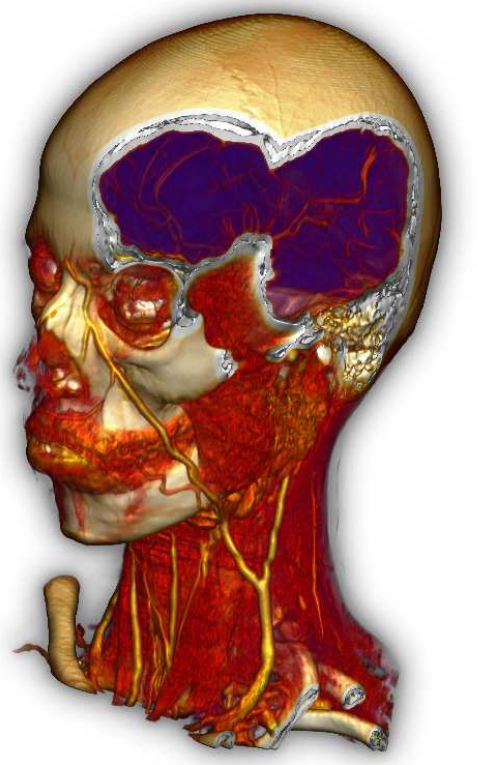
Stefan Bruckner

Efficient Volume Visualization of Large Medical Datasets

Master's Thesis



supervised by

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Dipl.-Inform. Sören Grimm

Institut of Computer Graphics and Algorithms
Vienna University of Technology

Abstract

The size of volumetric datasets used in medical environments is increasing at a rapid pace. Due to excessive pre-computation and memory demanding data structures, most current approaches for volume visualization do not meet the requirements of daily clinical routine. In this diploma thesis, an approach for interactive high-quality rendering of large medical data is presented. It is based on image-order raycasting with object-order data traversal, using an optimized cache coherent memory layout. New techniques and parallelization strategies for direct volume rendering of large data on commodity hardware are presented. By using new memory efficient acceleration data structures, high-quality direct volume rendering of several hundred megabyte sized datasets at sub-second frame rates on a commodity notebook is achieved.

Kurzfassung

Die Größe von in der Medizin verwendeten Volumensdatensätzen nimmt rapide zu. Doch, aufgrund von exzessiver Vorberechnung und speicherintensiven Datenstrukturen, sind viele Visualisierungstechniken solchen Datenmengen nicht gewachsen. Im Rahmen dieser Diplomarbeit wird eine Methode präsentiert, die die direkte Volumsvisualisierung von großen Datensätzen auf Standardhardware ermöglicht. Der Ansatz basiert auf einem hochqualitativen Image-Order-Algorithmus mit Object-Order-Datenverarbeitung, der ein optimiertes Cache kohärentes Speicherlayout einsetzt. Des weiteren werden neue Techniken und Strategien zur Parallelisierung auf Standardhardware präsentiert. Durch den Einsatz von neuen speicheroptimierten Beschleunigungsdatenstrukturen, werden für Datensätze einer Größe von vielen hundert Megabyte Darstellungsraten von mehreren Bilden pro Sekunde auf einem Standard-Notebook erreicht.

Contents

Chapter 1

Introduction

The beginning of knowledge is the discovery of something we do not understand.

Frank Herbert

Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis [8].

On the computer science side, it uses techniques of computer graphics and imaging. On the human side, perceptual and cognitive capabilities of the viewer determine the conditions the process needs to take into account. Successful visualization can reduce the time it takes to understand the underlying data, to perceive relationships, and to extract significant information.

Scientific visualization is a tool that enables scientists to analyze, understand, and communicate the numerical data generated by scientific research. In recent years, humans have been collecting data at a rate beyond what can be studied and comprehended. Scientific visualization uses computer graphics to process numerical data into two- and three-dimensional visual images. This visualization process includes gathering, processing, displaying, analyzing, and interpreting data. It is revolutionizing the way scientists do science

as well as changing the way people deal with large amounts of information.

Volume visualization is a field within scientific visualization, which is concerned with volume data. Volume data are 3D entities that may have information inside them, might not consist of surfaces and edges, or might be too voluminous to be represented geometrically. Volume visualization is a method of extracting meaningful information from volumetric data using interactive graphics and imaging, and it is concerned with volume data representation, modeling, manipulation, and rendering. Volume datasets are obtained by sampling, simulation, or modeling techniques. For example, a sequence of 2D slices obtained from Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) is three-dimensionally reconstructed into a volume model and visualized for diagnostic purposes, planning of treatment, or surgery. The same technology is often used for non-destructive inspection of composite materials or mechanical parts. Similarly, confocal microscopes produce data which is visualized to study the morphology of biological structures. In many computational fields, such as in computational fluid dynamics, the results of simulation typically running on a supercomputer are often visualized as volume data for analysis and verification. Recently, many traditional geometric computer graphics applications, such as CAD and simulation, have been exploiting the advantages of volume techniques called volume graphics for modeling, manipulation, and visualization.

Over the years many techniques have been developed to visualize volumetric data. Since methods for displaying geometric primitives were already well-established, most of the early methods involve approximating a surface contained within the data using geometric primitives. Common methods include contour tracking [13], opaque cubes [12], marching cubes [27], marching tetrahedra [48], dividing cubes [3], and others. These algorithms fit geometric primitives, such as polygons or patches, to constant-value contour surfaces in volumetric datasets. After extracting this intermediate representation, hardware-accelerated rendering can be used to display the surface primitives. In general, these methods require to make a decision for every data sample whether or not the surface passes through it. This can produce false positives (spurious surfaces) or false negatives (erroneous holes in surfaces),

(a) (b)

Figure 1.1: Comparison of surface and volume rendering. (a) traditional surface representation of a volumetric dataset. (b) direct volume rendering of the same dataset.

particularly in the presence of small or poorly defined features. As information about the interior of objects is generally not retained, a basic drawback of these methods is that one dimension of information is essentially lost.

In response to this, direct volume rendering techniques were developed that attempt to capture the entire 3D data in a single 2D image. Volume rendering techniques convey more information than surface rendering methods, but at the cost of increased algorithm complexity, and consequently increased rendering times. To improve interactivity in volume rendering, many optimization methods as well as special-purpose volume rendering hardware have been developed. Direct volume rendering algorithms include approaches such as raycasting [24], splatting [56], and shear-warp [20]. Instead of extracting an intermediate representation, volume rendering provides a method for directly displaying the volumetric data. The original samples are projected onto the image plane in a process which interprets the data as an amorphous cloud of particles. It is thus possible to simultaneously visualize information about surfaces and interior structures without making any assumptions about the underlying structure of the data. Volume rendering comprises more information in a single image than traditional surface representations

(see Figure 1.1), and is thus a valuable tool for the exploration and analysis of data. However, due to the increased computational effort required and the enormous size of volumetric datasets, the ongoing challenge of research in volume rendering is to achieve fully interactive performance.

This diploma thesis presents new methods and extensions to existing techniques for interactive direct volume rendering of large medical data. A high-quality volume visualization system has been developed which is capable of interactively handling large datasets on commodity hardware. An overview of algorithms for volume rendering is given in Chapter 2. In Chapter 3, we propose several techniques to enable the handling of large datasets, which are combined in a single framework to form a high-performance volume rendering algorithm. We discuss an alternative storage scheme that can significantly improve the cache behavior of a volume rendering algorithm. Furthermore, we present parallelization strategies which are well-suited for commodity hardware, and introduce memory efficient acceleration data structures. Chapter 4 focuses on the concepts used in the implementation of our algorithm. In Chapter 5, the performance of our techniques is discussed and our results are presented. Finally, in Chapter 6, the contents of this diploma thesis is summarized.

Chapter 2

State of the Art in Volume Rendering

Art is making something out of nothing and selling it.

Frank Zappa

This chapter will give a brief scientific background about volume rendering techniques in general. We start by introducing a common optical model and then discuss several algorithms and optimizations that have been proposed.

2.1 Optical Model for Volume Rendering

Optical models for direct volume rendering view the volume as a cloud of particles [30]. Light from a source can either be scattered or absorbed by particles. In practice, models that take into account all the phenomena tend to be very complicated. Therefore, practical models use several simplifications. A common approximation for the volume rendering integral is given by [32]:

$$\mathrm{I}_\lambda(x,r) = \int_0^L C_\lambda(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds \tag{2.1}$$

Hereby, I_λ is the amount of light of wavelength λ coming from a ray direction r that is received at location x on the image plane. L is the length of the ray r and μ is the density of volume particles which receive light from the light sources and reflect it towards the observer according to their material properties. C_λ is the light of wavelength λ reflected and/or emitted at location s in the direction of r. The equation takes into account emission and absorbtion effects, but discards more advanced effects such as scattering and shadows.

In general, Equation 2.1 cannot be computed analytically. Hence, most volume rendering algorithms use a numeric solution of the equation. This results in the common compositing equation:

$$\mathrm{I}_\lambda(x,r) = \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \cdot \prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{2.2}$$

Here $\alpha(s_i)$ are the opacity samples along a ray and $C_\lambda(s_i)$ are the local color values derived from the illumination model. C and α are referred to as transfer functions. These functions assign color and opacity to each intensity value in the volume.

2.2 Volume Rendering Techniques

In general, a volumetric dataset consists of samples arranged on a regular grid. These samples are also referred to as voxels. While most volume rendering techniques are based on the theoretical framework presented in Section 2.1, several different techniques implementing this optical model have emerged. In the following, we use a taxonomy based on the processing order of the data. We distinguish between image-order, object-order, and hybrid-order. Image-order methods start from the pixels on the image plane and compute the contribution of the appropriate voxels to these pixels. Object-order techniques traverse the voxels and compute what their contribution to

the image is. Hybrid-order methods try to combine both approaches. Techniques based on the texture mapping capabilities of the graphics hardware as well as dedicated volume rendering hardware solutions are each discussed in a separate section.

2.2.1 Image-Order Volume Rendering

The image-order approach to volume rendering determines, for each pixel on the image plane, the data samples which contribute to it. Raycasting [24] is an image-order algorithm that casts viewing rays through the volume. At discrete intervals along the ray, the three-dimensional function is reconstructed from the samples and the optical model is evaluated. This process is illustrated in Figure 2.1. As the accumulation is performed in front-to-back order, viewing rays that have accumulated full opacity can be terminated. This very effectively avoids processing of occluded regions and is one of the main advantages of raycasting. One challenge in raycasting is the efficient skipping of non-contributing regions (i.e., regions that have been classified as transparent). As typical medical datasets commonly contain a large number of such voxels, this has a major performance impact.

Numerous approaches for improving the performance of raycasting have been presented. Most of them rely on one or more of the following principles:

Image-Space Coherency

There is high coherency between pixels in image space, i.e., it is highly probable that between two pixels having identical or similar color we will find another pixel having the same (or similar) color. This observation is exploited by adaptive refinement [26]. The method works by initially casting rays only from a subset of screen pixels. "Empty"pixels residing between pixels with similar values are assigned an interpolated value.

Object-Space Coherency

Datasets contain regions having uniform or similar values. One way to increase the performance of raycasting therefore is to avoid sampling within

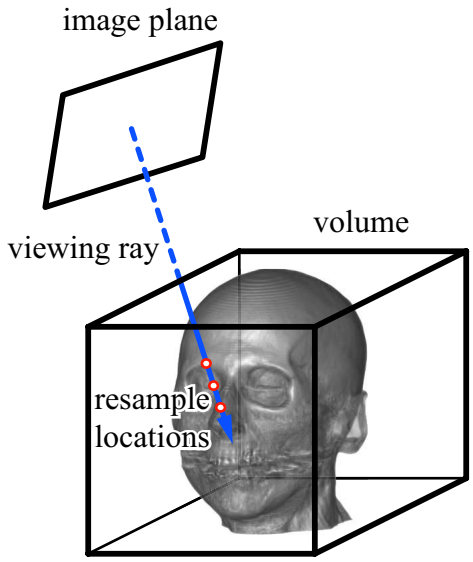


Figure 2.1: Illustration of raycasting. A ray starting at an image pixel is cast through the volume, evaluating the optical model at each resample location.

these regions. In an approach by van Walsum et al. [53] a ray starts sampling the volume at low frequency (i.e., large spacing between sample points). If a large value difference is encountered between two adjacent samples, additional samples are taken. This idea can be extended to lower the sample rate in regions where only small contributions of opacity are made.

Inter-Ray Coherency

For orthographic viewing the increased coherency between rays can be exploited. All rays, although having different origin, have the same slope. To avoid computations involved in advancing the ray through voxel space, the idea of template-based raycasting has been presented [57]. The sample points encountered by a ray are pre-computed and stored in a template. All rays can then be generated by applying the ray template.

Inter-Frame Coherency

In interactive viewing the differences between subsequent frames are usually small. The C-Buffer approach [58] works by storing, at each pixel location,

the object-space coordinates of the first non-empty voxel hit by the corresponding ray. This information is used to estimate the initial position of a ray in the consecutive frame. For each change of viewing parameters, the C-Buffer is transformed accordingly. In the case of rotation, a transformed buffer goes through a process of eliminating coordinates that might have become hidden.

Empty Space Skipping

As datasets usually contain large regions which are classified as transparent, several methods have been suggested to rapidly traverse empty space. Levoy presented an approach called hierarchical spatial enumeration [25]. The algorithm first creates a binary pyramid of the volume, which encodes empty and non-empty space. Raycasting is started at the top level of the pyramid. Whenever a ray reaches a non-empty cell, the algorithm moves down one level, entering whichever cell encloses the current location. Otherwise, the intersection point with the next cell is calculated and the ray is forwarded to this position. Following this idea, a min-max octree based on the volume's data values can be generated. This octree can be used to efficiently create the pyramid data structure whenever the classification changes. Another approach is space leaping [4, 50, 6]. Here, a distance transform is applied to the volume to calculate a "proximity"or "skip"value for each empty cell which encodes the distance to the nearest opaque cell. The value therefore is the distance that can be safely skipped along any ray that samples this cell. A drawback of this method is that it requires extensive processing every time the transfer function is changed.

Efficient Memory Access

For large datasets, memory access has a considerable impact on the overall processing time of a raycasting algorithm. The most simple memory layout for raycasting is a three-dimensional array. However, using this storage scheme leads to view-dependent render times, due to changing memory access patterns for different viewing directions. This can greatly affect the

performance for large datasets. Another common storage scheme is bricking [40], where the volume data is stored as sub-cubes (blocks) of a fixed size. In general, this approach reduces the view dependent performance variations but does not increase the memory consumption. Law and Yagel have developed a thrashless raycasting method based on such a memory layout [22]. In their approach, all resample locations within one block are processed before the algorithm continues to process the next block. Knittel [17] and Mora et al. [35] achieved impressive performance by using a spread memory layout. The main drawback of such an approach is the enormous memory usage. In both systems, the memory usage is approximately four times the data size.

2.2.2 Object-Order Volume Rendering

In contrast to image-order techniques, object-order methods determine, for each data sample, how it affects the pixels on the image plane. In its simplest form, an object-order algorithm loops through the data samples, projecting each sample onto the image plane. Splatting [56] is a technique that traverses and projects footprints (known as splats) onto the image plane, (see Figure 2.2). Voxels that have zero opacity, and thus do not contribute to the image, can be skipped. This is one of the greatest advantages of splatting, as it can tremendously reduce the amount of data that has to be processed. But there are also disadvantages: Using pre-integrated kernels introduces inaccuracies into the compositing process, since the 3D reconstruction kernel is composited as a whole. This can cause color bleeding artifacts (i.e. the colors of hidden background objects may "bleed"into the final image).

To remedy these artifacts, an approach has been developed which sums voxel kernels within volume slices most parallel to the image plane. However, this leads to severe brightness variations in interactive viewing. Mueller et. al. introduced a method which eliminates these drawbacks [36]. Their approach processes voxel kernels within slabs aligned parallel to the image plane. All voxel kernels that overlap a slab are clipped to the slab and summed into a sheet buffer. Once a sheet buffer has received all contributions, it is composited with the current image, and the slicing slab is

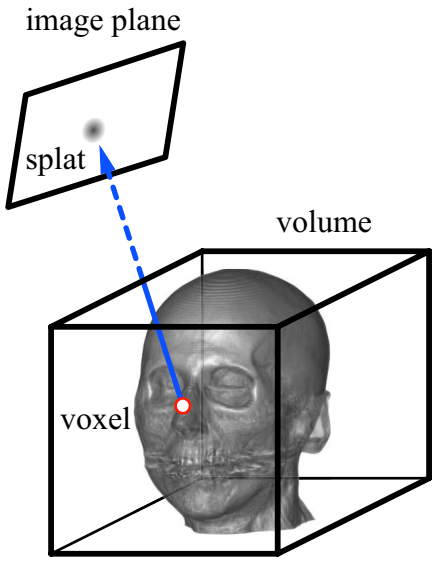


Figure 2.2: Illustration of splatting. The optical model is evaluated for each voxel and projected onto the image plane using a footprint (splat).

advanced forward. Mueller et. al. also presented an acceleration technique called early splat elimination which allows to skip footprint rasterization for occluded voxels [37]. However, the projection transformation still has to be performed for these voxels, hence, this optimization is not as effective as early ray termination in raycasting.

2.2.3 Hybrid-Order Volume Rendering

Image-order and object-order algorithms have very distinct advantages and disadvantages. Therefore, some effort has been spent on combining the advantages of both approaches.

Shear-warp [20] is such an algorithm. It is considered to be the fastest software-based volume rendering algorithm. It is based on a factorization of the viewing transformation into a shear and a warp transformation. The shear transformation has the property that all viewing rays are parallel to the principal viewing axis in sheared-object-space. This allows volume and image to be traversed simultaneously. Compositing is performed into an intermediate image. A two-dimensional warp transformation is then applied

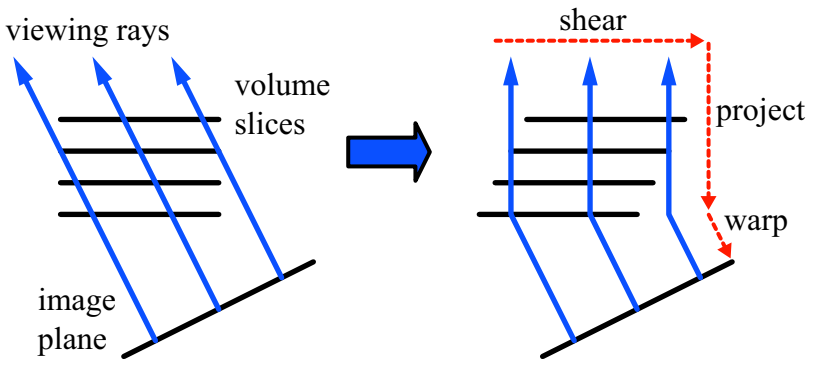


Figure 2.3: Illustration of the shear-warp mechanism. The volume slices are sheared so that all viewing rays are parallel to the major viewing axis. After the projection process has been performed, the distorted intermediate image is warped into the final image.

to the intermediate image, producing the final image. This basic mechanism is illustrated in Figure 2.3.

The aligned traversal is the basis for many optimizations: A runlength-encoding of the intermediate image allows an efficient early-ray termination approach. Additionally, runlength-encoding of the volume for each of the three major viewing axes allows skipping of transparent voxels. Additionally, an approach for empty space skipping which is based on a min-max octree has been presented. In contrast to runlength-encoding, this approach allows fast classification and does not require three copies of the volume.

The problem of shear-warp is the low image quality caused by using only bilinear interpolation for reconstruction, a varying sample rate which is dependent on the viewing direction, and the use of pre-classification. Some of these problems have been solved [51], however, the image quality is still inferior when compared to other methods, such as raycasting.

2.2.4 Texture Mapping Techniques

With graphics hardware becoming increasingly powerful, researchers have started to utilize the features of commodity graphics hardware to perform volume rendering. These approaches exploit the increasing processing power and flexibility of the Graphics Processing Unit (GPU). Nowadays, GPU-

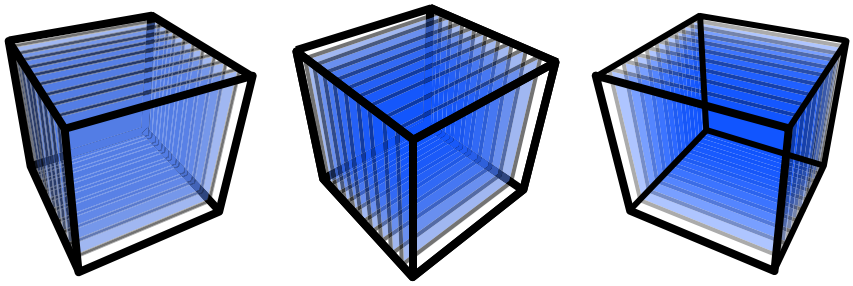


Figure 2.4: Volume rendering using 2D textures. For each of the three major viewing axes, a stack of 2D textures is stored. During rendering the stack corresponding to the axis most parallel to the current viewing direction is chosen and rendered in back-to-front order as textured quads using alpha blending.

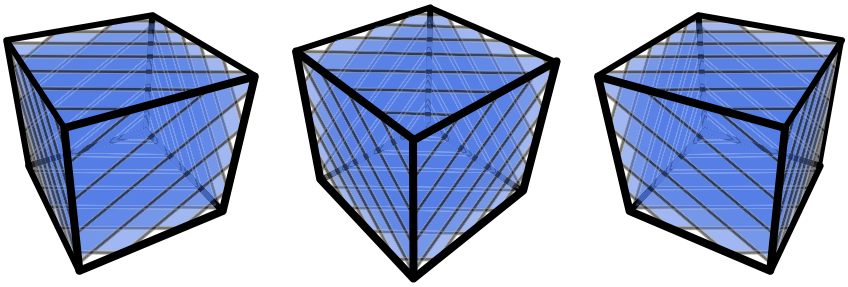


Figure 2.5: Volume rendering using 3D textures. The volume is stored as a single 3D texture. A set of planes parallel to the image plane is rendered in back-to-front order using alpha blending with appropriately specified texture coordinates.

accelerated solutions are capable of performing volume rendering at interactive frame rates for medium-sized datasets on commodity hardware.

One method to exploit graphics hardware is based on 2D texture mapping [45]. This method stores stacks of slices for each major viewing axis in memory as two-dimensional textures. The stack most parallel to the current viewing direction is chosen. These textures are then mapped on object-aligned proxy geometry which is rendered in back-to-front order using alpha blending (see Figure 2.4). This approach corresponds to shear-warp factorization and suffers from the same problems, i.e., only bilinear interpolation within the slices, and varying sampling rates depending on the viewing direction.

Approaches that use 3D texture mapping [2, 7, 55, 31] upload the whole volume to the graphics hardware as a three-dimensional texture. The hardware is then used to map this texture onto polygons parallel to the viewing plane which are rendered in back-to-front order using alpha blending (see Figure 2.5). 3D texture mapping allows to use trilinear interpolation supported by the graphics hardware and provides a consistent sampling rate. A problem of these approaches is the limited amount of video memory. If a dataset does not fit into this memory, it has to be subdivided. These blocks are uploaded and rendered separately, making the bus bandwith a bottleneck. One way to overcome this limitation is the use of compression [11].

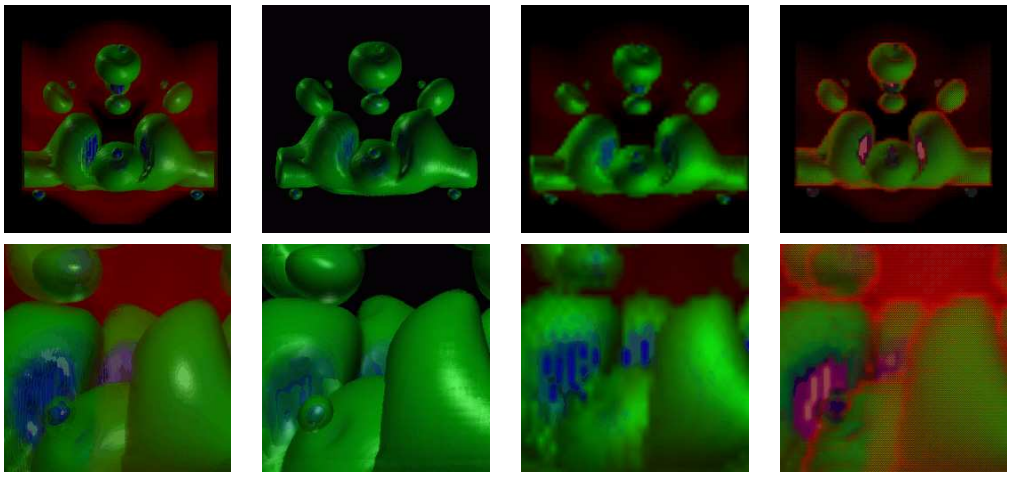
The increasing programmability of the graphics hardware has enabled several researches to apply acceleration techniques to GPU-based volume rendering [46, 18]. The performance of these approaches, however, is heavily dependent on the hardware implementation of specific features.

2.2.5 Special-Purpose Hardware

Due to the high computational cost of direct volume rendering, several researchers have proposed special-purpose volume rendering architectures. Most recent research has focused on accelerators for raycasting of regular datasets. Several architectures, such as VOGUE [16], VIRIM [10], VIZARD II [33], and EM-Cube [39], have been proposed. A comprehensive comparison of these architectures can be found in [44]. The EM-Cube architecture also served as a basis for the commercially available VolumePro board [41], which is capable of rendering a 512^3 dataset at 30 frames/second.

2.3 Comparison of Volume Rendering Algorithms

A extensive comparison of available algorithms for volume rendering has been performed by Meißner et al. [32]. While research has progressed since this study was performed, their basic findings are still valid. They conclude that the raycasting and splatting yield to similar image quality. The render times



(a) (b) (c) (d)

Figure 2.6: Comparison of volume rendering algorithms [32]. (a) raycasting. (b) splatting. (c) shear-warp. (d) 3D texture mapping.

of these methods are very much dependent on the type of dataset and transfer function. Shear-warp and 3D texture mapping provide high performance, but at the cost of degraded image quality. Recent work has been able to improve the quality of texture mapping approaches [5]. Figure 2.6 displays an excerpt of the result images by Meißner et al.

Chapter 3

Volume Rendering of Large Datasets

Many a small thing has been made large by the right kind of advertising.

Mark Twain

This chapter describes the fundamental components of our volume visualization system. First, we define a set of features that will be supported and derive the implications of these requirements. We then give an overview of the volume rendering pipeline. Next, the memory storage and access scheme that is used, is explained. We then go on to examine our approaches for parallelization and reconstruction. Finally, we show several optimizations to increase the performance of the algorithm and present a technique for increasing interactivity.

3.1 Introduction

When developing a volume visualization system, one must carefully evaluate the requirements with respect to its applications. These requirements depend on the type of datasets that will be processed, the desired image quality,

the degree of interactivity, etc. They influence, among others, the type of algorithm to be chosen and the optimizations that can be integrated.

3.1.1 System Requirements

For our system, be have carefully selected a number of features that we want to support:

Medical Datasets

The system will be primarily used to visualize medical datasets of anatomic nature aquired by CT or MRI.

High Quality

The system has to provide high quality. The user should be able to change the method of reconstruction, gradient estimation, and the sampling interval at run-time.

Interactive Viewing

The system has to allow interactive modification of viewing parameters. It must be capable of allowing the user to rotate, zoom and translate the volume.

Interactive Classification

It has been reported that transfer function design is a non-trivial task, which cannot easily be automated [14]. It is therefore essential that a volume rendering system allows fully interactive modification of the transfer function.

Low Memory Consumption

The system has to be capable of handling large datasets ($> 512^3$) on commodity hardware. In general, a medical visualization system consists of several modules. As these modules all share the same resources, our approach

should have the lowest possible memory footprint. Even sacrificing some performance, it should rather compute on the fly, than hold large pre-computed data structures in memory.

Parallelization Support

In order to achieve maximum performance, our system has to be capable of supporting multiple processors. Additionally, advanced features of the latest commodity hardware, such as Simultaneous Multithreading, have to be supported. None of these features, however, must be mandatory. Instead, the system has to automatically adapt to the given hardware configuration.

Pure Software

While graphics hardware acceleration can be exploited to increase the performance of a volume rendering system, certain problems arise when these kind of algorithms have to run in a heterogenous hardware environment. Supported features might differ, drivers might be outdated, etc. To avoid these problems, our system must not rely on any advanced features of the graphics hardware.

3.1.2 Implications

The requirements listed in the previous section have a number of implications on the choice of techniques that are useful. As hardware-accelerated techniques cannot be used, the choice of algorithms is limited to software algorithms. Shear-warp does not provide sufficient image quality, which basically reduces the choice to either a splatting or a raycasting approach. It has been shown that current splatting and raycasting approaches have distinct advantages and disadvantages. While splatting can efficiently skip empty space, raycasting performs better for high pixel content [32]. Considering the fact that the highest performance for splatting can only be reached if the graphics hardware is exploited for footprint rasterization and that raycasting is generally better suited for parallelization, the basic algorithm of our volume visualization system will be an image-order raycasting approach.

The kind of datasets which will be primarily used with our system do not profit from perspective projection. Since orthogonal viewing allows many optimizations, our system will only support parallel projection. The system will support voxels of up to 12 bits, as this is the common format for medical datasets. Due to performance issues, the voxels have to be aligned to byte boundaries, which leaves 4 bits. These 4 bits will be used to store segmentation information, which allows up to 16 different objects. Gradients will not be pre-computed and stored in memory, as it is common practice. Instead, they will be computed on-the-fly during rendering. Gradients require a considerable amount of memory. Storing gradients as uncompressed single precision floating-point values would require $3 \times 4 = 12$ bytes additional memory for every voxel. Even using gradient quantization, still requires at least 10 to 12 additional bits at each voxel for sufficient quality, which would double memory usage. Furthermore, using pre-computed gradients does not allow to quickly change the gradient estimation method at run-time without recomputing gradients for the entire dataset. Since demands for both high quality and interactive viewing exist, an effective scheme for automatic adaption is required. During classification and viewing parameter changes, the system has to transparently adapt rendering parameters to achieve interactive frame rates.

3.2 The Raycasting Pipeline

The goal of raycasting is the (approximate) calculation of the volume rendering integral for every viewing ray originating at the image plane and passing through the volume. This evaluates to advancing rays through the volume with regular increments Δs (i.e. the object sample distance) and performing the following steps at each location:

Reconstruction Reconstruction is the process of constructing a continuous function from the dataset. This step is necessary, since we want to sample the data at evenly spaced positions (resample positions) along each ray.

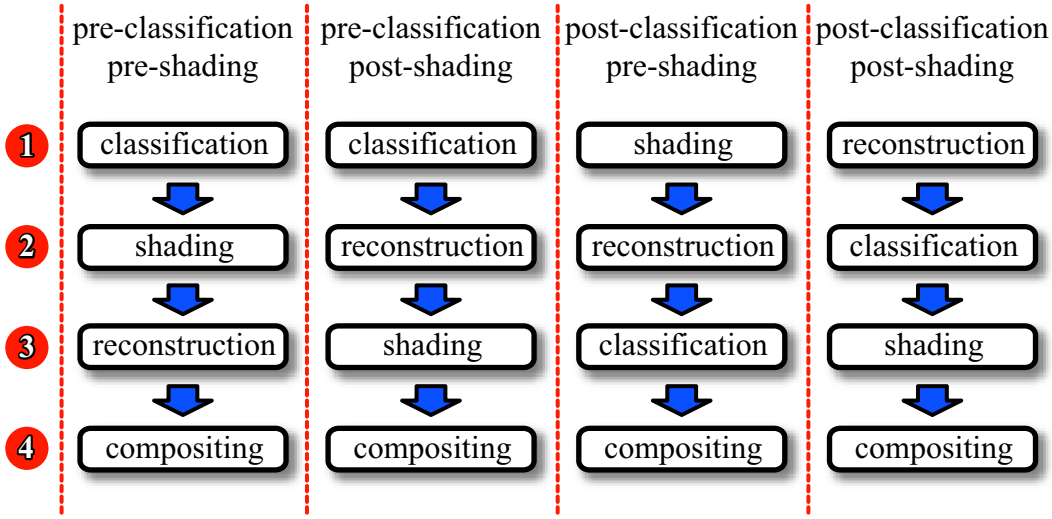


Figure 3.1: Four versions of the volume rendering pipeline. Both, classification and shading, can be performed before or after reconstruction.

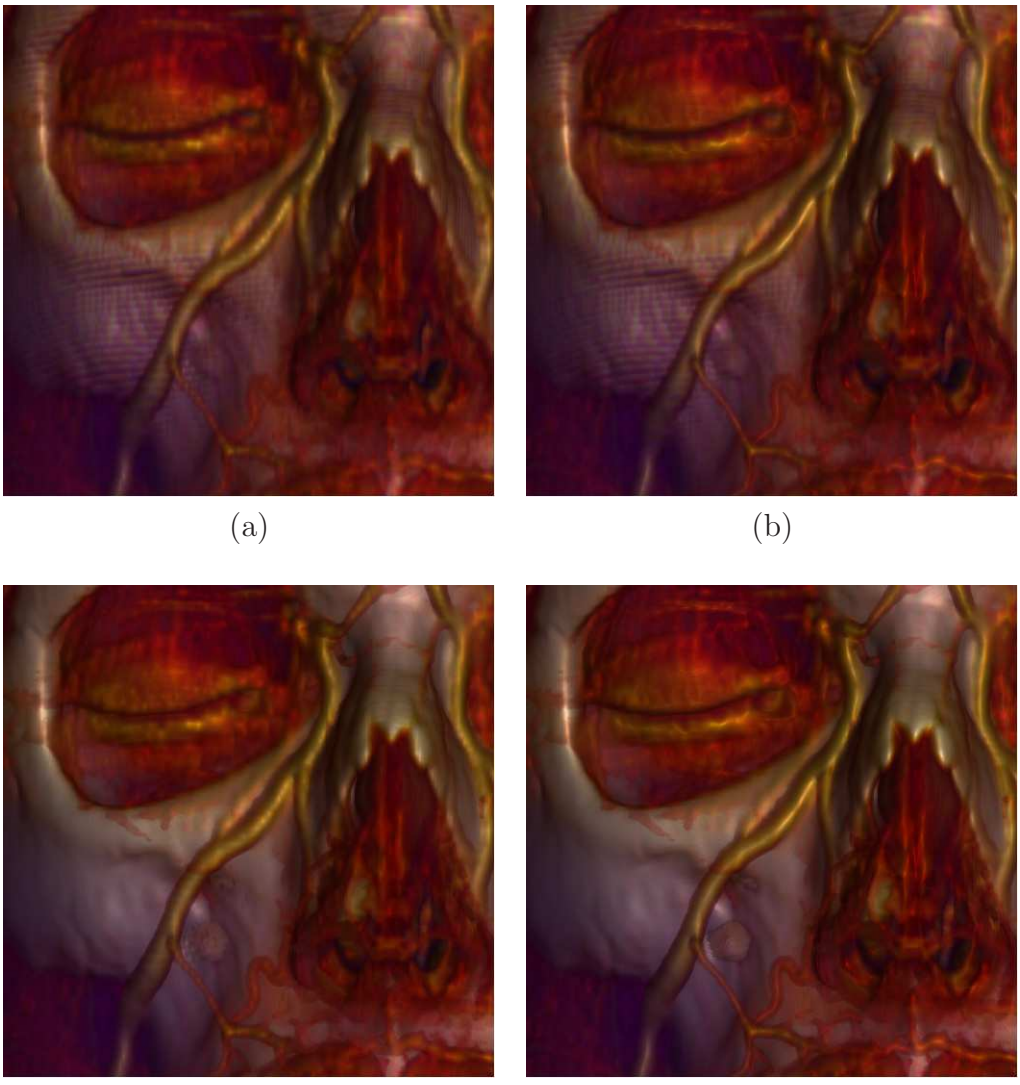
Classification Classification is the process of assigning a color and an opacity to a data value.

Shading We use the term shading for the process of evaluating the illumination model.

Compositing Compositing determines the contribution of a classified and shaded sample to the final image.

There are several possibilities in which order these steps can be performed (see Figure 3.1): Both classification and shading can occur before reconstruction (pre-classification, pre-shading) or after reconstruction (post-classification, post-shading).

Pre-classification assigns a color and an opacity to the samples before applying a reconstruction filter. Post-classification, on the other hand, applies to reconstruction filter to the original sample values and then classifies the reconstructed function values. Pre- and post-classification will produce different results, whenever the reconstruction does not commute with the transfer functions. As the reconstruction is usually non-linear, it will only commute with the transfer functions if the transfer functions are constant or



(a) (b)

(c) (d)

Figure 3.2: Comparison of classification and shading orders. (a) pre-classification and pre-shading. (b) pre-classification and post-shading. (c) post-classification and pre-shading. (d) post-classification and post-shading.

identity.

Pre-shading means that the illumination model is evaluated at the grid points only. Post-shading reconstructs the gradient (and/or other parameters required for evaluating the illumination model) at every resample location and then evaluates the illumination model. Again, pre-shading and post-shading are only equivalent if the lighting model is constant or an identity function of its parameters.

The loss in image quality caused by pre-classification and pre-shading has been previously discussed [5]. We have performed a comparison of all four variants using floating-point precision calculations throughout the pipeline. As shown in Figure 3.2, pre-classification causes the most severe artifacts. The differences between pre-shading and post-shading are more subtle, but still clearly visible. Our approach therefore uses both, post-classification and post-shading.

The following sections will discuss reconstruction, classification, shading, and compositing in detail.

3.2.1 Reconstruction

To render images from volume datasets, it is necessary to reconstruct a continuous function from the data samples. In many cases the first derivative, or gradient, of this function also has to be reconstructed, e.g. for shading.

Function Reconstruction

A point sample can be represented as a scaled Dirac impulse function. Sampling a signal is equivalent to multiplying it by a grid of impulses, one at each sample point, as shown in Figure 3.3 [29]. The Fourier transform of a two-dimensional impulse grid with frequency f_x in x and f_y in y is itself a grid of impulses with period f_x in x and f_y in y. If we call the impulse grid $k(x, y)$ and the signal $g(x, y)$, then the Fourier transform of the sampled signal, $\hat{gk}$, is $\hat{g} * \hat{k}$. Since k is an impulse grid, convolving $\hat{g}$ with $\hat{k}$ amounts to duplicating $\hat{g}$ at every point of $\hat{k}$, producing the spectrum shown at bottom right in Figure 3.3. The copy of $\hat{g}$ centered at zero is the primary spectrum, and

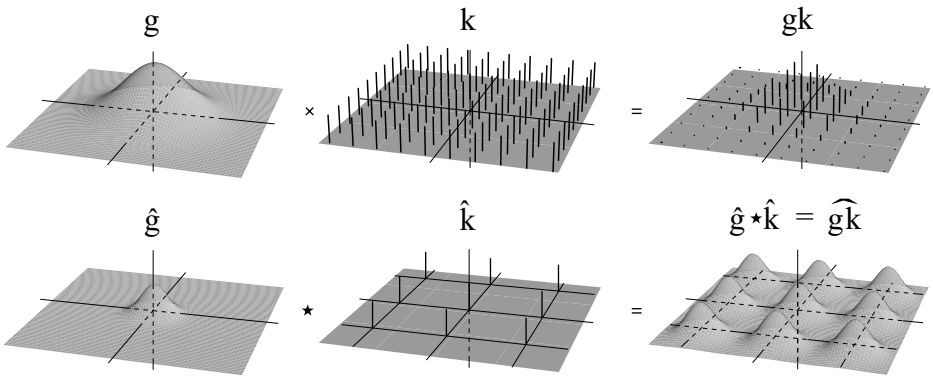


Figure 3.3: Two dimensional sampling in the space and frequency domain [29]. In the space domain (top), sampling corresponds to the multiplication of the original function with an impulse grid. In the frequency domain (bottom), sampling corresponds to the convolution with an impulse grid.

the other copies are called alias spectra. If $\hat{g}$ is zero outside a small enough region that the alias spectra do not overlap the primary spectrum, then $\hat{g}$ can be recovered by multiplying $\widehat{gk}$ by a function $\hat{h}$ which is one inside that region and zero elsewhere. Such a multiplication is equivalent to convolving the sampled data gk with h, the inverse transform of $\hat{h}$. This convolution with h allows us to reconstruct the original signal g by removing, or filtering out, the alias spectra, so we call h a reconstruction filter.

Thus, the goal of reconstruction is to extract the primary spectrum and to suppress the alias spectra. Since the primary spectrum comprises the low frequencies, the reconstruction filter is a low-pass filter. It is clear from Figure 3.3 that the simplest region to which we could limit $\hat{g}$ is the region of frequencies that are less than half the sampling frequency along each axis. We call this limiting frequency the Nyquist frequency and the region the Nyquist region. An ideal reconstruction filter can then be defined to have a Fourier transform that has the value one in the Nyquist region and zero outside it.

Extending the above to three-dimensional signals as used in volume rendering, the sampling grid becomes a three-dimensional lattice and the Nyquist region a cube. Given this Nyquist region, the ideal convolution filter is the

inverse transform of a cube, which is the product of three sinc functions.

$$h_I(x, y, z) = (2f_N)^3 \text{sinc}(2f_N x)\text{sinc}(2f_N y)\text{sinc}(2f_N z) \tag{3.1}$$

Thus, in principle, a volume signal can be exactly reconstructed from its samples by convolving with h_I. In practice, however, h_I cannot be implemented, because it has infinite extent in the spacial domain. Practical filters will therefore introduce some artifacts into the reconstructed function. A practical filter takes a weighted sum of a limited number of samples to compute the reconstruction of a point. That is, it is zero outside some finite region. This region is called the region of support. Filters with a larger region of support are generally more expensive since more samples have to be weighted.

The simplest interpolation function is known as zero-order interpolation, which is actually just a nearest neighbor function. The value at any location is simply the value of the closest sample to that location. One common interpolation function is a piecewise function known as first-order interpolation, or trilinear interpolation. With this function, the value is assumed to vary linearly along directions parallel to one of the major axes. Let the point p lie at location $(x_p, y_p, z_p)^T$ within a cubic cell and let $v_{000}, ..., v_{111}$ be the sample values at the eight corners of the cell. The value v_p at location p, according to trilinear interpolation, is then:

$$\begin{aligned} v_p \;=\; & v_{000}(1-x_p)(1-y_p)(1-z_p)+ \\ & v_{100}x_p(1-y_p)(1-z_p)+ \\ & v_{010}(1-x_p)y_p(1-z_p)+ \\ & v_{001}(1-x_p)(1-y_p)z_p+ \\ & v_{011}(1-x_p)y_pz_p+ \\ & v_{101}x_p(1-y_p)z_p+ \\ & v_{110}x_py_p(1-z_p)+ \\ & v_{111}x_py_pz_p \end{aligned} \tag{3.2}$$

Marschner and Lobb [29] have examined various reconstruction filters. The best results were achieved with windowed sinc filters. While they pro-

vide superior reconstruction quality, they are also about two orders of magnitude more expensive than trilinear interpolation. Therefore, when interactive performance is desired, trilinear reconstruction is often the method of choice.

Gradient Reconstruction

For volume rendering, not only the original function needs to be reconstructed. Especially the first derivative of the three-dimensional function, which is called the gradient, is quite important since it can be interpreted as the normal of an iso-surface. Gradients are used in shading and thus have considerable influence on image quality. Möller et al. [34] even concede gradient reconstruction having a greater impact on image quality than function reconstruction itself. The ideal gradient reconstruction filter is the derivative of the sinc filter, called cosc, which again has infinite extent and therefore cannot be used in practise.

Möller et al. [34] have examined different methods for gradient reconstruction. They have identified the following basic approaches:

Derivative First (DF) This method computes gradients at grid points of the rectilinear grid formed by the data samples. The gradient at a resample location is determined by interpolation between the gradients at the neighboring grid points.

Interpolation First (IF) This methods computes the derivative from a set of additionally interpolated samples at the resample location.

Continuous Derivative (CD) This approach uses a derivative filter which is pre-convolved with the interpolation filter. The gradient at a resample location is computed by convolving the volume by this combined filter.

Analytic Derivative (AD) This method uses a special gradient filter derived from the interpolation filter for gradient estimation.

In their work, they prove that DF, IF and CD are numerically equivalent and show that the AD method delivers bad results in some cases. An impor-

tant point of their work is the conclusion that the IF method outperforms the common DF method.

They state that the cost (using caching) of the DF method is $nE_D + m(4E_H)$ and the cost of the IF method is $m(E_D+E_H)$, where m is the number of voxels, n is the number of samples, E_D is the computational effort of gradient estimation, and E_H is the computational effort of the interpolation.

However, we recognize that if an expensive gradient estimation method is used, i.e. E_D is much larger than E_H, and the sampling rate is high, i.e. m is much larger than n, the DF method has advantages. Since the gradient estimation only is performed at grid points, a higher sampling rate does not increase the number of necessary gradient estimations. Additionally, from a practical point of view the DF method has other advantages: Modern CPUs provide SIMD extensions which allow to perform operations simultaneously on multiple data items. For the DF method this means that, assuming the same interpolation method is used for gradient and function reconstruction, the interpolation of function value and gradient can be performed simultaneously (i.e., since the function value has to be interpolated anyway, gradient interpolation almost comes for free). Using the IF method, this is not possible, since different filters are used for function and gradient reconstruction.

Central and intermediate differences are two of the most popular gradient estimation methods. However, since they use a small neighborhood they are very sensitive to noise contained in the dataset. Filters which use larger neighborhood therefore in general result in better image quality. This is especially true for medical datasets, which are often strongly undersampled in z-direction.

Neumann et al. [38] have presented a theoretical framework for gradient reconstruction based on linear regression which is a generalization of many previous approaches. The approach linearly approximates the three-dimensional function $f(x, y, z)$ according to the following formula:

$$f(x, y, z) \approx Ax + By + Cz + D \tag{3.3}$$

The approximation tries to fit a 3D regression hyperplane onto the sam-

pled values assuming that the function changes linearly in the direction of the plane normal $n = (A, B, C)^T$. The value D is the approximate density value at the origin of the local coordinate system. They derive a 4D error function and examine its partial derivatives for the four unknown variables. Since these partial derivatives have to equal zero at the minimum location of the error function, they end up with a system of linear equations. Assuming the voxels to be located at regular grid points leads to a diagonal coefficient matrix. Thus, the unknown variables A, B, C, D can be calculated by simple linear convolution:

$$\begin{aligned} A = w_A \sum_{k=0}^{26} w_k f_k x_k, B = w_B \sum_{k=0}^{26} w_k f_k y_k, \\ C = w_C \sum_{k=0}^{26} w_k f_k z_k, D = w_D \sum_{k=0}^{26} w_k f_k \end{aligned} \tag{3.4}$$

with

$$\begin{aligned} w_A = \frac{1}{\sum_0^{26} w_k x_k^2}, w_B = \frac{1}{\sum_0^{26} w_k y_k^2}, \\ w_C = \frac{1}{\sum_0^{26} w_k z_k^2}, w_D = \frac{1}{\sum_0^{26} w_k} \end{aligned} \tag{3.5}$$

The w_k are the weights of the weighting function, an arbitrary spherically symmetric function, which is monotonically decreasing as the distance from the local origin is getting larger. k denotes the index of a voxel with an offset $(x, y, z)^T$ in the 26-neighborhood of the current voxel and is defined as:

$$k = 9(z + 1) + 3(y + 1) + (x + 1) \tag{3.6}$$

The vector $(A, B, C)^T$ is an estimate for the gradient at the local origin and the value D is the filtered function value at the local origin. Using the filtered values instead of the original samples leads to strong correlation between the data values and the estimated gradients. These low-pass filtered values come as by-product of gradient estimation at little additional cost. Using this estimation method for an arbitrary resample location, however, requires additional computational effort. It is necessary to perform a matrix inversion and a matrix multiplication at each location. Thus, the gradient estimation using Neumann's approach is much cheaper, if gradients are only computed at grid points. However, we do not pre-compute the gradients since

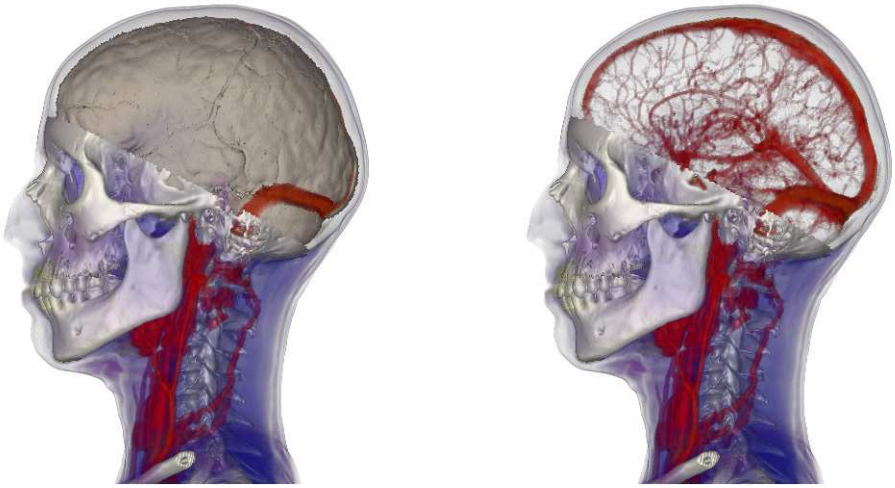
this would require a considerable amount of additional memory. Instead, the gradients and filtered values are computed on-the-fly for each cell. Trilinear interpolation is then used to calculate the function value and gradient at each resample location.

Additionally, this approach has other advantages: Since nothing is pre-computed, different gradient estimation and reconstruction filters can be implemented and simply changed at run-time without requiring further processing. It also helps to solve the problem of using the filtered values instead of the original samples, because the original dataset is still present and the additional filtering can be disabled by the user. This is important in medical applications, since some fine details might disappear due to filtering.

3.2.2 Classification

Classification is the process of assigning a color and opacity to a reconstructed function value. Transfer functions, usually implemented as lookup tables, are used for this mapping. During rendering, the reconstructed function value serves as an index for the lookup tables, which contain color and opacity values. Levoy first suggested the use of one-dimensional piecewise linear transfer functions [24]. Additionally, he used the gradient magnitude for opacity modulation, which effectively adds a second dimension. Opacity modulation enhances regions with high gradients and reduces the opacity of homogenous regions. Multi-dimensional transfer functions are a more general approach which has proven to provide more control over the appearance of the rendering [15]. However, these functions also require higher order derivatives. These have to be either pre-computed, which increases memory usage, or calculated on-the-fly, which decreases performance.

We therefore support one dimensional transfer functions including optional opacity modulation based on the gradient magnitude. Additionally, we support segmentation by assigning an object index to each voxel. For every object, an independent transfer function can be defined. Since 12 bit voxels are standard for medical datasets, the remaining 4 bits can be used for the segmentation information, as the data has to be aligned to byte bound-



(a) (b)

Figure 3.4: Example of segmented dataset. Four objects have been segmented: skullcap, blood vessels, brain, and background. (a) the skullcap has been disabled to display the brain. (b) skullcap and brain have been disabled to reveal occluded blood vessels.

aries for better performance. This allows up to 16 objects to be defined. A separate transfer function can be assigned to each of these objects and individual objects can be enabled or disabled (see Figure 3.4).

3.2.3 Shading

Despite its lacking physical validity, the Phong illumination model [42] is still widely used in computer graphics. Its popularity is most probably based on its simplicity. Phong's model is a local illumination model, which means only direct reflections are taken into account. While this may not be very realistic, it allows illumination to be computed efficiently. The model consists of an independent ambient, diffuse and specular term. It has the following parameters:

Light vector L The light vector is the normalized vector from a location in space to the light source. In case of a directional light source, this vector is the same for all points in a scene.

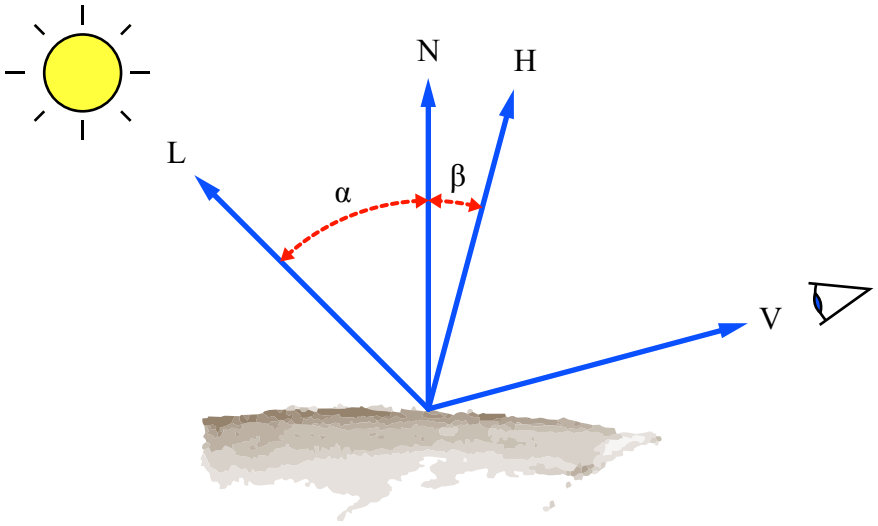


Figure 3.5: Parameters of the Phong illumination model. The light vector L points towards the light source and the view vector V points towards the viewer. N is the surface normal at the point the model is evaluated at. The half-way vector H is the vector half way between L and V.

View vector V The view vector is the normalized vector from a location in space along a viewing ray to its origin on the image plane. In case of parallel projection, this vector is the same for all points in a scene.

Surface normal N The Phong illumination model was originally designed for the rendering of surfaces. In volume rendering, the surface normal is approximated by the normalized gradient at the resample location.

These parameters are illustrated in Figure 3.5. Additionally, the half-way vector $H = \frac{1}{2}(L + V)$ is displayed.

Three constants, $F_{ambient}$, $F_{diffuse}$, and $F_{specular}$, control the contribution of each term to the final light intensity. The shaded color is computed by multiplying the input color (e.g. the color of a sample as obtained through the transfer function) by the sum of the three terms (see Equation 3.7). We assume here that the color of the light source is always white and can therefore disregard its color contribution.

$$c_{out} = c_{in}(I_{ambient} + I_{diffuse} + I_{specular}) \tag{3.7}$$

The ambient term (Equation 3.8) is constant. Its purpose it to simulate the contribution of indirect reflections, which are otherwise not accounted for by the model.

$$I_{ambient} = F_{ambient} \tag{3.8}$$

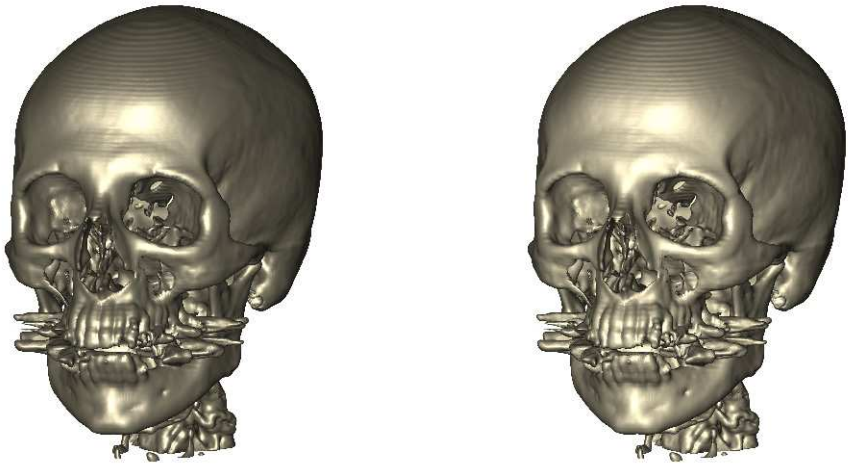
The diffuse term (Equation 3.9) is based on Lambert's cosine law which states that the reflection of a perfect rough surface is proportional to the cosine of the angle α between the light vector L and the surface normal N.

$$I_{diffuse} = F_{diffuse} \max(L \cdot N, 0) \tag{3.9}$$

The specular term (Equation 3.10)adds an artificial highlight to simulate specular reflections. For computing the specular term, Blinn proposed to use the half-way vector H [1], which is a vector halfway between the light vector and the view vector. The specular lighting intensity is then proportional to the cosine of the angle β between the half-way vector H and the surface normal N raised to the power of n, where n is called the specular exponent of the surface and represents its shininess. Higher values of n lead to smaller, sharper highlights, whereas lower values result in large and soft highlights.

$$I_{specular} = F_{specular} \max((H \cdot N)^n, 0) \tag{3.10}$$

Despite the low complexity of this illumination model, shading still has considerable impact on performance. One way to speed up the evaluation is the use of reflectance maps [52], which contain pre-computed illumination information. However, the use of such data structures requires a considerable amount of additional memory and can lead to cache thrashing. Furthermore, they have to be re-computed every time the illumination properties change. Thus, we choose to evaluate the illumination model on-the-fly. The most time consuming part of the model is the exponentiation used in the specular term. However, since this is a purely empirical model, every function that evokes a similar visual impression can be used instead of the exponentiation. Schlick therefore proposed to use the following approximation [47]:



(a) (b)

Figure 3.6: Visual comparison of specular highlights obtained with Phong's and Schlick's approach. (a) Phong. (b) Schlick. The specular exponent is 16 in both cases.

$$x^n \approx \frac{x}{n - nx + x} \tag{3.11}$$

Schlick's approximation is generally much faster to compute and yields to very similar results (see Figure 3.6). Figure 3.7 shows a comparison of the original function and the approximation for different values of n.

Our system uses the Phong illumination model with Schlick's approximation for the specular term. One directional light source is supported. This allows us to compute shading at little cost. This setup is well suited for medical applications, where the user generally does not benefit from (and might even be disturbed by) increased photorealism.

3.2.4 Compositing

In raycasting, the volume rendering integral is approximated by repeated application of the over-operator [54, 43] in front-to-back order. That is, at each resample location, the current color and alpha values for a ray are computed in the following way:

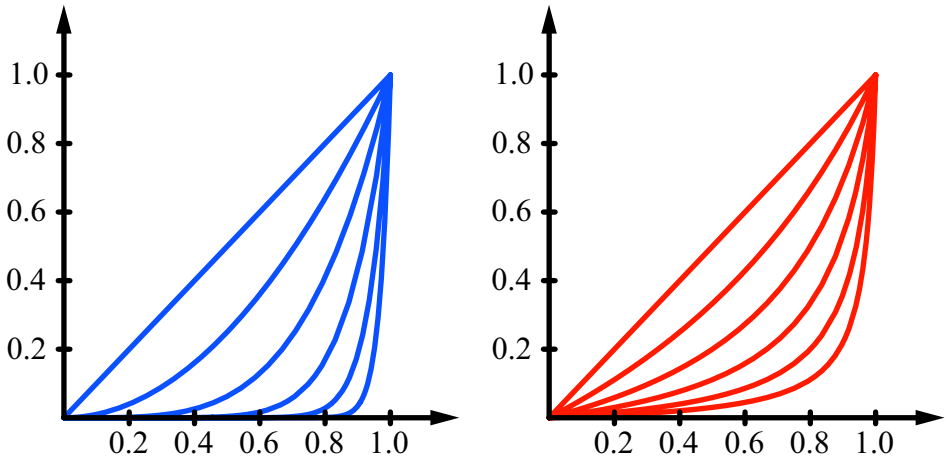


Figure 3.7: Comparison of Phong's and Schlick's approach for specular highlight simulation. (a) Phong: $f(x) = x^n$. (b) Schlick: $f(x) = \frac{x}{n-nx+x}$. $n = 1, 2, 4, 8, 16, 32$.

$$\begin{aligned} c_{out} &= c_{in} + c(x)\alpha(x)(1-\alpha_{in}) \\ \alpha_{out} &= \alpha_{in} + \alpha(x)(1-\alpha_{in}) \end{aligned} \tag{3.12}$$

c_{in} and α_{in} are the color and opacity the ray has accumulated so far. x is the reconstructed function value and $c(x)$ and $\alpha(x)$ are the classified and shaded color and opacity for this value. The advantage of using the front-to-back formulation of the over-operator is the possibility of early ray termination. As soon as a ray has accumulated full opacity (i.e., $\alpha_{out} = 1$), no further processing has to be done for this ray.

This formulation is only valid if compositing is performed at evenly-spaced locations at a distance of 1. If the object sample distance, i.e., the distance between subsequent samples along a ray, differs from 1, opacity correction is needed. Assuming an equal object sample distance for all rays, opacity correction can be achieved by using a corrected lookup table for the opacity values:

$$\alpha_{corr}(x) = 1 - (1 - \alpha(x))^{\Delta s} \tag{3.13}$$

where α_{corr} is the corrected opacity transfer function, α is the original

Level	Latency	Size
Register	1 ns - 3 ns	1 KB
Level 1 Cache	2 ns - 8 ns	8 KB - 128 KB
Level 2 Cache	5 ns - 12 ns	0.5 MB - 8 MB
Main Memory	10 ns - 60 ns	256 MB - 2 GB
Hard Disk	8 ms - 12 ms	100 GB - 200 GB

Table 3.1: Memory hierarchy of modern computer architectures. Memory is structured as a hierarchy of successively larger but slower storage technology.

opacity transfer function, and Δs is the object sample distance. In Equation 3.12, α is then replaced by α_{corr}.

3.3 Memory Management for Large Datasets

The past years have shown that the discrepancy between processor and memory performance is rapidly increasing, making memory access a potential bottleneck for applications which have to access large amounts of data. Raycasting, in particular, is prone to cause problems, since it generally leads to irregular memory access patterns. This section discusses strategies to improve memory access patterns taking advantage of the memory hierarchy.

The memory of contemporary computers is structured in a hierarchy of successively larger, slower, and cheaper memory levels. Each level contains a working copy or cache of the level above. Recent developments in processor and memory technology imply an increasing penalty if programs do not take optimal advantage of the memory hierarchy. The memory hierarchy of a x86-based system is shown in Table 3.1. The L1 cache is used to temporarily store instructions and data, making sure the processor has a steady supply of data to process while the memory catches up delivering new data. The L2 cache is the high speed memory between the processor and main memory.

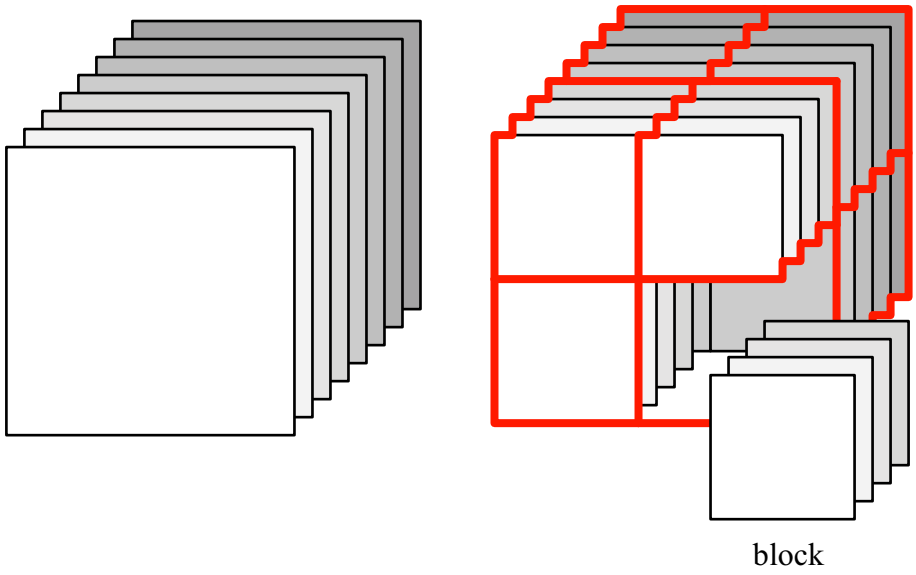
Going up the cache hierarchy towards the CPU, caches get smaller and faster. In general, if the CPU issues an operation on a data item, the request is propagated down the cache hierarchy until the requested data is found. It is very time consuming if the data is only found in a slow cache. This is due to the propagation itself as well as to the back propagation of data

through all the caches. For good performance, frequent access to the slower caches has to be avoided. Accessing the slower caches, like hard disk and main memory, only once would be optimal. We assume that there is enough main memory available to hold the volume data and all other data structures necessary - the hard disk only has to be accessed when a volume is loaded. Thus, the main focus lies in optimizing main memory access.

3.3.1 Bricking

The most common way of storing volumetric data is a linear volume layout. Volumes are typically thought of as a stack of two-dimensional images (slices) which are stored in an array linearly. The work-flow of a standard volume raycasting algorithm on a linearly stored volume is as follows: For every pixel of the image plane a ray is cast through the volume and the volume data is resampled along this ray. At every resample position resampling, gradient computation, shading, and compositing is performed. The closer the neighboring rays are to each other, the higher the probability is that they partially process the same data. Given the fact that rays are shot one after the other, it is very likely that the same data has to be read several times from main memory, because in general the cache is not large enough to hold the processed data of a single ray. This problem can be targeted by a technique called tile casting. Here, rather than processing one ray completely, each pass processes only one resample point for every ray. However, different viewing directions still cause a different amounts of cache line requests to load the necessary data from main memory which leads to a varying frame-rate.

The concept of bricking supposes the decomposition of data into small fixed-sized data blocks (see Figure 3.8). Each block is stored in linear order. The basic idea is to choose the block size according to the cache size of the architecture so that an entire block fits into a fast cache of the system. It has been shown that bricking is one way to achieve high cache coherency, without increasing memory usage [40]. However, accessing data in a bricked volume layout is very costly.



(a) (b)

Figure 3.8: Linear and bricked volume layouts. (a) linear volume layout stored as a stack of slices. (b) bricked volume layout stored as a set of blocks.

3.3.2 Addressing

In raycasting the eight samples closest to the current resample location are required in every processing step when trilinear interpolation is used. In a linear volume layout, these samples can be addressed by adding constant offsets to one known address. The necessary address computations are given in Algorithm 1.

Algorithm 1 ComputeSampleAddressesLinear(i,j,k)

$$
\begin{aligned}
sample_{i,j,k} &= i + j \cdot V_x + k \cdot V_x \cdot V_y \\
sample_{i+1,j,k} &= sample_{i,j,k} + 1 \\
sample_{i,j+1,k} &= sample_{i,j,k} + V_x \\
sample_{i+1,j+1,k} &= sample_{i,j,k} + 1 + V_x \\
sample_{i,j,k+1} &= sample_{i,j,k} + V_x \cdot V_y \\
sample_{i+1,j,k+1} &= sample_{i,j,k} + 1 + V_x \cdot V_y \\
sample_{i,j+1,k+1} &= sample_{i,j,k} + V_x + D_x \cdot V_y \\
sample_{i+1,j+1,k+1} &= sample_{i,j,k} + 1 + V_x + V_x \cdot V_y
\end{aligned}
$$

$V_{\{x,y,z\}}$ are the volume dimensions and i, j, k are the integer parts of the

current resample position. This addressing scheme is very efficient. Once the lower left sample is determined the other needed samples can be accessed just by adding an offset.

The evolution of CPU design shows that the length of CPU pipelines grows progressively. This is very efficient as long as conditional branches do not initiate pipeline flushes. Once a long instruction pipeline is flushed there is a significant delay until it is refilled. Most of the present systems use branch prediction. The CPU normally assumes that if-branches will always be executed. It executes the if-branch before actually checking the outcome of the if-clause. If the if-clause returns false, the else-branch has to be executed. This means that the CPU flushes the pipeline and refills it with the else-branch. This is very time consuming, due to the increasing size of the pipelines.

Algorithm 2 ComputeAddressBricked(i,j,k)

$u = (i \bmod B_x) + (j \bmod B_y) \cdot B_x + (k \bmod B_z) \cdot B_x \cdot B_y$
$v = (i/B_x) + (j/B_y) \cdot (V_x/B_x) + (k/B_z) \cdot (V_x/B_x) \cdot (V_y/B_y)$
return $(u \cdot B_x \cdot B_y \cdot B_z + v)$

Algorithm 3 ComputeSampleAddressesBricked(i,j,k)

$sample_{i,j,k}$ = ComputeAddressBricked(i,j,k)
if $(i \bmod B_x) < B_x - 1$ and $(j \bmod B_y) < B_y - 1$ and $(k \bmod B_z) < B_z - 1$ **then**

$sample_{i+1,j,k}$	=	$sample_{i,j,k} + 1$
$sample_{i,j+1,k}$	=	$sample_{i,j,k} + B_x$
$sample_{i+1,j+1,k}$	=	$sample_{i,j,k} + 1 + B_x$
$sample_{i,j,k+1}$	=	$sample_{i,j,k} + B_x \cdot B_y$
$sample_{i+1,j,k+1}$	=	$sample_{i,j,k} + 1 + B_x \cdot B_y$
$sample_{i,j+1,k+1}$	=	$sample_{i,j,k} + B_x + B_x \cdot B_y$
$sample_{i+1,j+1,k+1}$	=	$sample_{i,j,k} + 1 + B_x + B_x \cdot B_y$

else

$sample_{i+1,j,k}$	=	ComputeAddressBricked($i+1$,j,k)
$sample_{i,j+1,k}$	=	ComputeAddressBricked(i,$j+1$,k)
$sample_{i+1,j+1,k}$	=	ComputeAddressBricked($i+1$,$j+1$,k)
$sample_{i,j,k+1}$	=	ComputeAddressBricked(i,j,$k+1$)
$sample_{i+1,j,k+1}$	=	ComputeAddressBricked($i+1$,j,$k+1$)
$sample_{i,j+1,k+1}$	=	ComputeAddressBricked(i,$j+1$,$k+1$)
$sample_{i+1,j+1,k+1}$	=	ComputeAddressBricked($i+1$,$j+1$,$k+1$)

end if

Using a bricked volume layout one will encounter this problem. The

addressing of data in a bricked volume layout is more costly than in a linear volume layout. To address one data element, one has to address the block itself and the element within the block. The necessary address computation is given in Algorithm 2. $B_{\{x,y,z\}}$ are block dimensions, $V_{\{x,y,z\}}$ are the volume dimensions, i, j, and k are the integer parts of the current resample position.

In contrast to this addressing scheme, a linear volume can be seen as one large block. To address a sample it is enough to compute just one offset. In algorithms like volume raycasting, which need to address a certain neighborhood of data in each processing step, the computation of two offsets has a considerable impact on performance. To avoid this performance penalty, one can construct an if-else statement (see Algorithm 3). The if-clause checks if the needed data elements can be addressed within one block. If the outcome is true, the data elements can be addressed as fast as in a linear volume. If the outcome is false, the costly address calculations have to be done. This simplifies address calculation, but the involved if-else statement incurs pipeline flushes. In the following, we address this problem.

To avoid the costly if-else statement and the expensive address computations, one can create a lookup table to address all the needed samples. The straight-forward approach would be to create a lookup table for each possible sample position in a block. For a block of 32^3 this would lead to 32^3 different lookup tables to address the neighboring samples. In the resampling case, 7 neighbors need to be addressed - accordingly the size of the lookup tables would be $32^3 \cdot 7 \cdot 4$ bytes = 896 KB (4 bytes per offset). For accessing a 26-neighborhood a table of $32^3 \cdot 26 \cdot 4$ bytes = 3.25 MB (4 Bytes per offset) would be required. Such a large lookup table is not preferable, due to the limited size of cache. However, the addressing of such a lookup table would be straight-forward, because the indices in the lookup table would be the corresponding offsets of the current sample position, assuming the offset is given relative to the block memory address.

To reduce the size of the lookup table, the possible sample positions can be distinguished by the locations of the needed neighboring samples. The first sample location (i, j, k) is defined by the integer parts of the current resample position. Assuming trilinear interpolation, during resampling neighboring

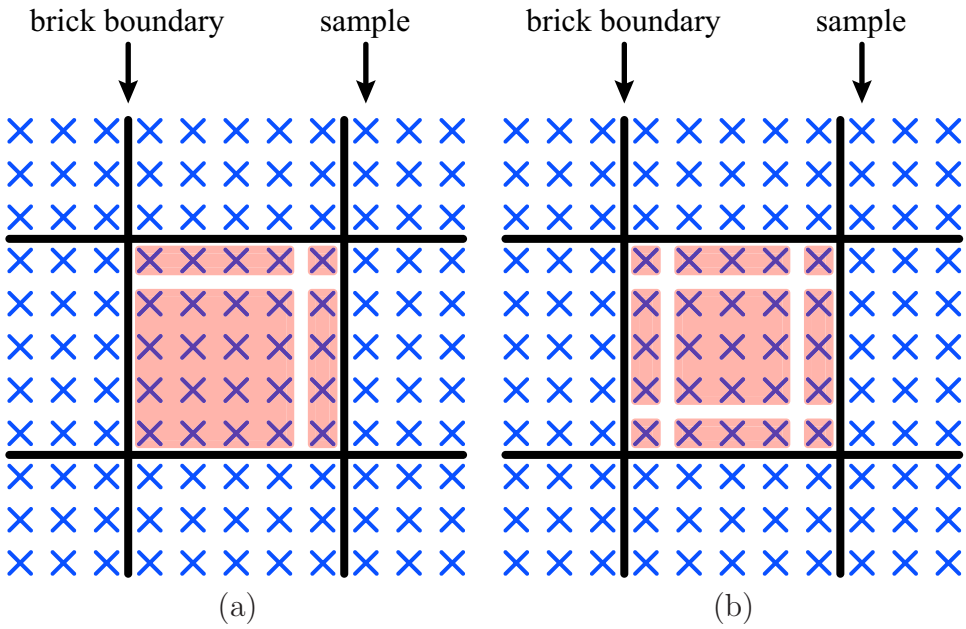


Figure 3.9: Access patterns during resampling and gradient computation. (a) typical access pattern during resampling (8-neighborhood). (b) typical access pattern during gradient computation (26-neighborhood).

Case	$(i \bmod B_x) \in$	$(j \bmod B_y) \in$	$(k \bmod B_z) \in$
0	$\{0, ..., B_x - 2\}$	$\{0, ..., B_y - 2\}$	$\{0, ..., B_z - 2\}$
1	$\{0, ..., B_x - 2\}$	$\{0, ..., B_y - 2\}$	$\{B_z - 1\}$
2	$\{0, ..., B_x - 2\}$	$\{B_y - 1\}$	$\{0, ...,\mathrm{B}_z - 2\}$
3	$\{0, ..., B_x - 2\}$	$\{B_y - 1\}$	$\{B_z - 1\}$
4	$\{B_x - 1\}$	$\{0, ...,\mathrm{B}_y - 2\}$	$\{0, ...,\mathrm{B}_z - 2\}$
5	$\{B_x - 1\}$	$\{0, ...,\mathrm{B}_y - 2\}$	$\{B_z - 1\}$
6	$\{B_x - 1\}$	$\{B_y - 1\}$	$\{0, ..., B_z - 2\}$
7	$\{B_x - 1\}$	$\{B_y - 1\}$	$\{B_z - 1\}$

Table 3.2: Cases for the position within a block for 8-neighborhood addressing. For every case, the range of each component of the three-dimensional position within a block is displayed.

samples to the right, top, and back of the current location are required. A block can be subdivided into subsets. For each subset, we can determine the blocks in which the neighboring samples lie. Therefore, it is possible to store these offsets in a lookup table [9]. This is illustrated in Figure 3.9 (a). We see that there are four basic cases, which can be derived from the current sample location. This can be mapped straightforwardly to 3D, which leads to eight distinct cases. These eight cases are shown in Table 3.2.

In the following, we construct a function to efficiently address the lookup table. The input parameters of the lookup table addressing function are the sample position (i, j, k) and the block dimensions B_x, B_y, and B_z. We assume that the block dimensions are a power of two, i.e., $B_x = 2^{N_x}$, $B_y = 2^{N_y}$, and $B_z = 2^{N_z}$. As a first step, the block offset parts of i, j, and k are extracted by a conjunction with the corresponding $B_{\{x,y,z\}} - 1$. The next step is to increase all by one to move the maximum possible value of $B_{\{x,y,z\}} - 1$ to $B_{\{x,y,z\}}$. All the other possible values stay within the range $[1, B_{\{x,y,z\}} - 1]$. Then a conjunction of the resulting value and the complement of $B_{\{x,y,z\}} - 1$ is performed, which maps the input values to $[0, B_{\{x,y,z\}}]$. The last step is to add the three values and divide the result by the minimum of the block dimensions, which maps the result to [0,7]. This last division can be exchanged by a shift operation. Algorithm 4 shows the lookup table index computation and Algorithm 5 shows how the neighboring sample offsets

Case	i_a	j_a	k_a	i_b	j_b	k_b	i'	j'	k'	$index$
0	0-30	0-14	0-6	1-31	1-15	1-7	0	0	0	0
1	0-30	0-14	7	1-31	1-15	8	0	0	8	1
2	0-30	15	0-6	1-31	16	1-7	0	16	0	2
3	0-30	15	7	1-31	16	8	0	16	8	3
4	31	0-14	0-6	32	1-15	1-7	32	0	0	4
5	31	0-14	7	32	1-15	8	32	0	8	5
6	31	15	0-6	32	16	1-7	32	16	0	6
7	31	15	7	32	16	8	32	16	8	7

$$
\begin{aligned}
index &= ((i' + j' + k') \gg \min(N_x, N_y, N_z)) \\
\{i', j', k'\} &= \underbrace{(\underbrace{(\{i, j, k\} \,\&\, (B_{\{x,y,z\}} - 1))}_{\{i_a, j_a, k_a\}} + 1) \,\&\, \sim (B_{\{x,y,z\}} - 1)}_{\{i_b, j_b, k_b\}}
\end{aligned}
$$

Table 3.3: Lookup table index calculation for 8-neighborhood. The calculation of the lookup table index is shown for $B_x = 32$, $B_y = 16$, $B_z = 8$

can be computed using the lookup table. We use & to denote a *bitwise and* operation, | to denote a *bitwise or* operation, $\gg$ to denote a *right shift* operation, and $\sim$ to denote a *bitwise negation*. In Table 3.3 we give an example of the calculation for a block size of $32 \times 16 \times 8$.

Algorithm 4 ComputeResampleLookupIndex(i,j,k)

$i' = ((i \,\&\, (B_x - 1)) + 1) \,\&\, \sim (B_x - 1)$
$j' = ((j \,\&\, (B_y - 1)) + 1) \,\&\, \sim (B_y - 1)$
$k' = ((k \,\&\, (B_z - 1)) + 1) \,\&\, \sim (B_z - 1)$
return $((i' + j' + k') \gg \min(N_x, N_y, N_z))$

Algorithm 5 ComputeSampleAddressesResampleLookup(i,j,k)

$sample_{i,j,k}$	=	ComputeAddressBricked(i,j,k)
$index$	=	ComputeResampleLookupIndex(i,j,k)
$sample_{i+1,j,k}$	=	$sample_{i,j,k} + LUT[index][0]$
$sample_{i,j+1,k}$	=	$sample_{i,j,k} + LUT[index][1]$
$sample_{i+1,j+1,k}$	=	$sample_{i,j,k} + LUT[index][2]$
$sample_{i,j,k+1}$	=	$sample_{i,j,k} + LUT[index][3]$
$sample_{i+1,j,k+1}$	=	$sample_{i,j,k} + LUT[index][4]$
$sample_{i,j+1,k+1}$	=	$sample_{i,j,k} + LUT[index][5]$
$sample_{i+1,j+1,k+1}$	=	$sample_{i,j,k} + LUT[index][6]$

A similar approach can be done for the gradient computation. We present a general solution for a 26-connected neighborhood. Here we can, analogous to the resample case, distinguish 27 cases. For 2D, this is illustrated in Figure

Case	i_a	j_a	k_a	i_b	j_b	k_b	i_c	j_c	k_c	i'	j'	k'	$index$
0	1-30	1-14	1-6	0-29	0-13	0-5	2-30	2-14	2-6	0	0	0	0
1	1-30	1-14	7	0-29	0-13	6	2-30	2-14	8	0	0	1	1
2	1-30	1-14	0	0-29	0-13	15	2-30	2-14	16	0	0	2	2
3	1-30	15	1-6	0-29	14	0-5	2-30	16	2-6	0	1	0	3
4	1-30	15	7	0-29	14	6	2-30	16	8	0	1	1	4
5	1-30	15	0	0-29	14	15	2-30	16	16	0	1	2	5
6	1-30	0	1-6	0-29	31	0-5	2-30	32	2-6	0	2	0	6
7	1-30	0	7	0-29	31	6	2-30	32	8	0	2	1	7
8	1-30	0	0	0-29	31	15	2-30	32	16	0	2	2	8
9	31	1-14	1-6	30	0-13	0-5	32	2-14	2-6	1	0	0	9
10	31	1-14	7	30	0-13	6	32	2-14	8	1	0	1	10
11	31	1-14	0	30	0-13	15	32	2-14	16	1	0	2	11
12	31	15	1-6	30	14	0-5	32	16	2-6	1	1	0	12
13	31	15	7	30	14	6	32	16	8	1	1	1	13
14	31	15	0	30	14	15	32	16	16	1	1	2	14
15	31	0	1-6	30	31	0-5	32	32	2-6	1	2	0	15
16	31	0	7	30	31	6	32	32	8	1	2	1	16
17	31	0	0	30	31	15	32	32	16	1	2	2	17
18	0	1-14	1-6	63	0-13	0-5	64	2-14	2-6	2	0	0	18
19	0	1-14	7	63	0-13	6	64	2-14	8	2	0	1	19
20	0	1-14	0	63	0-13	15	64	2-14	16	2	0	2	20
21	0	15	1-6	63	14	0-5	64	16	2-6	2	1	0	21
22	0	15	7	63	14	6	64	16	8	2	1	1	22
23	0	15	0	63	14	15	64	16	16	2	1	2	23
24	0	0	1-6	63	31	0-5	64	32	2-6	2	2	0	24
25	0	0	7	63	31	6	64	32	8	2	2	1	25
26	0	0	0	63	31	15	64	32	16	2	2	2	26

$$
\begin{aligned}
index &= (9i' + 3j' + k') \\
\{i', j', k'\} &= (((((\underbrace{\{i, j, k\} \,\&\, (B_{\{x,y,z\}} - 1)}_{\{i_a, j_a, k_a\}}) - 1) \,\&\, (2B_{\{x,y,z\}} - 1)) \mid 1) + 1) \gg N_{\{x,y,z\}}
\end{aligned}
$$

$$
\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\{i_b, j_b, k_b\}}
$$

$$
\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\{i_c, j_c, k_c\}}
$$

Table 3.4: Lookup table index calculation for 26-neighborhood. The calculation of the lookup table index is shown for $B_x = 32$, $B_y = 16$, $B_z = 8$.

3.9 (b). Depending on the position of sample (i, j, k) a block is subdivided into 27 subsets.

The first step is to extract the block offset, by a conjunction with $B_{\{x,y,z\}}-1$. Then we subtract one, and conjunct with $B_{\{x,y,z\}}+B_{\{x,y,z\}}-1$, to separate the case if one or more components are zero. In other words, zero is mapped to $2 \cdot B_{\{x,y,z\}}-1$. All the other values stay within the range $[0, B_{\{x,y,z\}}-2]$. To separate the case of one or more components being $B_{\{x,y,z\}}-1$, we add 1, after the previous subtraction is undone by a disjunction with 1, without loosing the separation of the zero case. Now all the cases are mapped to $\{0, 1, 2\}$ to obtain a ternary system. This is done by dividing the components by the corresponding block dimensions. These divisions can be replaced by faster shift operations. Then the three ternary variables are mapped to an index in the range of $[0, 26]$. The final lookup table index computation is given in Algorithm 6. In Table 3.4 we give an example of the calculation for a block size of $32 \times 16 \times 8$.

Algorithm 6 ComputeGradientLookupIndex(i,j,k)

$i' = (((((i \;\&\; (B_x - 1)) - 1) \;\&\; (2B_x - 1)) \mid 1) + 1) \gg N_x$
$j' = (((((i \;\&\; (B_x - 1)) - 1) \;\&\; (2B_x - 1)) \mid 1) + 1) \gg N_y$
$k' = (((((k \;\&\; (B_z - 1)) - 1) \;\&\; (2B_z - 1)) \mid 1) + 1) \gg N_z$
return $(9i' + 3j' + k')$

The presented index computations can be performed efficiently on current CPUs, since they only consist of simple bit manipulations. The lookup tables can be used in raycasting on a bricked volume layout for efficient access to neighboring samples. The first table can be used if only the eight samples within a cell have to be accessed (e.g., if gradients have been pre-computed). Compared to the if-else solution which has the costly address computation in the else branch, we get a speedup of about 30%. The benefit varies, depending on the block dimensions. For a $32 \times 32 \times 32$ block size the else-branch has to be executed in 10% of the cases and for a $16 \times 16 \times 16$ block size in 18% of the cases.

The second table allows full access to a 26-neighborhood. This approach reduces the cost of addressing by 40% compared to a if-else solution, as the else-branch has to be executed more often for a larger neighborhood. The

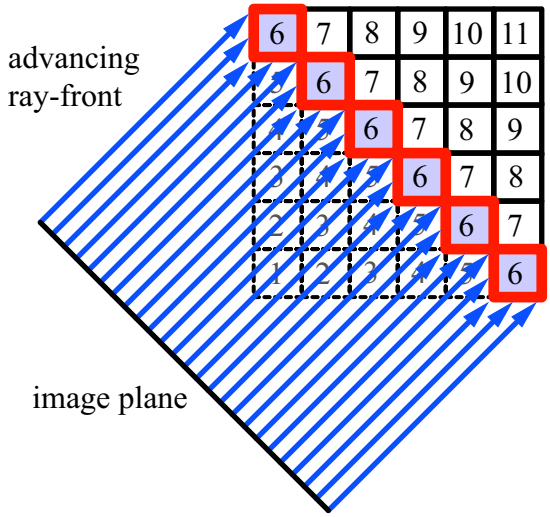


Figure 3.10: Blockwise raycasting scheme. A ray-front is advancing through the volume processing one list of blocks in each pass. The numbers inside the blocks identify their block list.

lookup table has a size of 27 cases · 26 offsets · 4 bytes per offset = 2808 bytes. This can be reduced by a factor of two due to symmetry reasons. Therefore we have a very small lookup table of 1404 bytes. This is an improvement of a factor of 2427 compared to the straight-forward solution.

Another possible option to simplify the addressing is to inflate each block by an additional border of samples from the neighboring blocks [11]. However, such a solution increases the overall memory usage considerably. For example, for a block size of $32 \times 32 \times 32$ the total memory is increased by approximately 20%. This is an inefficient usage of memory resources and the storage redundancy reduces the effective memory bandwidth. Our approach practically requires no additional memory, as all blocks share one global address lookup table.

3.3.3 Traversal

As stated in the previous sections, it is important to ensure that data once replaced in the cache will not be requested again, thus avoiding thrashing. Law and Yagel have presented a thrashless distribution scheme for parallel

raycasting [22, 23, 21]. Their scheme relies on an object space subdivision of the volume. While their method was essentially developed in the context of parallelization, avoiding multiple distribution of data, it is also useful for a single-processor approach.

The volume is subdivided into blocks, as described in Section 3.3.1. These blocks are then sorted in front-to-back order depending on the current viewing direction. The ordered blocks are placed in a set of block lists in such a way that no ray that intersects a block contained in a block list can intersect another block from the same block list. Each block holds a list of rays whose current resample position lies within the block. The rays are initially assigned to the block which they first intersect. The blocks are then traversed in front-to-back order by sequentially processing the block lists. The blocks within one block list can be processed in any order, e.g., in parallel. For each block, all rays contained in its list are processed. As soon as a ray leaves a block, it is removed from its ray list and added to the new block's ray list. When the ray list of a block is empty, processing is continued with the next block. Figure 3.10 illustrates this approach.

The generation of the block lists does not have to be performed for each frame. For parallel projection there are eight distinct cases where the order of blocks which have to be processed remains the same. Thus, the lists can be pre-computed for these eight cases. Figure 3.11 shows this for 2D where there are four cases.

3.4 Parallelization Strategies for Commodity Hardware

Raycasting has always posed a challenge on hardware resources, thus, numerous approaches for parallelization have been presented. As our target platform is consumer hardware, we have focused on two parallelization schemes available in current stand-alone PCs: Symmetric Multiprocessing (SMP) and Simultaneous Multithreading (SMT).

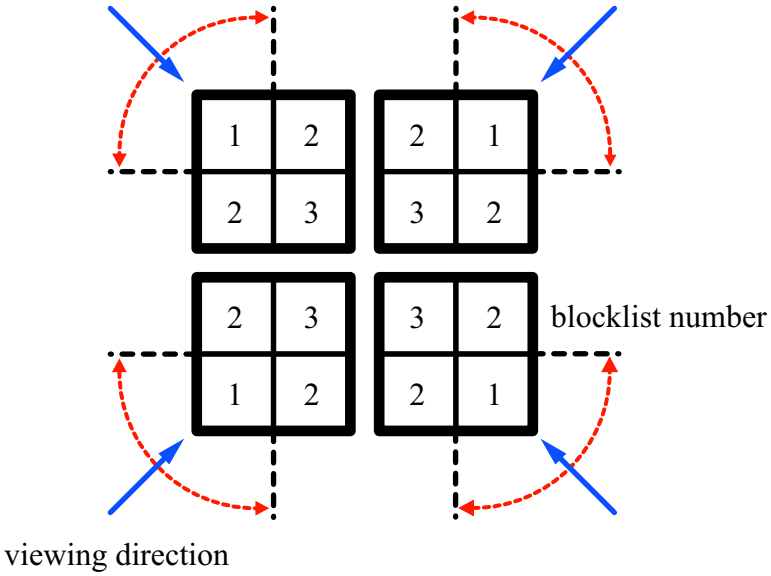


Figure 3.11: Front-to-back orders of blocks. In an interval of 90 degrees of the viewing direction the front-to-back order remains constant. The numbers inside the blocks identify their block list, and thus the designated processing order.

3.4.1 Symmetric Multiprocessing

Architectures using multiple processors within the same computer are referred to as Symmetric Multiprocessing systems. Multiprocessor architectures improve overall performance by allowing threads to execute in parallel. As Law and Yagel's traversal scheme [22] was originally developed for parallelization, it is straight-forward to apply it to SMP architectures. Since the blocks contained in each block list are independent, they can be distributed among the available physical CPUs.

A possible problem occurs when rays from two simultaneously processed blocks have the same subsequent block, as shown in Figure 3.12. As blocks processed by different CPUs can contain rays which have the same subsequent block, race conditions occur when both CPUs simultanously try to assign rays to the ray list of one block. One way of handling these cases would be to use synchronization primitives such as mutexes or critical sections to ensure that only one thread can assign rays at a time. However, the required overhead

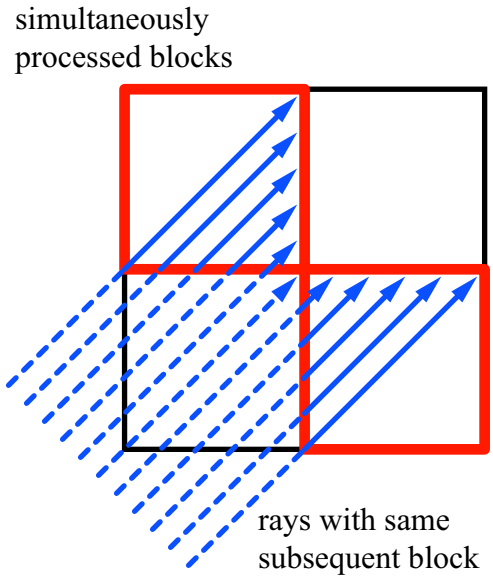


Figure 3.12: Concurrency problem in parallel block processing. The two highlighted blocks are processed by different CPUs. When both CPUs try to add their rays to the next block's ray list, race conditions can occur.

can decrease the performance drastically. Therefore, to avoid race conditions when two threads try to add rays to the ray list of a block, each block has a ray list for every physical CPU. When a block is being processed, the rays of all these lists are cast. When a ray leaves the block, it is added to the new block's ray list corresponding to the CPU currently processing the ray.

The basic algorithm processes the pre-generated block lists in passes. The *ProcessVolume* procedure (see Algorithm 7) is executed by the main thread and distributes the blocks of each pass among the available processors. It starts the execution of *ProcessBlocks* (see Algorithm 8) in a thread for each of the processors. *ProcessBlocks* traverses the list of blocks assigned to a processor and processes the rays of each block. *ProcessRay* performs resampling, gradient estimation, shading, and compositing for a ray, until it leaves the current block or is terminated for another reason (e.g., early ray termination). It returns true if the ray enters another block and false if no further processing of the ray is necessary. *ComputeBlock* returns the new block of a ray when it has left the current block. In the listed procedures,

$count_{physical}$ is the number of physical CPUs in the system.

Algorithm 7 ProcessVolume(*blocklists*)

```
for all lists l in blocklists do
  Partition l = l_0 ∪ ... ∪ l_{count_physical − 1}
  for i = 0 to count_physical − 1 do
    Begin execution of ProcessBlocks(l_i,i) in thread T_{i+1} on physical CPU i
  end for
  Wait for threads T_1,...,T_{count_physical} to finish
end for
```

Algorithm 8 ProcessBlocks($blocklist$,$id_{physical}$)

```
for all blocks b in blocklist do
  for i = 0 to count_physical − 1 do
    for all rays r in b.raylist[i] do
      if ProcessRay(r) then
        {the ray has entered another block}
        Remove(b.raylist[i],r)
        newBlock = ComputeBlock(r)
        Insert(newBlock.raylist[id_physical],r)
      else
        {the ray has been terminated or has left the volume}
        Remove(b.raylist[i],r)
      end if
    end for
  end for
end for
```

3.4.2 Simultaneous Multithreading

Simultaneous Multithreading is a well-known concept in workstation and mainframe hardware. It is based on the observation that the execution resources of a processor are rarely fully utilized. Due to memory latencies and data dependencies between instructions, execution units have to wait for instructions to finish. While modern processors have out-of-order execution units which reorder instructions to minimize these delays, they rarely find enough independent instructions to exploit the processor's full potential. SMT uses the concept of multiple logical processors which share the resources (including caches) of just one physical processor. Executing two

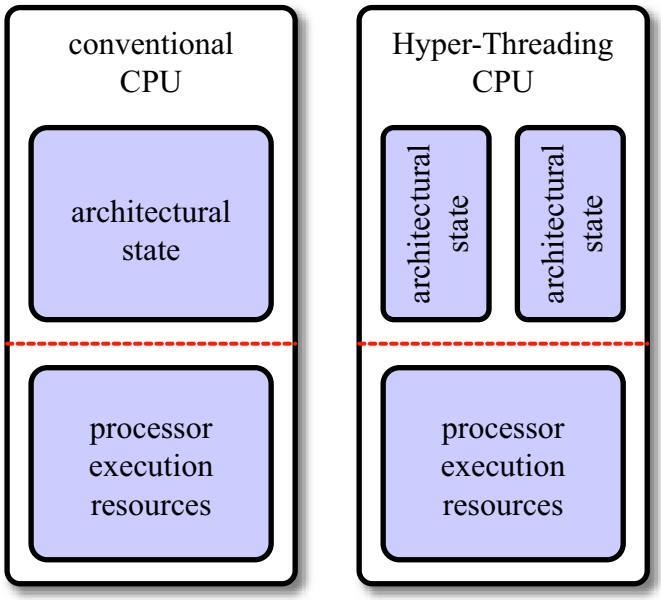


Figure 3.13: Comparison of conventional CPU and Hyper-Threading CPU. (a) conventional CPU with single architectural state. (b) Hyper-Threading CPU with duplicated architectural state, one for each logical processor.

threads simultaneously on one processor has the advantage of more independent instructions being available, and thus leads to more efficient CPU utilization. This can be implemnted by duplicating state registers, which only leads to little increases in manufacturing costs. Intel's SMT implementation is called Hyper-Threading [28] and was first available on the Pentium 4 CPU. Currently, two logical CPUs per physical CPU are supported (see Figure 3.13). Hyper-Threading adds less than 5% to the relative chip size and maximum power requirements, but can provide performance benefits much greater than that.

Exploiting SMT, however, is not as straight-forward as it may seem at first glance. Since the logical processors share caches, it is essential that the threads operate on neighboring data items. Therefore, in general, treating the logical CPUs in the same way as physical CPUs leads to little or no performance increase. Instead, it might even lead to a decrease in performance, due to cache thrashing. Thus, our processing scheme has to be extended in

order to allow multiple threads to operate within the same block.

The blocks are distributed among physical processors as described in the previous section. Additionally, within a block, multiple threads each executing on a logical CPU simultaneously process the rays of a block. Using several threads to process the ray list of a block would lead to race conditions and would therefore require expensive synchronization. Thus, instead of each block having just one ray list for every physical CPU, we now have $count_{logical}$ lists per physical CPU, where $count_{logical}$ is the number of threads that will simultaneously process the block, i.e., the number of logical CPUs per physical CPU. Thus, each block has $count_{physical} \cdot count_{logical}$ ray lists $raylist[id_{physical}][id_{logical}]$ where $id_{physical}$ identifies the physical CPU and $id_{logical}$ identifies the logical CPU relative to the physical CPU. A ray can move between physical CPUs depending on how the block lists are partitioned within in each pass, but they always remain in a ray list with the same $id_{logical}$. This means that for equal workloads between threads, the rays have to be initially distributed among these lists, e.g., by alternating the $id_{logical}$ of the list a ray is inserted into during ray setup.

The basic algorithm described in the previous section is extended in the following way: The *ProcessBlocks* procedure (see Algorithm 9) now starts the execution of *ProcessRays* for each logical CPU of the physical CPU it is executed on. *ProcessRays* (see Algorithm 10) processes the rays of a block for one logical CPU. All other routines remain unchanged.

Algorithm 9 ProcessBlocks(*blocklist*,$id_{physical}$)

for all blocks b in *blocklist* **do**
  for $i = 0$ to $count_{logical} - 1$ **do**
    Begin execution of ProcessRays(b,$id_{physical}$,i) in thread $T_{id_{physical} \cdot count_{logical}+i+1}$ on logical CPU i of physical CPU $id_{physical}$
  end for
  Wait for threads $T_{id_{physical} \cdot count_{logical}+1}$,...,$T_{id_{physical} \cdot count_{logical}+count_{logical}}$ to finish
end for

Figure 3.14 depicts the operation of the algorithm for a system with two physical CPUs, each allowing simultaneous execution of two threads, i.e. $count_{physical} = 2$ and $count_{logical} = 2$. In the beginning seven treads, $T_0, ..., T_6$, are started. T_0 performs all the preprocessing. In particular, it

Algorithm 10 ProcessRays($block$,$id_{physical}$,$id_{logical}$)

```
for i = 0 to count_physical − 1 do
  for all rays r in block.raylist[i][id_logical] do
    if ProcessRay(r) then
      {the ray has entered another block}
      Remove(currentBlock.raylist[i][id_physical],r)
      block_new = ComputeBlock(r)
      Insert(block_new.raylist[id_physical][id_logical],r)
    else
      {the ray has been terminated or has left the volume}
      Remove(block.raylist[i][id_logical],r)
    end if
  end for
end for
```

has to assign the rays to those blocks through which the rays enter the volume first. Then it has to choose the lists of blocks which can be processed simultaneously, with respect to the eight to distinguish viewing directions. Each list is partitioned by T_0 and sent to T_1 and T_2. After a list is sent, T_0 sleeps until its slaves are finished. Then it continues with the next pass. T_1 sends one block after the other to T_3 and T_4. T_2 sends one block after the other to T_5 and T_6. After a block is sent, they sleep until their slaves are finished. Then they send the next block to process, and so on. T_3, T_4, T_5, and T_6 perform the actual raycasting. Thereby T_3 and T_4 simultaneously process one block, and T_5 and T_6 simultaneously process one block.

3.5 Memory Efficient Acceleration Data Structures

Applying efficient memory access and parallelization techniques still is not sufficient to efficiently handle the huge processing loads caused by large datasets. In this section, we introduce optimization techniques to reduce this workload. We present three different techniques which each can achieve a significant reduction of rendering times. Our focus lies in minimizing the additional memory requirements of newly introduced data structures.

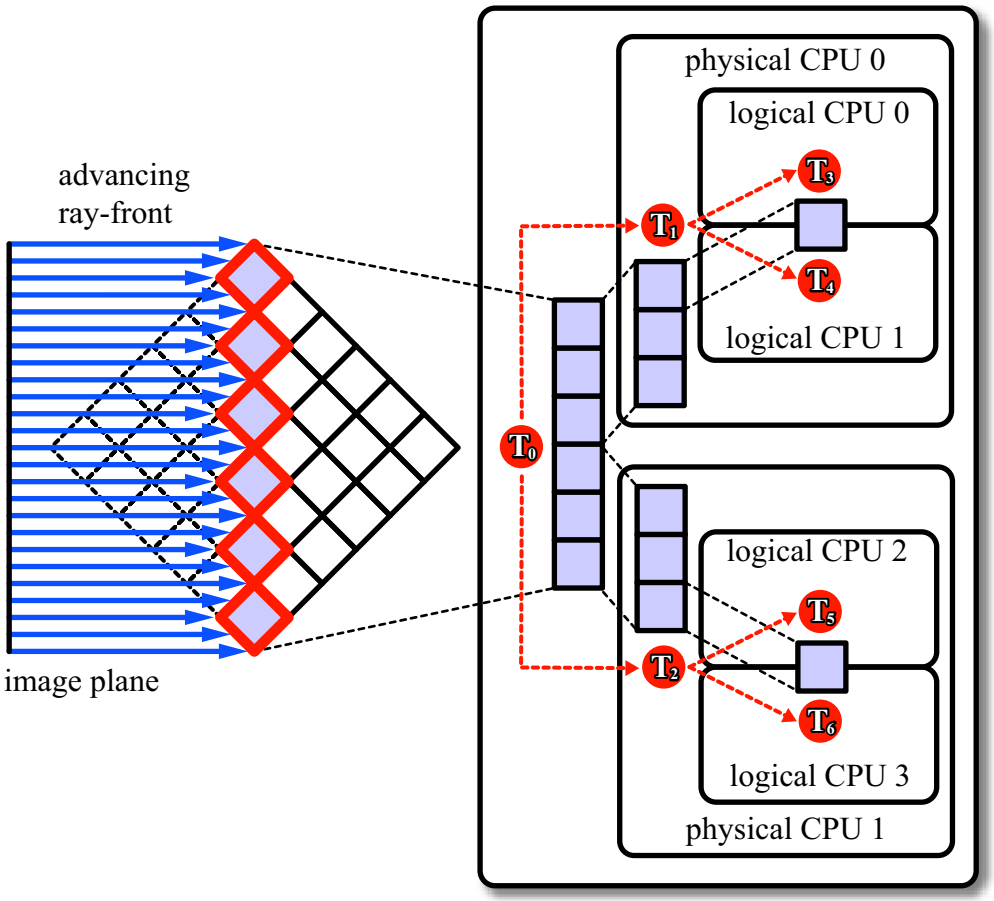


Figure 3.14: Simultaneous Multithreading enabled raycasting. The work is distributed among the threads T_i executing on different logical CPUs.

3.5.1 Gradient Cache

It has been argued that the quality of the final image is heavily influenced by the gradients used in shading [34]. High-quality gradient estimation methods have been developed, which generally consider a large neighborhood of samples. This implies higher computational costs. Due to this, many approaches use expensive gradient estimation techniques to pre-compute gradients at the grid points and store them together with the original samples. The additional memory requirements, however, limit the practical application of this approach. For example, using 2 bytes for each component of the gradient increases the size of the dataset by a factor of four (assuming 2 bytes are used for the original samples). In addition to the increased memory requirements of pre-computed gradients, this approach also reduces the effective memory bandwidth. We therefore choose to perform gradient estimation on-the-fly. Consequently, when using an expensive gradient estimation method caching of intermediate results is inevitable if high performance has to be achieved. An obvious optimization is to perform gradient estimation only once for each cell. When a ray enters a new cell, the gradients are computed at all eight corners of the cell. These gradients are then re-used during resampling within the cell. The benefit of this method is dependent on the number of resample locations per cell, i.e., the object sample distance.

However, the computed gradients are not reused for other cells. This means that each gradient typically has to be computed eight times, as illustrated in Figure 3.15. For expensive gradient estimation methods, this can considerably reduce the overall performance. It is therefore important to store the results in a gradient cache. However, allocating such a cache for the whole volume still has the mentioned memory problem.

Our blockwise volume traversal scheme allows us to use a different approach. We perform gradient caching on a block basis. Our cache is capable of storing one gradient for every grid point of a block. Thus, the required cache size is $(B_x+1)\times(B_y+1)\times(B_z+1)$ where B_x, B_y, B_z are the block dimensions. The block dimensions are increased by one to enable interpolation across block boundaries. Each entry of the cache stores the three compo-

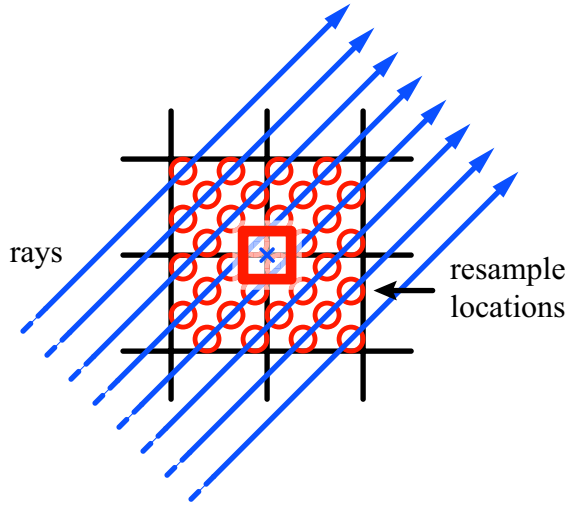


Figure 3.15: Redundant gradient computation at grid positions. Without caching, the gradient at the highlighted grid position has to be recomputed multiple times.

nents of a gradient, using a single precision floating-point number for each component. Additionally, a bit set has to be stored that encodes the validity of an entry in the cache for each grid point of a block.

When a ray enters a new cell, for each of the eight corners of the cell the bit set is queried. If the result of a query is zero, the gradient is computed and written into the cache. The corresponding value of the bit set is set to one. If the result of the query is one, the gradient is already present in the cache and is retrieved.

The disadvantage of this approach is that gradients at block borders have to be computed multiple times. However, this caching scheme still greatly reduces the performance impact of gradient computation and requires only a modest amount of memory. Furthermore, the required memory is independent of the volume size, which makes this approach applicable to large datasets.

3.5.2 Entry Point Buffer

One of the major performance gains in volume rendering can be achieved by quickly skipping data which is classified as transparent. In particular, it is important to begin sampling at positions close to the data of interest, i.e., the non-transparent data. This is particularly true for medical datasets, as the data of interest is usually surrounded by large amounts of empty space (air). The idea is to find, for every ray, a position close to its intersection point with the visible volume, thus, we refer to this search as entry point determination. The advantage of entry point determination is that it does not require additional overhead during the actual raycasting process, but still allows to skip a high percentage of empty space. The entry points are determined in the ray setup phase and the rays are initialized to start processing at the calculated entry position. The basic goal of entry point determination is to establish a buffer, the entry point buffer, which stores the position of the first intersection with the visible volume for each ray.

As blocks are the basic processing units of our algorithm, the first step is to find all blocks which do not contribute to the visible volume using the current classification, i.e., all blocks that only contain data values which are classified as transparent. It is important that the classification of a whole block can be efficiently calculated to allow interactive transfer function modification. We store the minimum and maximum value of the samples contained in a block and use a summed area table of the opacity transfer function to determine the visibility of the block [19].

A summed area table is a data structure for integrating a discrete function. We use a one-dimensional summed area table to evaluate the integral of the opacity transfer function α over the voxels represented by a block. The summed area table is computed in the following way: $S(0)$ is equal to $\alpha(0)$ and $S(i) = S(i-1) + \alpha(i)$ for all $i > 0$.

The integral of the discrete function α over the interval $[i_{min}, i_{max}]$ can then be evaluated in constant time by performing two table lookups:

$$\sum_{i_{\min}}^{i_{\max}} \alpha(i) = S(i_{\max}) - S(i_{\min} - 1) \tag{3.14}$$

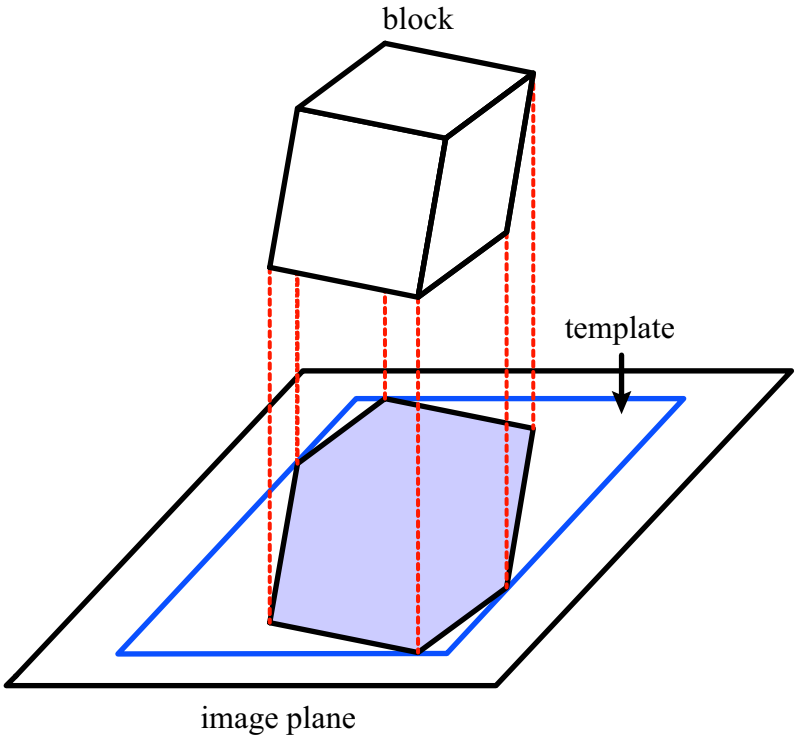
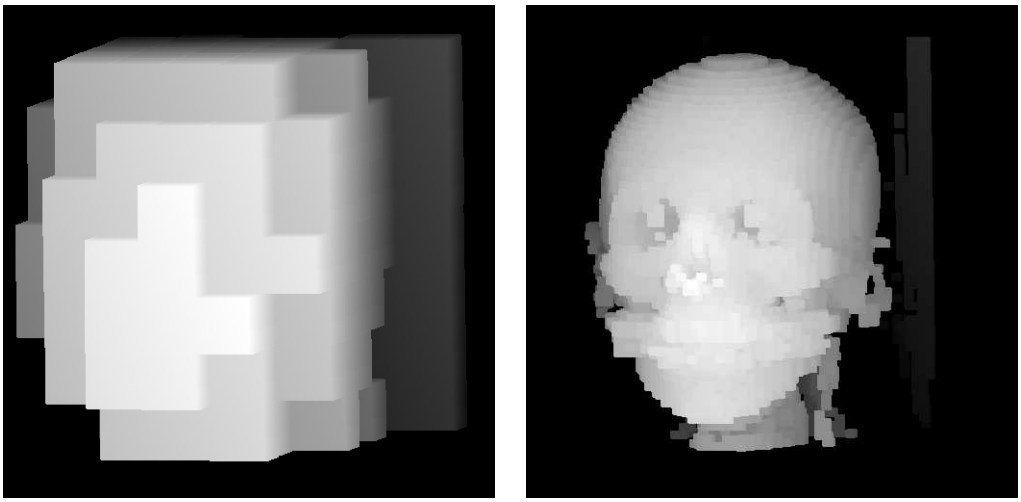


Figure 3.16: Block template generation. The block is projected onto the image plane, its depth values are rasterized and stored in a template image.

We then perform a projection of each non-transparent block onto the image plane with hidden surface removal to find the first intersection point of each ray with the visible volume [49]. The goal is to establish an entry point buffer of the same size as the image plane, which contains the depth value for each ray's intersection point with the visible volume. For parallel projection, this step can be simplified. As all blocks have exactly the same shape, it is sufficient to generate one template by rasterizing one block under the current viewing transformation (see Figure 3.16). Projection is then performed by translating the template by a vector $t = (t_x, t_y, t_z)^T$ which corresponds to the block's position in three-dimensional space in viewing coordinates. Thus, t_x and t_y specify the position of the block on the image plane (and therefore the location where the template has to be written into the entry point buffer) and t_z is added to the depth values of the template. The Z-buffer algorithm is used to ensure correct visibility. During ray setup, the depth values stored



(a) (b)

Figure 3.17: Block and octree projection. (a) projection of non-transparent blocks. (b) projection of non-transparent octree nodes.

in the entry point buffer are used to initialize the ray positions.

The disadvantage of this approach is that it requires an addition and a depth test at every pixel of the template for each block. This can be greatly reduced by choosing an alternative method. The blocks are projected in back-to-front order. The back-to-front order can be easily established by traversing the generated block lists (see Section 3.3.3) in reverse order. For each block the Z-value of the generic template is written into the entry point buffer together with a unique index of the block. After the projection has been performed, the entry point buffer contains the indices and relative depth values of the entry points for each ray. During ray setup, the block index is used to determine the translation vector t for the block and t_z is added to the relative depth value stored in the buffer to find the entry point of the ray. The addition only has to be performed for every ray that actually intersects the visible volume.

We further extend this approach to determine the entry points in a higher resolution than block granularity. We replace the minimum and maximum values stored for every block by a min-max octree. Its root node stores the minimum and maximum values of all samples contained in a block. Each ad-

ditional level contains the minimum and maximum value for smaller regions, resulting in a more detailed description of parameter variations inside the block. The resulting improvement in entry point determination is depicted in Figure 3.17.

The advantage of a min-max octree is, that it stores only information about the data values, which are independent of the transfer functions and viewing parameters. Thus, the octree can be generated in a preprocessing step. When the classification changes, the summed area table is used to efficiently determine the visibility of each octree node.

For determining the visibility of an octree node, the integral over the interval defined by the minimum and maximum values of the node is evaluated using the summed area table. If the integral is zero, then all voxels represented by the node are transparent. Otherwise, a more detailed level of the octree is used to refine the estimated classification.

When the classification changes, the summed-area table is recursively evaluated for all blocks. The classification information itself can be stored efficiently using a technique called hierarchy compression. Our octree has a maximum depth of three for each block. All nodes except the most detailed octree level (level 2) have three states (see Figure 3.18):

- opaque, i.e., none of the children of the node is transparent,
- transparent, i.e., all of the node's children are transparent,
- inhomogeneous, i.e., the node has transparent and non-transparent children.

For efficient storage and visibility determination, we use the following scheme: The information whether a node of level 2 is transparent or opaque is stored in one bit. Since each node of level 1 contains eight nodes of level 2, no additional information has to be stored for level 1. The state of a level 1 node can be determined by testing the byte which contains all the bits of its children. For level 0 such a hierarchy compression scheme would require to test 8 bytes to determine the state of a level 0 node and 64 bytes for the whole block. Thus, for level 0 we explicitly store the state information.

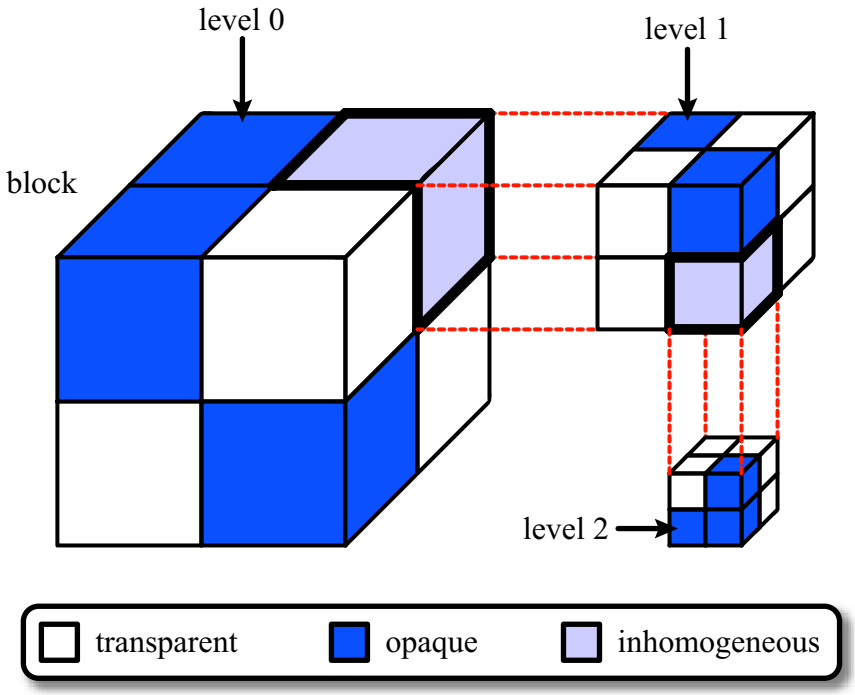


Figure 3.18: Octree classification of a block. The block is recursively subdivided into nodes which are either classified as fully transparent, fully opaque, or inhomogeneous. Leaf nodes can only be fully transparent or fully opaque.

Since there are three states, we need two bits for each level 0 node. Thus, two additional bytes are stored which contain the two bit state information for all level 0 nodes.

The projection algorithm is modified as follows. Instead of one block template there is now a template for each octree level. The projection of one block is performed by recursively traversing the hierarchical classification information in back-to-front order and projecting the appropriate templates for each level, if the corresponding octree node is non-transparent. In addition to the block index, the entry point buffer also stores an index for the corresponding octree node (node index). During ray setup, the depth value in the entry point buffer is translated by the t_z component of the translation vector plus the sum of the relative offsets of the node in the octree.

The node index encodes the position of a node's origin within the octree. It can be calculated in the following way:

$$index(node) = \sum_{i=0}^{N-1} octant_i(node) \cdot 8^{N-i-1} \tag{3.15}$$

where N is the depth of the octree, $octant_i$ is the octant of level i where the node is located. For an octree of depth N there are 8^N different indices. The relative translational offsets for the octree nodes can be pre-computed and stored in a lookup table of 8^N entries indexed by the node index.

3.5.3 Cell Invisibility Cache

As the depth of the octree does not reach down to cell level, the initial position of a ray might not be its exact intersection point with the visible volume. Thus, some transparent regions are still processed. We therefore introduce a cell invisibility cache to skip the remaining transparent region at cell level. We can skip the resampling and compositing in a cell if all eight samples of the cell are classified as transparent. To determine the transparency, a transfer function lookup has to be performed for each of these samples. For large zoom factors, several rays can hit the same cell and for each of these rays the same lookups would have to be performed.

A cell invisibility cache is attached at the beginning of the traditional volume raycasting pipeline. This cache is initialized in such a way that it reports every cell as visible. In other words every cell has to be classified. Now, if a ray is sent down the pipeline, every time a cell is classified invisible this information is stored in the cache. If a cell is found to be invisible, this information is stored by setting the corresponding bit in the cell invisibility cache. As the cache stores the combined information for eight samples of a cell in just one bit, this is more efficient than performing a transfer function lookup for each sample. The information stored in the cell invisibility cache remains valid as long as no transfer function modifications are performed. During the examination of the data, e.g., by changing the viewing direction, the cache fills up and the performance increases progressively.

The advantage of this technique is that no extensive computations are required when the transfer function changes. The reset of the buffer can be performed with virtually no delay, allowing fully interactive classification. As transfer function specification is a non-trivial task, minimizing delays initiated by transfer function modifications greatly increases usability.

3.5.4 Load Balancing

For every pass of the rendering algorithm, a list of blocks is distributed among the available processors. A straight-forward approach for this distribution is to simply divide the list into equally sized portions. This works well if the algorithm does not use any optimizations for skipping empty space. However, employing the optimizations presented in the previous sections can lead to a very unequal distribution, and thus, reduce the performance gains due to using multiple CPUs.

We use a simple estimate to determine the computational effort required for processing a whole block. For a list of N blocks $L = \{B_0, ..., B_{N-1}\}$ we compute for each block $B \in L$:

$$load(B) = \frac{rays(B)}{1 + C_t \cdot transparency(B)} \tag{3.16}$$

$rays(B_i)$ is the number of rays entering the block B_i and $transparency(B_i)$

is a measure for the number of transparent samples in the block. An estimate measure for the transparency can be retrieved from the classified min-max octree. Depending on how many octree levels are taken into account, the estimate will become more accurate. For example, a simple way to obtain such a measure is to sum up the transparency information of the top-level octree nodes by counting fully transparent nodes as $\frac{1}{8}$, fully opaque nodes as 0, and inhomogeneous nodes as $\frac{1}{16}$. The factor C_t determines the fraction of processing cost between a fully opaque and a fully transparent block. If C_t is 0, the cost of processing an opaque and a transparent block is the same, if C_t is 1, the cost of processing a fully transparent block is half the cost of processing a fully opaque block.

We then want to find a partition of $L = L_0 \cup ... \cup L_{M-1}$ for M CPUs, so that the workload for each CPU is approximately equal. We sort all blocks $B \in L$ in descending order of $load(B)$. A greedy algorithm then traverses the sorted list and always assigns a block to the CPU with the smallest current load, until all blocks have been assigned to a CPU. This approximative algorithm results in better balanced loads while introducing little overhead.

3.6 Maintaining Interactivity

In a volume visualization system interaction is very important. The user has to be able to freely move the viewpoint and zoom in and out. However, since the performance of the algorithm cannot be predicted for all types of datasets and transfer functions, it is necessary to use an adaptive scheme for modifying rendering parameters to achieve reactivity during interaction.

We identify the following rendering parameters that represent a trade-off between quality and speed:

Image sample distance The distance in x and y direction on the image plane between neighboring rays.

Object sample distance The distance between subsequent samples along a ray.

Reconstruction quality level The method used for resampling and gradient estimation.

Since the rendering time is approximately proportional to the number of rays cast, the image sample distance has quadratic influence on rendering time. The influence of the object sample distance is very much dependent on the transfer function and the dataset itself, thus, in contrast to the image sample distance there is no general rule. For resampling and gradient quality there is also no general rule. Though the different methods can be ordered according to their complexity, their influence on the actual render time cannot be predicted.

Our adaption scheme uses a user-supplied desired render time $t_{desired}$, the minimum and maximum values for the image sample distance isd_{min} and isd_{max}, the minimum and maximum values for the object sample distance osd_{min} and osd_{max}, and the minimum and maximum values for the reconstruction quality level rql_{min} and rgl_{max}. The reconstruction quality level defines the method used for resampling and gradient estimation. The methods are ordered according to their quality and complexity in the following way: $quality(rql_i) > quality(rql_{i+1})$ and $complexity(rql_i) > complexity(rql_{i+1})$.

The basic adaption procedure given in Algorithm 11 computes the values for $settings_{new}$, for a desired render time $t_{desired}$ based on the values of $settings_{old}$ and the render time t_{old} achieved with these settings. Δ_{osd} and Δ_{rql} define the increments in which to increase or decrease the objects sample distance and reconstruction quality level.

First, the image sample distance is adjusted according to the assumption that it has quadratic influence on the render time. If the resulting image sample distance is lower than isd_{min}, the image sample distance is set to isd_{min} and the object sample distance is adjusted. If the object sample distance cannot be adjusted, i.e., the resulting value is lower than osd_{min}, then it is set to osd_{min} and the reconstruction quality level is adjusted, if possible. If the adjusted image sample distance is greater than isd_{max}, it is set to isd_{max} and the object sample distance is adjusted. If the object sample distance cannot be adjusted, i.e., the resulting value is greater than osd_{max},

then it is set to osd_{max} and the reconstruction quality level is adjusted, if possible.

Algorithm 11 ComputeAdaption($t_{desired}$,$settings_{new}$,t_{old},$settings_{old}$)

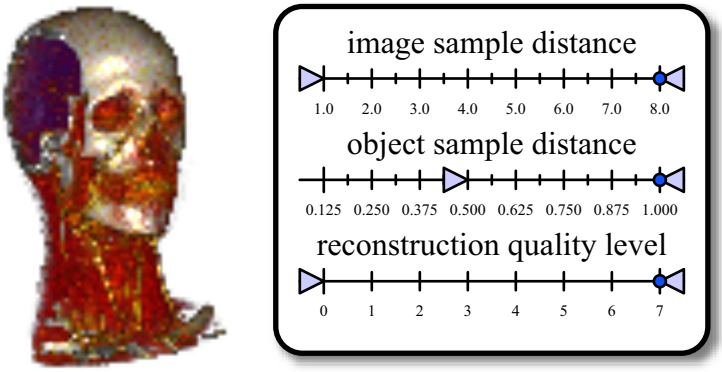
$settings_{new}.isd = settings_{old}.isd\sqrt{\frac{t_{old}}{t_{desired}}}$
if $settings_{new}.isd < isd_{min}$ **then**
  $settings_{new}.isd = isd_{min}$
  $settings_{new}.osd = settings_{old}.osd - \Delta_{osd}$
  if $settings_{new}.osd < osd_{min}$ **then**
    $settings_{new}.osd = osd_{min}$
    $settings_{new}.rql = \max(settings_{old}.rql - \Delta_{rql}, rql_{min})$
  else
    $settings_{new}.rql = settings_{old}.rql$
  end if
else if $settings_{new}.isd > isd_{max}$ **then**
  $settings_{new}.isd = isd_{max}$
  $settings_{new}.osd = settings_{old}.osd + \Delta_{osd}$
  if $settings_{new}.osd > osd_{max}$ **then**
    $settings_{new}.osd = osd_{max}$
    $settings_{new}.rql = \min(settings_{old}.rql + \Delta_{rql}, 0, rql_{max})$
  else
    $settings_{new}.rql = settings_{old}.rql$
  end if
else
  $settings_{new}.osd = settings_{old}.osd$
  $settings_{new}.rql = settings_{old}.rql$
end if

An application using this adaption scheme can supply different degrees of interactivity corresponding to different values for $t_{desired}$. For example, it is common to provide an interactive mode while the user modifies the viewing parameters (camera position, lighting setting, etc.) and a high-quality mode. The interactive mode would have a low value for $t_{desired}$, e.g. 0.1 seconds (10 frames/second). The high quality mode would use a very high value or even ∞ for $t_{desired}$ to ensure the best possible quality. However, since the adaption for the object sample distance and the reconstruction quality level is only incremental, it is possible that a transition from interactive mode to high quality mode does not lead to the best quality.

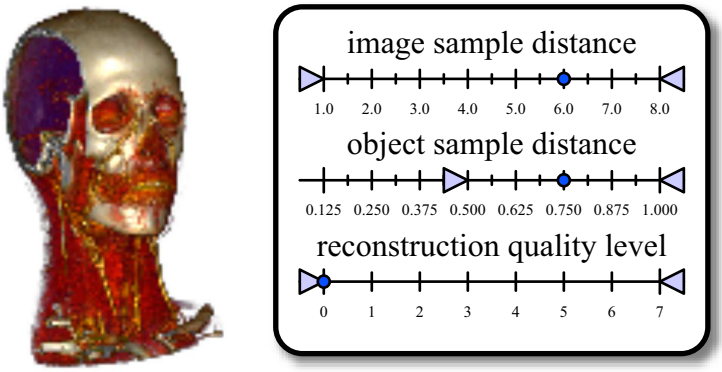
A solution to this problem is to base the adaption on the last known values for the current mode. In a table, for every possible value of $t_{desired}$ the values for image sample distance, object sample distance, reconstruction quality

level, and the actual render time achieved with these settings are stored. Before *ComputeAdaption* is called, $t_{desired}$ is used to retrieve $settings_{old}$ and t_{old} from this table. After rendering has been performed, the measured render time and the corresponding settings are written into the table again. This ensures that the adaption is always based on the last known values for the current render mode. An application can use this method to define any number of different render modes. It is even possible to define new modes at run-time by simply specifying a new value for $t_{desired}$. Using a value that is not found in the table causes a new entry filled with default values to be generated.

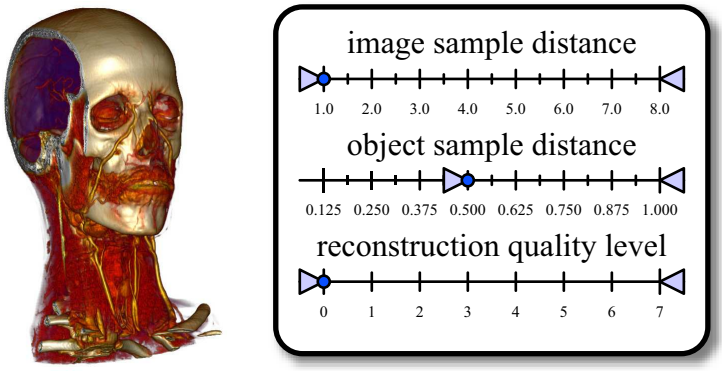
Figure 3.19 shows an example of the adaption scheme used in our prototype application. The interactive mode is activated when the user presses a mouse button and moves the mouse to rotate the camera. When the mouse button is held down longer than one second without moving the mouse, a preview mode rendering is automatically performed. A high quality mode rendering is performed as soon as the user releases the mouse button.



(a)



(b)



(c)

Figure 3.19: Example of interaction modes. (a) interactive mode - desired render time: 0.25s. (b) preview mode - desired render time: 1.0s. (c) high-quality mode: desired render time: ∞.

Chapter 4

Implementation

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra

The volume rendering algorithm was implemented in C++. Care was taken to avoid additional overhead of virtual method calls, etc. for critical parts of the algorithm. Additionally, the code was written according to optimization guidelines. Some parts were optimized using inline assembler, however, our experience has shown that heavy use of inline assembler code can result in bad performance. This is due to the fact that inline assembler affects code reordering, i.e., the compiler cannot reorder instructions to achieve optimal performance. Inlining of methods has proven to be one of the most valuable strategies for manual code optimization.

We have developed an interactive prototype application. The prototype allows the management of multiple datasets and segmented objects, the manipulation of camera settings, light settings, transfer functions, and the serialization of projects. Additionally, a wrapper library has been developed which provides access to the core functionality via a thin interface. The library was deliberately designed to provide a simple interface, and thus, only

provides limited flexibility. Its purpose is to enable easy integration of the implemented algorithms into existing applications.

4.1 Architecture

To build an extensible framework, we have developed several basic concepts which allow the easy integration of different algorithms. In the following, we give a brief overview over the principal building blocks of our architecture. Figure 4.1 depicts the interaction among these basic components.

4.1.1 Environment

The environment is a description of the scene which contains information about camera, light sources, and actors. An actor is the concrete representation of a volumetric dataset within the scene. It stores properties like transfer functions, illumination settings, transformation, etc.

4.1.2 Volumes

A volume implements the storage scheme for volumetric data, such as a linear volume layout or a bricked volume layout. The implementation uses the concept of volume iterators. An iterator provides access to the volume data hiding the internal data representation. Thus, when the internal data representation changes the remaining modules are not affected. There are general-purpose iterators for volume traversal in predefined orders, random-access iterators, and specialized iterators. Specialized iterators include iterators for casting a single ray (used for picking) and an iterator which implements Law and Yagel's volume traversal scheme [22].

4.1.3 Renderers

A renderer uses one or more iterators to implement a specific rendering method. The renderer determines the system configuration (number of CPUs, support of Hyper-Threading, etc.) and instantiates its iterators accordingly.

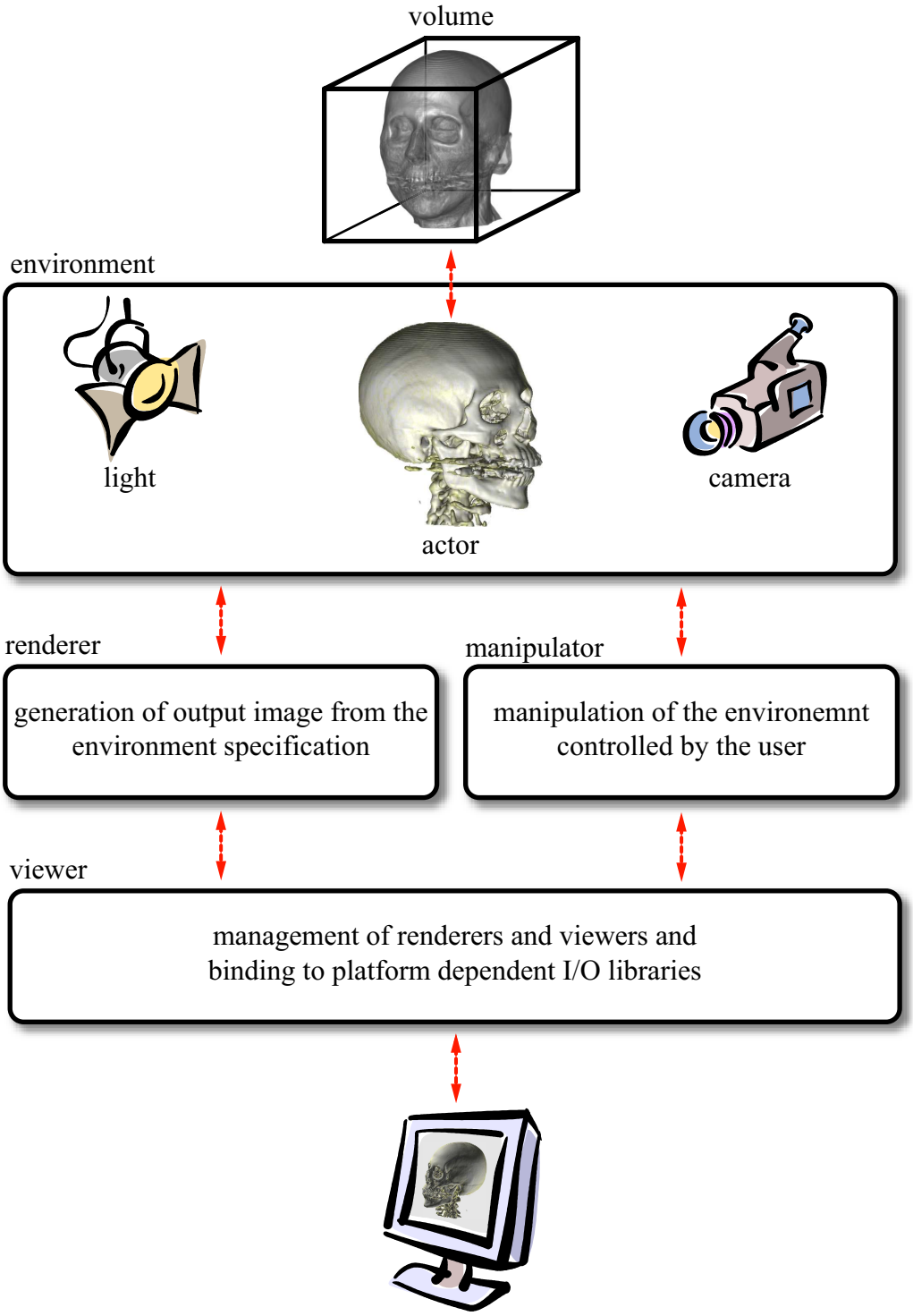


Figure 4.1: Architecture overview. The interrelationships between the basic components are depicted.

Additionaly, means for overriding these automatically detected settings are provided. Each stage of the volume rendering pipeline is freely configurable using templates. This implementation enables the compiler to produce highly optimized code, because it is able to generate different code versions at compile time. Apart from volume renders, other renders have been implemented (e.g., for two-dimensional on-screen display of dataset information).

4.1.4 Manipulators

Manipulators provide means to interactively modify the scene. This includes the modification of the camera (translation, rotation, zoom, etc.), light sources and actors. A manipulator uses an abstract event interface and is therefore independent of the used windowing toolkit or operating system.

4.1.5 Viewers

A viewer manages a set of renderers and manipulators. The viewer treats each renderer as a separate layer. The output images of all renderers are collected and composed into a final image for display. Events are passed on to the corresponding renderer or manipulator. The viewer base class defines interfaces for communication with renderers and manipulators. Derived classes establish a binding to a concrete windowing toolkit, such as Qt or GLUT.

Chapter 5

Results

Insanity: doing the same thing over and over again and expecting different results.

Albert Einstein

In this chapter we state the results that were determined by thorough analysis and benchmarking of our algorithms. We examine the effects of bricking, analyze the performance of our parallelization strategies, and demonstrate the effectiveness of our acceleration data structures. We compare the quality of reconstruction filters currently available in our implementation. Finally, we present visualization results obtained with real-world medical datasets.

5.1 Memory Management for Large Datasets

For a comparison of bricked and linear volume layouts, we use a Dual Intel Pentium Xeon 2.4 GHz equipped with 512 KB level-2 cache, 8 KB level-1 data cache, and 1 GB of Rambus memory.

In our system, we are able to support different block sizes, as long as each block dimension is a power of two. If we set the block size to the actual volume dimensions, we have a common raycaster which operates on a simple linear

volume layout. This enables us to make a meaningful comparison between a raycaster which operates on simple linear volume layout and a raycaster which operates on a bricked volume layout. To underline the effect of bricking we benchmarked different block sizes. Figure 5.1 shows the actual speedup achieved by blockwise raycasting. For testing, we specified a translucent transfer-function, such that the impact of all high level optimizations was overridden. In other words, the final image was the result of brute-force raycasting of the whole data. The size of the dataset had no influence on the actual optimal gains.

Furthermore, we did a worst-case comparison with respect to the viewing direction. In case of small blocks the worst case is similar to the best case. In contrast to that, using large bricks shows enormous performance decreases depending on the viewing direction. This is the well known fact of view-dependent performance of a raycaster operating on a linear volume layout. The constant performance behavior of small blocks is one of the main advantages of a bricked volume layout. There is nearly no view dependent performance variation anymore.

Going from left to right in the chart shown in Figure 5.1, first we have a speedup of about 2.0 with a block size of 1 KB. Increasing the block size up to 64 KB also increases the speedup. This is due to more efficient use of the cache. The chart shows an optimum at a block size of 64KB ($32 \times 32 \times 32$) with a speedup of about 2.8. This number is the optimal tradeoff between the needed cache space for ray data structures, sample data, and lookup tables. Larger block sizes lead to performance decreases, as they are too large for the cache, but still suffer from the overhead caused by bricking. This performance drop-off is reduced, once the block size approaches the volume size. With only one volume-sized block, the rendering context corresponds to a common raycaster operating on a linear volume layout.

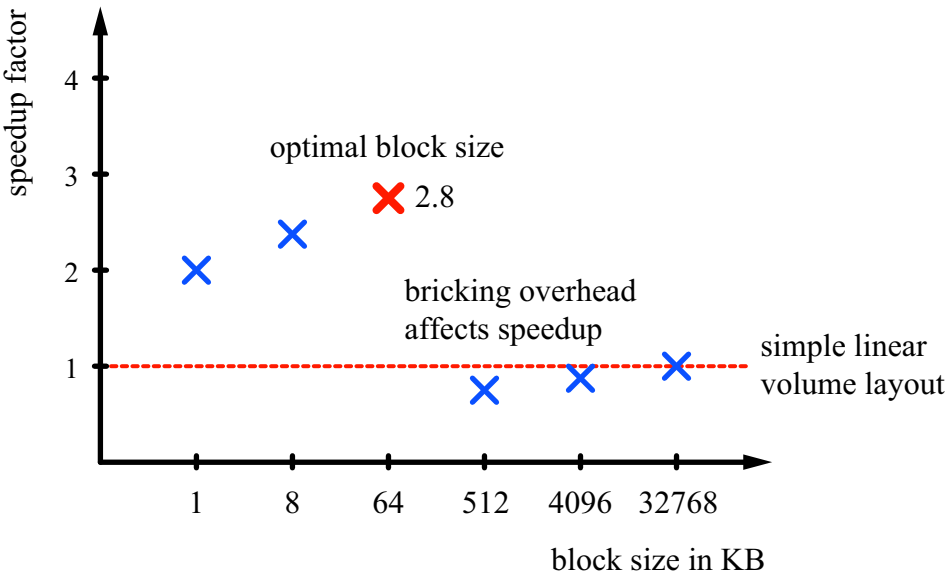


Figure 5.1: Block-based raycasting speedup compared to raycasting on a linear volume layout

5.2 Parallelization Strategies for Commodity Hardware

To evaluate the performance of our parallelization strategies, we use the same test system as in the previous section. This system has two CPUs and supports Hyper-Threading.

Our system is able to force threads on specific physical and logical CPUs. By following this mechanism we tested different configurations to obtain figures for the speedup achieved by the presented techniques. All test runs consistently showed the same speedup factors.

The achieved speedups for Symmetric Multiprocessing and Simultaneous Multithreading are shown in Figure 5.2. Testing Simultaneous Multithreading on only one CPU showed an average speedup of 30%. While changing the viewing direction, the speedup varies from 25% to 35%, due to different transfer patterns between the level 1 and the level 2 cache. Whether Hyper-Threading is enabled or disabled adding a second CPU approximately reduces the computational time by 50%, i.e., Symmetric Multiprocessing and

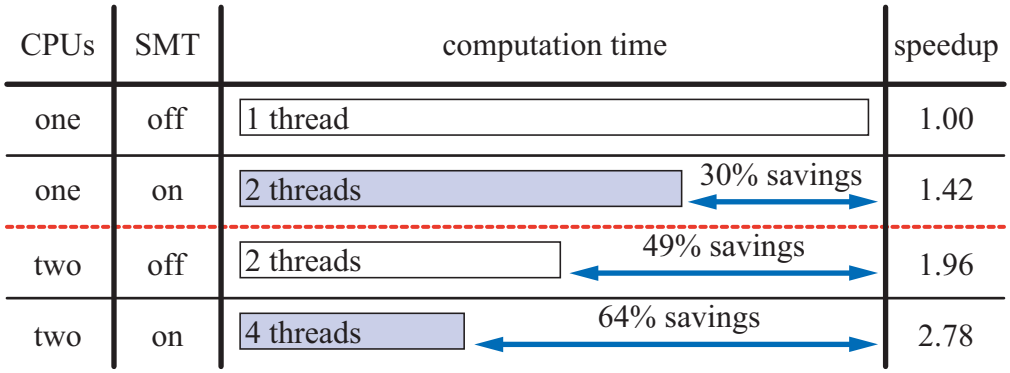


Figure 5.2: Symmetric Multiprocessing and Simultaneous Multithreading speedups

Simultaneous Multithreading are independent. This shows that our Simultaneous Multithreading scheme scales well on multi-processor machines. The Hyper-Threading benefit of approximately 30% is maintained if the second hyper-threaded CPU is enabled.

Figure 5.3 shows the Simultaneous Multithreading speedup for different block sizes. The speedup significantly decreases with larger block sizes. Once the level 2 cache size is exceeded, the two threads have to request data from main memory. Therefore, the CPU execution units are less utilized. Very small block sizes suffer from a different problem. The data fits almost into the level 1 cache. This means that one thread can utilize the execution units more efficiently, and the second thread is idle during this time. But the overall disadvantage is the inefficient usage of the level 2 cache. The optimal speedup $\frac{100}{100-30} \approx 1.42$ is achieved with a block size of 64 KB ($32 \times 32 \times 32$). This is also the optimal block size for the bricked volume layout.

5.3 Memory Efficient Acceleration Data Structures

To demonstrate the impact of our high-level optimizations we used a commodity notebook system equipped with an Intel Centrino 1.6 GHz CPU, 1 MB level 2 cache, and 1 GB RAM. This system has one CPU and does not

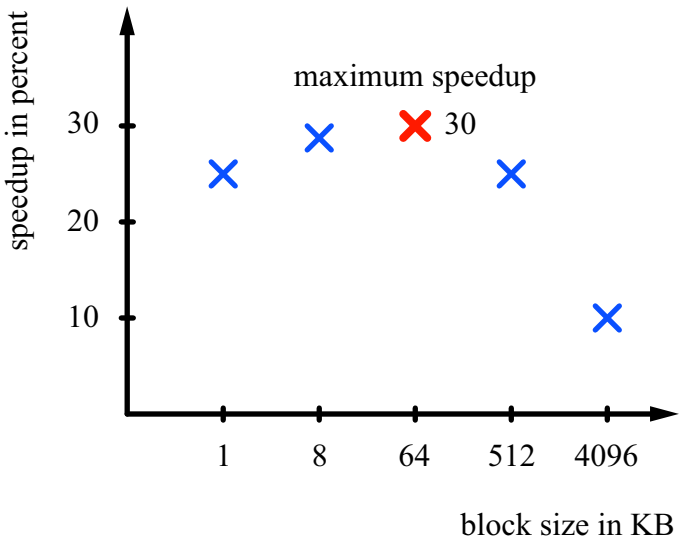


Figure 5.3: Simultaneous Multithreading speedup for different block sizes

support Hyper-Threading, so the presented results only reflect performance increases due to our high-level acceleration techniques.

The memory consumption of the gradient cache is not related to the volume dimensions, but determined by the fixed block size. We use $32 \times 32 \times 32$ blocks, the size of the gradient cache therefore is is $(33)^3 \cdot 3 \cdot 4$ byte ≈ 422 KB. Additionally we store for each cache entry a validity bit, which adds up to $33^3/8$ byte ≈ 4.39 KB.

Figure 5.4 shows the effect of per block gradient caching compared to per cell gradient caching and no gradient caching at all. Per cell gradient caching means that gradients are reused for multiple resample locations within a cell. We chose an adequate opacity transfer function to enforce translucent rendering. The charts from left to right show different timings for object sample distances from 1.0 to 0.125 for three different zoom factors 0.5, 1.0, and 2.0. In case of zoom factor 1.0 we have one ray per cell, already here per block gradient caching performs better than per cell gradient caching. This is due to the shared gradients between cells. For zooming out (0.5) both gradient caching schemes perform equally well. The rays are so far apart such that nearly no gradients can be shared. On the other hand, for zooming in (2.0), per block caching performs much better than per cell caching. This

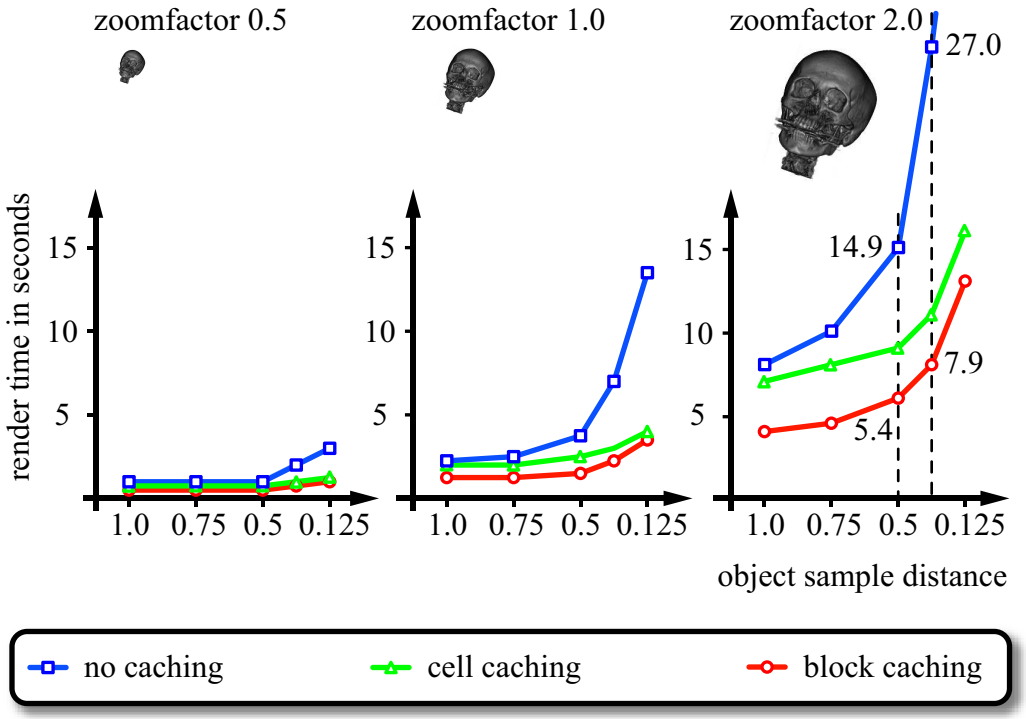


Figure 5.4: Comparison of different gradient caching strategies

is due to the increased number of rays per cell. For this zoom factor, per block gradient caching achieves a speedup of approximately 3.0 compared to no gradient caching at a typical object sample distance of 0.5.

The additional memory usage of the acceleration data structures is rather low. The cell invisibility cache has a size of 32^3 bit = 4096 byte. The min-max octree has a depth of three storing 4 byte at each node (a 2 byte minimum and maximum value) and requires at most 2340 byte. Additionally, the classification information is stored, which requires 66 byte. We use blocks of size $32 \times 32 \times 32$ storing 2 bytes for each sample, which is a total of 65536 bytes. Our data structures increase the total memory requirements by approximately 10%.

Figure 5.5 compares our acceleration techniques for three large medical datasets. In the fourth column of the table, the render times for entry point determination using block granularity is displayed. Column five shows the render times for octree based entry point determination. In the fifth col-

umn, the render times for octree based entry point determination plus cell invisibility caching are displayed. Typically, about 2 frames per second are achieved for these large data sets.

5.4 Comparison of Reconstruction Filters

In order to evaluate the quality of a reconstruction filter, Marschner and Lobb have defined a test signal [29]:

$$\rho(x,y,z) = \frac{1 - \sin(\frac{\pi z}{2}) + \alpha(1 + \rho_r(\sqrt{x^2+y^2}))}{2(1+\alpha)} \tag{5.1}$$

where

$$\rho_r(r) = \cos(2\pi f_M \cos(\frac{\pi r}{2})) \tag{5.2}$$

They sampled this signal on a $40 \times 40 \times 40$ grid in the range $-1 < x, y, z < 1$, with $f_M = 6$ and $\alpha = 0.5$. This signal has the property that a significant amount of its energy lies near the Nyquist frequency making it a very demanding filter test.

We use this function to evaluate the reconstruction quality of our system. We support gradient estimation using Neumann's method [38], central differences, and intermediate differences. Either first-order interpolation (trilinear interpolation) or zero-order interpolation (nearest neighbor) is used for function value and gradient. Additionally, we support using filtered values computed by Neumann's 4D linear regression approach instead of the actual density value. In Figure 5.6, we apply these reconstruction techniques to the Marschner-Lobb function sampled on different grid sizes and compare them with an analytic evaluation of the function and its derivative.

The effects of the different gradient estimation methods on real datasets can be seen in Figure 5.7. Neumann's method produces less fringing artifacts than the other methods. Also note that using the filtered density value causes some details to disappear. While this effect might be beneficial to the visual appearance of the image, this typically cannot be tolerated in medical

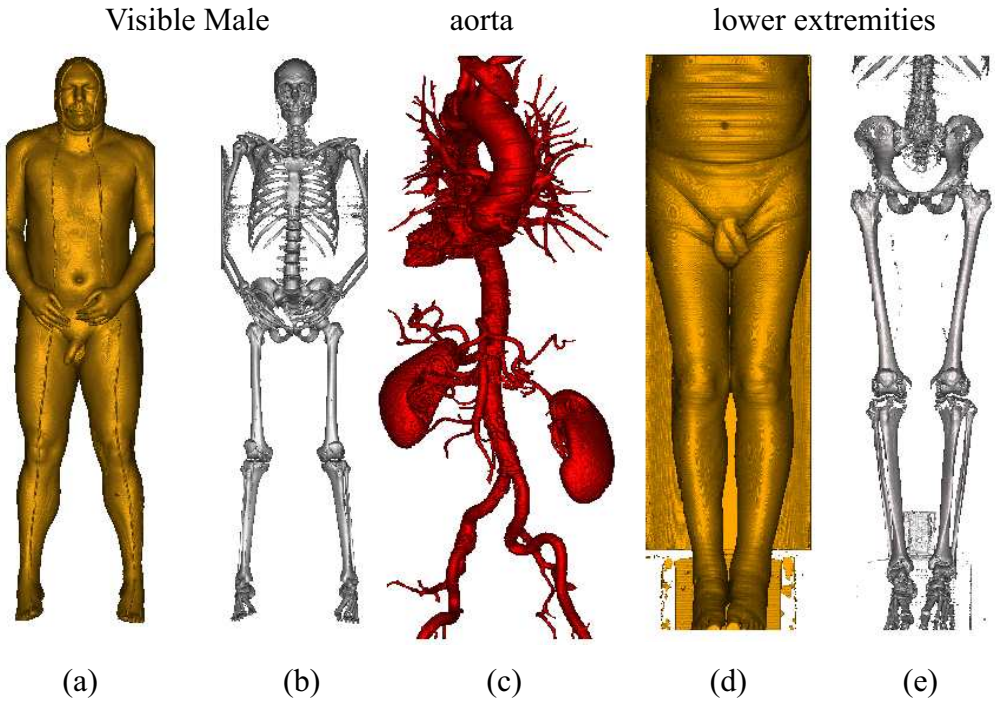


Image	Dimensions	Size	Block	Octree	Cell
(a)	$587 \times 341 \times 1878$	0.70 GB	0.61 s	0.46 s	0.40 s
(b)	$587 \times 341 \times 1878$	0.70 GB	0.68 s	0.53 s	0.45 s
(c)	$512 \times 512 \times 1112$	0.54 GB	1.16 s	0.93 s	0.61 s
(d)	$512 \times 512 \times 1202$	0.59 GB	0.86 s	0.70 s	0.64 s
(e)	$512 \times 512 \times 1202$	0.59 GB	0.69 s	0.46 s	0.37 s

Figure 5.5: Acceleration techniques tested on different datasets. Column four lists the render times for entry point determination at block level. The fifth column gives the render times for entry point determination using octree projection. The last column lists render times for octree projection plus additional cell invisibility caching.

environments.

5.5 Visualization Results

To demonstrate the applicability of the presented methods, we display visualization results for clinical datasets in Figures 5.8, 5.9, 5.10, and 5.11. The images show anatomic features and/or pathologies.

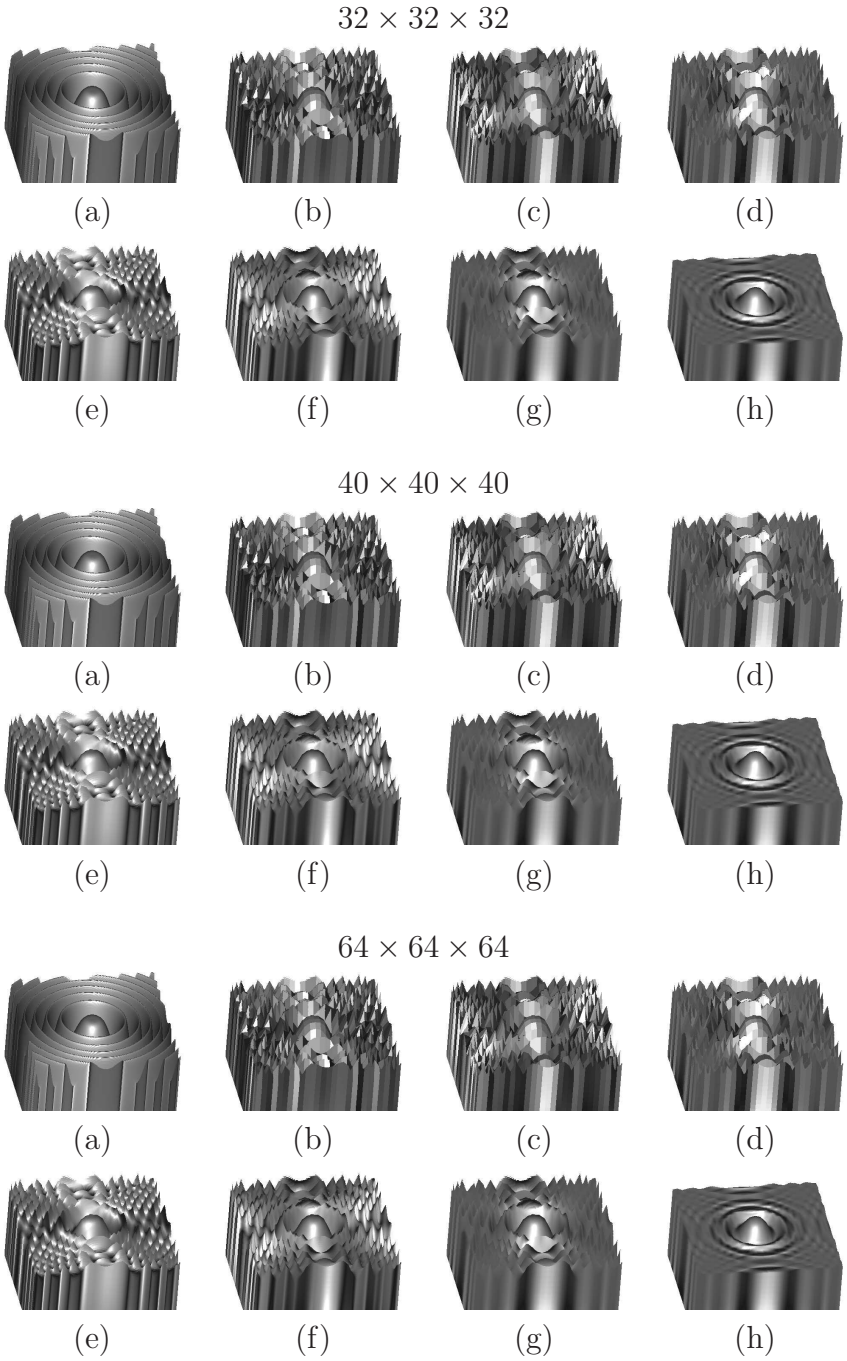


Figure 5.6: Comparison of reconstruction methods using the Marschner-Lobb test signal. (a) analytic evaluation. (b) zero-order interpolation, intermediate differences gradients. (c) zero-order interpolation, central differences gradients. (d) zero-order interpolation, Neumann gradients. (e) first-order interpolation, intermediate differences gradients. (f) first-order interpolation, central differences gradients. (g) first-order interpolation, Neumann gradients. (h) first-order interpolation, Neumann gradients and filtering.

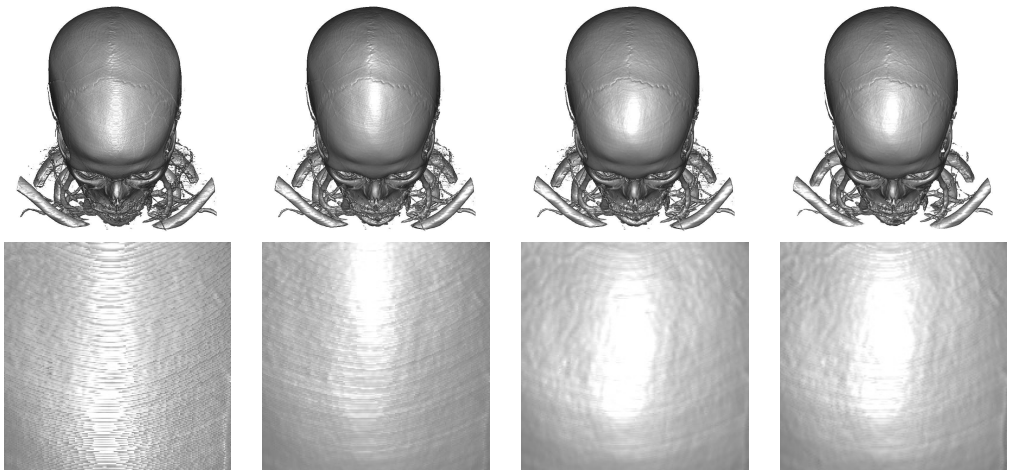


Figure 5.7: Comparison of gradient reconstruction methods. (a) first-order interpolation, intermediate differences gradients. (b) first-order interpolation, central differences gradients. (c) first-order interpolation, Neumann gradients. (d) first-order interpolation, Neumann gradients and filtering.

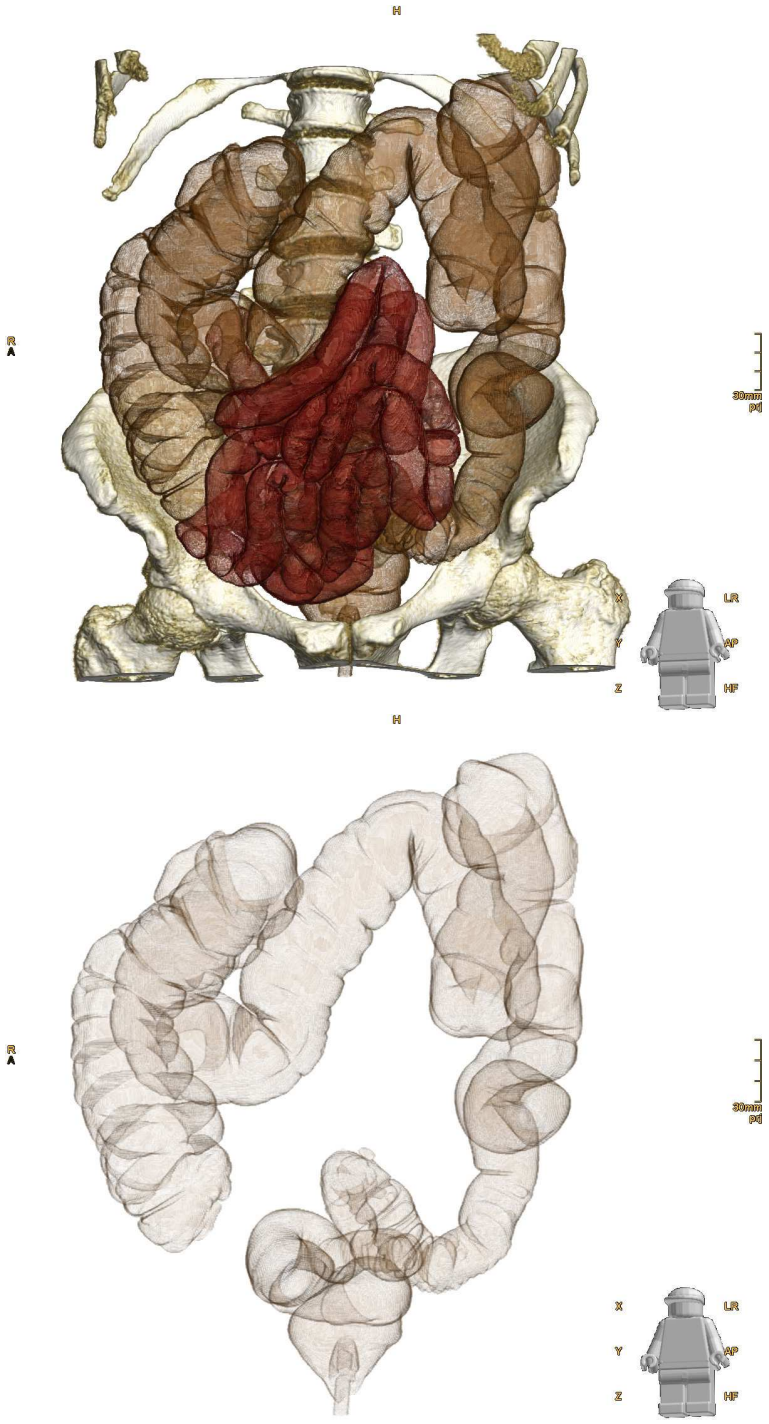


Figure 5.8: CT scan of colon. Bones and colon are displayed in the top image. The bottom image shows the colon without bones.

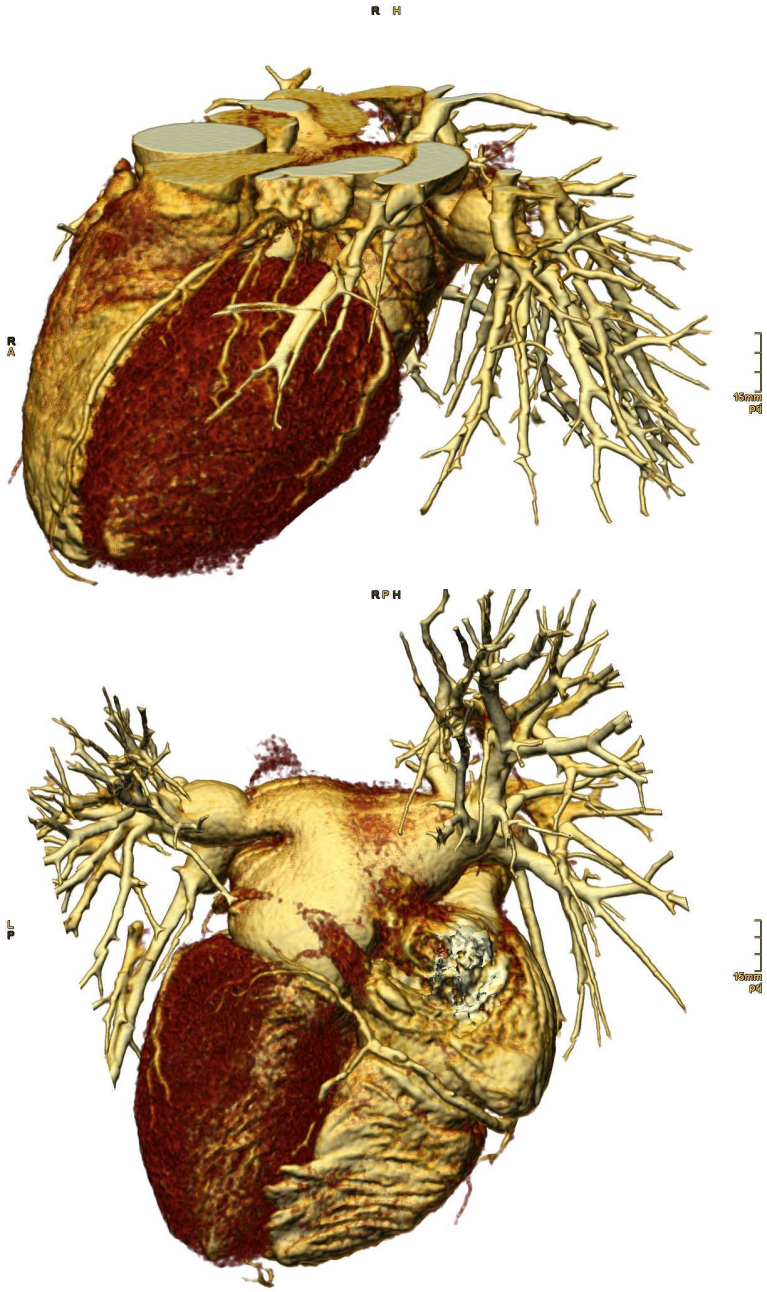


Figure 5.9: CT scan of heart. The myocardal muscle is displayed in red, the coronary vessels are depicted in yellow tones.

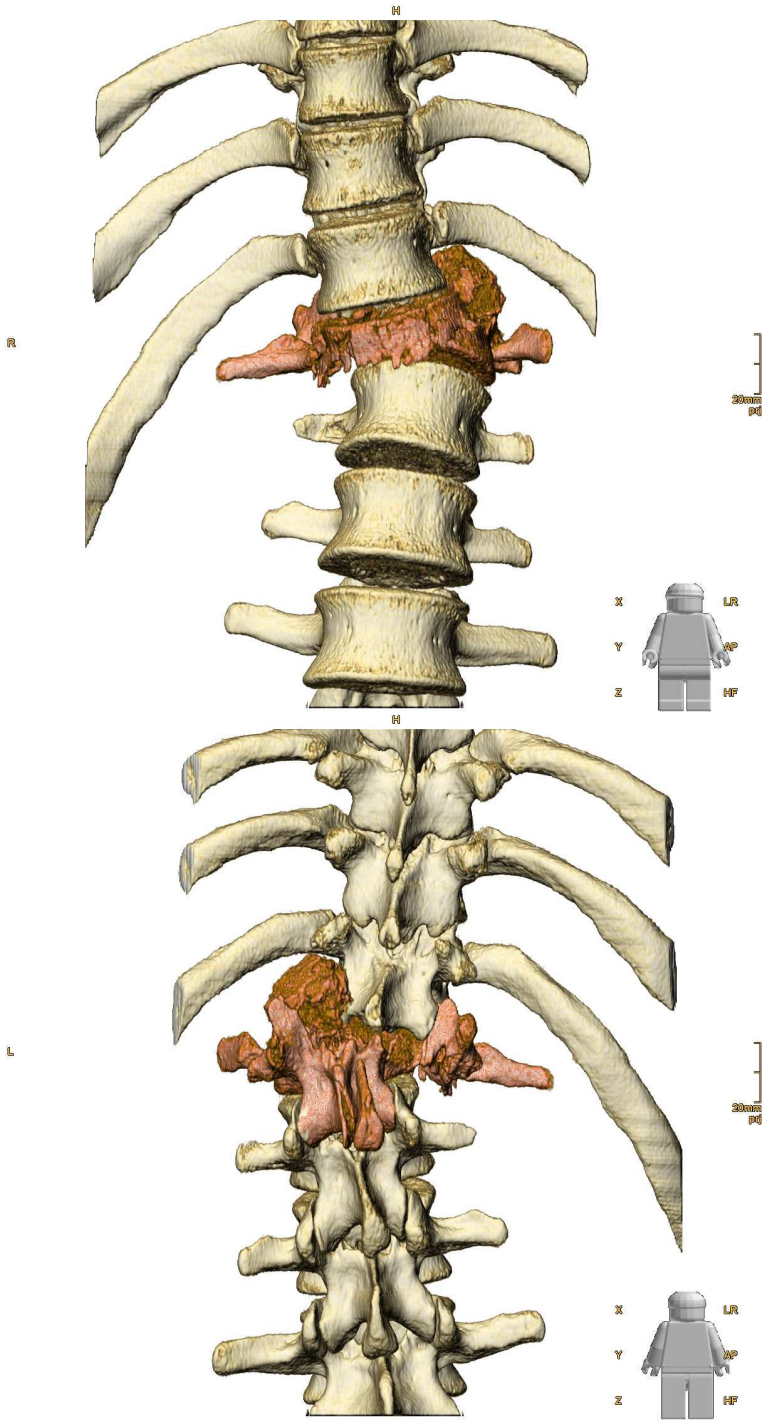


Figure 5.10: CT scan of lumbar spine. A fracture of a lumbar vertebra is highlighted.

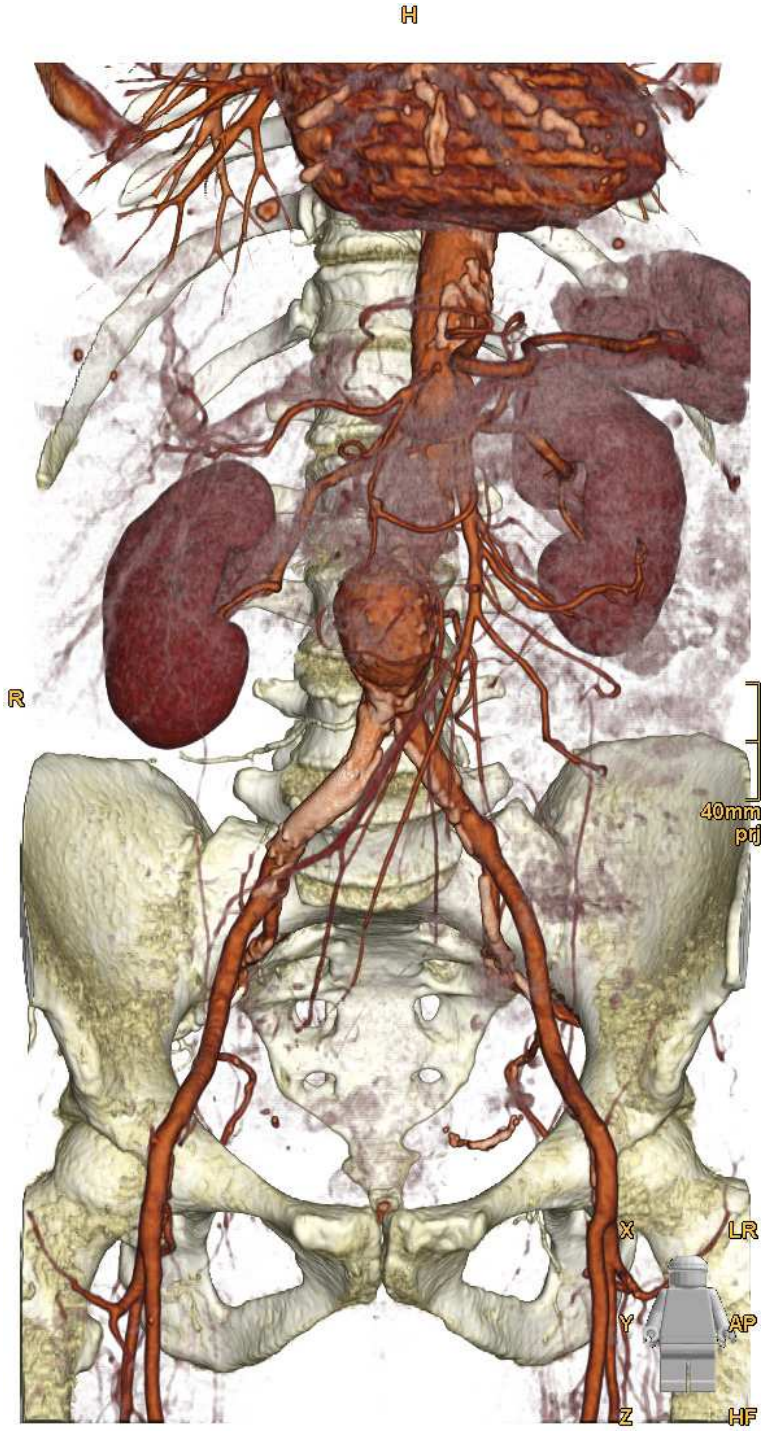


Figure 5.11: CT scan of abdomen. Through enhancement of the abdominal vascular structure an aorta aneurysma can be recognized.

Chapter 6

Summary

A conclusion is the place where you got tired of thinking.

Martin H. Fischer

In this final chapter, we summarize the main contributions presented in this diploma thesis.

6.1 Introduction

Direct volume rendering (DVR) is a powerful technique to visualize complex structures within volumetric data. Its main advantage, compared to standard surface rendering, is the ability to concurrently display information about the surface and the interior of objects. This aids the user in conveying spatial relationships of different structures.

In medicine, visualization of volumetric datasets acquired by computed tomography (CT), magnetic resonance imaging (MRI), or ultrasound imaging helps to understand patient's pathological conditions, improves surgical planning, and has an important role in education. However, a typical data size of today's clinical routine is about $512 \times 512 \times 1024$ (12 bit CT data)

and will increase in the near future due to technological advances in acquisition devices. Conventional slicing is of limited use for such large datasets due to the enormous amount of slices. However, providing interactive three-dimensional volume visualization of such large datasets is a challenging task.

6.2 Memory Management for Large Datasets

The past years have shown that the discrepancy between processor and memory performance is rapidly increasing, making memory access a potential bottleneck for applications which have to process large amounts of data. Raycasting, in particular, is prone to cause problems, since it generally leads to irregular memory access patterns. We discussed practical methods to improve memory access patterns taking advantage of the cache hierarchy.

6.2.1 Bricking

The most common way of storing volumetric data is a linear volume layout. Volumes are typically thought of as a number of two-dimensional images (slices) which are kept in an array. While this three-dimensional array has the advantage of simple address calculation, it has disadvantages when used in raycasting: Given the fact that rays are shot one after the other, the same data has to be read several times from main memory, because the cache is not large enough to hold the processed data of a single ray. This problem can be targeted by a technique called tile casting. Here, rather than processing one ray completely, each pass processes only one resample point for every ray. However, different viewing directions still cause a different amount of cache-line requests to load the necessary data from main memory, which leads to a varying frame-rate. The concept of bricking supposes the decomposition of data into small fixed-size data blocks. Each block is stored in linear order. The basic idea is to choose the block size according to the cache size of the architecture so that an entire block fits in a fast cache of the system.

6.2.2 Addressing

The addressing of data in a bricked volume layout is more costly than in a linear volume layout. To address one data element, one has to address the block itself and the element within the block. In contrast to this addressing scheme, a linear volume can be seen as one large block. To address a sample it is enough to compute just one offset. In algorithms like volume raycasting, which need to access a certain neighborhood of data in each processing step, the computation effort for two offsets instead of one generally cannot be neglected. In a linear volume layout, the offsets to neighboring samples are constant. Using bricking, the whole address computation would have to be performed for each neighboring sample that has to be accessed. To avoid this performance penalty, one can construct an if-else statement. The if-clause consists of checking if the needed data elements can be addressed within one block. If the outcome is true, the data elements can be addressed as fast as in a linear volume. If the outcome is false, the costly address calculations have to be done. This simplifies address calculation, but the involved if-else statement incurs pipeline flushes.

We therefore applied a different approach. We distinguished the possible sample positions by the locations of the needed neighboring samples. The first sample location (i, j, k) is defined by the integer parts of the current resample position. Assuming trilinear interpolation, during resampling neighboring samples to the right, top, and back of the current location are required. A block can be subdivided into subsets. For each subset, we can determine the blocks in which the neighboring samples lie. Therefore, it is possible to store these offsets in a lookup table.

The lookup table contains $8 \cdot 7 = 56$ offsets. We have eight cases, and for each sample (i, j, k) we need the offsets to its seven adjacent samples. The seven neighbors are accessed relative to the sample (i, j, k). Since each offset consists of four bytes the table size is 224 bytes. The basic idea is to extract the eight cases from the current resample position and create an index into a lookup table, which contains the offsets to the neighboring samples.

The input parameters of the lookup table addressing function are the

sample position (i, j, k) and the block dimensions B_x, B_y, and B_z. We assume that the block dimensions are a power of two, i.e., $B_x = 2^{N_x}$, $B_y = 2^{N_y}$, and $B_z = 2^{N_z}$. As a first step, the block offset part from i, j, and k is extracted by a conjunction with the corresponding $B_{\{x,y,z\}} - 1$. The next step is to increase all by one to move the maximum possible value of $B_{\{x,y,z\}} - 1$ to $B_{\{x,y,z\}}$. All the other possible values stay within the range $[1, B_{\{x,y,z\}} - 1]$. Then a conjunction of the resulting value and the complement of $B_{\{x,y,z\}} - 1$ is performed, which maps the input values to $[0, B_{\{x,y,z\}}]$. The last step is to add the three values and divide the result by the minimum of the block dimensions, which maps the result to [0,7]. This last division can be exchanged by a shift operation. In summary, the lookup table index for a position (i, j, k) is given by:

$$\begin{aligned} i' &= ((i \;\&\; (B_x - 1)) + 1) \;\&\; \sim (B_x - 1) \\ j' &= ((j \;\&\; (B_y - 1)) + 1) \;\&\; \sim (B_y - 1) \\ k' &= ((k \;\&\; (B_z - 1)) + 1) \;\&\; \sim (B_z - 1) \\ index &= (i' + j' + k') \gg \min(N_x, N_y, N_z) \end{aligned} \tag{6.1}$$

We use & to denote a *bitwise and* operation, | to denote a *bitwise or* operation, $\gg$ to denote a *right shift* operation, and $\sim$ to denote a *bitwise negation*.

A similar approach can be done for the gradient computation. We presented a general solution for a 26-connected neighborhood. Here we can, analogous to the resample case, distinguish 27 cases. The first step is to extract the block offset, by a conjunction with $B_{\{x,y,z\}} - 1$. Then we subtract one, and conjunct with $B_{\{x,y,z\}} + B_{\{x,y,z\}} - 1$, to separate the case if one or more components are zero. In other words, zero is mapped to $2 \cdot B_{\{x,y,z\}} - 1$. All the other values stay within the range $[0, B_{\{x,y,z\}} - 2]$. To separate the case of one or more components being $B_{\{x,y,z\}} - 1$, we add 1, after the previous subtraction is undone by a disjunction with 1, without loosing the separation of the zero case. Now all the cases are mapped to $\{0, 1, 2\}$ to obtain a ternary system. This is done by dividing the components by the corresponding block dimensions. These divisions can be replaced by faster shift operations. Then the three ternary variables are mapped to an index in the range of $[0, 26]$. In

summary, the lookup table index computation for a position (i, j, k) is:

$$\begin{aligned} i' &= (((((i \,\&\, (B_x - 1)) - 1) \,\&\, (2B_x - 1)) \mid 1) + 1) \gg N_x \\ j' &= (((((j \,\&\, (B_y - 1)) - 1) \,\&\, (2B_y - 1)) \mid 1) + 1) \gg N_y \\ k' &= (((((k \,\&\, (B_z - 1)) - 1) \,\&\, (2B_z - 1)) \mid 1) + 1) \gg N_z \\ index &= 9i' + 3j' + k' \end{aligned} \tag{6.2}$$

The presented index computations can be performed efficiently on current CPUs, since they only consist of simple bit manipulations. The lookup tables can be used in raycasting on a bricked volume layout for efficient access to neighboring samples. The first table can be used if only the eight samples within a cell have to be accessed (e.g., if gradients have been pre-computed). The second table allows full access to a 26-neighborhood. Compared to the if-else solution which has the costly computation of two offsets in the else branch, we get a speedup of about 30%. The benefit varies, depending on the block dimensions. For a $32 \times 32 \times 32$ block size the else-branch has to be executed in 10% of the cases and for a $16 \times 16 \times 16$ block size in 18% of the cases.

6.2.3 Traversal

It is most important to ensure that data once replaced in the cache will not be required again to avoid thrashing. Law and Yagel have presented a thrashless distribution scheme for parallel raycasting [22]. Their scheme relies on an object space subdivision of the volume. While their method was essentially developed in the context of parallelization, to avoid redundant distribution of data blocks over a network, it is also useful for a single-processor approach.

The volume is subdivided into blocks. These blocks are then sorted in front-to-back order depending on the current viewing direction. The ordered blocks are placed in a set of block lists in such a way that no ray that intersects a block contained in a block list can intersect another block from the same block list. Each block holds a list of rays whose current resample position lies within the brick. The rays are initially added to the list of the block which

they first intersect. The blocks are then traversed in front-to-back order by sequentially processing the block lists. The blocks within one block list can be processed in any order, e.g., in parallel. For each block, all rays contained in its list are processed. As soon as a ray leaves a block, it is removed from its list and added to the new block's list. When the ray list of a block is empty, processing is continued with the next block. Due to the subdivision of the volume, it is very likely that a block entirely remains in a fast cache while its rays are being processed, provided the block size is chosen appropriately. The generation of the block lists does not have to be performed for each frame. For parallel projection there are eight distinct cases where the order of blocks which have to be processed remains the same. Thus, the lists can be pre-computed for these eight cases.

6.3 Parallelization Strategies for Commodity Hardware

Raycasting has always posed a challenge on hardware resources. Thus, numerous approaches for parallelization have been presented. As our target platform is consumer hardware, we have focused on two parallelization schemes available in current stand-alone PCs: Symmetric Multiprocessing (SMP) and Simultaneous Multithreading (SMT).

6.3.1 Symmetric Multiprocessing

Computer architectures using multiple similar processors connected via a high-bandwidth link and managed by one operating system are referred to as Symmetric Multiprocessing systems. Each processor has equal access to I/O devices. As Law and Yagel's traversal scheme [22] was originally developed for parallelization, it is straight-forward to apply to SMP architectures. The blocks in each of the block lists can be processed simultaneously. Each block list is partitioned among the $count_{physical}$ CPUs available.

A possible problem occurs when rays from two simultaneously processed blocks have the same subsequent block. One way of handling these cases

would be to use synchronization primitives, such as mutexes or critical sections, to ensure that only one thread can assign rays at a time. However, the required overhead can decrease the performance drastically. Therefore, to avoid the race conditions when two threads try to add rays to the ray list of a block, each block has a list for every physical CPU. When a block is being processed, the rays of all these lists are processed. When a ray leaves the block, it is added to the new block's ray list corresponding to the CPU currently processing the ray.

6.3.2 Simultaneous Multithreading

Simultaneous Multithreading is a well-known concept in workstation and mainframe hardware. It is based on the observation that the execution resources of a processor are rarely fully utilized. Due to memory latencies and data dependencies between instructions, execution units have to wait for instructions to finish. While modern processors have out-of-order execution units which reorder instructions to minimize these delays, they rarely find enough independent instructions to exploit the processor's full potential. SMT uses the concept of multiple logical processors which share the resources of just one physical processor. Executing two threads simultaneously on one processor has the advantage of more independent instructions being available, thus increasing CPU utilizations. This can be achieved by duplicating state registers, which only leads to little increases in manufacturing costs. Intel's SMT implementation is called Hyper-Threading and was first available on the Pentium 4 CPU. Currently, two logical CPUs per physical CPU are supported.

For exploiting SMT, it is essential that the threads operate on neighboring data items, since the logical processors share chaches. Therefore, treating the logical CPUs in the same way as physical CPUs leads to little or no performance increase. Instead, it might even lead to a decrease in performance, due to cache thrashing. Thus, the processing scheme has to be extended in order to allow multiple threads to operate within the same block.

The blocks are distributed among physical processors as described in the

previous section. Additionally, within a block, multiple threads each executing on a logical CPU simultaneously process the rays of a block. Using several threads to process the ray list of a block would lead to race conditions and would therefore require expensive synchronization. Thus, instead of each block having just one ray list for every physical CPU, we now have $count_{logical}$ lists per physical CPU, where $count_{logical}$ is the number of threads that will simultaneously process the block, i.e., the number of logical CPUs per physical CPU. Thus, each block has $count_{physical} \cdot count_{logical}$ ray lists $raylist[id_{physical}][id_{logical}]$ where $id_{physical}$ identifies the physical CPU and $id_{logical}$ identifies the logical CPU relative to the physical CPU. A ray can move between physical CPUs depending on how the block lists are partitioned within in each pass, but they always remain in a ray list with the same $id_{logical}$. This means that for equal workloads between threads, the rays have to be initially distributed among these lists, e.g., by alternating the $id_{logical}$ of the list a ray is inserted to during ray setup.

6.4 Memory Efficient Acceleration Data Structures

Applying efficient memory access and parallelization techniques still is not sufficient to efficiently handle the huge processing loads caused by large datasets. We presented algorithmic optimizations to reduce this workload. We introduced three techniques which each can achieve a significant reduction of rendering times. Our goal was to minimize the additional memory requirements of newly introduced data structures.

6.4.1 Gradient Cache

When using an expensive gradient estimation method, caching of intermediate results is inevitable if high performance has to be achieved. An obvious optimization is to perform gradient estimation only once for each cell. When a ray enters a new cell, the gradients are computed at all eight corners of the cell. The benefit of this method is dependent on the number of resample

locations per cell, i.e., the object sample distance. However, the computed gradients are not reused for other cells. This means that each gradient typically has to be computed eight times. For expensive gradient estimation methods, this can considerably reduce the overall performance. It is therefore important to store the results in a gradient cache. However, allocating such a cache for the whole volume still has the mentioned memory problem.

Our blockwise volume traversal scheme allows us to use a different approach. We perform gradient caching on a block basis. The cache is able to store one gradient entry for every grid position contained in a cell of the current block. Thus, the required cache size is $(B_x + 1) \times (B_y + 1) \times (B_z + 1)$ where B_x, B_y, B_z are the block dimensions. The block dimensions have to be increased by one to enable interpolation across block boundaries. Each entry of the cache stores the three components of a gradient, using a 4 byte single precision floating-point number for each component. Additionally, a bit array has to be stored that encodes the presence of an entry in the cache for each grid position in a cell of the current block.

When a ray enters a new cell, for each of the eight corners of the cell the bit set is queried. If the result of a query is zero, the gradient is computed and written into the cache. The corresponding value of the bit set is set to one. If the result of the query is one, the gradient is already present in the cache and is retrieved. The disadvantage of this approach is that gradients at block borders have to be computed multiple times. However, this caching scheme still greatly reduces the performance impact of gradient computation and requires only a modest amount of memory. Furthermore, the required memory is independent of the volume size, which makes this approach applicable to large datasets.

6.4.2 Entry Point Buffer

One of the major performance gains in volume rendering can be achieved by efficiently skipping data which is classified as transparent. In particular, it is important to begin sampling at positions close to the data of interest, i.e., the non-transparent data. This is particularly true for medical datasets, as

the data of interest is usually surrounded by large amounts of empty space (air). The idea is to find, for every ray, a position close to its intersection point with the visible volume, thus, we refer to this search as entry point determination. The advantage of entry point determination is that it does not require additional overhead during the actual raycasting process, but still allows to skip a high percentage of empty space. The entry points are determined in the ray setup phase and the rays are initialized to start processing at the calculated entry position. The basic goal of entry point determination is to establish a buffer, the entry point buffer, which stores the position of the first intersection with the visible volume for each ray.

As blocks are the basic processing entities of our algorithm, the first step is to find all blocks which do not contribute to the visible volume using the current classification, i.e., all blocks that only contain data values which are classified as transparent. It is important that the classification of a whole block can be calculated efficiently to allow interactive transfer function modification. We store the minimum and maximum value of the samples contained in a block and use a summed area table of the opacity transfer function to determine the visibility of the block. We then perform a projection of each non-transparent block onto the image plane with hidden surface removal to find the first intersection point of each ray with the visible volume. The goal is to establish an entry point buffer of the same size as the image plane, which contains the depth value for each ray's intersection point with the visible volume. For parallel projection, this step can be simplified. As all blocks have exactly the same shape, it is sufficient to generate one template by rasterizing the block under the current viewing transformation. Projection is performed by translating the template by a vector $t = (t_x, t_y, t_z)^T$ which corresponds to the block's position in three-dimensional space in viewing coordinates. Thus, t_x and t_y specify the position of the block on the image plane (and therefore the location where the template has to be written into the entry point buffer) and t_z is added to the depth values of the template. The Z-buffer algorithm is used to ensure correct visibility. In ray setup, the depth values stored in the entry point buffer are used to initialize the ray positions.

The disadvantage of this approach is that it requires an addition and a

depth test at every pixel of the template for each block. This can be greatly reduced by choosing an alternative method. The blocks are projected in back-to-front order. The back-to-front order can be easily established by traversing the generated block lists in reverse order. For each block the Z-value of the generic template is written into the entry point buffer together with a unique index of the block. After the projection has been performed, the entry point buffer contains the indices and relative depth values of the entry points for each ray. In ray setup, the block index is used to find the translation vector t for the block and t_z is added to the relative depth value stored in the buffer to find the entry point of the ray. The addition only has to be performed for every ray that actually intersects the visible volume.

We further extended this approach to determine the entry points in a finer resolution than block granularity. We replaced the minimum and maximum values stored for every block by a min-max octree. Its root node stores the minimum and maximum values of all samples contained in the block. Each additional level contains the minimum and maximum value for smaller regions, resulting in a more detailed description of parameter variations inside the block. Every time the classification changes, the summed area table is recursively evaluated for each octree node and the classification information is stored as linearized octree bit encoding using hierarchy compression.

The projection algorithm was modified as follows. Instead of one block template there is now a template for every octree level. The projection of one block is performed by recursively traversing the hierarchical classification information in back-to-front order and projecting the appropriate templates for each level, if the corresponding octree node is non-transparent. In addition to the block index, the entry point buffer now also stores an index for the corresponding octree node. In ray setup, the depth value in the entry point buffer is translated by the t_z component of the translation vector plus the sum of the relative offsets of the node in the octree. The relative translational offsets for the octree nodes can be pre-computed and stored in a lookup table.

6.4.3 Cell Invisibility Cache

We introduced a cell invisibility cache to skip the remaining transparent regions at cell level. We can skip the resampling and compositing in a cell if all eight samples of the cell are classified as transparent. To determine the transparency, a transfer-function lookup has to be performed for each of these samples. For large zoom factors, several rays can hit the same cell and for each of these rays the same lookups would have to be performed.

A cell invisibility cache is attached at the beginning of the traditional volume raycasting pipeline. This cache is initialized in such a way that it reports every cell as visible. In other words every cell has to be classified. Now, if a ray is sent down the pipeline, every time a cell is classified invisible this information is stored in the cache. If a cell is found to be invisible, this information is stored by setting the corresponding bit in the cell invisibility cache. As the cache stores the combined information for eight samples of a cell in just one bit, this is more efficient than performing a transfer function lookup for each sample. The information stored in the cell invisibility cache remains valid as long as no transfer function modifications are performed. During the examination of the data, e.g., by changing the viewing direction, the cache fills up and the performance increases progressively.

The advantage of this technique is that no extensive computations are required when the transfer function changes. The reset of the buffer can be performed with virtually no delay, allowing fully interactive classification. As transfer function specification is a non-trivial task, minimizing delays initiated by transfer function modifications greatly increases usability.

6.5 Results

We performed a comprehensive performance evaluation of the proposed techniques. The results were obtained by thorough experiments on diverse hardware.

6.5.1 Memory Management for Large Datasets

For a comparison of bricked and linear volume layouts, we used a Dual Intel Pentium Xeon 2.4 GHz equipped with 512 KB level-2 cache, 8 KB level-1 data cache, and 1 GB of Rambus memory.

In our system, we are able to support different block sizes, as long as each block dimension is a power of two. If we set the block size to the actual volume dimensions, we have a common raycaster which operates on a simple linear volume layout. This enables us to make a meaningful comparison between a raycaster which operates on a simple linear volume layout and a raycaster which operates on a bricked volume layout. To underline the effect of bricking we benchmarked different block sizes. We have a speedup of about 2.0 with a block size of 1 KB. Increasing the block size up to 64 KB also increases the speedup. This is due to more efficient use of the cache. The chart shows an optimum at a block size of 64KB ($32 \times 32 \times 32$) with a speedup of about 2.8. This number is the optimal tradeoff between the needed cache space for ray data structures, sample data, and lookup tables. Larger block sizes lead to performance decreases, as they are too large for the cache, but still suffer from the overhead caused by bricking. This performance drop-off is reduced, once the block size approaches the volume size. With only one volume-sized block, the rendering context is that of a common raycaster operating on a linear volume layout.

6.5.2 Parallelization Strategies for Commodity Hardware

To evaluate the performance of our parallelization strategies, we used the same test system as in the previous section. This system has two CPUs and supports Hyper-Threading.

Our system is able to force threads on specific physical and logical CPUs. By following this mechanism we tested different configurations to obtain figures for the speedup achieved by the presented techniques. All test runs consistently showed the same speedup factors. Testing Simultaneous Multithreading on only one CPU showed an average speedup of 30%. While

changing the viewing direction, the speedup varies from 25% to 35%, due to different transfer patterns between the level 1 and the level 2 cache. Whether Hyper-Threading is enabled or disabled adding a second CPU approximately reduces the computational time by 50%, i.e., Symmetric Multiprocessing and Simultaneous Multithreading are independent. This shows that our Simultaneous Multithreading scheme scales well on multi-processor machines. The Hyper-Threading benefit of approximately 30% is maintained if the second hyper-threaded CPU is enabled.

For different block sizes, the speedup for Simultaneous Multithreading varies. The speedup significantly decreases with larger block sizes. Once the level 2 cache size is exceeded, the two threads have to request data from main memory. Therefore, the CPU execution units are less utilized. Very small block sizes suffer from a different problem. The data fits almost into the level 1 cache. This means that one thread can utilize the execution units more efficiently, and the second thread is idle during this time. But the overall disadvantage is the inefficient usage of the level 2 cache. The optimal speedup $\frac{100}{100-30} \approx 1.42$ is achieved with a block size of 64 KB ($32 \times 32 \times 32$). This is also the optimal block size for the bricked volume layout.

6.5.3 Memory Efficient Acceleration Data Structures

To demonstrate the impact of our high-level optimizations we used a commodity notebook system equipped with an Intel Centrino 1.6 GHz CPU, 1 MB level 2 cache, and 1 GB RAM. This system has one CPU and does not support Hyper-Threading so the presented results only reflect performance increases due to our high-level acceleration techniques.

The memory consumption of the gradient cache is not related to the volume dimensions, but determined by the fixed block size. We use $32 \times 32 \times 32$ sized blocks, the size of the gradient cache therefore is is $(33)^3 \cdot 3 \cdot 4$ byte $\approx$ 422 KB. Additionally we store for each cache entry a validity bit, which adds up to $33^3/8$ bytes $\approx$ 4.39 KB.

The impact of our gradient caching scheme is determined by the zoom factor and the object sample distance. In case of zoom factor 1.0 we have one

ray per cell, already here per block gradient caching performs better than per cell gradient caching. This is due to the shared gradients between cells. For zooming out both gradient caching schemes perform equally well. The rays are so far apart such that nearly no gradients can be shared. On the other hand, for zooming in, per block caching performs much better than per cell caching. This is due to the increased number of rays per cell. For a zoom factor of 2.0, per block gradient caching achieves a speedup of approximately 3.0 compared to no gradient caching at a typical object sample distance of 0.5.

The additional memory usage of the acceleration data structures is rather low. The cell invisibility cache has a size of 32^3 bit = 4096 byte. The min-max octree has a depth of three storing 4 byte at each node (a 2 byte minimum and maximum value) and requires at most 2340 byte. Additionally, the classification information is stored, which requires 66 byte. We use blocks of size $32 \times 32 \times 32$ storing 2 bytes for each sample, which is a total of 65536 bytes. Our data structures increase the total memory requirements by approximately 10%.

Our tests have shown that a combination of the proposed optimizations achieves render times of about 2 frames per second for various large datasets.

6.6 Conclusion

We have presented different techniques for volume visualization of large datasets on commodity hardware. We have shown that efficient memory management is fundamental to achieve high performance. Our work on parallelization has demonstrated that well-known methods for large parallel systems can be adapted and extended to exploit evolving technologies, such as Simultaneous Multithreading. Our memory efficient data structures provide frames per second performance even for large datasets. A key point of our work was to demonstrate that commodity hardware is able to achieve the performance necessary for real-world medical applications. In future work, we will investigate out-of-core and compression methods to permit the use of even larger datasets.

Acknowledgements

So long, and thanks for all the fish.

Douglas Adams

First of all, I want to thank my supervisor Sören Grimm for all his advise and help. I further thank Eduard Gröller, our beloved master, for his supervision and support. Many thanks to Rainer Wegenkittel for creating fantastic images and for being a driving factor for continuous improvement of our implementation. Thanks to Armin Kanitsar for helpful discussions and suggestions, as well as successful management of the AD**A**PT project. I want to express my gratitude to *Tiani Medgraph* for making this work possible.

The presented work has been funded by the AD**A**PT project (FFF-804544). AD**A**PT is supported by *Tiani Medgraph*, Austria (`http://www.tiani.com`), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See `http://www.cg.tuwien.ac.at/research/vis/adapt` for further information on this project. The used datasets are courtesy of Univ.-Klinik Innsbruck, AKH Vienna, Univ.-Klinikum Freiburg, and NLM.

Bibliography

[1] James F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics*, 11(2):192–198, 1977.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the Symposium on Volume Visualization 1994*, pages 91–98, 1994.

[3] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the reconstruction of surfaces from tomograms. *Medical Physics*, 15(3):320–327, 1988.

[4] D. Cohen and Z. Sheffer. Proximity clouds: An acceleration technique for 3D grid traversal. *The Visual Computer*, 11(1):27–38, 1994.

[5] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the Workshop on Graphics Hardware 2001*, pages 9–16, 2001.

[6] J. L. Freund and K. Sloan. Accelerated volume rendering using homogenous region encoding. In *Proceedings of Visualization 1997*, pages 191–196, 1997.

[7] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of the Symposium on Volume Visualization 1996*, pages 23–30, 1996.

[8] N. Gershon. From perception to visualization. In L. Rosenblum, R.A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann, editors, *Scientific Visualization - Advances and Challenges*, pages 129–139. Academic Press, 1994.

[9] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5), 2004. To appear.

[10] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. *Computers & Graphics*, 19(5):705–710, 1995.

[11] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *Proceedings of the Visualization 2002*, pages 53–60, 2002.

[12] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing*, 9(1):1–21, 1979.

[13] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[14] G. Kindlmann and J. W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of the Symposium on Volume Visualization 1998*, pages 79–86, 1998.

[15] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[16] G. Knittel. A scalable architecture for volume rendering. *Computers & Graphics*, 19(5):653–665, 1995.

[17] G. Knittel. The UltraVis system. In *Proceedings of the Symposium on Volume Visualization 2000*, pages 71–79, 2000.

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

[19] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, Computer Systems Laboratory, 1995.

[20] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series):451–458, 1994.

[21] A. Law and R. Yagel. Exploiting spatial, ray, and frame coherency for efficient parallel volume rendering. In *Proceedings of GRAPHICON 1996*, pages 93–101, 1996.

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

[23] A. Law and R. Yagel. An optimal ray traversal scheme for visualizing colossal medical volumes. In *Proceedings of Visualization in Biomedical Computing 1996*, pages 43–52, 1996.

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

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

[26] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, 1990.

[27] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–168, 1987.

[28] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[29] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization 1994*, pages 100–107, 1994.

[30] N. Max. Optical models for volume rendering. In M. Göbel, H. Müller, and B. Urban, editors, *Visualization in Scientific Computing*, pages 35–40. Springer-Verlag Wien, 1994.

[31] M. Meißner, U. Hoffmann, and W. Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. In *Proceedings of Visualization 1999*, pages 207–214, 1999.

[32] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings of the Symposium on Volume Visualization 2000*, pages 81–90, 2000.

[33] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, and R. Proksa. VIZARD II: A reconfigurable interactive volume rendering system. In *Proceedings of the Workshop on Graphics Hardware 2002*, pages 137–146, 2002.

[34] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. A comparison of normal estimation schemes. In *Proceedings of Visualization 1997*, pages 19–26, 1997.

[35] B. Mora, J.-P. Jessel, and R. Caubet. A new object-order ray-casting algorithm. In *Proceedings of Visualization 2002*, pages 203–210, 2002.

[36] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *Proceedings of Visualization 1998*, pages 239–246, 1998.

[37] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[38] L. Neumann, B. Csébfalvi, A. König, and M. E. Gröller. Gradient estimation in volume data using 4D linear regression. In *Proceedings of Eurographics 2000*, pages 351–358, 2000.

[39] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An architecture for low-cost real-time volume rendering. In *Proceedings of the Workshop on Graphics Hardware 1997*, pages 131–138, 1997.

[40] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[41] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Proceedings of SIGGRAPH 1999*, pages 251–260, 1999.

[42] Bui-Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[43] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, 1984.

[44] H. Ray, H. Pfister, D. Silver, and T. A. Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[45] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multitextures and multi-stage rasterization. In *Proceedings of the Workshop on Graphics Hardware 2000*, pages 109–118, 2000.

[46] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualisation 2003*, pages 231–238, 2003.

[47] C. Schlick. A fast alternative to Phong's specular model. In P. Heckbert, editor, *Graphics gems IV*, pages 385–387. Academic Press, 1994.

[48] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, 1990.

[49] R. Srinivasan, S. Fang, and S. Huang. Volume rendering by template-based octree projection. In *Proceedings of the Workshop on Visualization in Scientific Computing 1997*, pages 155–163, 1997.

[50] B. T. Stander and J. C. Hart. A Lipschitz method for accelerated volume rendering. In *Proceedings of the Symposium on Volume Visualization 1994*, pages 107–114, 1994.

[51] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *Proceedings of the Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization 2002*, pages 95–104, 2002.

[52] J. van Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proceedings of the Workshop on Graphics Hardware 1995*, pages 51–55, 1995.

[53] T. van Walsum, A. J. S. Hin, J. Versloot, and F. H. Post. Efficient hybrid rendering of volume data and polygons. In F. H. Post and A. J. S. Hin, editors, *Advances in Scientific Visualization*, pages 83–96. Springer-Verlag Berlin-Heidelberg, 1992.

[54] B. A. Wallace. Merging and transformation of raster images for cartoon animation. *Computer Graphics*, 15(3):253–262, 1981.

[55] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH 1998*, pages 169–178, 1998.

[56] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, 1990.

[57] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):153–167, 1992.

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

List of Figures

List of Tables