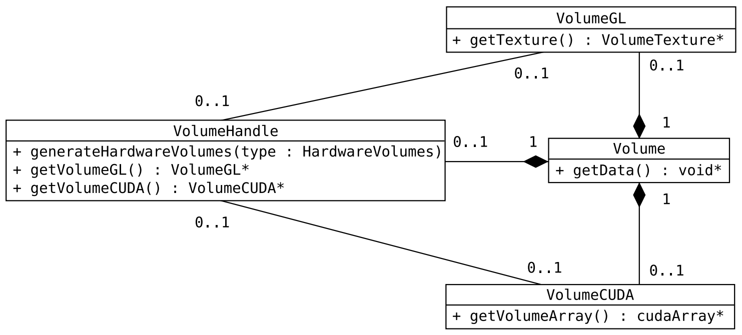

VolumeGL
+ getTexture() : VolumeTexture*
VolumeHandle
+ generateHardwareVolumes(type : HardwareVolumes)
+ getVolumeGL() : VolumeGL*
+ getVolumeCUDA() : VolumeCUDA*
Volume
+ getData() : void*
VolumeCUDA
+ getVolumeArray() : cudaArray*
0..1
1

Figure 6.6: Overview of the classes for managing volumes and their OpenGL and CUDA hardware representations in Voreen.

representation is created only when a processor inside the network requests it. In a network containing both a shader-based and a CUDA-based raycaster, both an OpenGL texture and a CUDA array would be created automatically. Usually only one of the types would be requested in a typical network, but besides from the doubled memory requirements, there is no reason to disallow multiple hardware representations.

`VolumeCUDA` encapsulates a `cudaArray` into which the data of the associated `Volume` is copied. The CUDA array is created as a three-dimensional array and filled with the data using `cudaMemcpy3D()`. To speed up the data transfer, the volume data could be copied into page-locked memory before uploading to the GPU, which would allow for slightly higher throughput and running the upload in the background while another CUDA kernel is active. However, for static data the upload needs to be performed only once, so there is only little potential for an overall performance gain. In addition, the cost of copying the data into a page-locked part of system memory would introduce additional overhead—if possible at all, since page-locked memory is a scarce resource that might not suffice for large data sets.

6.5.2 Network integration

Since we can use the proxy geometry and ray parameter textures generated using OpenGL, only the processor that implements the actual raycasting needs to be replaced for implementing CUDA-based raycasting in Voreen. Consequently, the data-flow network shown in Figure 6.7 looks very similar to a standard volume raycasting network. The new `CUDARaycaster` processor must convert the two ray parameter textures generated by `EntryExitPoints` to make them accessible to the raycasting kernel. In contrast to the ray parameter textures, the volume texture can

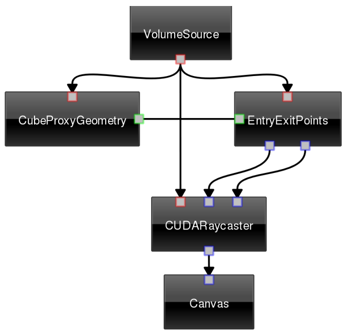


Figure 6.7: Minimal data-flow network for implementing CUDA-based raycasting in Voreen. Compared to a standard network, only the raycasting processor had to be replaced with a `CUDARaycaster`.

be accessed directly, as the `CUDARaycaster` can retrieve the `VolumeHandle` through its volume inport, from which it can request the associated `VolumeCUDA` that contains the volume texture as a `cudaArray`. Hence, the implementation of `CUDARaycaster` is limited to data initialization, converting ray parameters and transfer function texture for use by CUDA, and executing the actual raycasting kernel.

As discussed in Section 6.3.2, there are some issues when converting the ray parameter textures from OpenGL for access by the CUDA kernel. This step is necessary because the raycasting approach relies on rendering the proxy geometry to generate the information for ray setup, and using OpenGL is the natural solution for such a rendering task. Generating the ray parameter textures directly with CUDA would require implementing the complete functionality of rasterizing the proxy geometry. Kainz et al. (2009) accomplished this goal, however, not to avoid dependencies on OpenGL, but to get access to the internals of the geometry rendering, in order to allow an efficient implementation of multi-volume raycasting. Using such a CUDA-based rasterization and completely abandoning OpenGL would add no further advantage, as, for example, support for OpenGL is still much more prevalent in graphics hardware than support for stream processing APIs. Besides, visualizations of volume data are often combined with complex geometry, for which it would be pointless to replace an existing and proven graphics API with a new implementation just to use stream processing throughout the entire process. The approach taken with OpenCL, i. e., sharing data such as textures or vertices between the graphics and stream processing subsystems, seems more reasonable. The same can be achieved using CUDA, at least with newer revisions. Hence, we combine a single CUDA-based raycasting processor with OpenGL-based processors for rendering the proxy geometry or adding a background image.

6.6 Results

6.6.1 Testing methodology

In order to obtain meaningful performance data for comparing CUDA and fragment shader raycasting, we have implemented feature-identical versions of a raycaster for both cases, using CUDA version 2.1 and OpenGL shaders implemented in GLSL, running on Linux. To get comparable results, our measurements were confined to the actual fragment shader or kernel call, not counting time for rendering the proxy geometry or converting textures from OpenGL to CUDA format. Tests showed that rendering of the proxy geometry and further initializations take less than 5% of the total rendering time. The CUDA kernels were timed using the asynchronous event mechanism from the CUDA API, while for shader raycasting a high-precision timer was used, enclosing the shader execution with calls of `glFinish()` to ensure correct results. Each volume object was rotated around the Y-axis while measuring the average frame rate when rendering 100 frames. The tests were conducted on two different systems, one equipped with an Intel Core 2 Duo E6300 CPU and an NVIDIA GeForce 8800 GT, the other with a Core 2 Quad Q9550 and a GeForce GTX 280.

To obtain a better insight into the performance characteristics of the different parts of a full raycasting implementation, we tested with several variants of the raycaster that implement a subset of the complete functionality. We started with the minimal raycasting algorithm performing a simple ray traversal (*RC*), which directly maps voxel intensity to luminance. In the next step a transfer function was added (*TF*), which uses a one-dimensional texture as a lookup table. Adding Phong lighting with on-the-fly gradient calculation using central differences and early ray termination resulted in a full-featured raycaster (*PH*). Finally, as an example for an expensive technique, we added gradient filtering, which computes central differences for all surrounding voxels and averages the result (*GF*). Table 6.1 lists the number of texture

technique	regs	fetches
basic raycasting (RC)	15	1
transfer function (TF)	19	2
Phong shading (PH)	33	8
gradient filtering (GF)	57	56

Table 6.1: CUDA register usage and number of texture fetches per sample point for the different raycasting techniques. The gradient filtering requires a large number of texture fetches, since it computes the gradient for all voxels surrounding the sample point and averages the result. This also leads to excessive register requirements.

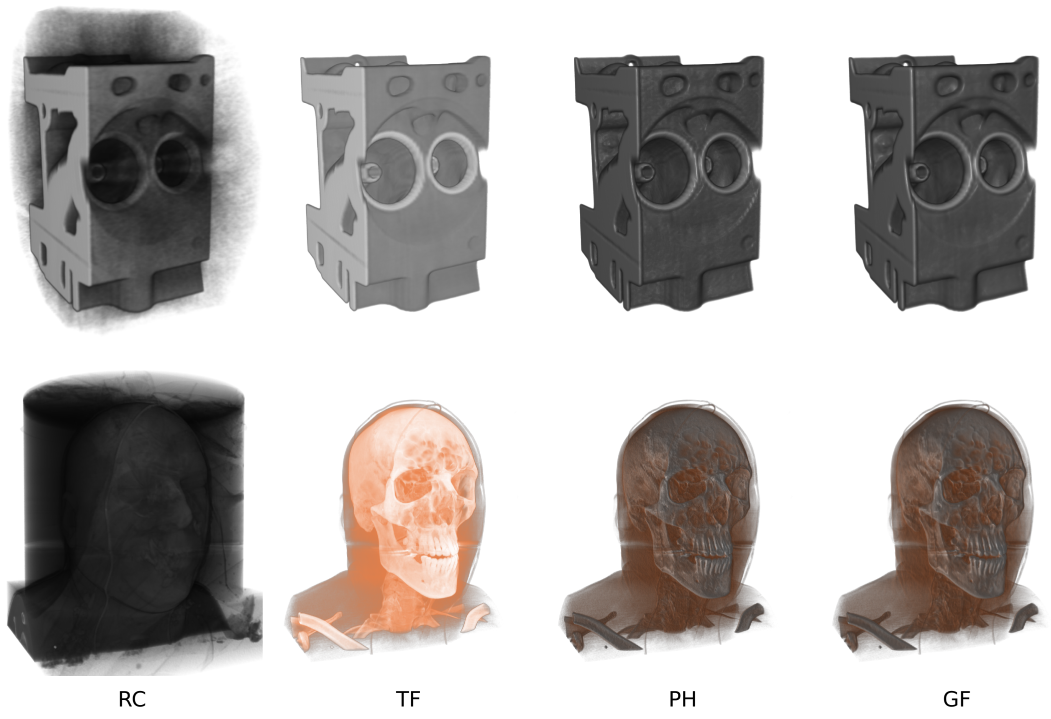


Figure 6.8: Results of rendering the *engine* and the *vmhead* data sets with different raycasting techniques.

fetches per sample point for the individual techniques. Advanced techniques include all previous features, e. g., Phong lighting includes a transfer function.

We have tested our implementations with several data sets and chose two representative volumes of different sizes and with different transfer functions for comparing the different techniques. The *engine* is dense with hardly any transparency, while the larger *vmhead* is semi-transparent. Renderings of the data sets with the different techniques are shown in Figure 6.8.

6.6.2 Basic raycaster

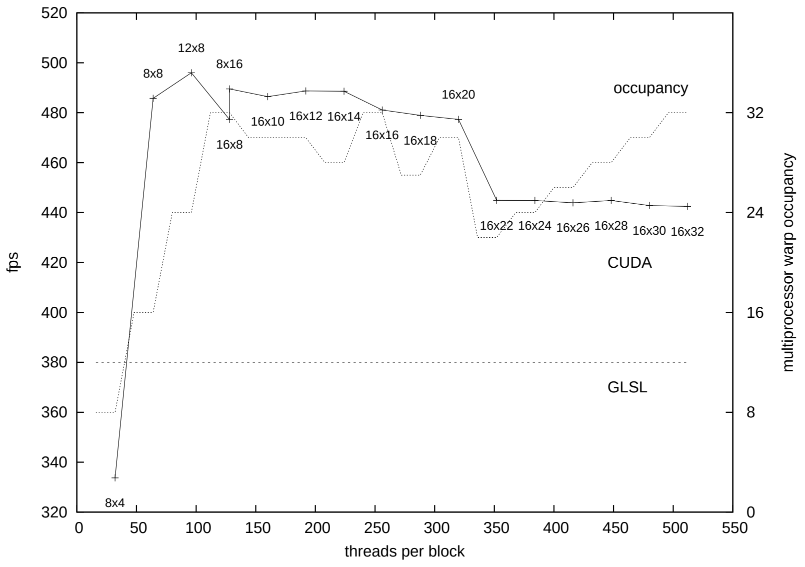
Table 6.2 lists frame rates of our basic raycasting implementations, tested with different GPUs, viewport resolutions, and data sets. It is notable that the GeForce GTX 280 achieves significant speedups for the CUDA implementation for all techniques except *PH*, while with the 8800 GT significant speedups are only found with the *RC* technique and 1024^2 viewport size, the GLSL implementation being close to equal or faster for all other cases. The frame rate differences between GLSL and CUDA reach

		view-	*engine* data set ($256^2 \times 128$, 8 bit)				*vmhead* data set ($512^2 \times 294$, 16 bit)			
techn.	device	port	GLSL	CUDA	speedup	bs_{opt}	GLSL	CUDA	speedup	bs_{opt}
RC	8800GT	512^2	291.1	300.4	+3.2%	16×8	72.2	64.1	−11.2%	16×20
		1024^2	81.0	96.9	+19.6%	16×32	48.3	56.3	+16.6%	16×28
	GTX280	512^2	380.0	496.0	+30.5%	12×8	121.2	158.5	+30.8%	8×8
		1024^2	124.5	147.8	+18.7%	16×8	70.5	100.2	+42.1%	16×18
TF	8800GT	512^2	194.2	173.5	−10.7%	8×16	68.1	59.4	−12.8%	8×16
		1024^2	61.1	62.3	+2.0%	16×24	38.2	37.4	−2.1%	16×24
	GTX280	512^2	317.4	358.1	+12.8%	16×12	118.0	153.9	+30.4%	8×16
		1024^2	100.9	110.6	+9.6%	8×16	64.6	82.7	+28.0%	16×16
PH	8800GT	512^2	60.2	43.6	−27.6%	8×8	21.5	22.0	+2.3%	8×16
		1024^2	17.1	14.6	−14.6%	16×12	12.0	9.9	−17.5%	16×12
	GTX280	512^2	95.2	77.6	−18.5%	8×8	40.7	38.1	−6.4%	8×16
		1024^2	25.5	22.5	−13.3%	16×8	18.0	17.2	−4.4%	16×8
GF	8800GT	512^2	8.9	6.7	−24.7%	8×16	4.6	3.4	−26.1%	8×16
		1024^2	2.5	2.1	−16.0%	8×16	1.7	1.6	−5.9%	8×16
	GTX280	512^2	9.5	10.4	+9.5%	8×8	4.6	5.6	+21.7%	12×8
		1024^2	2.5	2.9	+16.0%	8×8	1.8	2.3	+27.8%	8×8

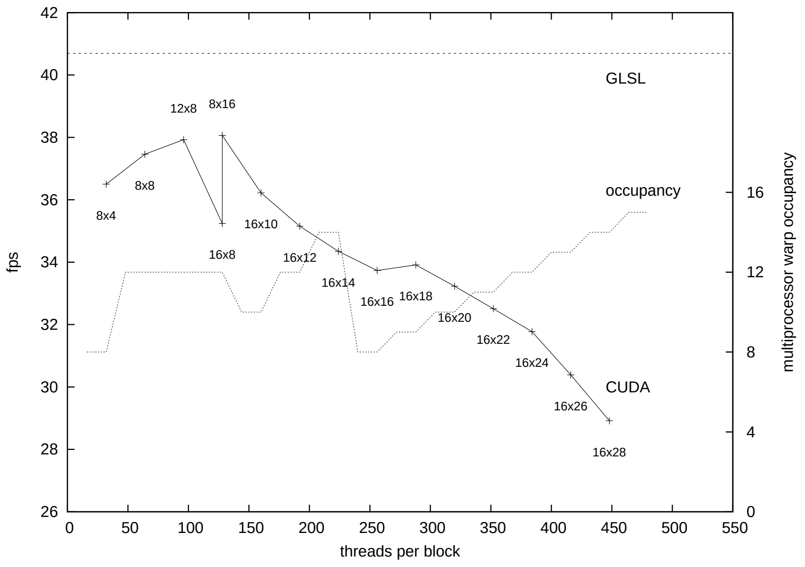
Table 6.2: Performance results in frames per second for basic raycasting implemented with GLSL and CUDA. The CUDA raycasting was run with different block sizes, and results for the optimal block size bs_{opt} are given.

up to 30%, with one outlier even at +42%. For the 8800 GT increasing the viewport size also increases the speedup, while the speedup for the 280 GTX mostly stays the same. Switching from the *engine* data set to the larger *vmhead* increases the speedup for the 280 GTX, while this is less significant for the 8800 GT.

As the selection of thread block size can have a tremendous influence on the performance of a CUDA kernel, we tested all raycaster variants with several block sizes to find the optimal block size bs_{opt}. This was achieved using a script that automatically configures and compiles the CUDA kernel and shader code for each test run and then executes the benchmark. Figure 6.9 shows the effect of the block size on the frame rate when using between 32 and the maximum of 512 threads per block. The diagrams also include the multiprocessor warp occupancy for each block configuration, i. e., the number of active warps per multiprocessor. Maximizing the active warps up to the maximum of 32 is recommended for optimal hardware utilization (NVIDIA, 2010, p. 83), and some correlation between occupancy and frame rate is visible in the diagrams. The frame rate follows the warp occupancy up to a block size of 16×22 for *RC*, but increasing the block size further has no influence on the frame rate. For *PH* no similar behavior is visible and also the warp occupancy only reaches half of the maximum of 32, as the high register requirements limit the number of concurrent block executions.

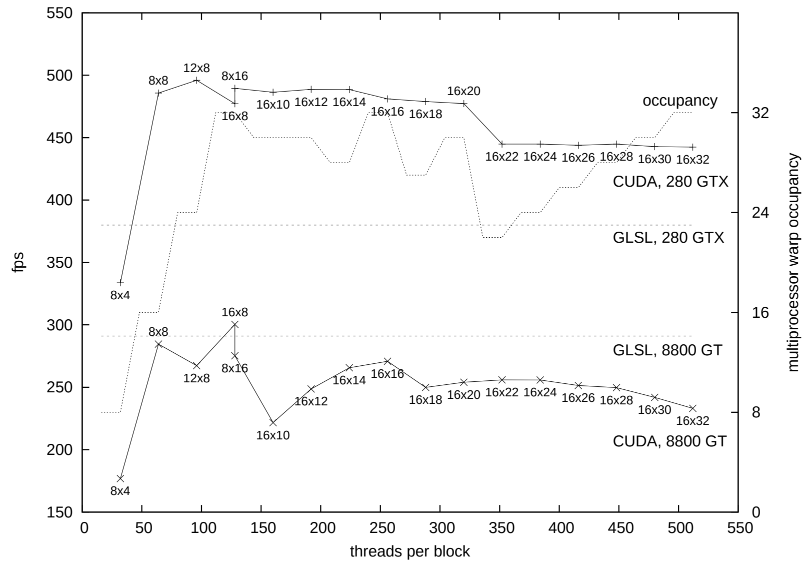


(a) engine data set, basic raycasting (RC)

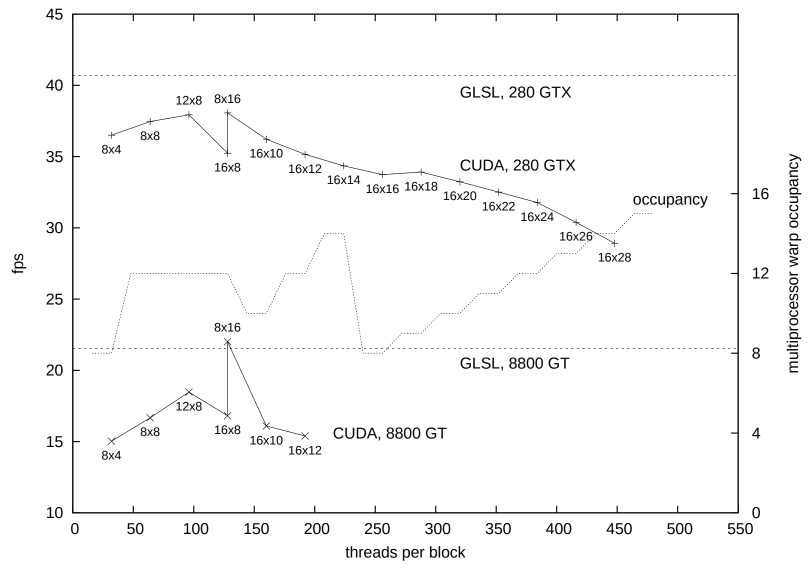


(b) vmhead data set, Phong lighting (PH)

Figure 6.9: Influence of block size on rendering performance, rendered on a GeForce GTX 280, viewport size is 512^2. The frame rate obtained with a GLSL fragment shader is included for reference as a horizontal dashed line.



(a) engine data set, basic raycasting (RC)



(b) vmhead data set, Phong lighting (PH)

Figure 6.10: Comparing the influence of block size between a GeForce 8800 GT and a GeForce GTX 280, viewport size is 512^2. With the more complex PH kernel, the number of thread blocks is limited by the high register requirements of the kernel. Therefore the maximum block size possible on the 8800 GT is only 16×12, in contrast to the simpler RC kernel, for which the maximum of 16×32 = 512 threads can be used.

When choosing a single general block size for CUDA raycasting, we would recommend 8×16, as it gives close to optimal results for most of the observed cases. As an alternative to a static block size, an auto-tune approach could be used in an application, i.e., automatically testing different block sizes at runtime and measuring the rendering times to find the optimal configuration for each practical use case. However, this would increase both complexity and performance overhead, and therefore would only be justified when a static block size proves to be insufficient.

While the advanced raycasting techniques are more costly since they perform more texture fetches, they also require more hardware registers to run (compare Table 6.1). Due to the limited availability of hardware registers, this restricts the number of active thread blocks per multiprocessor. The GTX 280 has twice as many registers available as the 8800 GT and therefore allows larger block sizes for kernels that use many registers. It is notable that for gradient filtering (*GF*), with both a very high register count and a large number of texture fetches, the GTX 280 can achieve a significant speedup, while it was slower than GLSL for *PH*. Figure 6.10 compares the effect of block size between the two tested graphics processors. The different processors show a roughly similar behavior with regard to block size, albeit on a different overall performance level. The high register requirements of the *PH* kernel are a limiting factor on the 8800 GT, hence the block size is limited to 16×12 for this kernel. However, looking at the other results, this does not directly limit the maximum frame rate on this GPU, which would probably still be achieved for the block size of 8×16 even if more registers would be available.

6.6.3 Slab-based raycaster

We tested slab-based raycasting on the GTX 280 only, as this GPU proved to be influenced less by high register requirements. The shared memory cache contains $bs_x \times bs_y \times sd$ sampled voxels, depending on the thread block size bs and the slab depth sd. The optimal slab depth sd_{opt} depends on the data set, just as the block size. Results of the slab-based raycaster are presented in Table 6.3. We added an intermediate viewport size of 768^2 for this test to be able to better analyze the connection between viewport size and speedup factor.

For the basic *RC* technique each sampled voxel is accessed only once, hence caching the slabs cannot improve performance. However, this allows us to measure the overhead for managing the shared memory cache and for fetching additional border voxels. For the tested configurations the overhead is between 23% and 67%. When applying Phong lighting, volume data is accessed multiple times by the gradient calculation, and the slab caching can result in a significant speedup compared to the

			engine data set ($256^2 \times 128$, 8 bit)				
technique	regs	viewport	basic	slab	speedup	bs_{opt}	sd_{opt}
RC	22	512^2	496.0	186.4	−62.4%	8×16	31
		768^2	251.6	85.7	−65.9%	16×16	31
		1024^2	147.8	49.2	−66.7%	8×16	31
PH	34	512^2	77.6	77.1	−0.6%	8×16	31
		768^2	36.2	35.2	−2.8%	16×16	31
		1024^2	22.5	19.6	−12.9%	8×16	31

			vmhead data set ($512^2 \times 294$, 16 bit)				
technique	regs	viewport	basic	slab	speedup	bs_{opt}	sd_{opt}
RC	22	512^2	158.5	122.0	−23.0%	16×14	16
		768^2	131.1	74.1	−43.5%	16×14	16
		1024^2	100.2	43.4	−56.7%	16×30	16
PH	34	512^2	38.1	67.9	+78.2%	16×30	16
		768^2	27.1	34.1	+25.8%	16×30	16
		1024^2	17.2	19.5	+12.7%	16×30	16

Table 6.3: Performance results for the CUDA implementation of slab-based raycasting on a GeForce GTX 280. Note that the RC technique is only used to measure the overhead of the slab-based approach.

basic CUDA raycasting. A performance increase between 12% and 78% is found only with the large 16-bit *vmhead* data set, presumably since the hardware texture cache is less efficient with larger volumes, as for the 8-bit *engine* data set a slight performance decrease is measured. Another reason might be that the early ray termination is less efficient with the slab approach, since only complete slabs can be terminated, not individual rays. The *engine* is solid and the rays are terminated much earlier than with the semi-transparent *vmhead*. It is notable that the speedup decreases with increasing viewport size. When the viewport gets larger, adjacent rays hit the same voxels more often. Hence, there is more locality of texture fetches resulting in more hits in the hardware texture cache. However, the slab cache is most efficient in the opposite case, when the data set resolution is high compared to the viewport size.

The amount of shared memory required by the raycasting kernels depends on block size and slab depth. For *vmhead* the optimal configuration results in 15,360 bytes which is close to the maximum of 16 kB, hence only one thread block can be active per multiprocessor. Although *engine* only uses up to 7,936 bytes, this does not result in more concurrent thread blocks because of the high register requirements. Nevertheless, this configuration is faster than one with a smaller block size, which would allow multiple concurrent thread blocks. This shows that the stream processing approach is not fully effective in hiding the latency of the large number of texture fetches performed by the raycasting algorithm, as running more thread blocks concurrently results in no performance advantage in this case.

6.6.4 Discussion

Performance evaluation

The number of required hardware registers seems to be a major factor influencing kernel performance compared to a feature-equivalent shader implementation. Maršálek et al. (2008) reported saving registers by moving long-living variables into shared memory. In our tests, however, moving variables into shared memory did not reduce the number of registers allocated by the CUDA compiler. It is possible that they used an earlier version of the compiler that was less efficient in register optimization. Since shaders receive the same benefits as CUDA kernels from the double bandwidth and twice the number of scalar processors of the GTX 280 compared to the 8800 GT (see Table 2.1, p. 8), we suspect that the reason for the larger speedups for the CUDA kernels achieved with the GTX 280 is its support for more hardware registers.

Hence, it seems that our CUDA implementation is less efficient in utilizing hardware registers than shaders are, therefore profiting more when more registers are available. This claim is supported by results recently published by Smelyanskiy et al. (2009), who implemented a CUDA-based raycaster to compare it to an implementation on a simulated Larrabee processor architecture. They compared their results to our initial findings (Mensmann et al., 2009) and to our more detailed results, which we shared through personal communication. For their tests, they modified our raycasting fragment shader in Voreen to match the features of their CPU-based reference implementation. However, their CUDA implementation was developed independently from ours. They report that their fragment shader implementation is between 1.12x and 1.26x faster than their CUDA implementation. Hence, they obtained results similar to ours, which makes it unlikely that the discrepancy in performance between CUDA and shader implementations can be explained purely by inefficiencies or errors in our implementation. In their discussion Smelyanskiy et al. conclude that both CUDA and shader-based implementations of raycasting are comparable in performance, which coincides with our results. They also propose that the performance differences might be caused by empty space skipping in our shader implementation that takes advantage of hardware rasterization, which is not possible in their CUDA raycaster. However, empty space skipping as discussed in Chapter 5 is not automatically available in Voreen through the use of the Krüger-Westermann proxy geometry approach. Therefore we disagree with this explanation.

The slab-based raycasting can increase rendering efficiency when the same volume data is accessed multiple times, e. g., for gradient calculation. However, it should be noted that the algorithm can be compared to the basic raycasting only to a certain extent, as the gradients are less exact for the slab data. Nonetheless, the results show how much of a difference the use of shared memory can make. We demonstrated that the method is most efficient for high resolution data sets. This is advantageous for typical applications of volume rendering, e. g., medical imaging, where data sets typically have a much higher resolution than *engine* and improvements in scanner technology are constantly increasing the quality of data sets. The algorithm is also more efficient with semi-transparent than with non-transparent data. For data with no transparency this is not a real issue as well, as in this case also simpler techniques such as isosurface rendering could be used. Our slab-based method is designed for use with direct volume rendering, which is most useful for semi-transparent data.

Getting good global memory coalescing is often described as one of the most important optimizations for CUDA kernels (e. g., NVIDIA, 2010, p. 85). For our raycasting kernels, coalescing did only have a minor effect on the overall performance, as global memory is used just for reading of the ray parameters and writing of the raycasting results. The vast majority of memory accesses use the texturing hardware, which is not affected by coalescing issues.

Further observations

A practical result we observed when testing the different kernel configurations is that the integration of the GLSL compiler into the graphics driver compared to the standalone CUDA compiler is an advantage for visualization tasks. Compilation at runtime is not supported by CUDA, but is the default mode of operation for GLSL and also OpenCL. With many different configuration options that need to be set at runtime, it is advantageous to control these using the preprocessor. This approach is also used extensively for implementing the complex raycasting shaders in Voreen (Meyer-Spradow et al., 2010). To achieve the same with CUDA, multiple kernel variants would need to be defined. Newer CUDA releases have added support for C++ templates, which partly addresses the problem. While CUDA includes a mechanism for just-in-time compilation (JIT) since version 2.1 (NVIDIA, 2009), this only works on device code in PTX assembly form (NVIDIA, 2010, p. 16) and is not an alternative to full runtime compilation.

Comparing frame rates of the two most simple versions of the raycasting, *RC* and *TF*, a significant performance difference can be noticed, although the kernels differ only in one additional texture lookup in the one-dimensional transfer function texture.

Even though this doubles the number of texture accesses, we would have expected a smaller decrease in frame rate, as the transfer function texture is relatively small and should support efficient caching. It seems that the access pattern of the transfer function texture together with the additional volume texture accesses prevents a better utilization of the texture cache, leading to the overall speed penalty compared to *RC*. Hence, as an optimization for certain applications it should be analyzed whether the transfer function can be represented analytically, and if the transfer function could then be evaluated directly instead of using a lookup table. As the performance of the raycasting kernel is mainly limited by texture access, additional computations could be added to the kernel without significantly limiting its performance. Even when no analytical representation of the transfer function is available, a discrete representation as a lookup table might fit into shared memory, effectively replacing the use of the texturing hardware.

Our tests also showed that texture filtering—although performed by dedicated hardware—comes not totally for free. When we switched from filter mode *linear* to *nearest* for the CUDA volume texture, the frame rate increased slightly by about 4% to 10%. This overhead, however, is much less than what would be expected when doing the filtering manually in the CUDA kernel. Manual texture filtering can be compared in complexity to gradient computation. Although the *PH* raycaster also includes Phong lighting calculation in addition to computing the gradient, the performance difference between *TF* and *PH* can be used as an indicator for the performance of manual filtering. Frame rates acquired with *PH* are nearly five times slower than those for *TF*. Hence, the overhead of up to 10% for the hardware texture filtering is acceptable in comparison.

6.7 Summary

We have demonstrated that the CUDA programming model is suitable for volume raycasting and that a CUDA-based raycaster—while not a "silver bullet"—can be more efficient than a shader-based implementation. Factors influencing the speedup are the type of GPU, thread block size, and data set size. We have also shown that using shared memory can bring a substantial performance increase when the same volume data is accessed multiple times. However, hardware restrictions need to be taken into account, as managing the shared memory and especially handling border voxels can introduce a significant overhead. We have demonstrated that a direct translation of a visualization algorithm from a fragment shader to a CUDA kernel does not guarantee the same performance results, and that an adaptation to the specific features of the new programming model is necessary.

The amount of information contained in a single voxel is relatively small, therefore advanced visualization techniques take surrounding voxels into account to improve expressivity of visual results. Hence, memory bandwidth becomes the limiting factor for raycasting, as no complex calculations are applied and the compute intensity, i. e., the ratio of arithmetic operations to volume texture fetches, is rather low. Caching brings some improvements, but the overall access pattern for the ray traversal is quite complex with a low level of data locality.

Other factors besides rendering performance should also be taken into account when choosing a programming model for a raycasting application. Currently a shader implementation will support a wider range of graphics hardware and does not depend on a single vendor. Also the integration into existing volume rendering frameworks is easier, e. g., by being able to directly use 2D and 3D textures and render targets from OpenGL. Many of these issues will hopefully be removed by implementations of the OpenCL standard, which is vendor-neutral and supports better integration with OpenGL.

As future work it should be investigated whether more complex visualization techniques, such as ambient occlusion, can benefit from the additional hardware resources accessible through stream processing APIs or upcoming hardware architectures such as Larrabee. The slab-based approach might prove valuable for exploiting these resources and efficiently utilizing them for visualization tasks. As the currently available on-chip memory is a scarce resource, particularly for storing volume data, volume rendering would especially benefit from improvements in this area, which are expected for future hardware. Finally, object-order volume rendering techniques could be evaluated for implementation on stream processing hardware, as their regular memory access patterns might use the on-device cache memory more efficiently than image-order raycasting.

Chapter 7

Lossless Compression for Rendering Time-Varying Volume Data

Since the size of time-varying volumetric data sets typically exceeds the amount of available GPU and main memory, out-of-core streaming techniques are required to support interactive rendering. To deal with the performance bottlenecks of hard-disk transfer rate and graphics bus bandwidth, we present a hybrid CPU/GPU scheme for lossless compression and data streaming that combines a temporal prediction model, which enables us to exploit coherence between time steps, and variable-length coding with a fast block compression algorithm. This combination becomes possible by taking advantage of the CUDA computing architecture for unpacking and assembling data packets on the GPU. The system allows near-interactive frame rates on desktop hardware even for rendering large real-world data sets with a low signal-to-noise-ratio, while not degrading image quality. It uses standard volume raycasting and can be easily combined with existing acceleration methods and advanced visualization techniques.

Improvements in programmable graphics hardware have made interactive volume visualization possible for data from many different domains such as medicine or seismology. While the resolution of these data sets is constantly increasing due to advancements in acquisition technology, graphics processors could keep up with the increasing amount of data by means of boosting computation performance and graphics memory. GPU-based raycasting can easily achieve interactive frame rates for large data sets even without including optimization algorithms. However, this only holds true as long as all data required by the visualization fits completely into GPU memory. When this is not possible, data needs to be streamed from system memory to GPU memory and potentially even from mass storage. The transfer bandwidth between the different levels of this memory hierarchy has not kept up with the advances in GPU performance and therefore serious performance degradations must be expected when data sets require out-of-core streaming. Time-varying volume data, i. e., series of volumes corresponding to individual time steps, can easily reach sizes in the range of gigabytes—and even more in the domain of petascale visualization. Time-varying volume data can be acquired by medical scanners, e. g., from a cardiac CT, although in this domain only a relatively low temporal resolution is used. In contrast, time-varying data sets with both high spatial and temporal resolution are routinely created in numerical simulations, especially in the fields of fluid dynamics and meteorology.

While the volumetric data resulting from large-scale simulations is often primarily intended for statistical analysis models, visualization can be essential for understanding unexpected results or spotting errors. There exist several approaches for conveying the information from multiple time steps in a single image (Meyer-Spradow et al., 2006), but they are only applicable for relatively simple data with a small number of time steps. Typically the data is visualized as a volumetric movie by rendering the time steps one after another. Precomputed animations are useful only up to a certain degree, since interactive modification of viewing parameters such as camera position and transfer function is generally seen as a necessity for visualization of volumetric data. The occlusion problem is more of an issue for volume series than for a static volume, as the camera position may need continuous updates to keep the region of interest in sight, which is not possible in a precomputed animation.

While complex simulations typically run on computing clusters or supercomputers, in practice much of the data analysis and visualization is performed on standard workstations. Often visualization is just an additional tool for analysis and not the main goal of the domain experts. Hence, no additional resources are available for data visualization, and many existing approaches—involving parallel render nodes or specialized hardware—are not applicable. To give a recent example for parallel visualization, Howison et al. (2010) were able to perform raycasting of a 4608^3 volume data set to a 4608^2 viewport in 0.56 seconds. This, however, required 216,000 compute cores of JAGUAR, a Cray XT5 system that is currently ranked as the fastest supercomputer in the world (with a Linpack performance of 1.76 petaflops according to the June 2010 TOP500 list).

To allow interactive rendering of large time-varying volumetric data sets on desktop machines, we must apply techniques to accelerate data streaming to the graphics processor. A common approach is to use data compression techniques in order to reduce the amount of data that needs to be transferred through bandwidth bottlenecks. Many compression techniques have been proposed for static as well as for time-varying data sets. However, lossless compression techniques for rendering time-varying data sets are rare, even though domain experts want to be able to access the original data without any loss of accuracy. Especially when considering the amount of time spent on simulations, it is of great interest to be able to inspect the data without losing information. Therefore we present a lossless compression scheme that meets these requirements and allows near-interactive frame rates even for large data sets. When visualizing data with our approach, the user can rely on the fact that all visible features are actually present in the data and do not occur due to compression artifacts.

To achieve this goal, our technique utilizes direct programming of the graphics processor through the CUDA programming interface. With this programming functionality, simple compression techniques can be brought directly to the GPU. Since typically the graphics processor is not used to capacity when data needs to be streamed, it has free resources to support this compression, which previously would have been handled exclusively by the CPU. Our hybrid approach allows the combination of several different compression methods, which are optimized for different parts of the data transfer pipeline. Thus, we are able to achieve an efficient lossless compression, which is essential to allow fast streaming. As most components of our approach work independently from each other, the lossless compression scheme can be modified by exchanging individual parts or adding further ones. It can therefore be viewed as a generic framework for combining CPU and GPU techniques to improve data throughput and rendering performance for time-varying volume data. Since decompression is performed transparently, the presented approach does not limit the visualization techniques that can be applied to the data.

The remainder of this chapter is structured as follows. In the next section we review previous work related to time-varying volume data and compressed volume rendering. Section 7.2 discusses our proposed compression scheme, while the accompanying GPU-supported decompression pipeline is described in Section 7.3. The integration of the approach into VOREEN is discussed in Section 7.4. We present our results in Section 7.5 and a summary in Section 7.6.

7.1 Related Work

There exists a multitude of approaches for compressed volume rendering and for visualization of time-varying volume data sets (Ma, 2003). They can use lossless or lossy compression, or a combination of both. A second distinction can be made between CPU-based, GPU-based, and hybrid CPU/GPU techniques.

Guthe et al. (2002) presented lossy CPU-based hierarchical wavelet decompression for rendering large volume data sets using hardware-accelerated slice rendering. Vector quantization was used by Kraus and Ertl (2002) in a GPU-based compression scheme for static and time-varying volumetric data sets. Sohn et al. (2004) described a compression scheme for encoding time-varying volumetric features to support isosurface rendering. It is based on a lossy wavelet transform with temporal encoding. A block-based transform coding scheme for compressed volume rendering using vector quantization was introduced by Fout and Ma (2007), which performs decompression on the GPU by rendering to slices of a 3D texture. Nagayasu et al. (2008) presented a hybrid pipeline rendering system for time-varying volume data. It uses a two-stage CPU/GPU decompression that combines lossy DXT/S3TC hardware texture compression on the GPU (Iourcha et al., 1999) with lossless LZO compression on the CPU (Oberhumer, 2008). Temporal coherence is exploited by packing voxels from three successive volumes into the RGB components of the compressed volume texture (Nagayasu et al., 2006), which is stored using VTC 3D texture compression (NVIDIA, 2004). Because it is based on the simple hardware compression originally intended for 2D textures, the system is limited to 8-bit scalar data and is prone to visual artifacts. While achieving interactive frame rates through a high compression ratio*, the authors report visible artifacts that could mislead the user, but assess the image quality as tolerable for time-series analysis.

Several lossless compression algorithms and prediction schemes for volumetric medical data were compared by Ait-Aoudia et al. (2006). While most of these techniques were developed for static data, Binotto et al. (2003) proposed a lossless compression approach for time-varying data using fragment shaders, based on the concept of adaptive texture maps as introduced by Schneider and Westermann (2003). It subdivides the volume into 3D blocks and replaces duplicate and homogeneous blocks by references. As this relies on exact matches between blocks, the approach is most effective for sparse data sets with a low noise level. This is rarely found with complex numerical simulation data—just quite the contrary: Artificial noise is often added intentionally to obtain a more natural behavior in simulations.

Smelyanskiy et al. (2009) used a slice-based variable-length coding to compress static volume data on the x86 and Larrabee architectures, which they report to be more effective and faster than ZLIB compression. Fraedrich et al. (2007) presented

* There seems to be no general consensus on the definition of the term *compression ratio*. We use the definition by Sayood (2000, p. 5), who defines it as the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression. For example, if a file is compressed to half of its original size, the compression ratio is 2:1, which we simply write as 2. According to the inverse definition by Salomon (2002, p. 5), the compression *factor* would be 2, while the compression *ratio* would be 1:2 or 0.5.

an implementation of lossless Huffman coding as a fragment shader that allows to store up to 3.2 times more volume data without loss of information. However, the decoding throughput of this technique lies in the range of the transmission rate of the PCI Express bus, undoing any savings achieved by the data compression, which demonstrates the difficulty of porting compression algorithms to a GPU architecture.

The GPU-based volume raycasting approach by Krüger and Westermann (2003) can be easily extended to accelerate the final rendering step also for time-varying data, especially by implementing an empty space skipping similar to our approach presented in Chapter 5. For example, Grau and Tost (2007) exploit frame-to-frame coherence in time-varying volume data to speed up rendering by preventing unnecessary raycasting.

7.2 Hybrid Compression Scheme

In this section we first discuss properties of time-varying data sets and related hardware restrictions, before introducing our lossless hybrid compression scheme and its components.

7.2.1 Data properties and hardware limitations

Volume data acquired from medical scanners is typically stored using 12-bit or 16-bit integer values. Numerical simulations, on the other hand, return floating-point data with a highly varying value range. While modern graphics processors directly support 32-bit float textures, we consider 16-bit integer data as sufficient for most volume visualization tasks. Additionally, floating point data stored in IEEE 754 format (IEEE, 2008) is not suitable as input for the standard block compression algorithms that we will discuss in Section 7.2.4. Hence, a specialized compression approach for float data would be necessary, for example, the one described by Lindstrom and Isenburg (2006) for integration into a large scale simulation cluster, which is beyond the scope of this thesis. Therefore we convert the available simulation test data from float to integer format during preprocessing, spreading the data values according to the minimum and maximum values found in the data set to make full use of the 16-bit value range. Depending on the actual application, a more elaborate mapping might be needed.

While there exist data sets with extremely high temporal as well as spatial resolution, for many applications a single time step of a time-varying data set can easily fit into graphics memory. Having the complete time step available to the GPU has several advantages for rendering, in contrast to splitting up data, e. g., into individual

data set	resolution	steps	step size	total size
convection	512×256^2	401	64 MB	11.4 GB
combustion	$448 \times 704 \times 128$	122	77 MB	25.1 GB
hurricane	$512^2 \times 128$	48	64 MB	3.0 GB

Table 7.1: Properties of the time-varying data sets tested with our approach (using 16-bit integer scalar values).

volume bricks. First, no overhead is introduced for managing several parts of the volume or for border handling. Second, availability of the complete volume allows us to use visualization and acceleration techniques that require more information than what is available in a single brick, such as for global illumination (e. g., Ropinski et al., 2010). Finally, implementation is greatly simplified. Hence, it is desirable for the decompression to assemble the data back into its original form as a single 3D texture in GPU memory. As our approach makes the uncompressed data accessible as a standard volume texture during rendering, no multi-pass bricking techniques need to be employed but standard rendering can be used. In the optimal case a volume rendering system can be extended to support such an out-of-core rendering of time-varying data by just replacing the modules for loading data from disk and uploading into a 3D texture, while the actual rendering code may stay untouched. This is an important aspect especially in the context of existing large-scale visualization systems such as Voreen. Furthermore our technique can be combined with multi-resolution approaches (Ljung et al., 2006). Since we employ a lossless compression, multi-resolution data can be compressed and streamed by using our approach without affecting its content. However, this would require additional functionality for deciding which resolution should be used for the individual parts of the volume.

One major reason for the high volume rendering performance achieved by current graphics processors is the graphics memory bandwidth. For example, an NVIDIA GeForce GTX 280 achieves 110 GB/s for an on-device copy. When data needs to be streamed from the CPU to the GPU via the PCI Express bus, the achievable throughput is more than an order of magnitude lower at 2.5 GB/s. Finally, when the data must be read from mass storage, current desktop hard drives achieve around 110 MB/s and server hard drives up to 170 MB/s. Combining several drives can improve throughput, but it is obvious that mass storage is the major bottleneck for streaming data to the GPU.

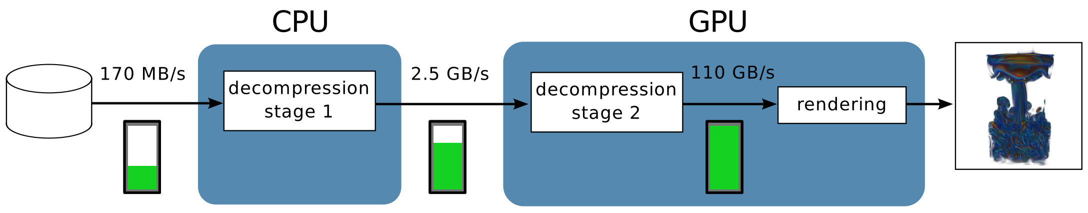


Figure 7.1: Decompression workflow of our hybrid compression scheme. Data moves through the mass storage bottleneck with the maximum compression ratio, before it is decompressed in two stages on the CPU and on the GPU. The GPU can then directly access the uncompressed data for rendering.

7.2.2 Two-stage compression approach

As the size of a single time step for typical time-varying data sets is already in the range of what a hard drive can transfer per second (compare Table 7.1), it becomes clear that any technique that aims at interactive rendering must minimize the amount of data that needs to be loaded from disk, i. e., it must maximize the compression ratio of the on-disk storage format.

It would be optimal to run the decompression completely on the GPU, as the data would then travel through both described bottlenecks, i. e., mass storage and the PCI Express bus, in compressed form. Unfortunately, the highly parallel architecture of current GPUs is not well suited for general data compression tasks. Most algorithms for data compression work in a serial fashion and show no coherent branching behavior, which does not map well to GPUs. Hence, the main decompression must be performed by the CPU. As the bandwidth between CPU and GPU is an order of magnitude larger than that of mass storage, getting maximum compression in this transfer is not as important as when loading from disk. But as decompression can use the CPU to full capacity, moving calculations to the GPU is beneficial. This is possible only for simple computations that fit into the highly parallel architecture, but even simple memory copy operations can benefit from the higher memory bandwidth on the GPU and may run much faster than on the CPU. Therefore we propose a *two-stage* or *hybrid* compression scheme, as illustrated in Figure 7.1. Data is compressed twice, first with a simple algorithm whose decompression component runs efficiently on the GPU, then with a CPU-based compression technique. Care must be taken that the output of the first compression is still suitable for the second compression step to be effective. Furthermore, an initial compression step that preprocesses data so that they can be compressed more efficiently by the second technique would be advantageous.

7.2.3 Subdividing the volume into bricks for compression

Volume data is usually stored using a simple memory layout where the two-dimensional slices that form the volume are saved one after another. Previous work often used 2D image compression techniques on these individual slices (Ait-Aoudia et al., 2006). Working with slices has the advantage that the memory format is identical to that of the final 3D texture used for rendering, but this comes at the cost of losing spatial coherence. Two voxels that are close together in volume space can actually lie far away in memory space and vice versa. This effect can be evaded by subdividing the volume into three-dimensional bricks (see Figure 7.2) and storing the contents of each brick as a continuous block in memory, hence reducing the memory range used per brick. This can improve the compression ratio of local phenomena by increasing spatial coherence and is also essential for the variable-length coding described in Section 7.2.6. It is noteworthy that the bricking scheme is only used for compression and data transfer, but not for rendering. Therefore, it does not introduce an overhead to the rendering component, but requires the bricks to be assembled back to the original 3D texture. Brick assembly is a simple operation that can be performed very efficiently on the GPU. Although OpenGL provides brick-wise updates of 3D textures, it does not support data reduction in these update operations. So even a brick with all zeros would need to be transferred completely. OpenGL also supports slice-based writing to 3D textures from a fragment shader, but this introduces considerable overhead and is limited in functionality. Using CUDA for assembling the bricks back into a complete volume on the GPU is more flexible and therefore allows for better data reduction.

The level of data locality could be increased further by applying a space-filling curve, e. g., by accessing the voxels in Z-order (Morton, 1966). However, this would introduce a more complex memory access pattern that is not suitable for a CUDA implementation. Furthermore, we do not expect that such an implementation would achieve significantly better compression results, since block compression algorithms would not be able to fully exploit the improved data locality.

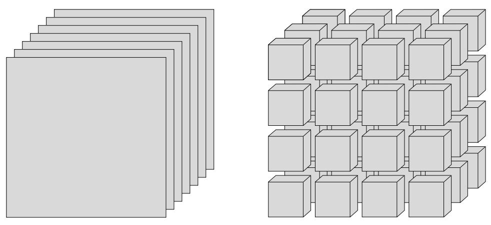
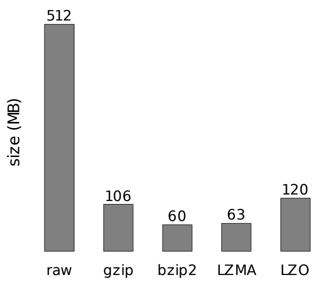
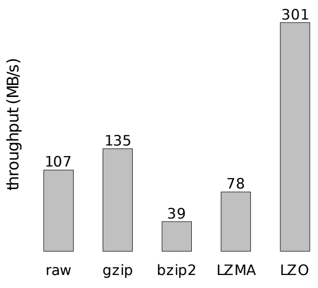


Figure 7.2: Slice-wise and brick-wise memory layout for storing volume data.



(a) compressed size



(b) decompression throughput

Figure 7.3: Compression efficiency and decompression throughput of the different lossless block compression algorithms. Only gzip and LZO can decompress faster than loading the uncompressed (raw) file from disk would take. The results include the time for reading the compressed data from disk.

A good opportunity for optimization after the volume has been decomposed into bricks is removing duplicate bricks and replacing them by references (Binotto et al., 2003). This, however, requires an exact match and therefore is only applicable to data without noise, i. e., mostly synthetic data. To be feasible, the brick size should be chosen as small as possible to increase the likelihood of duplicates, but this also increases the overhead of brick handling. Real-world data sets we tested did hardly contain any non-uniform duplicate bricks when choosing a feasible brick size, hence we do not consider this method as beneficial for our use case. The only type of duplicate bricks that appears regularly as a result of the prediction scheme discussed in Section 7.2.5 is a uniform brick where all voxels are set to zero. This case is efficiently handled by the variable-length coding described in Section 7.2.6.

7.2.4 Main compression algorithm

To choose a compression algorithm for our use case we must take both compression ratio and decompression speed into account. As a requirement, the time for reading and uncompressing a data block must be less than the time that would be needed for reading the uncompressed block. However, most lossless data compression tools and libraries such as gzip, bzip2, or LZMA are optimized for maximum compression ratio, but not speed. In contrast, the Lempel-Ziv-Oberhumer (LZO) real-time compression library (Oberhumer, 2008) was built with the main goal of providing fast decompression, since it is intended primarily for embedded systems.

We have evaluated the different compression algorithms by compressing a 512 MB part of the *convection* data set and decompressing it from hard disk. Figure 7.3a shows the compressed file sizes. The best compression ratios were achieved by bzip2, which is based on a Burrows-Wheeler transform (Burrows and Wheeler, 1994), and

LZMA (Pavlov, 2009). Both gzip and LZO use a Lempel-Ziv dictionary coder (Ziv and Lempel, 1977) and produce much larger compressed files. However, for our application not the compression ratio but the decompression throughput is most important. The results illustrated in Figure 7.3b reverse the ranking: Only gzip and LZO can decompress faster than it would take to read the uncompressed file from disk, hence bzip2 and LZMA are unsuitable for data streaming. Although having a slightly worse compression ratio, LZO achieves a throughput that more than doubles the throughput of gzip.

Hence, as also suggested by Nagayasu et al. (2008), we chose LZO as the CPU-based compression algorithm for our compression scheme. The LZO library supports multiple algorithms, from which we selected the LZO1X-999 variant, which yields the best compression ratio. It is the slowest of the available LZO compressors (up to 8 times slower than the default in our tests), but this does not influence the decompression speed.

7.2.5 Prediction schemes

The block compression algorithms described in the previous section can reduce the size of volume data by utilizing spatial coherence. But they cannot directly take advantage of temporal coherence between different time steps, because their *sliding window* is not large enough to cover several time steps. To utilize temporal coherence in time-varying data, a *prediction model* needs to be applied. Such a model tries to predict voxel values and replaces them by the error in the prediction (Ait-Aoudia et al., 2006; Fraedrich et al., 2007). For time-varying data it is promising to predict that the current voxel value will not change in the next time step and to store the difference to the actual value, i. e., the error in the prediction. This *differential pulse-code modulation* (DPCM) or *delta encoding* initially does not reduce the storage requirements. However, when the changes between time steps are not random, the error data will exhibit uniform structures. For example, all voxels in regions that do not change between time steps will get a delta value of zero, resulting in uniform bricks that can be compressed efficiently. Furthermore the resulting delta values will typically not use the full data range that is taken up by the original values, which might allow further compression.

A disadvantage of delta encoding is that it prevents jumping directly to a certain time step, as all previous time steps first have to be read to reconstruct the data. This can be resolved by saving the absolute values in addition to the delta values and loading them on demand, at the cost of increased storage requirements. Because jumping to a certain time step is less time-critical than sequential playback, the

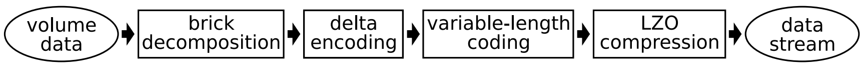


Figure 7.4: Block diagram of the complete volume compression scheme.

storage requirements can be limited by not saving the absolute values for every time step but only for every kth step. Hence, a maximum of $k - 1$ delta time steps will need to be loaded in addition to one absolute time step to reconstruct the volume data. This could be reduced further by a more elaborate algorithm.

7.2.6 Variable-length coding

As the scalar values inside a volume data set are usually not uniformly distributed, the value range of some bricks will be smaller than the value range of the entire volume. When the difference between the maximum and the minimum value in one brick is less than 2^n with $n < 16$, the brick size can be reduced by storing the minimum as the brick's *base value*, and for each voxel the difference from the base. Each of the difference values now only takes n bits to store, so this *variable-length coding* reduces the size needed for storing the brick.

This approach can be directly applied in combination with the previously described temporal prediction scheme. As the resulting delta values are typically smaller than the absolute voxel values they encode, the variable-length coding can achieve much higher compression ratios when applied to the delta values compared to when the time steps are compressed using absolute values. Uniform bricks are handled by the variable-length coding directly: All voxels in such a brick have the same value, so just the base value needs to be stored, while the delta values are reduced to "zero bits", i. e., are omitted.

7.2.7 Data preprocessing

The entire data set compression as shown in Figure 7.4 can run as an offline preprocessing step, with the aim of minimizing the overall data size. It creates a stream file that contains a sequence of compressed bricks with additional per-brick information, such as the number of bits used for storage (see Section 7.4.1 for more details). Preprocessing of the test data sets took 3 minutes for *hurricane*, 23 minutes for *combustion*, and up to 167 minutes for *convection*, with most of the time spent on the LZO compression algorithm. The runtime of this preprocessing is usually not an issue, as it is much less computationally intensive than the simulations used for creating the data in the first place. Hence, we did not yet optimize it for speed, e. g., by running multiple compression threads in parallel.

7.3 GPU-supported Decompression Pipeline

7.3.1 Multi-threaded loading and LZO decompression

We used a pipeline approach to overlap loading from disk, LZO decompression, and data upload to the GPU (see Figure 7.5). Each of these tasks runs as one or more independent threads. The loader thread was configured to load up to five time steps in advance and feed them to the decompression thread when it becomes idle. The main bottleneck is typically data loading, so building up of a long queue is only expected for bricks with a high compression ratio, for which loading from disk is faster than decompression. Consequently, we only assigned a single thread to LZO decompression, which might need to be changed when a faster storage device that can saturate a single CPU core is available.

7.3.2 Asynchronous data transfer to the GPU

Although much faster than disk I/O, uploading the uncompressed bricks to the GPU is still one of the bottlenecks of the decompression pipeline. To achieve optimal throughput, the data transfer to the GPU runs through a transfer buffer in main memory that is marked as *page-locked*, i. e., it will not be paged to disk by the operating system and therefore can be copied over the PCI Express bus using direct memory access (DMA) without involvement of the CPU. Using this "pinned" memory allows us to start an asynchronous memory copy that can run in parallel to CPU operations and GPU kernel executions. It achieves the maximum transfer bandwidth available by copying one large memory block that contains all bricks.

While the variable-length coding already handles "empty" (i. e., all-zero) bricks and those bricks that do not change between time steps, the data transfer could be further

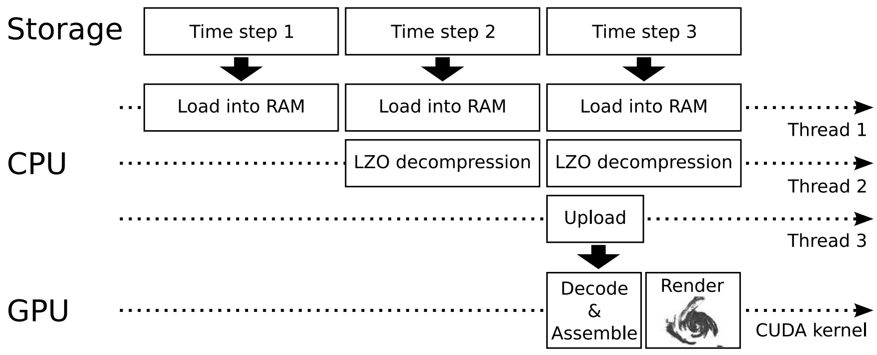


Figure 7.5: Our hybrid CPU/GPU decompression pipeline.

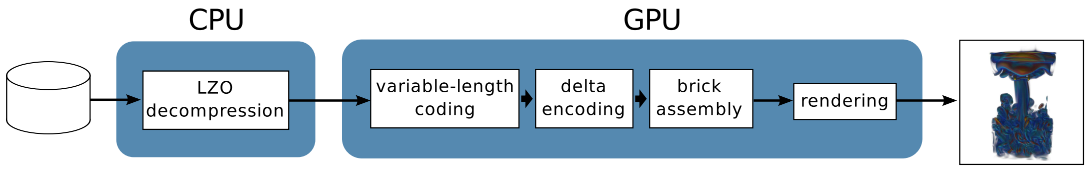


Figure 7.6: Decompression workflow.

reduced by ignoring bricks that are completely transparent because of the transfer function. A simple approximation for determining a brick's visibility is comparing the minimum and maximum intensity values inside the brick with the minimum and maximum intensity that is assigned non-zero opacity in the transfer function. This can be implemented efficiently but introduces two issues: First, not loading a brick because it is currently invisible breaks the delta encoding of upcoming time steps, as it requires data from all previous time steps to calculate the current value. Hence, the absolute value would need to be accessible as well, increasing disk usage. The second issue is that when the user modifies the transfer function, bricks that were previously hidden may get visible, therefore requiring a load operation, which might hamper the user experience. In addition, this optimization is not specific to our hybrid compression scheme, so we have not yet implemented it for the current system.

7.3.3 Decoding and brick assembly

In order to reconstruct the final 3D texture to be used for rendering, the data packets uploaded to the GPU need to be decoded and reassembled in three steps: Resolving variable-length coding, resolving delta encoding, and brick assembly (see Figure 7.6).

As the variable-length coding requires different addressing modes based on with how many bits a brick is stored, we have implemented individual kernels for handling each of the supported bit lengths. Based on analysis of our test data, we concluded that the compression ratio for just supporting 16, 8, 4, and 0 bits comes close enough to the optimal result so that the additional costs of supporting all possible numbers of bits are not justified.

To be able to benefit from the texturing hardware for linear filtering and border handling during rendering, the volume needs to be available to the raycasting kernel as a CUDA array. In contrast to data in global memory, a CUDA kernel cannot directly write to such an array. Therefore the decompression kernel uses a shadow copy of the volume texture located in global memory to write its results. The volume is later copied to the CUDA array by calling `cudaMemcpy3D()` from host code. As this is an on-device copy, it can theoretically make use of the full GPU memory bandwidth.

This intermediate step is anticipated to become unnecessary with the next generation of graphics processors, which are expected to allow writing to 3D textures from kernel code. The feature is already included in the OpenCL specification through the extension `cl_khr_3d_image_writes` (Munshi, 2009, p. 250), but this extension is not yet supported by current GPUs and drivers.

Implementing delta encoding is trivial, as the kernel just needs to add the calculated value to the existing value in the volume instead of overwriting it. To obtain optimal performance with CUDA kernels it is most important to satisfy the coalescing rules, i. e., to organize memory accesses in such a way that they require only the minimum number of memory transactions. We distribute the assembly of a brick onto blocks of 64 CUDA threads, where each thread is assigned an x-coordinate and processes all voxels in the brick belonging to this x-coordinate. Due to the memory layout, this results in adjacent threads accessing adjacent memory cells, and therefore the kernel shown in Listing 7.1 achieves full coalescing. By constructing a suitable two-dimensional CUDA grid of thread blocks, a single kernel launch is enough to start processing of all bricks.

7.3.4 Rendering

Since the compression scheme outputs the complete volume of the current time step into a CUDA array, we can directly use the basic CUDA volume raycaster described in Chapter 6 for rendering. The raycaster uses direct volume rendering with Phong lighting, on-the-fly gradient calculation, and early ray termination. Using a fragment shader for GPU-based raycasting would also be possible by making the CUDA array available to OpenGL through the *interop* mechanism available in recent revisions of CUDA. With OpenCL this would be even simpler, as OpenGL and OpenCL can share resources such as volume textures without any conversion.

7.4 Integration into Voreen

7.4.1 Preprocessing and on-disk storage format

We integrated the data preprocessing into the `voltool` command-line application, which is part of Voreen. The new `--stream` parameter expects a `.dat` text file with information about the `.raw` files that contain the actual volume data of each of the time steps to be processed—in contrast to the standard `.dat/.raw` format where only one volume file is specified per `.dat` file. The program outputs three files: a description file (`.dat`), a brick file (`.brk`), and a stream file (`.str`). The description file (Listing 7.2) is similar to a standard `.dat` file and contains general information

```
// Resolves delta encoding and variable-length coding for 4 bit bricks and assembles them into
// the output volume. Each thread processes all voxels with the given x-position in one brick.
__global__ void volumestream_delta4(dim3 volumeSize, int brickSize, unsigned char* brickData,
                                    ushort* baseValues, uint* brickPositions, ushort* output)
{
    const uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    const uint brickNum = blockIdx.y;                // the brick to process by this thread
    const uint brickLength = (brickSize * brickSize * brickSize) / 2;
    const ushort baseValue = baseValues[brickNum];    // for variable-length coding
    uint inPos = brickLength * brickNum + x;          // read position
    uint outPos = brickPositions[brickNum] + (x * 2); // write position

    for (int z = 0; z < brickSize; z++) {             // for each slice
        uint p = outPos;                              // the write position in the slice
        for (int y = 0; y < brickSize; y++) {         // for each row
            ushort value = brickData[inPos];
            // write two 16-bit output voxels from 8-bit input
            output[p + 0] += baseValue + (value >> 4);
            output[p + 1] += baseValue + (value & 0xF);

            // set up read/write positions for next row
            inPos += brickSize / 2;
            p += volumeSize.x;
        }
        outPos += volumeSize.x * volumeSize.y;        // move write position to next slice
    }
}
```

Listing 7.1: CUDA kernel for unpacking delta-encoded volume bricks with 4-bit variable-length coding. The kernels for decoding 8- and 16-bit bricks work similar.

```
Steps:          401                 # number of time steps
Mode:           delta/vlc           # direct, delta, delta/vlc
BrickSize:      64                  # brick side length
BrickCount:     8 4 4               # number of bricks in each dimension
Resolution:     512 256 256         # resolution of a single time step
SliceThickness: 1 1 1               # taken from source .dat file
Format:         USHORT              # USHORT or FLOAT
Compression:    lzo                 # none or lzo
BrickFileName:  convection_t.brk
StreamFileName: convection_t.str
```

Listing 7.2: Example `.dat` file for the *convection/T* data set.

```
enum PredictionType { DIRECT, DELTA };

struct BrickInfo {
    uint64_t offset_;               // position of brick data in stream file
    uint32_t size_;                 // compressed brick size
    uint16_t minValue_;             // minimum value in brick
    uint16_t maxValue_;             // maximum value in brick
    uint16_t baseValue_;            // base value for variable-length coding
    uint8_t bits_;                  // number of bits for variable-length coding
    PredictionType prediction_;     // absolute values or delta encoding
};
```

Listing 7.3: Data structure for brick information used in the binary `.brk` file.

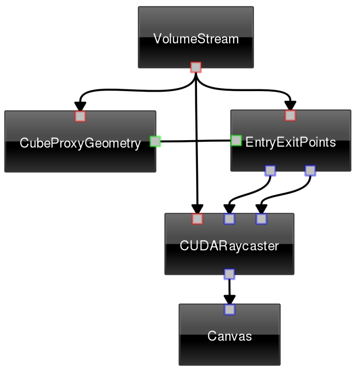


Figure 7.7: Data-flow network for rendering time-varying volume data using our lossless compression scheme. The standard `VolumeSource` processor was replaced by a `VolumeStream`. It reads the data of the current time step from the stream file and makes the uncompressed volume available to other processors through its outport.

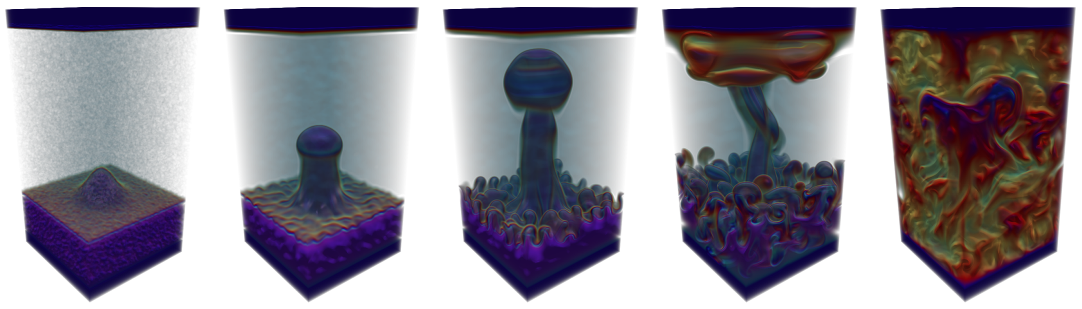
about the processed time-varying data set. Information about each brick is stored in the binary brick file (Listing 7.3), including the position of the compressed brick data in the stream file.

7.4.2 Network integration

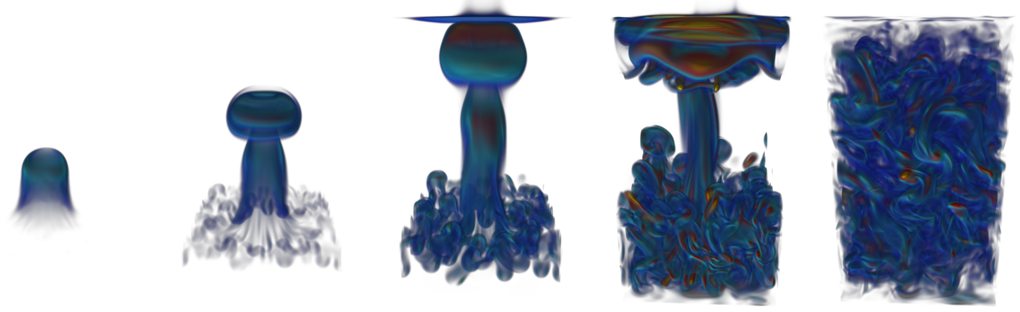
We integrated loading and streaming of time-varying data into Voreen by implementing a single `VolumeStream` processor, making use of the data-flow architecture. The new processor replaces `VolumeSource`, which only returns a static volume. When using `VolumeStream` the user selects the description file (`.dat`) of the compressed time-varying data set to load through the property mechanism. A further property of the processor can be used to set the time step to be loaded from the compressed data, which is then made available to other processors as a `VolumeHandle` through the single outport of the processor. Hence, for a data-flow network implementing rendering of compressed time-varying volume data (Figure 7.7) only a single processor needs to be replaced compared to a standard rendering network.

7.5 Results

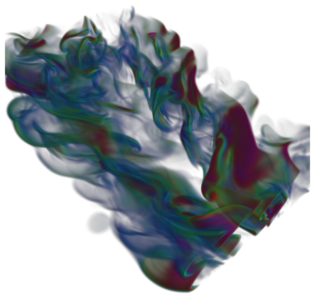
Tests were conducted on a workstation equipped with an Intel Core 2 Quad Q9550 CPU (2.83 GHz), an NVIDIA GeForce GTX 280 GPU, 4 GB RAM, and a 1.5 TB eSATA hard disk with a specified burst transfer rate of 105–115 MB/s. The system was running 64-bit Linux and used version 3.0 beta of the CUDA Toolkit. We later verified our results with the final 3.0 release.

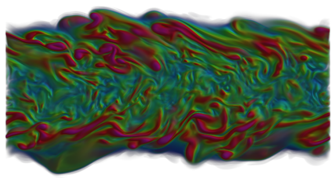


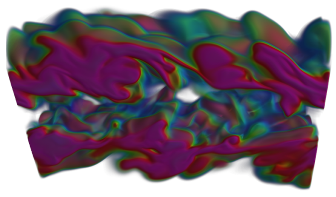
(a) convection/T



(b) convection/ens



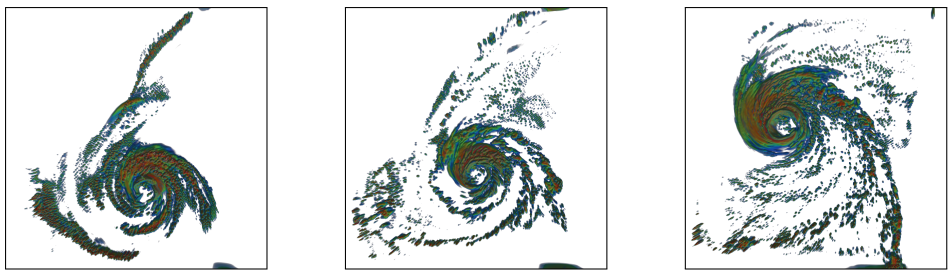




(c) combustion/chi

(d) combustion/vort

(e) combustion/y_oh



(f) hurricane/rain

Figure 7.8: Visualizations of individual time steps from the time-varying test data sets using direct volume rendering.

					compr.			brick bit usage			
pred.	vlc	bs	memory	bricks	size	σ	σ_{vlc}	16	8	4	0
none	no	64	512 kB	128	16.76 GB	1.50	—	—	—	—	—
delta	no	64	512 kB	128	11.53 GB	2.17	—	—	—	—	—
delta	yes	32	64 kB	1024	11.16 GB	2.24	1.33	61%	18%	20%	1%
delta	yes	64	512 kB	128	11.31 GB	2.22	1.14	78%	15%	6%	0%
delta	yes	128	4 MB	16	11.68 GB	2.15	1.04	93%	7%	0%	0%
delta	yes	256	32 MB	2	11.92 GB	2.10	1.02	96%	4%	0%	0%

Table 7.2: Effect of prediction scheme, variable-length coding (vlc), and brick size (bs) on compression, tested with the *convection/T* data set. Also listed are the total compression ratio σ, compression ratio σ_{vlc} obtained by variable-length coding alone, and the percentages of bricks that are encoded with a certain number of bits by the variable-length coding.

7.5.1 Test data sets

Renderings of the test data sets are shown in Figure 7.8. The *convection* data set is the result of a hydrodynamical simulation of a thermal plume. It contains two modalities, temperature (*T*) and enstrophy (*ens*), where the latter highlights swirling regions of the flow.[*] As can be seen in the first time step of this data set in Figure 7.8, the *T* modality contains a high level of noise, which was intentionally introduced into the simulation, and ends in a fully turbulent scenario. The *ens* modality is more uniform in the initial part, but also becomes fully turbulent towards the end. Three modalities *chi*, *vort*, and *y_oh* are available from a turbulent combustion simulation. The structure of this data set is turbulent as well, but the amount of empty space varies between the modalities. Finally, there is data from a simulation of the amount of rain in different levels of the atmosphere for Hurricane Isabel. This smaller data set contains many empty regions and is expected to achieve a high compression ratio.

7.5.2 Compression ratio

To evaluate the effect of the compression parameters, we have processed *convection/T* with several different compression options. The results are listed in Table 7.2. Note that the given raw sizes correspond to the data converted to 16-bit, the original float data would take up twice the amount of memory. First, we examined the effect of delta encoding without using variable-length coding. Delta encoding increased the

[*] In fluid dynamics, *enstrophy* is defined as the integral of the square of the vorticity, which is related to the amount of the local angular rate of rotation in a fluid flow.

data set	modality	raw size	compr. size	σ	σ_{vlc}	brick bit usage 16	8	4	0
convection	T	25.1 GB	11.3 GB	2.2	1.1	78%	15%	6%	0%
	ens	25.1 GB	4.8 GB	5.2	1.5	52%	19%	12%	17%
combustion	chi	11.4 GB	2.7 GB	4.3	2.0	48%	2%	1%	50%
	vort	11.4 GB	3.8 GB	3.0	1.9	50%	3%	6%	41%
	y_oh	11.4 GB	3.5 GB	3.3	2.0	49%	2%	2%	47%
hurricane	rain	3.0 GB	0.1 GB	25.1	2.6	36%	3%	1%	60%

Table 7.3: Results of our hybrid compression scheme. The compression uses 64^3 bricks, delta encoding, variable-length coding, and LZO1X-999 compression.

compression ratio of the LZO algorithm from 1.50 to 2.17, which is quite significant considering the low cost of calculating the delta values. The efficiency of variable-length coding depends on the brick size, as smaller bricks are more likely to contain data that fits into a smaller value range. As can be seen from the bit usage, only 4% of the bricks can be encoded with less than the full 16 bits when a brick side length of 256 voxels is used. This percentage increases with smaller brick size, thus increasing the compression ratio σ_{vlc} achieved by the variable-length coding alone. The best compression ratio is achieved for a brick side length of 32, but also the number of bricks increases to 1024 for this configuration. To keep the overhead for managing bricks reasonable, we chose a brick side length of 64 for all following tests. While this reduces σ_{vlc} from 1.33 to 1.14, the overall compression ratio, i. e., when including LZO compression, stays nearly the same. Hence, the LZO algorithm compensates differences in variable-length coding when the block size is chosen small enough.

As expected, the compression ratios achieved for the different data sets vary significantly (Table 7.3). The modality *T* of the *convection* data set has the lowest compression ratio both for variable-length coding as well as for total compression. This is the result of the high level of noise and low amount of empty space in the data set. The *ens* modality is more sparse and therefore has a much higher compression ratio of 5.2, with many more bricks encoded with less than 16 bits. The *combustion* data set contains a lot of empty space, so 41% to 50% of its bricks are empty and can be encoded with zero bits. It is notable that while σ_{vlc} only varies slightly between the modalities, the overall compression ratio σ varies between 3.0 and 4.3. Hence, the differences are a result of only the LZO compression. Finally, *hurricane* is a rather small and sparse data set that gets a high compression ratio of 25.1 and therefore shifts the system bottleneck from disk throughput to decompression speed.

7.5.3 Rendering speed

To determine the increase of overall rendering performance achieved by our method, we measured the time taken for rendering all time steps of the compressed data sets and compared this to the results of the uncompressed version (Table 7.4). The loader for the uncompressed files reads the 16-bit integer data of a time step into memory and immediately uploads it to the GPU for rendering. Disk caches were flushed between test runs. The raycasting sampling rate was set to 2 samples per voxel, and a viewport size of 512×512 pixels was chosen. As expected, the rendering speedups for most data sets resemble the compression ratios. For some (*convection/T*, *combustion/chi*) they even slightly exceed the compressed ratios, which we would explain with caching effects. The small *hurricane* data set is not limited by disk throughput, and therefore the rendering speedup attained by the compression technique is significantly smaller than the compression ratio ($s = 7.27 < \sigma = 25.1$). It is rendered with 10 fps, more than 7 times faster than without compression. The same effect, but less intense, can be seen for *convection/T*, which has the second highest compression ratio $\sigma = 5.2$, but only a rendering speedup of $s = 4.23$. With data sets for which disk throughput was the bottleneck we measured a disk transfer rate of up to 106 MB/s, which is close to the specified maximum transfer rate of the used hard drive.

Measuring the time needed for the on-device copy of the volume from global memory into the final 3D texture stored as a CUDA array (compare Section 7.3.3) gave results of about 24 ms for the *convection* data set, twice the time needed for brick assembly and even more than the time taken for rendering. This corresponds to a throughput of only about 5 GB/s, much less than the maximum of 110 GB/s. We presume that the low throughput for copying to a 3D CUDA array is related to the internal data format in GPU memory. According to Engel et al. (2006, p. 192), 3D texture data is often rearranged on the GPU to increase the locality of neighboring data values. Apparently, the data is automatically converted to this internal format when copying to the CUDA array, significantly decreasing throughput compared to a direct copy to global memory. This could be solved by having the data available in the internal format before copying; however, currently no such mechanism exists in CUDA. Future graphics processors that are expected to allow direct writing to 3D textures from CUDA kernels will hopefully resolve this problem by making the intermediate copy and data conversion unnecessary, and therefore remove the performance obstacle.

To examine the efficiency of the GPU-based brick assembly implemented as a CUDA kernel, we compared it to a CPU implementation that assembles the bricks into a memory buffer, which is then uploaded to the GPU. The results in Table 7.5

		raw		compressed		
data set	modality	time	fps	time	fps	*s*
convection	T	281	1.4	108	3.7	2.60
	ens	281	1.4	66	6.0	4.23
combustion	chi	126	1.0	28	4.4	4.55
	vort	117	1.0	39	3.1	2.98
	y_oh	125	1.0	36	3.4	3.47
hurricane	rain	35	1.4	5	10.0	7.27

Table 7.4: Timing results for rendering all time steps of the data sets using the raw data and our compression scheme. The table includes the measured time in seconds, frames per second, and the speedup factor *s*.

		CPU		GPU		
data set	modality	time	fps	time	fps	*s*
convection	T	117	3.4	108	3.7	1.08
	ens	92	4.4	66	6.0	1.38
combustion	chi	28	4.4	28	4.4	1.01
	vort	39	3.1	39	3.1	1.01
	y_oh	36	3.4	36	3.4	1.00
hurricane	rain	48	6.2	5	10.0	1.29

Table 7.5: Efficiency of running brick assembly on the GPU compared to the CPU. A rendering speedup $s > 1$ shows that the CPU was saturated by the brick assembly and that the CPU load could be reduced by offloading this work to the GPU.

show that the CUDA implementation is never slower than the CPU and can achieve a significant rendering speedup of up to 1.38x. For *combustion* no acceleration is possible, as this data set is disk-bound, i. e., the rendering performance for the data set is limited by the disk transfer rate. The CPU implementation writes directly into the 3D texture, so the speedup would increase further when the CUDA kernel would also be able to write directly to a 3D texture without the intermediate on-device copy.

7.6 Summary

In this chapter we have presented a framework that allows to increase rendering speed of time-varying volume data sets based on a lossless compression scheme. By utilizing both CPU and GPU, we could minimize the amount of data that needs to be transferred. Relocating work to the GPU allows us to use prediction models and brick-based instead of slice-based addressing to better maintain spatial coherence. Together with variable-length coding this can increase the efficiency of LZO compression without increasing load on the CPU. While the image quality is not affected by our approach, the compression ratio that can be achieved is highly dependent on the data set. We have demonstrated our technique with real-world data sets that contain a considerable level of noise. In all cases near-interactive frame rates were obtained at full image quality, and the compression improved performance so far that interactive frame rates are expected to be achieved when replacing the single hard disk by a faster storage device, e. g., a RAID system or a solid-state drive (SSD).

The exploited stream programming techniques make the compression scheme flexible and easily extensible. Thus, it would also be possible to integrate lossy compression for application cases where accuracy is not the highest priority. When using the proposed technique, the GPU-based volume rendering component needs no adaptation and can remain completely unchanged, as we have shown for Voreen. Therefore, combination with other conventional acceleration techniques, for example, empty-space skipping or temporal coherence optimization (Grau and Tost, 2007), is possible. However, for the types of data we tested, the core raycasting performance on the GPU was not the bottleneck.

Future work includes direct support for floating-point data, evaluating further prediction schemes, and combination with multi-resolution techniques. Each of these points is related to extending individual components of our compression scheme, hence its overall structure would stay unchanged. With new graphics processors it should also be investigated whether they better support the implementation of complex compression algorithms such as LZO, which would allow to offload further work to the GPU.

Chapter 8

Conclusions

EVEN STATE-OF-THE-ART IMPLEMENTATIONS of volume rendering leave room for optimization, although they often already utilize graphics hardware. In this thesis we have investigated and successfully exploited the spatial and temporal coherence found in both volume data and rendering algorithms to achieve significant rendering speedups for static as well as for time-varying volume data. The presented techniques were carefully designed to fully utilize the hardware resources of current graphics processing units, which were accessed either through a graphics API or using the stream processing paradigm. The implementations follow a modular approach and can be integrated easily into established, large-scale rendering systems.

The presented acceleration techniques target different parts of the volume rendering pipeline (compare Figure 8.1) and exploit different types of coherence. However, they share the common goal of increasing the overall rendering performance to enable more sophisticated visualizations. The occlusion frustum algorithm (Chapter 5) utilizes spatial data coherence of empty voxels in the volume together with temporal coherence in the images resulting from volume rendering. The purely GPU-based technique achieves a rendering speedup of up to a factor of two while not requiring

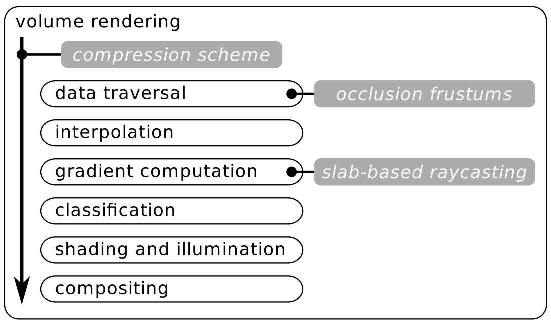


Figure 8.1: Position of the acceleration techniques in the volume rendering pipeline.

preprocessing. Unlike previous approaches, it does not cause image artifacts and is therefore suitable especially for interactive data exploration tasks. Slab-based raycasting (Chapter 6) exploits spatial coherence in the access pattern of the ray traversal. It can accomplish significant speedups by utilizing shared memory, which only became accessible through a stream processing implementation. Hence, the technique demonstrates the suitability of the stream processing paradigm for volume rendering. The hybrid compression scheme for time-varying data (Chapter 7) employs spatial coherence for block compression and temporal coherence for the prediction model. It achieves a compression ratio of up to 25:1 and a rendering speedup of up to 7x without affecting the image quality and therefore enables interactive visual data analysis without relying on specialized hardware.

As not all types of coherence exist in all types of volume data, it is difficult to recommend a specific acceleration technique for general use. An optimal solution would introduce only a negligible overhead even in the worst case, which unfortunately does not apply to the three presented techniques. It would be useful to be able to predict the performance of a rendering approach and to only activate the optimization when it has a chance to actually achieve an overall improvement. However, because of the complexity of the—largely undocumented—graphics hardware, a purely analytical approach for estimating rendering performance seems unrealistic. An experimental approach that constantly measures the frame rate and dynamically activates or deactivates acceleration techniques based on these measurements seems to be more promising. This would be a worthwhile addition for practical use of the presented optimizations, but also for other acceleration approaches.

When programming modern graphics processors, it is especially important to keep in mind the individual hardware restrictions. These limitations prevent the use of many standard approaches known from CPU programming. For GPU-based implementations, data structures as well as algorithms must be kept simple, since both an irregular memory access pattern and non-coherent branching behavior will introduce a severe performance penalty. Despite these limitations, volume raycasting can still use the graphics hardware efficiently. However, volume graphics differs significantly from the triangle-based graphics that GPUs and common graphics APIs support natively. Hence, implementations of volume rendering can benefit from approaches originally intended for general-purpose computations on GPUs. While only CUDA was used for evaluating volume raycasting through stream processing, the results can be generalized to most current stream processing architectures. Thanks to the standardization through OpenCL, the paradigm will play an increasingly important role in volume rendering in the future. However, it will probably not replace traditional graphics APIs such as OpenGL completely: Volume rendering is just one component of a full visualization system, which also consists of many other subsystems that use standard graphics methods, from rendering an additional mesh geometry, over image processing, to showing the rendering results on the screen. Therefore, future volume visualization systems will use a combination of graphics programming and stream processing.

Future progress both in graphics processors and in multi-core CPUs will offer new challenges but also new possibilities for volume rendering. The presented acceleration approaches would benefit directly from more cache memory and from higher storage system throughput. Such enhancements are currently arriving with the latest generation of GPUs and fast solid-state drives. Possible extensions of the presented approaches, such as incremental refinement of the occlusion frustum proxy geometry or implementation of data decompression completely on the GPU, depend on further hardware improvements. Upcoming hybrid processors that combine functionality of both CPUs and GPUs might open the way to novel optimizations, but exploiting coherence, with such techniques as discussed in this thesis, will most certainly be an important aspect of any new acceleration approach.

Appendix A

Source Code

These source code examples are included for illustration purposes and therefore may vary slightly from the actual implementation. The code is intended for integration into the VOREEN framework and therefore uses of some of the VOREEN infrastructure.

A.1 Fragment Shader for Volume Raycasting

```
//
// Implementation of volume raycasting as a fragment shader
//

#include "modules/mod_sampler2d.frag"
#include "modules/mod_sampler3d.frag"
#include "modules/mod_transfunc.frag"

uniform float raycastingQualityFactorRCP_;
uniform vec3 cameraPosition_;
uniform vec3 lightPosition_;

uniform SAMPLER2D_TYPE entryPoints_;        // ray entry points
uniform SAMPLER2D_TYPE exitPoints_;         // ray exit points

uniform sampler3D volume_;                  // volume dataset
uniform VOLUME_PARAMETERS volParams_;       // texture lookup parameters for volume_

// Gradient calculation using central differences
vec3 calcGradient(sampler3D vol, VOLUME_PARAMETERS volParams, vec3 samplePos) {
    const vec3 off = volParams.datasetDimensionsRCP_;

    float v0 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(off.x, 0, 0)).a;
    float v1 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(0, off.y, 0)).a;
    float v2 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(0, 0, off.z)).a;
    float v3 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(-off.x, 0, 0)).a;
    float v4 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(0, -off.y, 0)).a;
    float v5 = textureLookup3DUnnormalized(volume, volParams, samplePos + vec3(0, 0, -off.z)).a;

    vec3 gradient = vec3(v3 - v0, v4 - v1, v5 - v2) * 0.5;
    return gradient;
```

```
}

// Gradient calculation using filtered central differences
vec3 calcGradientFiltered(sampler3D volume, VOLUME_PARAMETERS volParams, vec3 samplePos) {
    const vec3 delta = volParams.datasetDimensionsRCP_;

    vec3 g0 = calcGradient(volume, volParams, samplePos);
    vec3 g1 = calcGradient(volume, volParams, samplePos + vec3(-delta.x, -delta.y, -delta.z));
    vec3 g2 = calcGradient(volume, volParams, samplePos + vec3( delta.x,  delta.y,  delta.z));
    vec3 g3 = calcGradient(volume, volParams, samplePos + vec3(-delta.x,  delta.y, -delta.z));
    vec3 g4 = calcGradient(volume, volParams, samplePos + vec3( delta.x, -delta.y,  delta.z));
    vec3 g5 = calcGradient(volume, volParams, samplePos + vec3(-delta.x, -delta.y,  delta.z));
    vec3 g6 = calcGradient(volume, volParams, samplePos + vec3( delta.x,  delta.y, -delta.z));
    vec3 g7 = calcGradient(volume, volParams, samplePos + vec3(-delta.x,  delta.y,  delta.z));
    vec3 g8 = calcGradient(volume, volParams, samplePos + vec3( delta.x, -delta.y, -delta.z));

    vec3 mix0 = mix(mix(g1, g2, 0.5), mix(g3, g4, 0.5), 0.5);
    vec3 mix1 = mix(mix(g5, g6, 0.5), mix(g7, g8, 0.5), 0.5);
    return mix(g0, mix(mix0, mix1, 0.5), 0.75);
}

// Standard Phong illumination, the material and light properties are hardcoded for brevity
vec3 phong(vec3 sample, vec3 color, vec3 gradient) {
    vec3 N = normalize(gradient);
    vec3 L = normalize(vec3(lightPosition_ - sample));
    vec3 V = normalize(vec3(cameraPosition_ - sample));

    float shade = 0.3;                           // ambient
    float NdotL = max(dot(N, L), 0.0);
    shade += NdotL * 0.4;                        // diffuse

    vec3 H = normalize(V + L);
    float NdotH = pow(max(dot(N, H), 0.0), 60.0);   // shininess
    shade += NdotH * 0.5;                        // specular

    return color * shade;
}

// Performs direct volume rendering and returns the final fragment color.
vec4 basicRaycaster(in vec3 first, in vec3 last) {
    vec4 result = vec4(0.0);

    // calculate ray parameters
    float stepIncr = raycastingQualityFactorRCP_;
    float t = 0.0;
    vec3 direction = last.rgb - first.rgb;

    float tend = length(direction);
    direction = normalize(direction);

    while (t <= tend) {
        vec3 sample = first.rgb + t * direction;
        vec4 voxel = textureLookup3D(volume_, volParams_, sample);

        float intensity = voxel.a;
#ifdef TF
        vec4 color = applyTF(voxel);
#else
        vec4 color = vec4(intensity);
#endif // TF

#ifdef PHONG
  #ifdef GRAD_FILTER
        vec3 grad = calcGradientFiltered(volume_, volParams_, sample);
  #else
        vec3 grad = calcGradient(volume_, volParams_, sample);
  #endif // GRAD_FILTER
        color.rgb = phong(sample, color.rgb, grad);
#endif // PHONG

        // perform compositing: multiply alpha by raycastingQualityFactorRCP_ to
        // accomodate for variable slice spacing
        color.a *= raycastingQualityFactorRCP_;
        result.rgb = result.rgb + (1.0 - result.a) * color.a * color.rgb;
        result.a = result.a + (1.0 - result.a) * color.a;

#ifdef ERT
        // early ray termination
        if (result.a >= 1.0) {
            result.a = 1.0;
            t = tend;
        }
#endif // ERT

        t += stepIncr;
    }

    return result;
}

void main() {
    vec3 frontPos = textureLookup2D(entryPoints_, gl_FragCoord.xy).rgb;
    vec3 backPos = textureLookup2D(exitPoints_, gl_FragCoord.xy).rgb;

    // determine whether the ray has to be casted
    if (frontPos == backPos) {
        // background needs no raycasting
        discard;
    } else {
        // frag coords are lying inside the bounding box
        gl_FragColor = basicRaycaster(frontPos, backPos);
    }
}
```

Listing A.1: Fragment shader implementation of basic volume raycasting.

A.2 CUDA Kernel for Volume Raycasting

```
//
// Implementation of volume raycasting as a CUDA kernel
//

#include <cuda.h>
#include "cutil_math.h"

texture<ushort, 3, cudaReadModeNormalizedFloat> volumeTex;       // 3D texture (16 bit)
texture<uchar4, 1, cudaReadModeNormalizedFloat> transferFuncTex;  // 1D RGBA texture (8 bit)

// Gradient calculation using central differences
inline __device__
float3 calcGradient(float3 sample) {
    const float3 offset = make_float3(1.f / off_x, 1.f / off_y, 1.f / off_z);

    float v0 = tex3D(volumeTex, sample.x + offset.x, sample.y, sample.z);
    float v1 = tex3D(volumeTex, sample.x, sample.y + offset.y, sample.z);
    float v2 = tex3D(volumeTex, sample.x, sample.y, sample.z + offset.z);
    float v3 = tex3D(volumeTex, sample.x - offset.x, sample.y, sample.z);
    float v4 = tex3D(volumeTex, sample.x, sample.y - offset.y, sample.z);
    float v5 = tex3D(volumeTex, sample.x, sample.y, sample.z - offset.z);

    return make_float3(v3 - v0, v4 - v1, v5 - v2) * 0.5f;
}

// Gradient calculation using filtered central differences
inline __device__
float3 calcGradientFiltered(float3 sample) {
    const float3 offset = make_float3(1.f / off_x, 1.f / off_y, 1.f / off_z);

    float3 g0 = calcGradient(sample);
    float3 g1 = calcGradient(sample+make_float3(-offset.x, -offset.y, -offset.z));
    float3 g2 = calcGradient(sample+make_float3( offset.x,  offset.y,  offset.z));
    float3 g3 = calcGradient(sample+make_float3(-offset.x,  offset.y, -offset.z));
    float3 g4 = calcGradient(sample+make_float3( offset.x, -offset.y,  offset.z));
    float3 g5 = calcGradient(sample+make_float3(-offset.x, -offset.y,  offset.z));
    float3 g6 = calcGradient(sample+make_float3( offset.x,  offset.y, -offset.z));
    float3 g7 = calcGradient(sample+make_float3(-offset.x,  offset.y,  offset.z));
    float3 g8 = calcGradient(sample+make_float3( offset.x, -offset.y, -offset.z));
    float3 lerp0 = lerp(lerp(g1, g2, 0.5f), lerp(g3, g4, 0.5f), 0.5f);
    float3 lerp1 = lerp(lerp(g5, g6, 0.5f), lerp(g7, g8, 0.5f), 0.5f);

    return lerp(g0, lerp(lerp0, lerp1, 0.5f), 0.75f);
}

// Standard Phong illumination, the material and light properties are hardcoded for brevity
inline __device__
float3 phong(float3 sample, float3 color, float3 gradient, float3 lightPos, float3 cameraPos) {
    float3 N = normalize(gradient);
    float3 L = normalize(lightPos - sample);
    float3 V = normalize(cameraPos - sample);
    float shade = 0.3f;                                 // ambient
    float NdotL = max(dot(N, L), 0.f);
    shade += NdotL * 0.4f;                              // diffuse

    float3 H = normalize(V + L);
    float NdotH = __powf(max(dot(N, H), 0.f), 60.f);    // shininess
    shade += NdotH * 0.5f;                              // specular

    return color * shade;
}

// Performs direct volume rendering
__global__
void basicRaycaster(float4* entryPoints, float4* exitPoints, float4* output,
                    uint width, uint height, float qualityFactorRCP,
                    float3 cameraPos, float3 lightPos)
{
    // Determine our position on the screen based on thread and block ID
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    if (x >= width || y >= height)
        return;

    uint index = (__umul24(y, width) + x);

    // enforce reading 4 floats although only 3 are accessed to get coalescing
    volatile float4 entry4 = entryPoints[index];
    volatile float4 exit4 = exitPoints[index];
    float3 first = { entry4.x, entry4.y, entry4.z };
    float3 last = { exit4.x, exit4.y, exit4.z };

    if (first == last) {
        output[index] = make_float4(0.f);
        return;
    }

    float4 result = make_float4(0.f);

    // calculate ray parameters
    float stepIncr = qualityFactorRCP;
    float t = 0.f;
    float3 direction = last - first;

    float tend = length(direction);
    direction = normalize(direction);

    while (t <= tend) {
        float3 sample = first + t * direction;
        float intensity = tex3D(volumeTex, sample.x, sample.y, sample.z);

#ifdef GRAD
  #ifdef GRAD_FILTER
```

```
        float3 gradient = calcGradientFiltered(sample);
  #else
        float3 gradient = calcGradient(sample);
  #endif // GRAD_FILTER
#endif // GRAD

  #ifdef TF
        float4 color = tex1D(transferFuncTex, intensity); // moved here to hide latency
  #else
        float4 color = make_float4(intensity);
  #endif // TF

#ifdef GRAD
  #ifdef PHONG
        float3 shadedColor = phong(sample, make_float3(color), gradient, lightPos, cameraPos);
        color.x = shadedColor.x;
        color.y = shadedColor.y;
        color.z = shadedColor.z;
  #endif // PHONG
#endif // GRAD

        t += stepIncr;

        // perform compositing
        color.w *= qualityFactorRCP;

        result.x = result.x + (1.f - result.w) * color.w * color.x;
        result.y = result.y + (1.f - result.w) * color.w * color.y;
        result.z = result.z + (1.f - result.w) * color.w * color.z;
        result.w = result.w + (1.f - result.w) * color.w;

#ifdef ERT
        // early ray termination
        if (result.w >= 1.f) {
            result.w = 1.f;
            t = tend + 1.f;
        }
#endif // ERT
    }

    // write output color
    output[index] = result;
}
```

Listing A.2: CUDA kernel for performing basic volume raycasting.

Appendix B

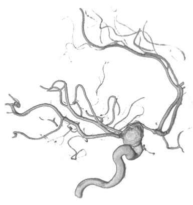
Index of Data Sets

The following contains information about the volume data sets used throughout this thesis. Some of them were resized for individual tests; this is noted where applicable in the respective chapter.

Static Data Sets

aneurysm

Rotational C-arm x-ray scan of the arteries in the right half of a human head, with an aneurism visible.

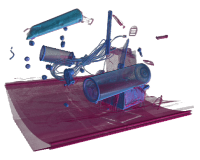


Dimensions:	256 × 256 × 256
Data type:	8 bit unsigned integer
Source:	`http://www.volvis.org`

Courtesy of Philips Research, Hamburg, Germany.

backpack

CT scan of a backpack filled with items.

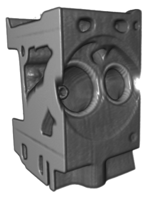


Dimensions:	512 × 512 × 373
Data type:	12 bit unsigned integer
Source:	`http://www.volvis.org`

Courtesy of Kevin Kreeger, Viatronix Inc., USA.

engine

CT scan of two cylinders of an engine block.

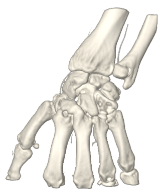


Dimensions:	256 × 256 × 128
Data type:	8 bit unsigned integer
Source:	`http://www.volvis.org`

Created by General Electric.

hand

CT scan of a human hand.



Dimensions:	256 × 256 × 256
Data type:	12 bit unsigned integer

Courtesy of Tiani Medgraph, Vienna, Austria.

stagbeetle

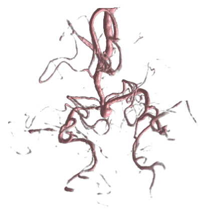
CT scan of a stag beetle sculpture.

Dimensions:	416 × 416 × 247
Data type:	16 bit unsigned integer
Source:	`http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/`

The stag beetle from Georg Glaeser, Vienna University of Applied Arts, Austria, was scanned with an industrial CT by Johannes Kastner, Wels College of Engineering, Austria, and Meister Eduard Gröller, Vienna University of Technology, Austria.

vertebra

Rotational angiography scan of a human head with an aneurysm. Only contrasted blood vessels are visible.



Dimensions:	512 × 512 × 512
Data type:	12 bit unsigned integer
Source:	`http://www.volvis.org`

Courtesy of Michael Meißner, Viatronix Inc., USA.

vmhead

Visible Human head CT.

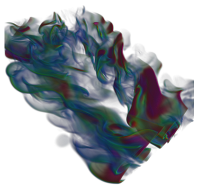
Dimensions:	512 × 512 × 294
Data type:	12 bit unsigned integer
Source:	`http://www.nlm.nih.gov/research/visible/getting_data.html`

Courtesy of the Visible Human Project, U.S. National Library of Medicine. The volume was constructed from the raw data using the instructions from `http://teem.sourceforge.net/nrrd/vmhead/`.

Time-Varying Data Sets

combustion

Turbulent combustion simulation.

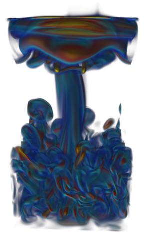


Dimensions:	480 × 720 × 120
Time steps:	122
Data type:	32-bit float
Modalities:	chi, vort, y_oh
Source:	`http://vis.cs.ucdavis.edu/Ultravis/datasets/`

This data set was made available by Dr. Jacqueline Chen at the Sandia National Laboratory through the SciDAC Institute for Ultra-Scale Visualization.

convection

Hydrodynamical simulation of a thermal plume.

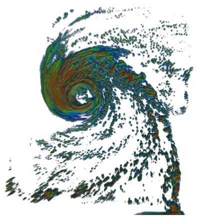


Dimensions:	512 × 256 × 256
Time steps:	401
Data type:	32-bit float
Modalities:	temperature (T), enstrophy (ens)

This data set was generated by Johannes Lülff and Michael Wilczek from the Institute for Theoretical Physics at the University of Münster.

hurricane

Simulation of Hurricane Isabel from the 2003 Atlantic hurricane season.



Dimension:	500 × 500 × 100
Time steps:	48
Data type:	32-bit float
Modalities:	qrain
Source:	`http://www.vets.ucar.edu/vg/isabeldata/`

The Hurricane Isabel data was produced by the Weather Research and Forecast (WRF) model, courtesy of NCAR, and the U.S. National Science Foundation (NSF).

Bibliography

Ait-Aoudia, S., Benhamida, F.-Z., and Yousfi, M.-A. (2006). Lossless compression of volumetric medical data. In *ISCIS: International Symposium on Computer and Information Sciences*, volume 4263 of *Lecture Notes in Computer Science*, pages 563–571. Springer-Verlag.

AMD (2009). *Stream Computing User Guide, Version 1.4-beta*. Advanced Micro Devices, Inc.

Avila, R., Sobierajski, L., and Kaufman, A. (1992). Towards a comprehensive volume visualization system. In *Proceedings of IEEE Visualization*, pages 13–20.

Binotto, B., Comba, J. L. D., and Freitas, C. M. D. (2003). Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *PVG '03: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 69–76.

Buck, I., Fatahalian, K., and Hanrahan, P. (2004a). Poster: GPUBench: Evaluating GPU performance for numerical and scientific applications. In *ACM Workshop on General-Purpose Computing on Graphics Processors*.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004b). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto.

Crow, F. C. (1977). Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, pages 242–248.

Cullip, T. J. and Neumann, U. (1994). Accelerating volume reconstruction with 3D texture hardware. Technical report, University of North Carolina at Chapel Hill.

Eisenmann, U., Freudling, A., Metzner, R., Hartmann, M., Wirtz, C. R., and Dickhaus, H. (2009). Volume rendering for planning and performing neurosurgical interventions. In *World Congress on Medical Physics and Biomedical Engineering*, volume 25/6 of *IFMBE Processings*, pages 201–204. Springer-Verlag.

Engel, K., Hadwiger, M., Kniss, J. M., and Weißkopf, D. (2006). *Real-Time Volume Graphics*. A K Peters.

Fout, N. and Ma, K.-L. (2007). Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1600–1607.

Fraedrich, R., Bauer, M., and Stamminger, M. (2007). Sequential data compression of very large data in volume rendering. In *VMV 2007: Proceedings of the Vision, Modeling, and Visualization Conference*, pages 41–50. Akademische Verlagsgesellschaft AKA.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 213–222.

Grau, S. and Tost, D. (2007). Frame-to-frame coherent GPU ray-casting for time-varying volume data. In *VMV 2007: Proceedings of the Vision, Modeling, and Visualization Conference*, pages 61–70. Akademische Verlagsgesellschaft AKA.

Grauer, B., Harlambang, N., and Hata, N. (2008). Volume rendering algorithm with CUDA for Slicer3. `http://www.slicer.org/slicerWiki/index.php/Slicer3:Volume_Rendering_With_Cuda`. Accessed 2008-12-03.

Grimm, S., Bruckner, S., Kanitsar, A., and Gröller, M. E. (2004). A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5):719–729.

Gudmundsson, B. and Randén, M. (1990). Incremental generation of projections of CT-volumes. In *Proceedings of the First IEEE Conference on Visualization in Biomedical Computing*, pages 27–34.

Guthe, S., Wand, M., Gonser, J., and Straßer, W. (2002). Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization*, pages 53–60.

Hadwiger, M., Sigg, C., Scharsach, H., Bühler, K., and Gross, M. H. (2005). Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum (Eurographics 2005)*, 24(3):303–312.

Havran, V., Bittner, J., and Seidel, H.-P. (2003). Exploiting temporal coherence in ray casted walkthroughs. In *SCCG '03: Proceedings of the 19th Spring Conference on Computer graphics*, pages 149–155. ACM.

Hernell, F., Ljung, P., and Ynnerman, A. (2007). Efficient ambient and emissive tissue illumination using local occlusion in multiresolution volume rendering. In *Eurographics/IEEE VGTC Symposium on Volume Graphics*, pages 1–8.

Houston, M. (2007). Understanding GPUs through benchmarking. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*.

Howison, M., Bethel, E. W., and Childs, H. (2010). MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *EGPGV '10: Eurographics Symposium on Parallel Graphics and Visualization*, Norrköping.

IEEE (2008). *IEEE 754-2008 Standard for Floating-Point Arithmetic*. Microprocessor Standards Committee of the IEEE Computer Society.

Iourcha, K., Nayak, K., and Hong, Z. (1999). System and method for fixed-rate block-based image compression with inferred pixel values. US Patent 5,956,431. S3 Incorporated.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. (2005). Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604.

Kainz, B., Grabner, M., Bornik, A., Hauswiesner, S., Muehl, J., and Schmalstieg, D. (2009). Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs. *ACM Transactions on Graphics*, 28(5):1–9.

Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., and Owens, J. D. (2003). Programmable stream processors. *IEEE Computer*, pages 54–62.

Kim, J. (2008). *Efficient Rendering of Large 3-D and 4-D Scalar Fields*. PhD thesis, University of Maryland, College Park.

Kim, J. and JaJa, J. (2008). Streaming model based volume ray casting implementation for Cell Broadband Engine. In *EGPGV '08: Eurographics Symposium on Parallel Graphics and Visualization*, pages 9–16.

Klein, T., Strengert, M., Stegmaier, S., and Ertl, T. (2005). Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Proceedings of IEEE Visualization*, pages 223–230.

Kraus, M. and Ertl, T. (2002). Adaptive texture maps. In *HWWS '02: Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 7–15.

Krüger, J. and Westermann, R. (2003). Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization*, pages 287–292.

Lakare, S. and Kaufman, A. (2004). Light weight space leaping using ray coherence. In *Proceedings of IEEE Visualization*, pages 19–26.

Law, A. and Yagel, R. (1996). Multi-frame thrashless ray casting with advancing ray-front. In *Proceedings of Graphics Interfaces*, pages 70–77.

Leung, W., Neophytou, N., and Mueller, K. (2006). SIMD-aware ray-casting. In *Eurographics/IEEE 5th International Workshop on Volume Graphics*, pages 59–62.

Levoy, M. (1988). Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37.

Levoy, M. (1990). Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261.

Li, W., Mueller, K., and Kaufman, A. (2003). Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of IEEE Visualization*, pages 317–324.

Lindstrom, P. and Isenburg, M. (2006). Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250.

Liu, B., Clapworthy, G. J., and Dong, F. (2009). Accelerating volume raycasting using proxy spheres. *Computer Graphics Forum (EuroVis 2009)*, 28(3):839–846.

Ljung, P., Lundström, C., and Ynnerman, A. (2006). Multiresolution interblock interpolation in direct volume rendering. In *EuroVis '06: Proceedings of the Eurographics/IEEE Symposium on Visualization*, pages 259–266.

Lorensen, W. E. and Cline, H. E. (1987). Marching Cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–169.

Luebke, D. and Parker, S. (2008). Interactive ray tracing with CUDA. Presentation at the NVISION 08 conference, San Jose.

Ma, K.-L. (2003). Visualizing time-varying volume data. *Computing in Science and Engineering*, 5(2):34–42.

Maršálek, L., Hauber, A., and Slusallek, P. (2008). Poster: High-speed volume ray casting with CUDA. *IEEE Symposium on Interactive Ray Tracing*, page 185.

Max, N. (1995). Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108.

Mensmann, J., Ropinski, T., and Hinrichs, K. (2008a). Accelerating volume raycasting using occlusion frustums. In Hege, H.-C., Laidlaw, D. H., Pajarola, R., and Staadt, O., editors, *Eurographics/IEEE 7th International Symposium on Volume and Point-Based Graphics*, pages 147–154, Los Angeles.

Mensmann, J., Ropinski, T., and Hinrichs, K. (2008b). Interactive cutting operations for generating anatomical illustrations from volumetric data sets. *Journal of WSCG – 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 16(1-3):89–96.

Mensmann, J., Ropinski, T., and Hinrichs, K. (2009). Poster: Slab-based raycasting: Efficient volume rendering with CUDA. SIGGRAPH/Eurographics Conference on High Performance Graphics, New Orleans.

Mensmann, J., Ropinski, T., and Hinrichs, K. (2010a). An advanced volume raycasting technique using GPU stream processing. In *GRAPP 2010: International Conference on Computer Graphics Theory and Applications*, pages 190–198, Angers.

Mensmann, J., Ropinski, T., and Hinrichs, K. (2010b). A GPU-supported lossless compression scheme for rendering time-varying volume data. In Westermann, R. and Kindlmann, G., editors, *IEEE/Eurographics 8th International Symposium on Volume Graphics*, pages 109–116, Norrköping.

Meyer-Spradow, J. (2009). *Interaktive Entwicklung Raycasting-basierter Visualisierungs-Techniken für medizinische Volumen-Daten mit Hilfe von Datenflussnetzwerken*. PhD thesis, Westfälische Wilhelms-Universität Münster.

Meyer-Spradow, J., Ropinski, T., Mensmann, J., and Hinrichs, K. (2009). Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13.

Meyer-Spradow, J., Ropinski, T., Mensmann, J., and Hinrichs, K. (2010). Interactive design and debugging of GPU-based volume visualizations. In *GRAPP 2010: International Conference on Computer Graphics Theory and Applications*, pages 239–245, Angers.

Meyer-Spradow, J., Ropinski, T., Vahrenhold, J., and Hinrichs, K. (2006). Illustrating dynamics of time-varying volume datasets in static images. In *VMV 2006: Proceedings of the Vision, Modeling, and Visualization Conference*, pages 333–340. Akademische Verlagsgesellschaft AKA.

Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada.

Munshi, A., editor (2009). *The OpenCL Specification, Version 1.0.48*. Khronos OpenCL Working Group.

Nagayasu, D., Ino, F., and Hagihara, K. (2006). Two-stage compression for fast volume rendering of time-varying scalar data. In *GRAPHITE 2006: Proceedings of the International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, pages 275–284.

Nagayasu, D., Ino, F., and Hagihara, K. (2008). A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics*, 32(3):350–362.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53.

NVIDIA (2004). OpenGL extension nv_texture_compression_vtc. `http://www.opengl.org/registry/specs/NV/texture_compression_vtc.txt`. Accessed 2010-06-12.

NVIDIA (2008a). CUDA GPU occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`. Accessed 2010-06-12.

NVIDIA (2008b). *CUDA Programming Guide, Version 2.1*. NVIDIA Corporation.

NVIDIA (2008c). CUDA SDK code samples. `http://www.nvidia.com/object/cuda_get_samples.html`. Accessed 2009-04-27.

NVIDIA (2009). CUDA Linux Release Notes, Version 2.1. `http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUDA_Release_Notes_2.1_linux.txt`. Accessed 2010-04-20.

NVIDIA (2010). *CUDA Programming Guide, Version 3.0*. NVIDIA Corporation.

Oberhumer, M. F. X. J. (2008). LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`. Accessed 2010-06-12.

Owens, J. (2005). Streaming architectures and technology trends. In Pharr, M., editor, *GPU Gems 2*, chapter 29, pages 457–470. Addison Wesley.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

Pavlov, I. (2009). LZMA software development kit. `http://www.7-zip.org/sdk.html`. Accessed 2010-06-12.

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

Praßni, J.-S., Ropinski, T., Mensmann, J., and Hinrichs, K. (2010). Shape-based transfer functions for volume visualization. In *IEEE Pacific Visualization Symposium (PacificVis)*, pages 9–16.

Ropinski, T., Döring, C., and Rezk-Salama, C. (2010). Interactive volumetric lighting simulating scattering and shadowing. In *IEEE Pacific Visualization Symposium (PacificVis)*, pages 169–176.

Ropinski, T., Meyer-Spradow, J., Diepenbrock, S., Mensmann, J., and Hinrichs, K. (2008). Interactive volume rendering with dynamic ambient occlusion and color bleeding. *Computer Graphics Forum (Eurographics 2008)*, 27(2):567–576.

Röttger, S., Guthe, S., Weiskopf, D., Ertl, T., and Straßer, W. (2003). Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the Symposium on Data Visualisation*, pages 231–238.

Salomon, D. (2002). *A Guide to Data Compression Methods*. Springer-Verlag.

Sayood, K. (2000). *Introduction to Data Compression*. Morgan Kaufmann, 2nd edition.

Scharsach, H., Hadwiger, M., Neubauer, A., Wolfsberger, S., and Bühler, K. (2006). Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *Eurographics/IEEE VGTC Symposium on Visualization*, pages 315–322.

Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. X. (2000). Conservative volumetric visibility with occluder fusion. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 229–238.

Schneider, J. and Westermann, R. (2003). Compression domain volume rendering. In *Proceedings of IEEE Visualization*, pages 293–300.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. (2008). Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15.

Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, 37(1):10–21.

Shreiner, D. (2009). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition.

Smelyanskiy, M., Holmes, D., Chhugani, J., Larson, A., Carmean, D. M., Hanson, D., Dubey, P., Augustine, K., Kim, D., Kyker, A., Lee, V. W., Nguyen, A. D., Seiler, L., and Robb, R. (2009). Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570.

Sohn, B.-S., Bajaj, C., and Siddavanahalli, V. (2004). Volumetric video compression for interactive playback. *Computer Vision and Image Understanding*, 96(3):435–452.

Šrámek, M. and Kaufman, A. (2000). Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):236–252.

Wald, I., Friedrich, H., Marmitt, G., and Seidel, H.-P. (2005). Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572.

Wald, I., Slusallek, P., Benthin, C., and Wagner, M. (2001). Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Eurographics 2001)*, 20(3):153–164.

Wan, M., Sadiq, A., and Kaufman, A. (2002). Fast and reliable space leaping for interactive volume rendering. In *Proceedings of IEEE Visualization*, pages 195–202.

Westermann, R. and Sevenich, B. (2001). Accelerated volume ray-casting using texture mapping. In *Proceedings of IEEE Visualization*, pages 271–278.

Westover, L. (1990). Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 367–376.

Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349.

Yagel, R. and Shi, Z. (1993). Accelerating volume animation by space-leaping. In *Proceedings of IEEE Visualization*, pages 62–69.

Yoon, I., Demers, J., Kim, T., and Neumann, U. (1997). Accelerating volume visualization by exploiting temporal coherence. In *Proceedings of IEEE Visualization*, pages 21–24.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343.

Acronyms

CT	computed tomography
CUDA	compute unified device architecture
DMA	direct memory access
DVR	direct volume rendering
EEP	entry/exit points (ray parameters)
ERT	early ray termination
FPS	frames per second
GLSL	OpenGL shading language
GPGPU	general-purpose computing on graphics processing units
GPU	graphics processing unit
JIT	just-in-time (compilation)
LUT	lookup table
MIMD	multiple instructions, multiple data
MIP	maximum intensity projection
MP	streaming multiprocessor (CUDA)
OpenCL	open computing language
OpenGL	open graphics library
PBO	pixel buffer object (OpenGL)
PTX	parallel thread execution (pseudo-assembly language used in CUDA)

RAID	redundant array of independent/inexpensive disks
SIMD	single instruction, multiple data
SIMT	single instruction, multiple threads
SSD	solid-state drive
SP	scalar processor (CUDA)