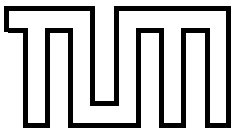
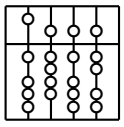


Technische Universität München
Fakultät für Informatik

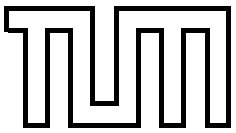


Diplomarbeit

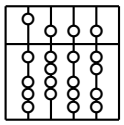
Volume Rendering Techniques for Medical Imaging

In collaboration with Siemens Corporate Research Inc., Princeton, USA

Peter Lücke



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Volume Rendering Techniques for Medical Imaging

In collaboration with Siemens Corporate Research Inc., Princeton, USA

Peter Lücke

Aufgabensteller:	Prof. Nassir Navab, Ph.D.
Betreuer:	Dipl.Inf. Wolfgang Wein (TUM) Ali Khamene, Ph.D. (SCR)
Abgabedatum:	15. April 2005

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. April 2005 Peter Lücke

Abstract

The visualization of volumetric data sets is an integral component of medical imaging applications. Usually, visualization of volumetric data is computational quite expensive. Therefore, early adoptions of direct volume rendering techniques were not able to deliver volume visualization at decent speeds, i.e interactive rates. Admittedly, high rendering performance is crucial to such applications as clinical diagnostics or image registration. For this reason, much time has been spent on refining the available volume rendering algorithms. Other approaches for reducing the rendering speeds of volumetric data sets are based on the use of dedicated graphical hardware. Unfortunately, such graphics hardware used in workstations has high monetary costs. However, due to the increased level of programmability and performance of commodity graphics hardware it is not only possible to perform traditional visualization tasks on a cheap personal computer instead of an expensive workstation, but to use rendering algorithms that previously could not be employed for real-time visualization at all. Although consumer graphics hardware was designed with only acceleration of textured polygons in mind, the performance of volume rendering applications based on commodity graphics hardware is astonishing.

In this diploma thesis I describe different methods for direct volume rendering and their respective implementation details. Therefore I integrated these techniques into the chair's program library. The implemented methods for direct volume rendering can be used for both image visualization and image registration. One part of the library provides methods for different direct volume rendering techniques on consumer graphics hardware. The other part provides a novel approach for accelerating direct volume rendering without the need for dedicated hardware, i.e that can be executed on general purpose processors.

Zusammenfassung

Die Visualisierung volumetrischer Daten ist ein integraler Bestandteil medizinischer Bildverarbeitungsanwendungen. Da die Visualisierung solcher Daten sehr rechenintensiv ist, waren die ersten Anwendungen nicht in der Lage Volumen in interaktiven Geschwindigkeiten darzustellen. Allerdings ist eine interaktive Darstellung sehr wichtig, sowohl für eine medizinische Diagnose als auch für die sogenannte Bildregistrierung. Daher wurde viel Zeit darauf verwendet, die vorhandenen Algorithmen zur Darstellung von Volumen zu verbessern. Andere Ansätze versuchten die Visualisierung durch die Verwendung von dedizierter Grafikhardware zu beschleunigen. Ein Nachteil dieser dedizierten Grafikhardware ist allerdings, dass entsprechende Workstations sehr hohe Anschaffungskosten haben. Allerdings ist es möglich, durch die zunehmende Programmierbarkeit und die zunehmende Leistung von - im Handel erhältlichen - Grafikbeschleunigern, gängige Visualisierungsaufgaben ebenso auf weitaus billigeren PCs durchzuführen. Darüber hinaus ermöglichen solche Grafikkarten auch die Anwendung neuer Darstellungsalgorithmen, die so bis jetzt im Echtzeit-bereich nicht möglich waren. Die Leistung, die bei der Verwendung entsprechender Grafikhardware erreicht werden kann ist erstaunlich und das, obwohl sie eigentlich nur zur Beschleunigung polygonaler Objekte entwickelt wurden.
Im Zuge dieser Arbeit wurde eine Programmbibliothek geschrieben, die sowohl für Visualisierung als auch für Bildregistrierung verwendet werden kann. Die Bibliothek stellt Methoden für eine hardware-beschleunigte Darstellung volumetrischer Daten zur Verfügung. Zusätzlich stellen wir einen neuartigen Ansatz vor, der dazu dienen soll die Darstellung zu beschleunigen ohne dabei auf spezielle Hardware zu setzen.

Preface

About this work The document at hand is my diploma thesis for the computer science degree at the *Technische Universität München*. A portion of it originates from my work at the *Chair for Computer Aided Medical Procedures* of the Technische Universität München. Further work was done during my internship at *Siemens Corporate Research, Inc.*, Princeton USA from September 2004 to January 2005. All presented techniques for visualizing volumetric data sets have been implemented in the chair's CAMPLIB library. Both software based techniques were developed during my time at SCR and implemented in the *RTReg* project.

Acknowledgements First, I would like to thank the department head Frank Sauer for inducing all necessary actions for my internship. I also have to thank my direct supervisor at SCR, Ali Khamene, for his helpful advice, his technical knowledge, and his patience.
I am very indebted to Univ.-Prof. Dr. Nassir Navab, head of the chair, for conducting this diploma thesis with me, as well. Additionally, I have to reciprocate my supervisor at the chair, Wolfgang Wein, for his advice and for recommending me to SCR.
Last but not least, I want to give everyone else at the Chair for Computer Aided Medical as well as all other interns at SCR props for exchanging technical issues with me.

Contents

List of Figures

Listings

1 Introduction

1.1 Motivation

Medical Imaging techniques have become a critical part of medical applications in general. These techniques can be used in many different fields of application, such as operation planning, segmentation, clinical diagnostics, and post-evaluation of performed surgical and radio-therapeutical treatment. Ever since the discovery of x-rays by Wilhelm C. Röntgen in 1895, x-rays have been a crucial component of medical applications. By exposing a patient to x-rays an 2-dimensional image can be generated that displays the radiation attenuation through the patient's body. Later, other techniques, like ultrasound or angiography were used to generate 2-dimensional images from a patient's body as well. However, 2-dimensional images are just one of the modalities used in medical imaging today. Additionally, 3-dimensional images describe another set of modalities that is crucial to medical imaging applications today. Such images can be thought of a set of several 2-dimensional images that comprise, if put in order, a 3-dimensional spatial region.

Typically, a 3-dimensional data set is referred to as a *volumetric data set* or *volume*. They can be acquired from many different sources, such as Computational Fluid Dynamics, or Seismic Data Measurement. In general, any data that can be represented as a 3-dimensional scalar field can be considered a volume. Volumetric data sets can , of course, be created synthetically, i.e procedurally [21, 50]. Such an approach is especially appropriate for describing fluids, gaseous objects, natural phenomena like clouds, fog, and fire, and visualizing molecular structures.

With respect to medical imaging, volumetric data sets can be acquired from many different medical methods as well, such as Computed Tomography (CT), Magnetic Resonance Imaging (MRI), Ultrasound (US), or Positron Emission Tomography (PET). An outstanding amount of anatomical details can be obtained, especially, by Computed Tomography and Magnetic Resonance scans. These techniques also allow for extracting a set of profiles from a patient's body. In general, a 3-dimensional representation of a 3-dimensional spatial region has more significance than 2-dimensional images of the same region acquired by methods, such as x-ray, angiography, and ultrasound. Therefore, volumetric data sets about a patient's physical condition are a vital part of *clinical diagnostics*, today.

Although deemed very valuable in practice, medical images are always acquired at specific expenses. These expenses can consist of high monetary costs, of a specific amount of effort necessary, and sometimes even of sanitary burden on the patient's physical health. In order to lessen these expanses, the application of the aforementioned techniques is kept to

a minimum. Therefore, the data stored within a volume has to be used to its full capacity. Due to this fact, data acquired by different type of sensors, or modalities, is often tried to be merged for subsequent diagnostics or operational treatment. Consequently, medical images have to be brought into spatial alignment so that the spatial relationships of anatomical structures within are patient's body match within all involved data. This process is commonly referred to as *registration*. Registered medical images can then be used for further treatment and diagnostics. Another important application of registration is the aligning of volumetric data sets that have been acquired at different points in time, thus allowing to give evidence about a patient's medical progress.

Both, clinical diagnostics and registration, need means to visualize the data contained within a volume. In the past, this visualization was done displaying each slice of the volumetric data set separately or by segmenting the volumetric data set into 3-dimensional objects, that can be visualized with traditional techniques. The later approach is commonly referred to as *indirect volume rendering*, due to the transforming of the volumetric representation into a polygonal representation of the same data. However, such a transforming approach is susceptible to loosing information stored in a volume. Therefore, *direct volume rendering* techniques have been proposed, that visualize the volumetric data set directly, i.e without changing is representation. The early methods, such as *volume ray-casting* [46, 83] or *volume splatting* [87] were not capable to visualize a volumetric data set at decent speeds due to the high computational demands of these methods. For example, a volume constituted from 512 images, each 512×512 in size, and storing its data at 16-bit precision, takes up 256 megabyte of host memory. Additionally, the bandwidth consumption when accessing a volumetric data set during the generation of a single frame skyrockets, as basically all data within the volume has to be transferred over the bus. However, it is crucial for clinical diagnostics applications that a volume can be visualized at interactive rates. Moreover, registration applications make even higher demands about the rendering speed of a direct volume rendering application, as the registration process consists of a iterative search for the optimal spatial alignment between two data sets. Therefore, the more views of a volume can be rendered in a certain amount of time the faster a registration application finishes the registration process.

Therefore, many researchers have worked independently on accelerating direct volume rendering techniques over the years. Briefer visualization times were achieved by developing new rendering techniques or by refining the existing ones. For example, shear-warp [40] was one of the first direct volume rendering techniques that was able to visualize a volumetric data set in decent times. Another technique offering decent times is *volume rendering using transgraph* [42]. Examples for refining existing rendering techniques include *empty space skipping* [90] or *early ray termination*. However, often these acceleration techniques come at a prize. Some generate images of lesser quality, like shear-warp, while others do not allow for arbitrarily modifying all viewing parameters, like volume rendering using transgraph.

Another approach that has been explored for accelerating direct volume rendering, was the development of dedicated volume rendering hardware, like the Mitsubishi VolumePro

[70] or the Silicon Graphics, inc. Visualization Systems [79]. However, such dedicated hardware for volume rendering is usually very expensive, thus only a few institutions can afford these. On the contrary, commodity graphics accelerator cost only a fraction. Additionally, due to the huge demand for high-performance 3-dimensional computer graphics generated by computer games, by now consumer graphics hardware has evolved to a point, that such hardware not only rival, but in many areas even surpass dedicated graphics workstations from just a couple of years ago. Current state-of-the-art commodity graphics accelerators like the NVIDIA GeForce 6800 Ultra [18], or the ATI Radeon X850 XT PLATINUM EDITION [10], offer a level of programmability and performance that not only makes it possible to execute traditional workstation applications on a cheap personal computer, but even enables the use of rendering algorithms that previously could not be employed for real-time graphics at all. Although, these graphics accelerators were purely designed for accelerating the drawing of polygonal scene descriptions, their increasing feature set and programmability allows for high-quality volume rendering, for instance with respect to the application of transfer functions, shading, and filtering. In spite of the tremendous requirements imposed by the sheer amount of data, a volumetric data set is constituted of, the rendering performance, the quality, and flexibility that can be achieved is astonishing.

1.2 Outline of this thesis

In the following chapter I will present a more detailed view about volume rendering in general. The next three chapters deal with volume rendering applications on consumer graphics hardware. First, I cover the basics about the mode of operation of such hardware as well as some of their inherited characteristics in chapter 3. Then different approaches for a volume rendering application based on the texturing capabilities of off-the-shelf graphics hardware are described in theory. The last section of chapter 3 concludes the topic by illustrating details about an actual implementation of the volume rendering techniques described earlier. The subsequent chapters shift the focus to volume rendering techniques that do not require any dedicated hardware, i.e that can be executed directly on a host CPU. Therefore, chapter 4 gives an overview over existing CPU-based volume rendering applications. Additionally, We will present a novel approach for accelerating traditional volume ray-casting. Details about an actual implementation can also be found in chapter 4. The last chapter of this diploma thesis deals with an evaluation of the performance of the presented techniques for direct volume rendering.

2 An Introduction to Volume Rendering

The term *volume rendering* comprises a set of techniques for rendering discrete three-dimensional, i.e volumetric, data sets. With respect to medical imaging, such data can be acquired from different sources, such as computed tomography, magnetic resonance imaging, ultrasound, or positron emission tomography.

Prior to the introduction of volume rendering, each 2-dimensional subset had to be visualized and interpreted separately. Although visualizing a volumetric data set as 3-dimensional entities is not an easy task, it is both worthwhile and rewarding. To summarize succinctly, volume rendering is a very powerful way for visualizing such 3-dimensional scalar fields. It also helps in the interpretation of the contained data values.

2.1 Volumetric Data Sets

In contrast to a surface data set, that is inherently 2-dimensional, even though surfaces are often embedded in a spatial domain of three dimensions, a volumetric data set is comprised of a 3-dimensional scalar field. In principle, its values are defined over a continuous 3-dimensional domain:

$$f(\vec{x}) \in R \quad \text{with} \quad \vec{x} \in R^3$$

With respect to volume rendering, a volumetric data set is stored as a 3-dimensional array of scalar values, i.e each of its values is stored at discrete locations. Its values are obtained by sampling the continuous 3-dimensional domain at these discrete locations. In analogy to the term pixel, denoting a single picture element of a discrete 2-dimensional scalar field, i.e image, the volume elements constituting the sampled volume are referred to as *voxels*.
It is convenient to think of a volume as being constituted of a collection of equisized cubes, where each individual sampling point is assumed to lie in the respective centers of these cubes. Although this aids visualizing the immediate vicinity of individual voxels, it is more accurate to identify each individual voxel with a sample obtained at an infinitesimally small point in R^3. Both models are depicted in figure 2.1.

In the later model (figure 2.1 (2)), it is understandable that the volumetric function is only defined at the exact sampling locations. Therefore, the most important task of volume rendering is to re-attain a continuous function that is defined for all points in R^3 in order to fully represent the original data. The process of attaining a continuous function from a collection of discrete samples is known as function or signal *reconstruction* [66, 73].

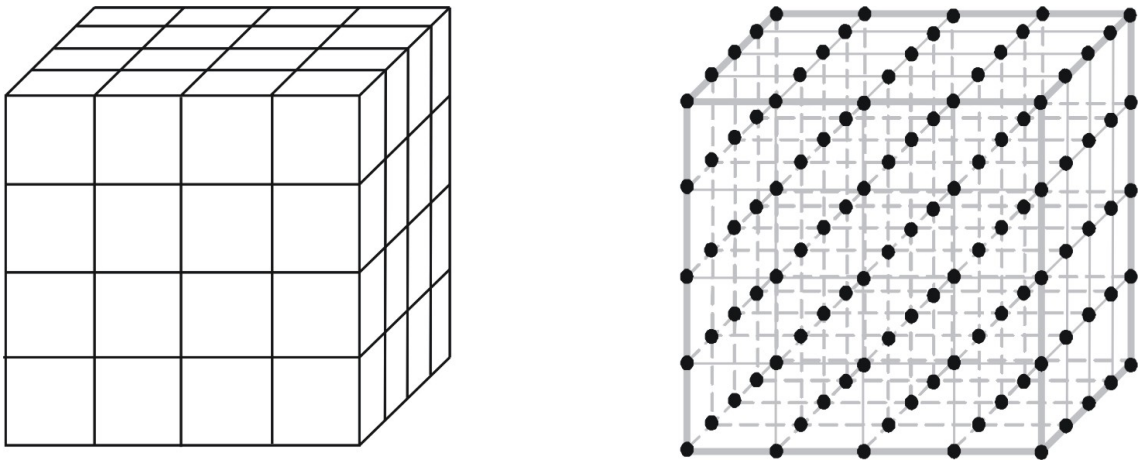


Figure 2.1: Two different models of a volumetric data set: (1) Comprised of a collection of equisized cubes each centered around an individual sampling point. (2) Comprised of a collection of scalar values each sampled at an infinitesimally small sampling points.

2.2 Sampling and Reconstruction

Storing the representation of a continuous function within a digital computer, the continuous representation has to be converted to a discretized representation by *sampling* the function at certain, usually equispaced, sampling locations [66, 73]. To yield a good representation these discrete sampling locations have to be uniformly distributed through the entire continuous domain, or at least through the entire domain of interest. Additionally to the discretization by sampling the function at certain locations, it is also necessary to quantize the resulting values in order to map continuous scalar values to quantities that can be represented as a discrete value. The sampled and quantized values are then stored in either integer, fixed-point, or floating point format.

Sampling a continuous function at discrete locations converts this function to a discrete function, that is only defined at the exact sampling locations. In order to obtain values of the corresponding continuous function, i.e to treat the discrete function as being a continuous one, the continuous function has to be constructed or reconstructed from the discretized values. This process is therefore referred to as *reconstruction* of a continuous function from a discrete one [66].

Reconstruction is done by applying a certain reconstruction filter to the discrete function. Thereby the filter kernel, i.e the function describing the filter, is convoluted with the discrete function. Over the years many different filters have been proposed. The simplest such filter is the box filter (figure 2.2 (1)). Applying a *box filter* to the discrete function's values for reconstruction results in nearest-neighbor interpolation between these values. To achieve the commonly used linear interpolation, a *tent filter* (figure 2.2 (2)) has to be convoluted with the discretized function. As both filters can easily be implemented and both offer fast reconstruction, both are widely supported by commodity graphics hardware.

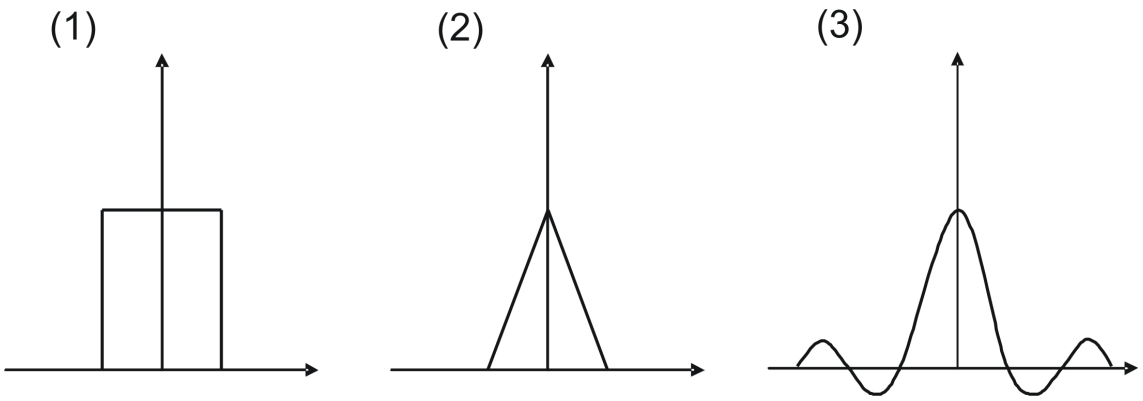


Figure 2.2: Some examples of different reconstruction filters: (1) The box filter resulting in nearest-neighbor interpolation, (2) the tent filter resulting in linear interpolation, and (3) the sinc filter resulting in "perfect"reconstruction

Sampling theory states that a continuous function can be reconstructed from its discretized representation in its entirety, if certain conditions are warranted during the sampling process of the continuous function. In order to uniquely reconstruct a continuous function from its sampled representation, its original frequency spectrum and the original spectrum's replications introduced by the convolution of the continuous function with the comb function, i.e the function used to determine the sampling locations, must not overlap. The distance between two adjacent replications in the frequency spectrum is $\frac{2\pi}{\omega}$, where ω is the distance between two adjacent sampling locations used for sampling. Therefore, all components of the original spectrum for frequencies equal or larger than $\frac{\pi}{\omega}$ must be equal to zero, i.e the continuous must not contain any frequencies above a certain threshold s_0. Functions fulfilling this requirement are referred to as being *band-limited* or s_0*-frequency-limited*. Additionally, the continuous function has to be sampled at different sampling locations with a frequency larger than twice the threshold, i.e larger than $2s_0$. This observation is also referred to as *Shannon's theorem*, whereas the critical sampling frequency is referred to as the *Nyquist frequency*. To enforce the requirement for a band-limited function, a low-pass filter can be applied to the continuous function prior to sampling. This low-pass filter must be chosen appropriately, so that it completely removes frequencies above the desired Nyquist frequency. Otherwise higher frequencies would result in aliasing as they would be interpreted as much lower frequencies during reconstruction due to overlapping in the discretized function's frequency spectrum.

However, the statement that a continuous function can be reconstructed entirely from a discrete representation stays theoretical, since the reconstruction filter would have to be "perfect". The "perfect", i.e ideal reconstruction filter is known as the *sinc filter*. Its frequency spectrum is box-shaped, thus allowing the complete elimination of all replications of the continuous function's frequency spectrum. In the spatial or time domain a box-shaped frequency spectrum is described by:

$$sinc(x) \;=\; \frac{sin(\pi x)}{\pi x}$$

A graph of this function is depicted in figure 2.2 (3). It is obvious, that the sinc filter has an infinite extent, i.e the filter function is non-zero from $-\infty$ to $+\infty$. Therefore, as the sinc filter cannot be implemented in practice, a trade-off between the extents of a reconstruction filter, i.e the time it takes to evaluate the filter function, and the reconstruction quality must be found. Even if the sinc filter could be evaluated completely, quantization during the sampling process could still yield artifacts. A good but short overview about sampling theory can be found in [3], whereas more comprehensive overviews can be found in [66, 73]. An evaluation of reconstruction filters for volumerendering in general can be found in [50].

In hardware amenable rendering, linear interpolation, i.e using a tent filter, is usually considered to be a reasonable trade-off between rendering performance and reconstruction quality. High-quality filters, also referred to as cubic filters, like the family of cardinal splines [37] including both the Catmull-Rom Spline and the BC-Spline [57], are usually only employed when the filtering operation is done in software. However, it has recently been shown that high-quality reconstruction filters can indeed be used for rendering on today's consumer graphics accelerators through the employment of shader programs [28, 29].

2.3 Volume Rendering Techniques

Over the years many different avenues have been taken for visualizing 3-dimensional scalar fields, i.e volumetric data. Basically, all these different approaches can be classified using one of the following general techniques.

2.3.1 Indirect Volume Rendering

Indirect volume rendering techniques involve converting the volumetric data set to an intermediate representation that can easily be visualized. Usually, this intermediate representation is a surface representation of the volume consisting of a set of polygons that can be rendered using traditional techniques. The polygonal representation is extracted from the volume according to a certain value, called the *isovalue*, i.e if a value of the 3-dimensional scalar field is in line with an given *isovalue* a polygon is inserted into the surface representation of the volume. The resulting surface is referred to as the *isosurface* corresponding to a specific *isovalue*.

$$f_{Vol}(x, y, z) \; = \; p_{iso} \pm e$$

When the 3-dimensional scalar field is stored as a structured set of point samples, the most common technique for generating a given *isosurface* is to create an explicit polygonal representation for the surface using a technique such as Marching Cubes [49], or Projected Tetrahedra [78]. Ideally, the isovalue can be controlled interactively by the user of the volume rendering application. However, in order to be able to acquire a surface representation of the essential data stored inside a volume, the algorithm used to extract an isosurface from a volumetric data set has to act on the following assumptions. First, the isosurface must exist and, second, the polygonal mesh representation has to model the true object structures at reasonable fidelity with a infinitely thin surface. Unfortunately, neither is always the case.

But even if a surface representation can be generated, the complexity of the extracted polygonal mesh can overwhelm the capabilities of the underlying graphics subsystem. Therefore, a direct volume rendering technique my prove more efficient [68]. Especially, when the visualized object is quite complex or large, or when the isosurface's corresponding isovalue is interactively varied and the repeated polygonal extraction overhead must be figured into the rendering performance evaluation [12].

2.3.2 Direct Volume Rendering

Direct volume rendering techniques render images of an entire 3-dimensional scalar, i.e a volume, without concentrating on, or explicitly extracting an surface corresponding to certain features of interest or a certain isovalue. In order to directly display the data stored in a volumetric data set a optical model is needed that describes how the scalars representing the volume interact with light, e.g. emission, absorption, reflection, or refraction [54].

In general, each scalar value contained in the volumetric data set is mapped to optical properties, like color and opacity, during rendering. Mapping scalar values to optical properties is usually achieved by evaluating a *transfer function* for each occurring sample value of the volume. The application of mapping scalar values to optical properties is also called *classification*. In general, a distinction is drawn between two types of classification. *Pre-classification* is done prior to reconstruction of the volumetric data set, whereas *post-classification* evaluates the transfer function on each reconstructed value.

The optical properties, and their optical effects respectively, are then integrated along viewing rays into the volume. In order to generate a projected image of a volumetric data set, many viewing rays have to be integrated with each ray corresponding to a pixel of the final image. This integral is also referred to as the *volume rendering integral*. The volume rendering integral is contingent upon the underlying optical model. Many different techniques have been developed to approximately solve the volume rendering integral [56]. Section 2.4 describes the volume rendering integral in more detail.

2.3.3 Maximum Intensity Projection

Maximum Intensity Projection, MIP for short, is a variant of direct volume rendering. Instead of compositing optical properties, Maximum Intensity Projection determines the final color of each pixel in the final rendering by the maximum value encountered during traversing of each corresponding ray. Visualizing medical 3-dimensional data sets obtained by a Magnetic Resonance Imaging, MRI for short, scanner is an important application of such a rendering mode. Such volumetric data sets typically exhibit a significant amount of noise that can make it hard use other rendering modes, as it is difficult to extract meaningful isosurfaces, or define a transfer function that aid the interpretation of the data set. However, data values of vascular structures acquired by MRI scanners are higher than the values of the surrounding tissue. This can easily be exploited by Maximum Intensity Projection volume rendering for visualizing such medical data. Figure 2.3 shows the a comparison of direct volume rendering and Maximum Intensity Projection used with the same data set.

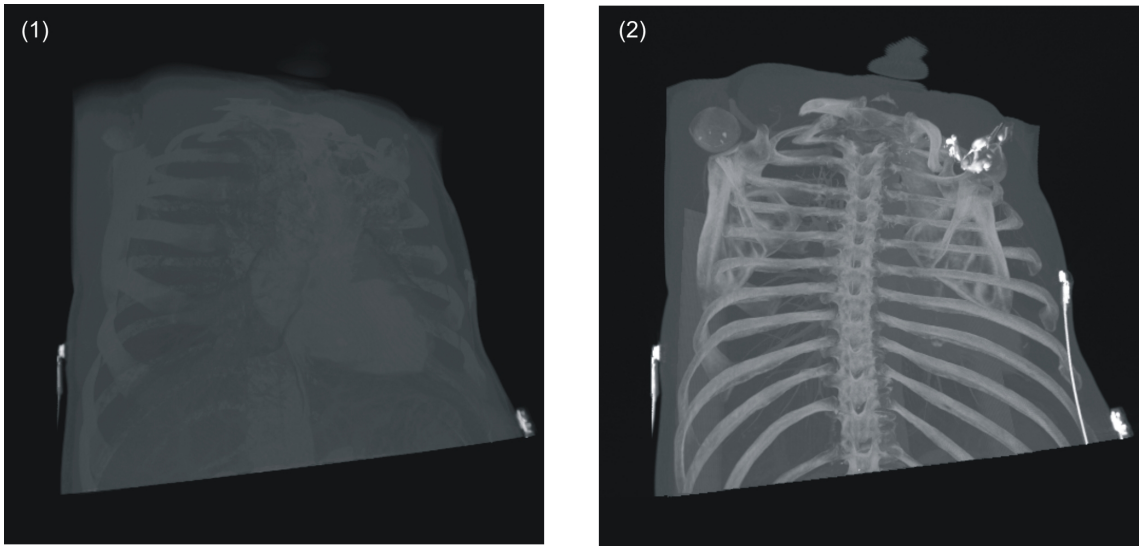


Figure 2.3: A comparison of direct volume rendering (1), and Maximum Intensity Projection (2) on the same data set.

2.3.4 Non-Polygonal Isosurfaces

In the context of volume rendering, the term *isosurface* stands for a contour surface extracted from a volumetric data set that corresponds to a certain feature of interest. Usually, this feature of interest is described by a specific value, also referred to as the corresponding *isovalue*. The term isosurfaces is also used to summarize boundary surfaces of regions of the volumetric data that are homogeneous with respect to certain attributes. For example, an isosurface could depict a region where the density of the volume is above or below a given threshold.

Usually an isosurface is constituted by a set of polygons characterizing an explicit surface. These polygons are usually extracted prior to actually displaying them in a preprocess. As section 2.3.1 summarizes, many different algorithms have been proposed for extracting isosurfaces from a given volumetric data set. For example, a variant of the Marching Cubes [49] algorithm is used to process the volume in order to generate an explicit geometric representation for the feature of interest, i.e a given isovalue. This geometric representation typically consists of thousands of polygons. It is clear, that the corresponding isosurface has to be recomputed each time the isovalue the extraction was based on changes during rendering. Rendering isosurfaces is the method of choice for indirect volume rendering applications.

However, isosurfaces can also be rendered using direct volume rendering techniques as the presence of explicit geometry is not a prerequisite. In this case, an isosurface is also referred to as a *non-polygonal isosurface*. A software based approach for displaying non-polygonal isosurfaces, is to use ray-casting (section 2.5) with special transfer functions [46]. The principle of a hardware amenable approach can be found in section 3.3.7.

2.4 The Volume Rendering Integral

With respect to volume visualization, direct volume rendering techniques produce projected images directly from the volumetric data set. These techniques do not involve construction of intermediate representations, such as a polygonal surface, of the data stored in the volume. Therefore, direct volume rendering techniques require some model of how the scalar values generate, reflect, scatter, or occlude light. As the volumetric data set is usually obtained by sampling a function regularly or irregularly at discrete sampling locations, it has to be interpolated to be used with one of the continuous optical models described here. A comparison of different interpolation techniques for volumetric data sets can be found in [61]. Here the interpolation is done somehow to give a scalar function $f(\vec{x})$ defined for all points $\vec{x}$ in the volume. Optical properties, such as color and opacity, can then be assigned as functions of the interpolated value $f(\vec{x})$.

Usually, optical models used for real-time rendering of volumetric data, that deliver a visual compelling visualization of the volume, are quite simplistic, as they do not consider effects like single or multiple scattering. However, real-time methods taking such effects into account are currently becoming available [31]. In each optical model a ray denoted by $\vec{x}(t)$ and parameterized by the distance to the eye t is cast into the volume. A brief survey of the most important optical models for direct volume rendering is given here, whereas a more extensive survey can be found in [54]:

Absorption only: The simplest participating medium has cold, perfectly black particles, which absorb all light that impinges on them. None of the particles emits or scatters light. The rate at which incoming light is occluded depending on the value of the particle is called the *extinction coefficient* or *absorption coefficient* $\tau(\vec{x}(t))$. Therefore, the *transparency* of the volume along the ray between 0 and D is

$$T(D) \;=\; e^{-\int_0^D \tau(f(\vec{x}(t)))dt} \tag{2.1}$$

The opacity α of the volume is

$$\alpha(D) \;=\; 1 - T(D)$$

Emission only: The volume is assumed to consist of particles that only emit light, but do not absorb any, as the absorption is negligible. The emitted light's *intensity* of the volumetric data set based on the emissive color values $c(\vec{x}(t))$ of each particle on a ray between 0 and D is

$$I(D) \;=\; \int_0^D c(f(\vec{x}(t)))dt \tag{2.2}$$

It is noteworthy that this integral does not have an upper bound, as the intensity can be added across an arbitrary thickness without attenuation.

Absorption plus Emission: In general, particles both occlude incoming light as well as add their own glow. Thus a realistic equation for computing the intensity along a ray

should include both source $I(D)$ (eqn. 2.2) and attenuation $T(D)$ (eqn. 2.1) terms. That way, the intensity along ray through the volume between 0 and D is

$$I(D) \;=\; \int_0^D c(f(\vec{x}(t)))\; e^{-\int_0^t \tau(f(\vec{x}(s)))ds}\; dt \tag{2.3}$$

This optical model is commonly used in volume rendering applications as it mimics the light transport most realistically compared to the other simplistic approaches. However, this is also a rather simplistic model since it does not take optical effects, such as single or multiple scattering, i.e more sophisticated interaction between light and particles contained in a volume, into account.

Scattering and Shading: The next step toward greater realism is to include scattering of illumination external to a particle in the volume, i.e indirect illumination. The *"Utah approximation"* model, a simplified model, assumes external illumination reaches a particle unimpeded by any intervening objects or volume absorption. The most general source term $c(\vec{x}(t), \vec{\omega})$ is the sum of a non-directional internal emissivity $E(\vec{x}(t))$, as described in the emission only model, and the reflection or scattering term $S(\vec{x}(t), \vec{\omega})$. The scattering term is dependent on the incoming illumination $i(\vec{x}(t), \vec{\omega}')$ at position $\vec{x}(t)$ and the bidirectional reflection distribution function $r(\vec{x}(t), \vec{\omega}, \vec{\omega}')$. This is subsumed in

$$S(\vec{x}(t), \vec{\omega}) \;=\; r(\vec{x}(t), \vec{\omega}, \vec{\omega}')i(\vec{x}(t), \vec{\omega}') \tag{2.4}$$

where $\vec{\omega}$ denotes the direction of the reflected light, and $\vec{\omega}'$ denotes the direction of the incoming light. That way, the source or color term can be evaluated by

$$c(\vec{x}(t), \vec{\omega}) \;=\; E(\vec{x}(t)) + S(\vec{x}(t), \vec{\omega}) \tag{2.5}$$

Shadows: In order to take shadows into account, the transparency of the volume density between the light source and the point $\vec{x}(t)$, as well as from $\vec{x}(t)$ to the ray source, should be taken into account. Therefore, the incoming illumination $i(\vec{x}(t), \vec{\omega}')$ at position $\vec{x}(t)$ of the scattering term (eqn. 2.4) is denoted by

$$i(\vec{x}(t), \vec{\omega}') \;=\; L\, e^{-\int_0^\infty \tau(\vec{x}(t) - t\vec{\omega})' dt}, \tag{2.6}$$

where L is the intensity from an infinite light source in direction $-\vec{\omega}'$. In practice, this integral does not run to ∞, but only to the boundaries of the volume. [53, 52, 35, 62] show how single scattering with shadowing can be evaluated under particular conditions. A more general two-pass numerical algorithm can be found in [34].

Multiple Scattering: Multiple scattering calculations are important for realistic rendering of high albedo media but are expensive in performance and are, usually, overkill for most scientific visualization applications. Multiple scattering involves directionally dependent light transport, thus making it necessary to find $I(\vec{x}(t), \vec{\omega})$, the intensity of each location $\vec{x}(t)$ in each light flow direction $\vec{\omega}$. Therefore, the color term has to be adjusted to take into account the scattered intensity from all possible incoming light directions $\vec{\omega}'$ on the 4π unit sphere. This is subsumed in

$$c(\vec{x}(t), \vec{\omega}) \;=\; \int_{4\pi} r(\vec{x}(t) - t\vec{\omega}, \vec{\omega}, \vec{\omega}')I(\vec{x}(t) - t\vec{\omega}, \vec{\omega}')d\vec{\omega}' \tag{2.7}$$

A good trade-off between rendering performance and a visual appealing visualization of a volumetric data set can be achieved when the absorption plus emission optical model is used. The term used to evaluate light transport in this model is often referred to as the *volume rendering integral*. The evaluation of this term, that integrates optical effects such as color and opacity, is common to all direct volume rendering techniques. In general, the evaluation of the volume rendering integral can be thought of as being evaluated along viewing rays cast into the volumetric data set, even if no explicit rays are actually employed by the underlying volume rendering technique. Therefore, ray casting described in section 2.5 could be seen as the most direct approach.

It is noteworthy that the volume rendering integral assumes both the volume and the mapping to optical properties to be continuous. However, in practice, of course, the evaluation of the volume rendering integral is done numerically. Therefore, together with additional approximations, the integral operation becomes a simple summation. As the volumetric data set itself is constituted by a collection of samples at discrete locations, a reconstruction has to be performed in order to get the corresponding continuous volume. This reconstruction is also only an approximation, and thus further impairs the evaluation of the volume rendering integral. The volume rendering integral is again denoted in figure 2.4.

$$I(D) \;=\; \int_0^D c(f(\vec{x}(t)))\; e^{-\int_0^t \tau(f(\vec{x}(s)))ds}\; dt$$

Figure 2.4: The volume rendering integral.

In order to obtain he intensity value of the color at a specific pixel on the image plane, a ray corresponding to the pixel's location is cast into the volumetric data set. Then the integration is performed along each ray cast into the volume. It is sufficient if the volume rendering integral is evaluated from the ray source to the point where the ray exits the volume. This happens after a certain distance D, where $t = D$. In the absorption plus emission optical model, the intensity at a location $\vec{x}(t)$ is constituted by the intensity emitted there, $c(f(\vec{x}(t)))$, multiplied by the cumulative, i.e integrated, absorption up to the position of emission, $e^{-\int_0^t \tau(f(\vec{x}(s)))ds}$.
It is important to note that the position of the interpolation and shading operator $f(\vec{x}(t))$ can affect the final color and opacity in the rendered image. Therefore, classification, i.e evaluation of the transfer function, is commonly distinguished between *pre-classification* and *post-classification*. In the case of pre-classification, the samples stored inside the volumetric data set are classified, i.e color and opacity values are assigned to each sample based on their values, *before* the interpolation along the ray takes place. One disturbing feature of pre-classification is that it tends to excessive blurring when the rendered image resolution exceeds that of the volumetric data set. This can occur in zoomed viewing or at wide perspectives [33, 60]. This blurriness can be eliminated, though, by switching the order of classification and ray resampling interpolation. Then, the original density volume is interpolated using the interpolation operator $f(\vec{x}(t))$ and the resulting sample values are passed to a transfer function for classification. The order of operations employed here, interpolation prior to classification, is commonly referred to as post-classification. Therefore, fine details

in the transfer function are readily expressed in the final image. However, post-classification is not without drawbacks: It is possible that a density is classified as a certain material that is not really present in the volume, due to the interpolation of densities.
When the color term of the volume rendering integral is constant, i.e the emission of light is neglected and only the absorption is considered, the result is a x-ray like image. This x-ray image accounts for the attenuation of the x-rays from the source by the density of the volumetric data set between the source and the film plane.

2.5 Ray-Casting - A Numerical Approximation of the Volume Rendering Integral

Ray-casting [46] can be seen as the the most straight-forward approach for numerical evaluation of the volume rendering integral (figure 2.4). Therefore, ray-casting is considered a direct volume rendering technique. But ray-casting can also be used for rendering non-polygonal isosurfaces (section 2.3.4) as well as for Maximum Intensity Projection (section 2.3.3).

For each pixel in the final image, a single ray is cast from the camera center through this pixel into the volume. The volume is then resampled at certain intervals along a particular ray. The distance between two adjacent resampling locations is commonly referred to as the sampling distance. Usually, the sampling distance is equidistant along each ray, but some methods have been proposed that use non-equal distance between two adjacent sampling locations. For example, one technique jitters the sampling locations to eliminate patterned sampling artifacts [19], whereas another technique increases the sampling distance in order to efficiently skip empty regions of the volumetric data set [90]. The sampling, i.e the reconstruction, of the volumetric data set is commonly done using trilinear interpolation. However, lower-order reconstruction filters, like nearest-neighbor, or higher-order reconstruction filters, like tri-cubic, can also be employed. After resampling, the interpolated scalar value is mapped to optical properties by evaluating the current transfer function for this value. This can be done efficiently, when precomputing the transfer function and storing the results in a lookup table. Instead of evaluating the transfer function for each scalar value, the scalar value is used as an index to access the precomputed values stored in the corresponding lookup table. Both methods, evaluating the transfer function on-the-fly or indexing into the lookup table, yield an RGBA color value for the corresponding location inside the volume that subsumes the respective emission and absorption coefficients [46]. The volume rendering integral is numerically approximated numerically by compositing these values in either front-to-back or back-to-front order. This process is illustrated in figure 2.5.

As the volume rendering integral can not be computed analytically, in general, ray-casting and other direct volume rendering algorithms discretize the volume rendering integral into a series of sequential intervals. First, the cumulative absorption up to a certain position $\vec{x}(t)$ along a single ray, from equation 2.3 or figure 2.4,

$$e^{-\int_0^t \tau(f(\vec{x}(s)))ds} \tag{2.8}$$

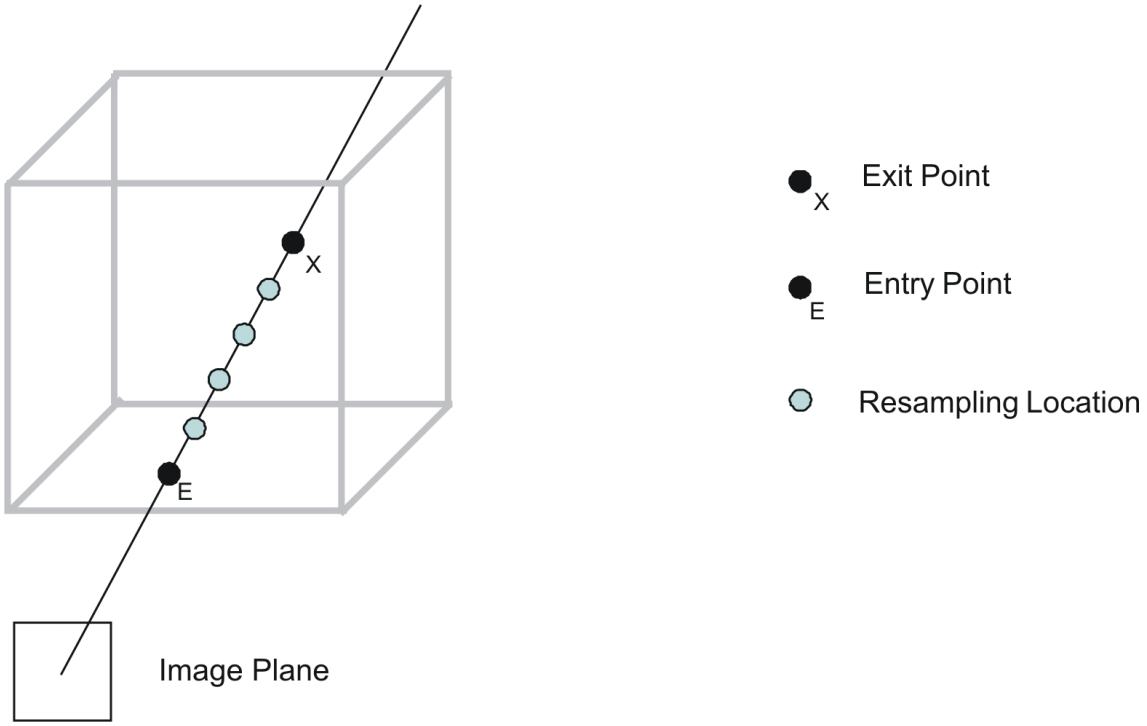


Figure 2.5: Ray-Casting steps.

can be approximated by

$$e^{-\sum_{i=0}^{\lfloor t/\triangle d \rfloor} \tau(f(\vec{x}(i\triangle d)))\triangle d}, \tag{2.9}$$

where $\triangle d$ denotes the distance between two successive resampling locations along each ray. The summation in the exponent can immediately be substituted by a multiplication of the exponentiation terms:

$$\prod_{i=0}^{\lfloor t/\triangle d \rfloor} e^{-\tau(f(\vec{x}(i\triangle d)))\triangle d} \tag{2.10}$$

As the exponent of the absorption term computes the transparency of a given resampling location i, the corresponding opacity α_i of this sample can be computed by

$$\alpha_i \;=\; 1 \;-\; e^{-\tau(f(\vec{x}(i\triangle d)))\triangle d} \tag{2.11}$$

With respect to this definition, equation 2.10 can be rewritten as

$$\prod_{i=0}^{\lfloor t/\triangle d \rfloor} (1 \;-\; \alpha_i) \tag{2.12}$$

Basically, α_i can as well be seen as an approximation for the absorption of the ith ray segment, instead of the absorption at a single point.

Similarly, the source term or color term, i.e the emission coefficient of the volume rendering integral, can be approximated by

$$\sum_{i=0}^{\lfloor D/\triangle d \rfloor} c(f(\vec{x}(i\triangle d)))\triangle d \,, \tag{2.13}$$

where $\triangle d$ denotes the distance between two successive resampling points. Instead of computing the emissive intensity for each resampling location seperately, the emissive intensity for the ith ray segment can be computed by

$$C_i \;=\; c(f(\vec{x}(i\triangle d)))\triangle d \tag{2.14}$$

Therefore, equation 2.13 can be rewritten as

$$\sum_{i=0}^{\lfloor D/\triangle d \rfloor} C_i \tag{2.15}$$

Using the approximation of both, the emission term (equation 2.15) and the absorption term (equation 2.12), the volume rendering integral shown in figure 2.4 can be evaluated using the following discretized approximation:

$$\sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 \;-\; \alpha_i) \,, \tag{2.16}$$

where n denotes the number of resampling locations specified by $n = \lfloor D/\triangle d \rfloor$. This equation is also referred to as the discretized volume rendering equation. It can be evaluated iteratively by compositing the color and opacity values in either front-to-back or back-to-front order.

In order to evaluate the discretized volume rendering equation in back-to-front order, equation 2.16 can be rewritten iteratively:

$$C_i^{accum} \;=\; C_i^{cur}\alpha_i^{cur} + (1 - \alpha_i^{cur})C_{i+1}^{accum}, \tag{2.17}$$

The stepping is done from $n-1$ to 0, i.e in back-to-front order. In each step a new accumulated value C_i^{accum} is calculated from the color C_i^{cur} and opacity α_i^{cur} at the current resampling position i along the ray, and the already accumulated color C_{i+1}^{accum} from the previous resampling location $i+1$. The starting condition is $C_n^{accum} = 0$.

Front-to-back evaluation of the discretized volume rendering integral can be done iteratively as well:

$$C_i^{accum} = C_{i-1}^{accum}\alpha_{i-1}^{accum} + (1 - \alpha_{i-1}^{accum})C_i^{cur}\alpha_i^{cur} \quad (2.18)$$

$$\alpha_i^{accum} = \alpha_{i-1}^{accum} + (1 - \alpha_{i-1}^{accum})\alpha_i^{cur} \quad (2.19)$$

It is important to note, that front-to-back compositing of color and opacity values requires separate tracking of opacity values, whereas back-to-front compositing does not. However, front-to-back rendering usually allows for an optimization commonly called *early ray termination*, where the progression along a ray is terminated when the accumulated opacity value does not allow for further accumulation, i.e the accumulated opacity value reaches 1.

3 Texture-based Techniques for Volume Rendering

3.1 Modern Graphics Hardware

Over the past few years the field of computer graphics hardware has been advancing at incredible rates. The sustained desire of the ever-growing gaming community to be stimulated and entertained visually, i.e the quest for achieving visual realism, is one of the driving forces behind this rapid pace of innovation. Achieving visual realism not only requires increasing the raw computing power of graphic accelerators but also the ability to compute more complex visual effects. Therefore hardware manufactures have included better performance and more hardware supported features as well as evolving programmability in their products with each generation.
In connection with this the two major 3D programming interfaces, OpenGL and Direct3D, are also continually evolving. Direct3D is part of DirectX a set of multimedia programming interfaces developed and maintained by Microsoft. OpenGL developed by Silicon Graphics is an open standard for 3D programming. It is available for Windows, UNIX, and MacOS whereas Direct3D is only available for Windows personal computers. Each new iteration of these APIs influences and incorporates the functionality and design of the computer graphics hardware.
Prior to the introduction of GPUs (short for Graphics Processing Unit), a term introduced by Nvidia in the late 1990s, graphic systems consisted of multiple chips that worked together to render images and to display them on a screen. Therefore graphic systems were quite expensive and not mass-market compatible but they already introduced many of the concepts, like vertex transformation and texture mapping, that we take for granted today. The *first generation* of GPUs (e.g. Nvidia's TNT2, ATI's Rage) were capable of rasterizing pre-transformed triangles, i.e the transformation of vertex data was part of the CPU's tasks, and applying one or two textures. The computation of vertex transformation and lighting (T&L) was offloaded from the CPU to the GPU by *second generation* graphic adapters (e.g. Nvidia's GeForce 256, ATI's Radeon 7500). Prior to that, fast vertex transformation was one of the key capabilities that differentiated high-end workstations from personal computers. Although the capabilities for coloring pixels and combining textures were expanded, these GPUs were merely more configurable not truly programmable. True programmability or at least partial programmability of the rendering pipeline was introduced roughly a year later. These *third generation* graphic accelerators (e.g. Nvidia's GeForce3, ATI's Radeon 8500, Microsoft's XBOX GPU) let the application specify a sequence of instructions for processing vertex data, thus offering vertex programmability. Still the processing of data at a pixel-level only allowed for more configurability rather than programmability. This remaining programmability issue has been addressed by *fourth generation* GPUs (e.g. Nvidia's GeForce FX, ATI's Radeon 9700). These GPUs provide both true vertex-level and pixel-level program-

mability, though the underlying programming model is quite simplistic. A good overview about the different generations of GPUs can be found in [24]. Current GPUs further improve upon these developments extending the programming model to include dynamic branching, i.e allowing true loops and conditionals. Today GPUs have reached the point of having five or more times as many transistors as the fastest consumer-level CPU by far exceeding Moore's Law. They also execute more than seven times as many floating point operations per second, while using an internal memory bandwidth of over 35 GB/s. Graphics accelerators will not stop there as their development is still one of the most dynamic segments of the semiconductor industry due to the ever-increasing demands on delivering visual realism in real-time graphics. In order for a programmer to harness the full potential of current GPUs the 3D programming interfaces are constantly advancing as well. Therefore programmers have recently been able to employ the capabilities of modern GPUs in other kinds of fields than generating images from 3-dimensional data, like computing database operations [26], solving sets of linear equations [39], audio processing [13], flow visualization [85], segmentation [76], Fast-Fourier-Transformation computation [58], or robot motion planning [45]. For a good overview and additional topics see [30].

Gen.	Year	Product	Multi-Texturing Fill Rate	Vertex Throughput	Memory Bandwidth	Notes
1^{st}	1998	Riva TNT	$50Mp/s$	$6Mv/s$		dual texture
2^{nd}	1999	GeForce 256	$480Mp/s$	$15Mv/s$	$4578MB/s$	fixed-function vertex processing, register combiners
3^{rd}	2001	GeForce3	$1920Mp/s$	$54Mv/s$	$7629MB/s$	vertex programs, quad-texturing, texture shaders
4^{th}	2003	GeForce FX	$3800Mp/s$	$356Mv/s$	$28992MB/s$	vertex and fragment programmability, floating-point pixels
5^{th}	2004	GeForce 6	$7200Mp/s$	$675Mv/s$	$36621MB/s$	Shader Model 3.0, fp16 blending, hdr

Table 3.1: Features and Performance Evolution of Selected NVIDIA GPUs by Generation

3.1.1 The Rendering Pipeline

A *pipeline* consists of a sequence of stages operating in parallel and in a fixed order. Each single stage receives data from its prior stage as input and sends its output to the next stage in the sequence after it is finished with its work. The rendering of virtual scenes on today's graphic hardware is implemented in such a pipelined fashion. The order of operations necessary for turning the geometry of a virtual scene into pixels that can be displayed on a screen or a portion of it, i.e a window, is called the *rendering pipeline* and used to be implemented in a fixed way. This order is illustrated in figure 3.1. As stated earlier recent developments have led to replacing the fixed function pipeline,specifically parts thereof, with a programmable one. A detailed description about the programmable parts of the rendering pipeline can be

found in section 3.1.2.
Input to the *rendering pipeline* is a stream of vertices describing the scene that can be joined to form geometric primitives, typically lines, triangles, quads, or polygons. It then computes a raster image of the virtual scene. The *rendering pipeline* can roughly be divided into five different stages (for a more detailed description of the following stages see [65], [8], [24], and [27]):

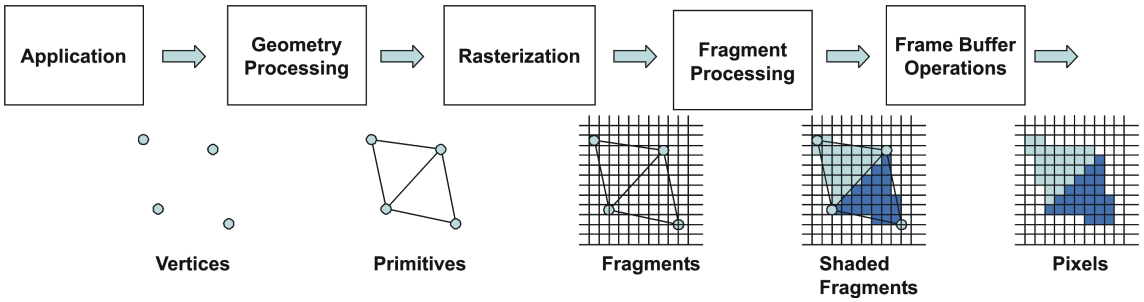


Figure 3.1: The fixed-function rendering pipeline

3.1.1.1 Application

Since the applications stage always executes in software completely a developer has full control over what happens. Therefore changing the implementation may yield differences in the performance of an application. The application takes care about the user input, the user interface, feedback handling, or any other computations not computed in other stages. Aside from rendering images its main function is to feed the subsequent stage with a stream of vertex data by calling functions defined by the underlying 3D programming interface.

3.1.1.2 Geometry Processing

The *geometry processing stage's* or simply the *geometry stage's* tasks include the majority of the per-geometric primitive or per-vertex operations, i.e operations responsible for modifying the incoming stream of vertex data. The geometry engine part of a modern GPU computes linear transformations, projective transformations, and evaluates local illumination models on a per-vertex basis. Therefore this part of the GPU is often referred to as *transform & lightning* (T&L). As figure 3.2 illustrates this stage can be further subdivided into several functional stages.

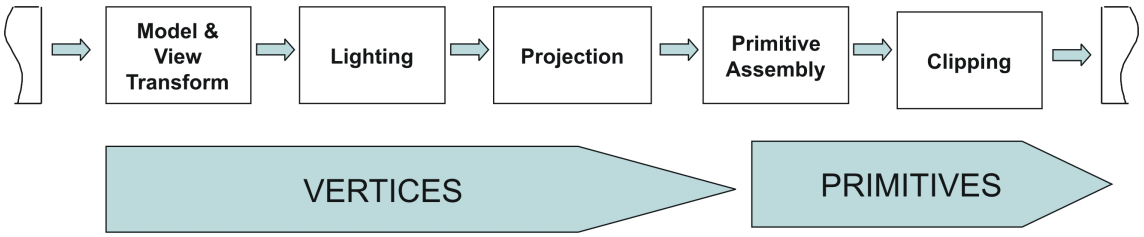


Figure 3.2: Geometry Processing Stage

Model and View Transform On its way to the screen, all models are transformed from their own model-coordinate space to world-coordinate space by the model transform. The purpose of the view transform is to place the camera origin at a specific location and aim it in a certain direction dependent on the underlying 3D application programming interface to facilitate subsequent projection and clipping operations. After a vertex has been transformed to this so-called view-coordinate space or eye-coordinate space it is sent to the next stage of the geometry stage. Model and view transformations are specified as as a 4×4 matrix using homogeneous coordinates usually concatenated into a single matrix, often called the modelview matrix, for performance reasons.

Lighting After all model and view transformations have been applied a local illumination model, e.g. Phong Lighting [71], is evaluated for each vertex. As this requires information about normal vectors and viewing direction it has to be computed after the model and view transformations. It also requires that all objects, i.e all models, the camera, and all light sources, reside in the same coordinate space, thus requiring a transformation of a scene's light sources to eye-coordinate space as well. This is made possible because all relative relationships, such as normals and distances, between light sources and models are preserved by the linear transformations from model-coordinate space to eye-coordinate space, for models, and from world-coordinate space to eye-coordinate space, for light sources.

Projection Next, geometric processing engines perform a perspective transformation, transforming the view volume, also called the frustum, into usually a unit cube with its extreme points at $(-1; -1; -1)$ and $(1; 1; 1)$, called the canonical view volume. Still it is considered to be a projection because after display the z-coordinate is not stored in the image generated. There are two different types of projections, namely orthographic projection and perspective projection. After either projection, vertex coordinates are stored in normalized device coordinates.

Primitive Assembly Geometric primitives are generated from the incoming stream of transformed vertices. Vertices are connected to lines and lines are joined together to form triangles. Arbitrary polygons are usually considered to be tessellated into triangles to ensure planarity.

Clipping Primitives fully inside the viewing frustum are passed as is to the next stage of the pipeline. Primitives completely outside are discarded. Only primitives that are partially inside the viewing volume are subject to clipping against the unit cube. For example, for a line where only one vertex is considered to be inside a new vertex has to be generated where the line intersects the viewing volume while the vertex outside can be discarded for further processing.

3.1.1.3 Rasterization Stage

Rasterization is the process of determining the set of screen pixels covered by a geometric primitive. So the result of the rasterization are a set of pixel locations as well as a set of corresponding *fragments*. It is important to distinguish between pixels and fragments. *Pixels*, short for picture elements, are parts of the final image that is displayed on a screen. Instead one can think about a *fragment* as a "potential pixel". A fragment is the same size as a

pixel, is associated with the same screen location as the pixel it corresponds to, and has a set of parameters such as color, depth value, and one or more sets of associated texture coordinates. These parameters a generated by the rasterizer during fragment generation through interpolation of the corresponding vertices of the geometric primitive. Figure 3.1 illustrates this process. Whether a *fragment* becomes a real pixel or not is decided through various tests at the end of the rendering pipeline (see section 3.1.1.5).

3.1.1.4 Fragment Processing

This stage can further be divided into two different tasks as shown in figure 3.3. Once a geometric primitive has been rasterized into a set of zero or more fragments it enters either the texture fetching stage or the fragment shading stage. It may skip the texture fetching stage regarding by the current state that is set by the underlying 3D programing interface, i.e if no texture lookup is to be executed.

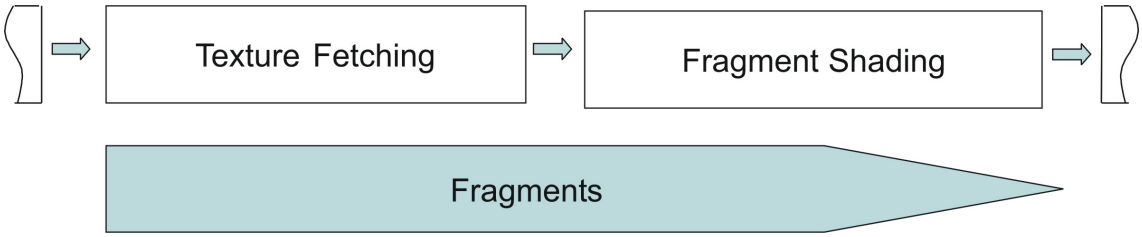


Figure 3.3: Fragment Processing Stage

Texture Fetching *Textures* are 1-dimensional or multi-dimensional images that can be "glued" to a 3-dimensional object. They are mapped onto geometric primitives in correspondence to the texture coordinates interpolated in the rasterization stage. This process yields an interpolated color value fetched from the texture. The order of interpolation is depending on the dimension of the texture target and the graphic hardware's capabilities. Current generation GPUs support the simultaneous fetching of multiple textures for each fragment without a hit in performance. Furthermore, these GPUs allow for enhanced controlling of the texture lookup itself. It is possible, for example, to use the color value returned by the first texture fetch as texture coordinates for consequent texture lookups. This is known as *dependent texturing*. Dependent texturing is important to implement different sorts of transfer functions for volume rendering. Other fragment attributes can be used as texture coordinates as well.

Fragment Shading The fragment shading stage applies further color operations on a given fragment to compute its final color. This stage is also capable of applying different math operations on a fragment's values. It may choose to change nearly every value of a fragment, e.g. the depth value, except for its screen location. Even allowing for the possibility that this stage may completely discard a fragment, thus preventing the fragment's corresponding screen pixel from being updated. The fragment shading stage emits one or zero completely colored fragments for each input fragment it receives.

3.1.1.5 Frame Buffer Operations

The frame buffer operations stage performs a set of per-fragment operations right before the fragment is turned into an actual pixel. The incoming fragment is at first checked based on number of different tests. If any of these tests fail the pixel operations stage immediately discards the specific fragment without updating its corresponding pixel's value stored in the frame buffer. All tests can be enabled or disabled by the programmer, though it is not possible to change either their order of sequence nor their functionality. If a fragment passes all the tests another set of operations is performed to update the values stored in the associated buffers. Thus the fragment has finally advanced to being a pixel. These operations can also be enabled or disabled The sequence of frame buffer operations is illustrated in figure 3.4.

Figure 3.4: Frame Buffer Operations

Scissor Test The scissor test is used to restrict drawing of pixels to a rectangular portion of the frame buffer. If a fragment lies inside this rectangle it is further processed by the subsequent tests.

Alpha Test The alpha tests compares the incoming fragment's opacity, i.e its alpha value, with a reference value. The fragment is accepted or rejected based on the outcome of this comparison.

Stencil Test The stencil test is typically used to mask out an irregularly shaped region of the frame buffer to prevent drawing from occurring within it. The pixel locations drawing is allowed or rejected on values stored in the *stencil buffer*, that is part of the actual frame buffer. Therefore it resemblances the frame buffer in width and height. The stencil buffer is essential to the application of the stencil test, without it every fragment passes the stencil test automatically. The stencil test itself involves a comparison of the fragment's associated pixel's stencil value stored in the stencil buffer with a reference value. Optionally this comparison can also take the associated pixel's depth value into account. If fragment passes the stencil test it may choose to update the value stored in the stencil buffer as well.

Depth Test The distance between the camera origin and an object, i.e the z-coordinate inside the view volume of an object, currently occupying a pixel location is stored in a specific buffer, namely the *depth buffer*. The depth buffer is also part of the frame buffer, therefore extending to the same dimensions as the frame buffer. The depth test decides whether a incoming fragment is occluded by a previously drawn pixel, by comparing the incoming fragment's depth value to the associated pixel location's depth value already stored in the depth buffer. If a fragment passes the depth test it may choose to update the depth buffer value with its own. The depth buffer together with the depth test therefore provide a convenient mechanism for depth ordering either partially or fully occluded objects on a per-fragment level.

Blending After a fragment has passed all the pixel tests its color values are then combined with the color values already stored in the frame buffer at the corresponding location. This combination is referred to as *blending*. Different blending operations can be applied, e.g. replacing or modulating depending on the stored alpha values, thus allowing for semi-transparent objects.

Dithering By dithering, color resolution can be improved at the expense of spatial resolution, on systems with only a small number of color bitplanes. If the hardware already has a high color resolution the enabling of dithering will end up doing nothing at all.

Logical Operations The final operation on a fragment is a logical operation, such as OR, XOR, and NEGATE. This operation is applied before the fragment is written to the frame buffer, thus becoming a pixel, to the incoming fragment's values and/or the values currently stored in frame buffer.

3.1.2 The Programmable Rendering Pipeline

Traditionally the *rendering pipeline* was implemented as a fixed-function sequence of stages as described in section 3.1.1. An application developer only had the choice to enable or disable certain stages, i.e functionalities, of the *rendering pipeline*, e.g. lighting or texturing. Additionally 3D programming interfaces along with the underlying hardware allowed for the possibility to set parameters for different stages, thus giving the programmer more configurability. But this made it nearly impossible for a programmer to implement other functionalities than what was implemented in hardware, such as the Gouraud Shading model [25], Phong's specular highlighting equation [71], or applying textures to surfaces [15]. Many of these algorithms have been invented over 20 years ago but still were the mainstay of graphics hardware for years. Therefore effects not implemented in hardware had to be computed using the regular CPU.

In order to free up CPU time for other computations than graphics processing graphics subsystems had to offer true programmability. As the traditional rendering pipeline was not assigned for programmability its design had to be extended. This resulted in the inclusions of two distinct programmable processors, namely the *programmable vertex processor* and the *programmable fragment processor*. How these programmable processors are incorporated into the rendering pipeline is depicted in figure 3.5. As figure 3.5 illustrates the fixed-function stages are still available. But by design it is not possible to use fixed and programmable functionality at the same time, i.e a programmer has to implement all functions of the fixed-function stage he is going to use while using either the programmable vertex processor or the programmable fragment processor. A Program written for either of the programmable processors is referred to as *vertex shader* or *fragment shader* respectively. Programs can be written through vendor-specific extensions to the 3D programming interface, using vendor-specific assembler code, or using one of the available high-level shader languages (see section 3.1.2.3 for a small overview). The capabilities of different shader versions are comprised in a specification called *shader model*. The current specification of shaders distinguishes four different shader models. But only the first three are currently supported by the newest graphics accelerators.

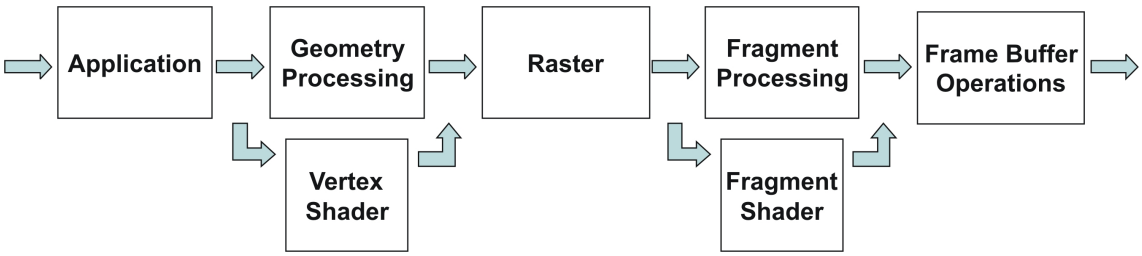


Figure 3.5: The Programmable Rendering Pipeline

Purely designed to allow for more visually compelling effects, such as High Dynamic Range Rendering, Displacement Mapping, or displaying natural phenomenas, a lot of effort has been undertaken to harness the offered functionality for the solution of scientific problems (see the end of section 1.1 for a quick overview). For another example, consumer graphics hardware was never intended for volume rendering in the first place.

3.1.2.1 Vertex Shaders

As their name and their place in the *rendering pipeline* suggests *vertex shaders* provide a way to modify parameters associated with each vertex, like its color, normal, texture coordinate(s), and position. First introduced with Microsoft's DirectX 8.0, and also available in OpenGL through extensions, vertex shaders helped to ease restrictions implied by the basic Gouraud/Phong model for lighting [25, 71] implemented in the fixed-function pipeline. When enabled, the hard-wired transform and lighting model is no longer available. In its place, the geometry processing stage of the pipeline is replaced by a vertex shader executing a series of commands written by an application developer. Both vertex shader and fixed-function geometry processing can by used in generating an image for displaying, by switching between one another; they simply cannot be used simultaneously. Therefore the developer also has to take care of implementing all or part of the fixed-function pipeline when a vertex shader is enabled. It is possible to emulate all of the functionality of the fixed-function geometric processing stage with a vertex shader, without a hit in performance [51]. Although there is a small overhead in setting up each vertex shader, as the shader itself must be uploaded to graphics accelerator when needed, a pure vertex shader implementation of a lighting model can actually be faster than the hard-wired path [55].

The vertex shader architecture is essentially a simple computer available on the graphics hardware. The programmable vertex processor branch of the rendering pipeline can be implemented as SIMD[1] parallel processing units, thus achieving additional speed overall [48].

Describing the whole vertex shader model or the underlying programming language specification in its entirety is beyond the scope of this document. For a more detailed account see [51, 89, 48, 23, 4, 24, 5, 6, 8]. However, a few comments should be made:

[1] Single Instruction Multiple Data

A vertex shader processes each vertex that is passed to it separately. It is not possible to pass results generated by one vertex to another vertex. Neither can a vertex shader create additional nor destroy superfluous vertices. A vertex shader's output must always at least consist of the vertex's homogeneous clip coordinates. But many other values, like diffuse color or texture coordinates, can be modified beyond this. As the complete shader architecture is tailored toward graphical computing many of the supported functions can be executed most efficiently on three- and four-element vectors.
Current hardware supports three different shader models for vertex shaders.

Shader Model 1.0 through **1.4** allows for a vertex shader to be comprised of 128 instruction steps. It does not allow for an explicit flow control, i.e statements like if, for, while, or goto are not available, as well as for early termination of the shader. This means that a vertex shader always takes the same time to finish for each vertex passed in. The amount of memory a vertex shader is able to use is restricted to 96 read-only and 12 read/write vector registers.

Shader Model 2.0 doubles the amount of possible instructions to 256. Likewise, the available read-only vector registers have risen to 256, while the read/write vector registers are now limited to 12-32. Flow control instructions are available as long as the loops they define can be completely unrolled, i.e they are controlled by constants. True dynamic branching is still not available. The calling of subroutines is also added to the specification.

Shader Model 3.0 offers significant improvements to vertex shaders. It allows a vertex shader to consist of nearly an unlimited amount of operations as it increases the maximum instruction count to 65535. The addition of true dynamic branching brings speedups to many algorithms that contain early-out opportunities. Another significant improvement is the ability to access up to four textures on any number of lookups during execution of a vertex shader, thus allowing true displacement mapping. To game developers the inclusion of instancing support is another key improvement as it makes it possible to draw many varied objects with just a single command.

Vertex Shaders are designed to increase the computation speed of more sophisticated lighting models [55]. They are able to compute values that slowly change over a surface, as further down the rendering pipeline these values will then be interpolated. By design, and as their name implies, they are not capable of shading fragments.

3.1.2.2 Fragment Shaders

In contrast to vertex shaders, *fragment shaders*, also called *pixel shaders*, are executed on a per-fragment basis during a rendering pass. Fragment shaders are essentially an evolutionary extension of the fixed-function multitexture fragment processing stage of the rendering pipeline. Both operate on constants and interpolated values. Also, both are capable of multiple texture retrievals to produce a pixel color, and optionally an associated alpha value. However, fragment shader capabilities surpass the limited functionality of the fixed-function multitexture fragment processing stage.

Fragment shaders are capable of using any result of a computation during their execution, even color values of a texture fetch, as texture coordinates for subsequent texture retrievals. Such a texture lookup is called *dependent texture lookup*. As section 3.2.7 shows, *dependent texture lookups* are an essential part of volume rendering applications on consumer graphics hardware. Fragment shaders can also modify the z-depth value associated with a fragment, as well as perform many other operations that do not fit well into the fixed-function multi-texture fragment processing stage concept.

First introduced with Microsoft's DirectX 8.0 their primary intend was to allow for implementation of a more convincing, i.e realistic, illumination model on a per-fragment basis. The fragment shader is an alternative path of the rendering pipeline that can replace the fragment processing stage. The fixed-function fragment processing stage is still available for use for during rendering of an image, but it cannot be used simultaneously with a fragment shader. Fragment shaders support many of the same math operations that vertex shaders do, as well as texturing operations. The interpolated diffuse and specular colors and alphas, constants, and sets of texture coordinates are the three sets of inputs passable to a fragment shader. Each of these can be a vector of up to four values. Inputs are treated as *rgba* data. A fragment shader consists of definitions of constants, a number of arithmetic instructions, and a number of texture address operations. After all instructions are performed the shaded fragment either continues down the rendering pipeline to be compared against the pixel tests and blended with the frame buffer, or, as of Shader Model 1.4, passed to programmable fragment processor a second time. After the second pass it is sent down the remaining pipeline. Another less efficient way to store intermediate results, but with the ability to perform more than just two rendering passes, is a technique called *render-to-texture* [67]. A fragment shader is also able to kill fragments, i.e discard fragments from further processing. On certain hardware the instructions used in a fragment shader might affect the performance of its execution, e.g. a fragment shader accessing more than two textures runs at half speed on NVIDIA GeForce3 or ATI Radeon 8500 based cards. Different version of the fragment shader model imply different restrictions on actual shaders.

Shader Model 1.0 through **1.4** limits a fragment shader to four texture addressing instructions, and eight arithmetic instructions. Discarding fragments is only partially supported. Statements like if, for, while, are only allowed if the corresponding loops can be completely unrolled, i.e dynamic branching is not supported. Values computed inside a shader are stored at low precision (8-bit) and clamped to the range $[0, 1]$. Shader Model 1.4 supports 16-bit values with the range $[-8, 8]$.

Shader Model 2.0 increases the amount of possible texture address instructions to 32 and the amount of arithmetic instructions to 64. Up to eight texture coordinates can be passed to a shader. 32 read-only vector registers are available, while the read/write vector registers are limited to 12-32. No improvements have been included for flow control statements. Still flow control instructions must define loops that can be completely unrolled. Values inside a shader are computed using `float` precision. NVIDIA further improved upon the Shader Model 2.0. Its *Shader Model 2.x* allowed for up to 1024 instructions per *extended fragment shader* with no restrictions on the number of actually used texture address instructions. Full support for `fixed` and `half` data types

has been added as well. If/else statements are fully supported whereas loop statements still have to be fully unrolled.

Shader Model 3.0 increases the maximum instruction count to 65535+ allowing for nearly unlimited shader programs. The addition of true dynamic branching saves performance by skipping complex shading on irrelevant fragments. The interpolated color values that are passed to a fragment shader are required to be at minimum of 32-bit precision. Whereas the support of multiple render targets was optional in shader model 2.0, a minimum of four simultaneous render targets is now required. The number of passable sets of texture coordinates has been increased from eight to ten. Shader Model 3.0 gives programmers full control over specular and fog computations, previously implemented as fixed-function stages. Additional back-face registers allow for two-sided lighting in a single pass.

For a closer look on *fragment* or *pixel shaders* refer to [24, 8, 23, 4, 5, 6]

3.1.2.3 High-Level Shading Languages

From their introduction, shader programs for consumer graphics hardware could be written using vendor specific extensions to 3D programming interface. Later, each vendor introduced its own assembler like shader language for programming its hardware. This made writing and reading code easier as it allowed for tools like macros or shader developing environments that helped to ease the burden of debugging. Nonetheless, this often led to portability problems, thus requiring to write shader programs for each particular hardware in its respective assembler language. The introduction of newer hardware can quickly make these shader programs obsolete or inefficient without constant maintenance. The next natural step, given an assembler like language, is to develop a high-level programming language. When Rob Cook, then at Lucasfilm Ltd., first published a SIGGRAPH paper about his concept of *shade trees* in 1984 [17] not a single consumer graphics accelerator had even been in development. Later, three different approaches for developing a high-level shader language have been taken: *non-interactive shading*, *multi-pass shading*, and *hardware-amenable shading*.

RenderMan, developed by Pixar in the late 1980s, uses the same basic concept of shade trees for surface shading [72, 84, 9]. RenderMan's shaders use a declarative or functional language to define expressions to be evaluated. Vertex and fragment shaders, in contrast, are programmed using imperative languages. Although at first intended to be implemented in hardware, RenderMan quickly grew too complex for this goal, thus becoming an *offline* or *non-interactive* shading language.

The SGI Interactive Shading Language used a *multi-pass* approach for delivering real-time shading [69]. The OpenGL graphics system as a whole can be thought of as a parallel SIMD computer [80, 20], with each rendering pass performing the equivalent of a single SIMD instruction. That way, a high-level description of the desired shading effect was split up into different rendering passes, with each pass sending its corresponding program down the OpenGL rendering pipeline for execution. In order to render a huge variety of RenderMan shaders two extensions had to be made to the

hardware: dependent texture lookups and a higher precision pipeline. Built upon OpenGL the SGI Interactive Shading Language is basically independent on the underlying hardware.

PixelFlow Shading Language and **Real-Time Shading Language** both were research projects to allow for hardware-amenable shading. PixelFlow [63, 43], developed at the Univerity of North Carolina during the mid-1990s, was an experimental SIMD multi-processor system capable of real-time shading. Developed at the Stanford University, the Real-Time Shading Language [74] was designed to run on second and third generation consumer graphics hardware. Shaders written in the Real-Time Shading Language could be compiled into one or more OpenGL rendering passes, depending on the underlying hardware's capabilities.

All of the three mentioned early high-level shader languages had their share in the development of the high-level shader languages commodity GPUs use today. Bill Mark, one of the original designers of the Stanford Real-Time Shading Language, even joined NVIDIA in 2001 to lead their effort to design and implement a shading language for their own GPUs. Currently there are three different shading languages supported by today's consumer graphics accelerators.

Cg, short for C for graphics, was developed by NVIDIA in collaboration with Microsoft and released in the end of 2002. It is inherited from earlier shading languages, like RenderMan or the Stanford Real-Time Shading Language, from 3D programming interfaces, like OpenGL or Direct3D, and, as its name implies, from the general-purpose programming language standard ANSI C. Several extensions and restrictions to standard C are present due to the nature of GPUs. One of the design goals during development was making Cg cross-platform compatible, i.e to allow shaders written in Cg to be executed on different hardware vendor's GPUs. Cg is also the only shading language currently available that integrates fully with the two major 3D programming interfaces. To achieve this cross-platform and cross-API compatibility the Cg compiler offers different profiles for each targeted system as well as the ability to compile Cg code during run-time execution of an application. In its initial release Cg offered support for Shader Models up to 2.0 as well as the NVIDIA extended Shader Model 2.x. The support for Shader Model 3.0 has been added to Cg through the inclusion of two new target profiles just recently. More profiles continue to be added to the specification in order to offer support for features implemented in both future hardware and 3D programming interfaces. In order to make Cg a industry-standard for shader programming additional tools have been released, such as CgFX and plug-ins for 3D modeling and CAD applications, easing the development and integration of shader programs. For a more closer account on Cg see [24, 5, 6, 1]

Microsoft's High-Level Shader Language, or HLSL [4] for short, is Microsoft's implementation of the Cg shader language. HLSL, just renamed for branding purposes, is an essential component of Microsoft's proprietary DirectX 9.0 multimedia framework and, unlike the Cg compiler, the HLSL compiler can therefore only generate DirectX executable code. Initially co-developed with NVIDIA, Microsoft and NVIDIA started evolving their respective shader languages independently from each other. Still most

shader programs written in either HLSL or Cg can easily be compiled using either compiler without major adjustments.

The OpenGL Shading Language is one of the major additions to OpenGL introduced in the release of the OpenGL 2.0 specification. Shaders are written using also a C-like language and encapsulated inside shader objects. Different shader objects can be combined to form program objects, a concept similar to texture and other buffer objects. The OpenGl 2.0 specification also added instructions for creating, linking, and enabling/disabling these program objects. These efforts have been taken to allow for a seamless and transparent integrations of shaders into existing applications, thus achieving a high acceptance among developers. The OpenGl Shader Language is often also referred to as GLSL or glslang. For an extensive look see [64, 36, 2].

Figure 3.6 summarizes the application of the available high-level shader languages and their respective 3D programming interfaces. Therefore Cg is best used if cross-platform, cross-API compatibility is to be achieved.

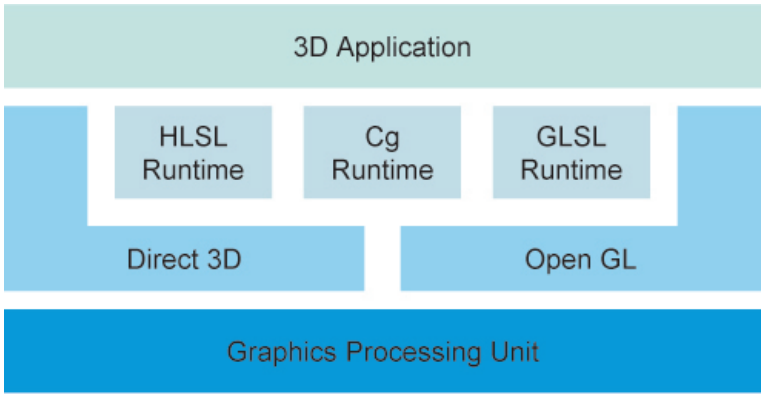


Figure 3.6: 3D programming interfaces and their supported shading languages

3.1.3 Characteristics of GPU Programming

In computer graphics the process of rendering an image can be subdivided in, first, transforming the primitives the scene is made of and, second, generating the color values for each output primitive in the final image, thus allowing for pipelining of the whole process. The current design of the rendering pipeline (sections 3.1.1, 3.1.2) resemblances this subdivision. Therefore two different types of shaders (sections 3.1.2.1, 3.1.2.2) have been introduced each working on its respective primitives. On current generation graphics hardware the number of fragment processing units is usually higher than the number of vertex processing units due to the realistic supposition that the currently in hardware implemented technique for image rendering, the Z-buffer rendering technique, usually generates much more fragments than the number of vertices describing a scene.

Each of the primitives used in each stage of the rendering pipeline, either vertices or fragments, are neither dependent on one of their predecessors nor on one of their successors.

Therefore huge improvements in rendering speeds can be achieved by parallelizing the rendering pipeline, i.e by adding more processing units to either the geometry processing stage or the fragment processing stage. Although this requires some sort of load balancing strategy [8], it still provides for huge speed-ups.

GPUs are a highly parallelized piece of hardware, by offering multiple pipelines for processing graphical primitives. Each stage of the rendering pipeline is designed to improve the computation of its associated graphical primitive, thus making GPUs also highly specialized processors.

NVIDIA's current high-end graphics accelerators, the GeForce 6 line of GPUs, sport up to 16 separate pixel pipelines and up to 6 vertex pipelines. Each fragment processing unit itself is capable of performing eight fragment shading operations per clock cycle. Both shader execution units offer support for the Shader Model 3.0 standard [18]. Across from this, ATI, NVIDIA's main competitor, offers a similar product, the series of X850 GPUs, matching the performance cornerstones but excluding support of Shader Model 3.0 [10].

Due to the fact that a single vertex requires approximately 100 individual precision floating point operations, with a single light source [7], GPUs have evolved to delivering several billion floating point operations per second. But still some limitations apply to the rendering pipeline in general:

Voxels , short for volume elements, are not accelerated on current generation GPUs.

Precision issues are still present. Although Shader Model 2.0 compliance requires to do all shader computations using full float precision the values passed down the rendering pipeline are clamped to the range $[0; 1]$ using a low precision fixed-format. Future accelerators promise to offer a full float precision pipeline.

Blending into the frame buffer is only performed at a fixed-precision as current implementations are only capable of displaying 8-bits per channel RBGA colors. Although GeForce 6 series GPUs offer the capability to do float-like 16-bit precision blending it is only possible to do this into an offscreen buffer.

Data transfer rate from graphics memory to host memory is very limited compared to bandwidth offered by the connection between GPU and graphics memory. Most of todays available motherboards offer AGP, short for Accelerated Graphics Port, for installing a graphics accelerator into a PC. Built upon PCI, AGP uses a point-to-point connection between the AGP device and the host memory, i.e an AGP device does not have to share its bandwidth with another device. It is asymmetric in design, thus offering different transfer rates for uploads and downloads. The current AGP 8x standard provides a theoretic upload rate of 2.1GB/s to graphics memory as opposed to a 133MB/s download rate from graphics memory. The recently introduced PCI Express standard, a replacement to both AGP and PCI, might help ease the issues given by the AGP connection. Like AGP, PCI Express uses a point-to-point architecture so no device will have to share its connection to the CPU or host memory with another device.

Furthermore, the new standard allows for PCI Express lanes to be bundled together to increase the bandwidth available to them. The PCI Express connection for graphics hardware uses 16 lanes simultaneously, thus providing a bandwidth of 4.2 GB/s. Another improvement is the fact that the bandwidth is offered in both directions, i.e for uploads to and downloads from the PCI Express device. Future considerations have been made to allow for even more lanes. PCI Express promises to be a major improvement over AGP in the future.

There are also some restrictions to be considered when using the programmable parts of the rendering pipeline:

Parameters passed to either the vertex or the fragment shaders are directly related to the rendering of images. In other words, transformation matrices, light sources, and color values, just to name a few, are the only values that are passed to a shader besides their respective primitives. While the shader model supports passing down global values or constants, called *uniform* parameters, for use during computation, neither their values can be changed during a shader's execution of a stream of primitives. Also, a shader cannot write to its input parameters, e.g change a transformation matrix, as this would break up the parallelized design concept.

Return values cannot be directly accessed by the CPU, as each result is passed down to the subsequent stage in the rendering pipeline, although some stages can be configured to which part of the subsequent stage their results should be passed to, e.g. a fragment shader can choose whether the computed result should be written to either the color buffer, the depth buffer, or both. Only the final result of the rendering process, i.e the computed image, can be download to host memory where it can be further processed by the host CPU.

Sharing information between the individual vertex processing units or fragment processing units is not possible by design. The great speed-up in image computation comes from the fact that each primitive is completely independent from one another, thus offering performance gains by allowing for parallelization of the rendering process. If different lanes of the rendering pipeline were able to communicate with each other and therefore change their execution instructions the independence of their respective primitives would get lost.

While current consumer graphics hardware promises very high performance many consideration have to be made considering how to harness their power.

3.2 Introduction to Texture-based Methods

In order to render an image of the entire volume, the volume data has to sampled at discrete locations (section 2.2). Strictly speaking, the sampling operation performed during rendering is actually a *resampling*, since the volume data are already discrete. In other words, the already sampled volume data is resampled at a different set of discrete locations than the set it was originally sampled at. This fact makes sampling the most fundamental task in volume rendering. The quality of the final image is dependent on the chosen resampling locations. After resampling, the obtained values then have to be mapped to optical properties, such as color and opacity, before they can be composited to create the final image. Compositing can be done in either *front-to-back* or *back-to-front* order.

The simplest approach to accomplish this resampling task is probably *ray-casting* (section 2.5). Besides the fact that ray-casting yields the most realistic results it is also the computational most expensive method because for each pixel in the final image a ray is cast into the volume, resampling the data at equispaced intervals along each ray.

Basically texture mapping operations perform a similar task to resampling. A equispaced grid of discrete *texels*, short for texture elements, is sampled repeatedly to obtain texture values at locations that do not coincide with the original grid. Therefore the nature of texture mapping operations as well as their available implementations in current graphics hardware make them an ideal candidate for performing the repetitive resampling operations that occur during volume rendering. When color and opacity values are stored in a texture map they can easily be mapped to resampled values just by sampling the texture. By exploiting hardware supported alpha blending (section 3.1.1.5) compositing can also be hardware accelerated. As ray-casting yields the most realistic results the most important question with regard to texture-based volume rendering is how to achieve the same or at least a sufficiently similar result in image quality.

A major drawback regarding volume rendering on commodity graphics hardware is that currently available hardware does not support the transformation, rasterization, or shading of voxels directly. Aside from just a few exceptions most of the currently available three-dimensional games base their rendering engines on polygonal scene descriptions. Therefore the *rendering pipeline* (section 3.1.1) of graphics accelerators is only designed to handle the transformation, rasterization, or shading of similar built scenes. Also, the design omits the acceleration of drawing textures directly to the frame buffer, i.e it is faster to draw a screen-filling quad with a texture applied to it than to directly display a texture by copying it to the frame buffer. To circumvent this drawback and to use the hardware accelerated sampling of texture for volume rendering the resampling locations have to be generated by rendering *proxy geometry* (section 3.2.1) with interpolated texture coordinates. Usually, this proxy geometry consists of a set of texture-mapped quads. Furthermore, the volume is rendered by compositing all slices of the proxy geometry from *back-to-front* via hardware accelerated alpha blending. The volume data itself has to be stored in one to several textures of three (section 3.2.4) or two dimensions, respectively. For example, a volume data set can either

be stored in a single 3D texture object or a stack of several 2D texture objects, each of which corresponds to an axis-aligned slice through the volume.

A volume can therefore be resampled at specific locations by rendering a set of texture-mapped proxy geometry and blending the generated fragments with the contents already stored in the frame buffer. Because this approach does not iterate over individual pixels on the image plane, but over parts of the volume itself, it can be considered an *object-order approach* in contrast to the image-order approach employed by ray-casting. Therefore the final result of a pixel is only available after all textured slices that contribute to that pixel's location have been processed.

3.2.1 Proxy Geometry

As the rendering pipeline of consumer graphics hardware is designed to handle large amounts of polygonal primitives rendering surfaces consisting of this kind of primitives is therefore accelerated. In the field of volume visualization only *indirect volume rendering* approaches allow for transforming the volume data into another representation before displaying it. For example, a variant of the marching cubes algorithm [49] can be used to build a geometric representation of the volume data by generating polygons that correspond to a certain iso-surface. In contrast, all *direct volume rendering* techniques do not allow changing the representation. Therefore the volume data has to be rendered as is, i.e the voxels contained in a volumetric data set have to be rendered directly. However, the rendering pipeline of commodity graphics hardware with texturing capabilities is designed to only render data built from vertices, i.e all fragments and ultimately all pixels computed by the graphics hardware are generated by rasterizing geometric primitives. Moreover, the sampling of textures is only supported in the interior of such geometric primitives specified by vertices.

A starting point for enabling consumer graphics hardware to render volumetric data directly is to place geometry inside the three-dimensional scalar field that constitutes the volume. When this geometry is rendered set texture coordinates are interpolated along the surface of the geometric primitive as well as other attributes. In the rasterization stage (section 3.1.1.3) each generated fragment is assigned its corresponding set of texture coordinates. This set of texture coordinates can later be used to sample one or several texture maps at the associated locations during the fragment processing stage (section 3.1.1.4). Now, to sample the volumetric data at arbitrary locations, the scalar field constituting the volume must be stored in one or several textures and the texture coordinates must be assigned to correspond to locations inside this scalar field. That way, the geometry exists solely for the purpose of generating locations for resampling the volume stored in the texture map(s), i.e it does not inhere any relations to the data contained in the volume. Therefore, this geometry is commonly referred to as *proxy geometry*.

Different kinds of geometries can be used to generated resampling locations for the visualization of the volumetric data set. However, it is important to note that the proxy geometry is closely related to the kind of textures used for storing the volume, such as 2-dimensional or

3-dimensional textures. For example, if the orientation of the slices of proxy geometry with respect to the actual volume can be arbitrary, the volumetric data set must be stored in a 3-dimensional textures. In this case, if the volume would be stored in a stack of 2-dimensional textures a single arbitrary slice would have to fetch data from several such textures of the stack. However, aligning the proxy geometry with the original volume data, i.e guaranteeing that all texture mapping operations for a single slice stay within the same texture, the proxy geometry can be comprised of a set of *object-aligned slices* (section 3.2.2), thus making 2-dimensional mapping capabilities sufficient.

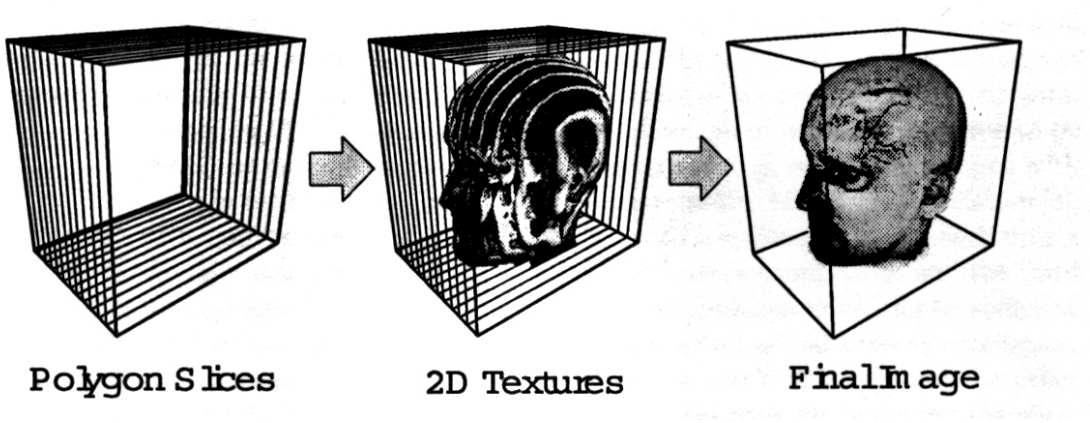


Figure 3.7: Object-aligned slices used as proxy geometry with 2-dimensional texture mapping, from [27].

Another approach is to render proxy geometry consisting of quads parallel to the image plane. The texture coordinates used to resample the texture object are interpolated over the interior of these slices by the rasterizer. The texture object has to be of three dimensions, as the orientation of the slices can be arbitrary in respect to the volumetric scalar field. As each slice of proxy geometry is always aligned to be parallel to the the image plane, i.e aligned to the viewport, this kind of proxy geometry is also referred to as *view-aligned slices* (section 3.2.5). Rendering view-aligned slices usually also involves clipping the view-aligned slices against the bounding box of the volume. A drawback of using 3-dimensional textures over 2-dimensional textures is that sampling 3-dimensional textures is computationally more expensive due to the use of trilinear interpolation as opposed to bilinear interpolation inside of 2-dimensional textures.

Another approach for generation resampling locations is to render view aligned spherical shells as proxy geometry. As spherical shells consist of significant more vertex primitives their computation is more complex and therefore more expensive.

3.2.2 Object-Aligned Slices using 2-dimensional Textures

Using only 2-dimensional texture mapping capabilities of current graphics hardware the volumetric data set must be stored in several 2-dimensional textures. This also implies that

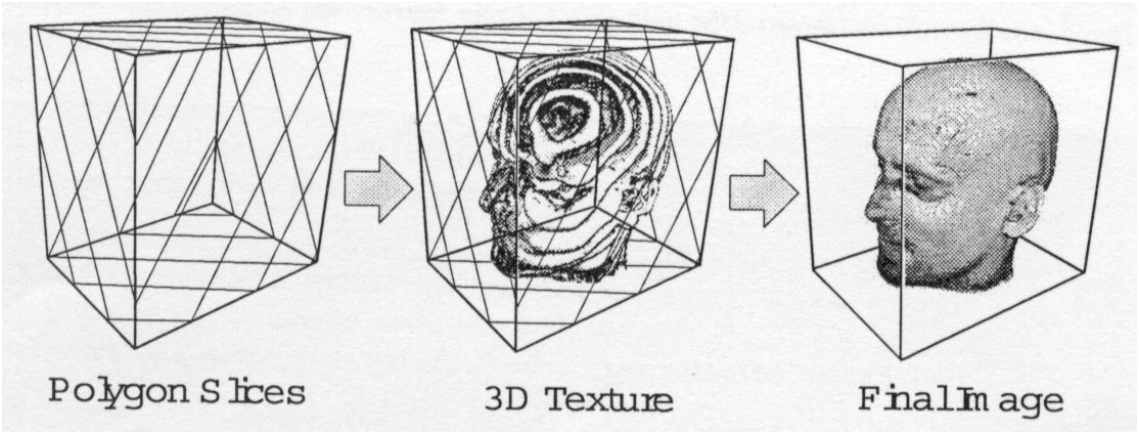


Figure 3.8: View-aligned slices used as proxy geometry with 3-dimensional texture mapping, from [27].

only 2-dimensional subsets of the volumetric data set are available for resampling. Proxy geometry, in this case, is constituted of a stack of planar slices. All slices of the stack are required to be object-aligned, i.e all slices have to be perpendicular to one of the major axes of the volume. During rendering the 2-dimensional textures are mapped to the different slices through hardware amenable bilinear filtering [16]. The slices must be aligned to either the x, y, or z-axis of the volume because each time a slice is rendered only two dimensions can be used to sample the 2-dimensional textures at the texture coordinates interpolated from the texture coordinates assigned to the vertices of the respective slices. Another reason is that bilinear filtering would not be sufficient for resampling otherwise. The third coordinate of the actual location of the drawn slice inside the volume is instead used to select the corresponding texture from the stack of textures. The proxy geometry is rendered in back-to-front order blending each slice on top of the slices already accumulated.

However, a single stack of 2-dimensional slices is not enough for visualization of the volume. Although it is sufficient for storing an entire volumetric data set, it would be possible to see through the individual slices when the point of view is rotated around the textured proxy geometry during rendering. This problem cannot be accounted for with just a single stack of slices. Therefore three stacks of slices must be stored, with each stack of slices aligned to one of the major axes. Then, the stack with slices most parallel to the image plane is chosen for rendering the volume as illustrated in figure 3.9. An obvious drawback of using object-aligned slices for volume visualization is the requirement for three slice stacks, that allocate three times the memory than the actual volume. Another drawback is that the switching of the stack currently used for rendering yields visible artifacts and a drop in rendering performance. If the three stacks of slices do not fit into local graphics memory each time a stack is switched all or part of it has to be re-uploaded from host memory to graphics memory. Therefore the frame that implied the stack switch might take longer to finish rendering. Also, when one stack is switched to another artifacts can become visible. The reason for this is that the actual locations of resampling points change abruptly with the change in stacks. This is illustrated in figure 3.10. When choosing a stack for rendering another consideration, in addition to the actual viewing direction, has to be taken into account. In order

to guarantee back-to-front order for rendering the stack of slices has to be rendered in one of two possible directions, i.e if the volume is looked at from the back, with respect to the order the slices are stored in the stack, the stack has to be rendered in reverse order, to achieve back-to-front rendering.

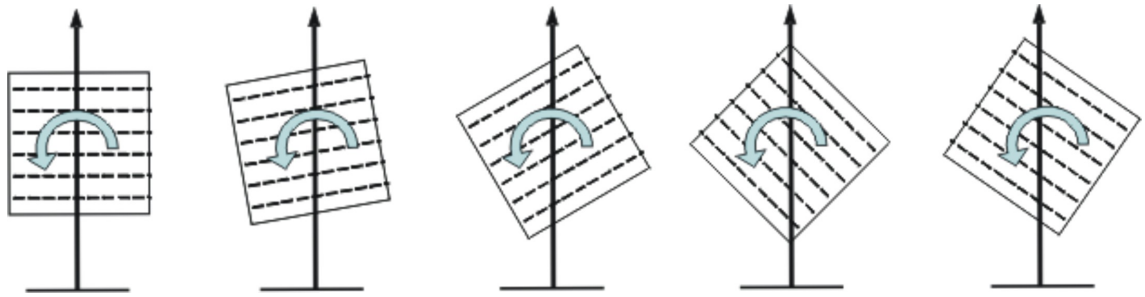


Figure 3.9: The stack of slices is chosen depending on the current viewing direction. Between image (3) and (4) the stack used for rendering has been swichted.

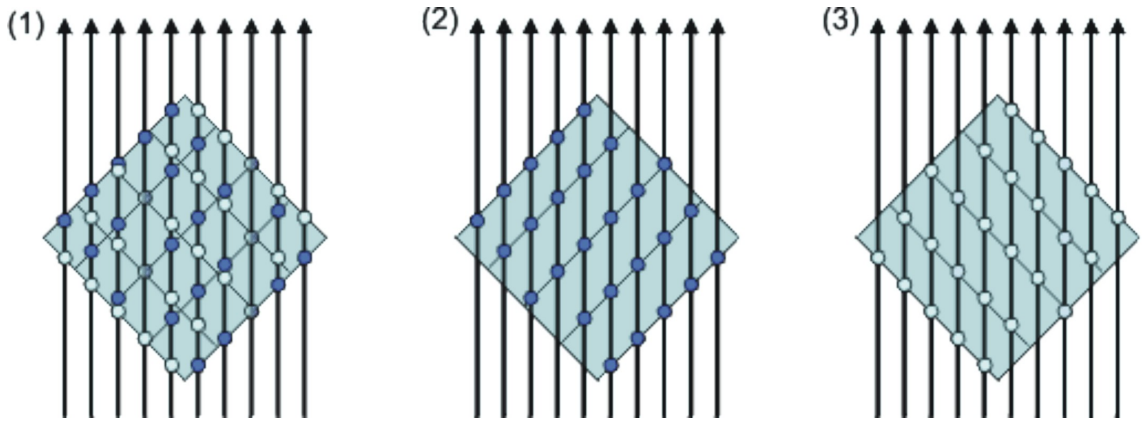


Figure 3.10: The location of resampling locations changes abruptly (1), when switching between one lice stack (1), to the next (2).

Using object-aligned slice stacks also leads to inconsistent sampling rates for different viewing directions. The reason for this is that alpha blending is used to accumulate the resampled values, thus performing a numerical integration of the volume rendering integral. This is used to achieve the same effect as compositing samples along a ray in ray-casting. The sampling distance at which the integral is approximated by the summation performed by alpha blending is dependent on distance between adjacent object-aligned slices.
The sampling distance can be incorporated into the numerical integration prior to execution, if the sampling distance is constant, i.e equidistant. However, when texture mapped object-aligned slices are used, the sampling distance between successive resampling locations not only is dependent on the slice distance, but also on the current viewing direction. This is illustrated in figure 3.11. It is easy to see that the sampling distance is only equal to the distance between adjacent slices when the viewing direction is parallel to the stack's major axis. When the view is rotated, the sampling distance increases. Therefore the output color that is blended into the frame buffer has to be modified accordingly on each change of the viewing

direction. Usually, this is done only approximately, simply by multiplying the stored intensity by the reciprocal of the cosine between the viewing vector and the stack's major axis. If a transfer function stores opacity-weighted RGB colors [88] a correction has also be applied to the respective color values. Although the aforementioned correction factor is used to correct the intensity values it can also be used to correct these color values.

Figure 3.11: The distance between adjacent ampling points is dependent on the viewing direction

The biggest advantage of using object-aligned slices for volume visualization is, one can achieve a very high rendering speed, since the employed bilinear interpolation requires only a lookup and weighting of four texel values for each resampling operation.
But this is also a major disadvantage of this approach as it limits the reconstruction of the volume data to use only 2-dimensional, i.e bilinear, interpolation. Another major disadvantage of this technique are the high memory requirements, due to the fact that the volume has to be stored in three stacks of textures, each aligned with a major axis of the volume. This approach is also limited to an upper number of slices that can be drawn as only a fixed number of textures is stored in each texture stack. The switching of the texture stacks can both be noticed visually, due to artifacts, and performance-wise, due to the possible uploading of texture data. Also, the sampling rate is inconsistent for different viewing direction.

3.2.3 Achieving Trilinear Interpolation using Object-Aligned Slices

The number of slices used for object-aligned volume visualization cannot be changed easily, as each slice corresponds to exactly one 2-dimensional slice from the stack of textures and vice versa. Additionally, the technique does not perform any interpolation between successive slices, since only bilinear interpolation is performed within each slice. These two properties can lead to visible artifacts when there are too few slices, and thus the sampling frequency is too low with respect to frequencies contained in a volume. With the algorithm explained in section 3.2.2 it is only possible to increase the sampling rate by generating additional slices, interpolated from the existing ones, before uploading them to the graphics hardware.

In order to increase the sampling rate without increasing the memory consumption of the volume itself, the inter-slice interpolation has to be performed on-the-fly by the graphics accelerator itself. On current generation consumer graphics hardware this can be achieved by using the available multi-texturing capabilities, i.e applying two textures simultaneously when rendering a single slice, as opposed to only one texture. Using the programmable

rendering pipeline the two bound textures are then linearly interpolated, thus achieving trilinear interpolation [75].

Therefore, the slice positions within the volume have to be specified using fractional numbers. The integer part corresponds to slices that actually exist in the current texture stack, and the fractional part determines the position between two adjacent slices. This way, the number of slices used to reconstruct the volume can now be adjusted arbitrarily, as the number of rendered slices is now independent from the number of slices stored in the volume.

For each slice to be rendered its two enclosing textures from the original stack of textures are bound to texture unit 0 and texture unit 1 respectively. The fractional part of the current's slice position is used as a weight for the interpolation between the values extracted from the textures. For each of the two neighboring slices standard bilinear interpolation is employed, and the interpolation between the two extracted results achieves trilinear interpolation (see figure 3.12). Therefore triliniear interpolation can be performed within the whole volume using only 2-dimensional textures and the programmable multi-texturing pipeline.

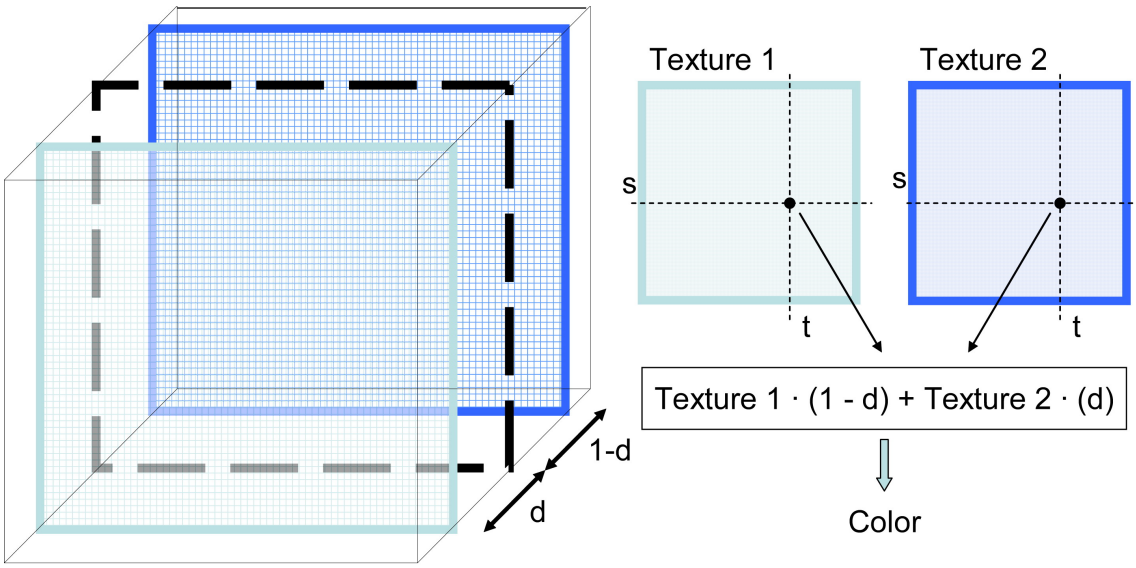


Figure 3.12: The values from both, front and back, textures are fetched using bilinear interpolation. Then their values are combined with respect to their distance to the current slice, thus achieving trilinear interpolation.

For volume visualization, using object-aligned slices with on-the-fly interpolation between two 2-dimensional textures from the current stack combines the advantages of object-aligned slice with the capability of arbitrarily controlling the number of slices, i.e the sampling rate. The trilinear interpolation, this method allows for, is not entirely comparable to trilinear interpolation that is employed while sampling 3-dimensional textures.
Unfortunately, all the other disadvantages inhered from using object-aligned slices, such as high memory consumption due to the three texture stacks and visible artifacts due to the switching between them, are still present in this approach.

3.2.4 Three-dimensional Textures

While 3-dimensional textures have been available in high-end graphics hardware for over a decade support for them only started to appear in consumer graphics hardware a few years ago. They were introduced first in ATI's Radeon 7500 series and later also appeared in NVIDIA's GeForce3 series of GPUs. But only a few applications were created that took advantage of them as shading in real-time graphics, especially games, is still largely a discipline of pasting one or multiple 2-dimensional textures onto 3-dimensional surfaces. But for delivering high-quality volume rendering they are an essential part.

The simplest way to think of 3-dimensional textures is as a stack of 2-dimensional ones. Though there are different ways about how the 3-dimensional textures are actual stored on the graphics hardware from manufacturer to manufacturer their use is the same. For example, on ATI Radeon X800 series GPUs a 3-dimensional texture is actually stored linearly, i.e like the volume is commonly stored in main memory. Whereas NVIDIA GPUs swizzle the volume while transferring it to a 3-dimensional texture stored in graphics memory. ATI's approach is simpler to implement and delivers fast rendering speeds when the 3-dimensional texture is accessed axis-aligned to the direction of the stack it is stored in. Rendering arbitrary orientations of the volume data results in big performance losses. NVIDIA's technique offers slightly less rendering speeds when the volume is rendered axis-aligned but the performance does not drop when viewed from arbitrary orientations. The texture coordinates of 3-dimensional textures used to access the texels stored within them are commonly called s, t, r, q. Some 3D programming interfaces use a different conventions for referring to width, height, depth, and projective coordinates respectively.

With the addition of an extra dimension the method of filtering textures changes a bit. Simple linear filtering of a 3-dimensional texture is referred to as trilinear filtering as it performs linear filtering along all three axes of the texture. In doing so colors are sampled from eight different texels. It is important not to mistake trilinear filtering used for a single 3-dimensional texture for trilinear filtering used for interpolating different levels in the mip-map chain of a 2-dimensional texture. With 3-dimensional textures, this case is referred to as quadrilinear filtering as is adds a linear filter along the fourth axis of volumetric mip-maps. Therefore the sampling of a textures doubles in expense when moving from 2-dimensional textures to 3-dimensional ones. Additionally, anisotropic filtering becomes more complex. Anisotropic filtering of 2-dimensional textures is understood as a filter along a line across the texture with a width of two and a length up to the maximum degree of anisotropy. For example, an anisotropic filter with a maximum anisotropy of eight requires $2 * 8 = 16$ texels. For 3-dimensional textures anisotropic filtering increases to a slab with width two and extending in the remaining two dimensions up to the maximum degree of anisotropy. Now, the mentioned example requires $2 * 8 * 8 = 128$ texels. This is 32 times the expense of a bilinear fetch of a 2-dimensional texture that is considered to be the basic unit of texture fetching performance.

With respect to filtering storing a 3-dimensional texture in graphics memory is computational more expensive. With an increase in resolution of the 3-dimensional texture its memory consumption grows extremely rapidly. For example, a texture of $256 \times 256 \times 256$ in dimension storing 32-bit RGBA values in each texel takes up $64MB$ in memory. Therefore, the number of color channels used in a 3-dimensional texture should be kept to a minimum. For volumetric data containing only intensity values, storing it in a 3-dimensional texture with only a single channel rather than RGBA is sufficient. Also, efficient texture compression techniques for 3-dimensional textures have been proposed [77]. Despite their high storage and filtering costs, 3-dimensional textures allow for higher-quality volume rendering than 2-dimensional textures (see section 3.2.5).

3.2.5 View-Aligned Slices using 3-dimensional Textures

In this approach the volume is stored in a single 3-dimensional texture. View aligned slices, usually clipped against the bounding box of the volume, are used to generate resampling locations for reconstructing the volume. In this case, 3-dimensional texture coordinates are interpolated over the interior of the view-aligned slices and then used directly for addressing the volume. This approach takes advantage of spatial coherence inside the volume. Unlike ray-casting, where each pixel of the final image is computed ray by ray, this approach processes all "rays"at a certain resampling distance simultaneously, i.e one 2-dimensional layer at a time.

One of the major advantages of using 3-dimensional textures over a stack of 2-dimensional textures is that slices can be oriented arbitrarily with respect to the volume. Thus allowing for arbitrarily oriented slices to be used for resampling the volume, i.e making it possible to display slices not perpendicular to one of the major axes of the volume. In order to mimic ray-casting as close as possible slices parallel to the image plane, i.e view-aligned, are rendered as proxy geometry. For orthogonal projection this approach offers an equidistant sampling rate for all viewing directions, thus mimicing ray-casting perfectly for each "ray", i.e for each final pixel (figure 3.13 (1)). However, in case of perspective projection, the distance between successive resampling locations is not equal for adjacent "rays", i.e for adjacent pixels (figure 3.13 (2)). Although this approach offers a good approximation of the final result, it is possible to render spherical shells as proxy geometry (section 3.2.6), which offer an equidistant sampling distance for perspective projection at the expense of more vertices that need to be processed.

While resampling the 3-dimensional texture at a location specified by the interpolated 3-dimensional texture coordinates, the graphics hardware performs general trilinear interpolation for each fragment. Therefore the number of slices can be chosen arbitrarily on-the-fly, without the need for setting up inter-slice interpolation manually.

The employing of trilinear interpolation for resampling the volume is a major advantage of using 3-dimensional textured view-aligned slices, as it results in an image of higher quality. Apart from that it is also possible to render slices with arbitrary orientation with respect to

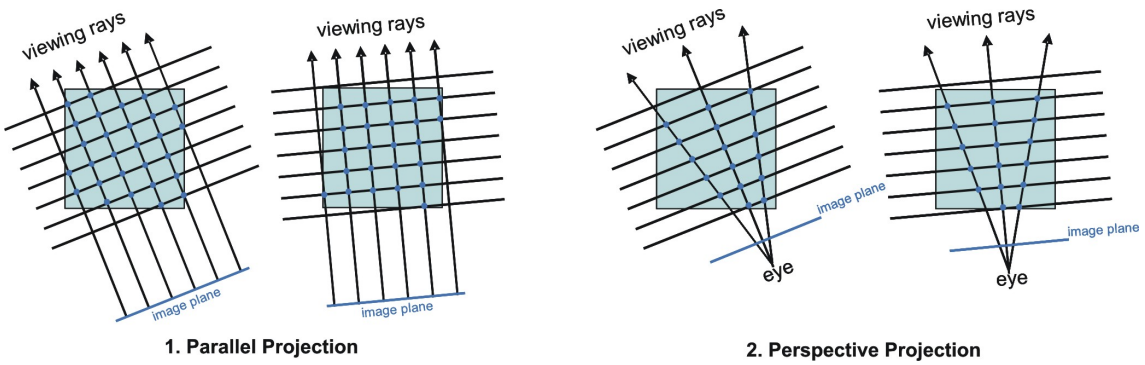


Figure 3.13: Resampling locations on view-aligned slices for (1) orthographic projection, and perspective projection (2), respectively. Notice how the distance between resampling points is equidistant for all points in (1), but not in (2)

the volumetric data set. This allows to maintain a constant distance between sampling points for all pixels and viewing directions, if orthogonal projection is used, as well as cutting out a single slice not aligned to a major axis of the volume. Furthermore, a single 3-dimensional textures allocates only a third of the memory the three object-aligned 2-dimensional textures stacks do.
The major disadvantage is that trilinear interpolation is significantly slower than bilinear interpolation, due to the requirement for using eight as opposed to four texels for each computed sample. This also requires texture fetch patterns that decrease the efficiency of texture caches on graphics memory.

3.2.6 Spherical Shells using 3-dimensional Textures

In the case of perspective projection, all types of proxy geometry that use planar slices for generating resampling locations, share the basic problem of a non-constant sampling distance between successive samples for adjacent pixels of the final image (figure 3.13 (2)). This pixel-to-pixel difference cannot be account for by incorporating the sampling distance in the numerical approximation of the volume rendering integral. A possible solution to this problem is the use of spherical shells instead of planar slices [41]. In order to closely mimic ray-casting, i.e to maintain a constant sampling distance for all pixels, the proxy geometry has to be comprised of concentric spheres. In practice it is sufficient to clip these spheres against both the viewing frustum and the bounding box of the volumetric data set. The remaining parts of the spheres are also referred to as shells.

However, a major drawback of this approach is that the spherical shells are much more complicated to setup than the otherwise used planar slices and also require more geometry to be rendered on screen, due to the fact that spheres have to be highly tessellated.

Therefore, this approach is only useful when perspective projection is used, and the artifacts caused by the not entirely accurate sampling distance for 3-dimensional textured view-aligned slices is deemed noticeable. As this is not likely to be the case, view-aligned slices

are usually sufficient even when perspective projection is used. Furthermore, the use of 3-dimensional texture mapping capabilities is mandatory for rendering spherical shells.

3.2.7 Transfer Functions

The role of transfer functions is to emphasize features in the data. Typically, a transfer function's job is to assign optical properties, such as color and opacity, to more abstract data values or other data measures defined by the volumetric data set. These optical properties are then used to generate a meaningful image from the volumetric data set. The simplest and most widely used transfer function are of one dimension only, as they map the range of values stored in the volume to color and opacity.
A transfer function can be evaluated by evaluating the definition of the transfer function during rendering of the volume. As this might increase the computational complexity of the rendering by a huge amount it is often better to precompute a transfer function's value and store them in a transfer function table. During rendering of the volume, this transfer function table can then be used to map the data values resampled from the volume to optical properties by a simple table lookup. There are two methods to accomplish this using graphics hardware.
The first method method uses a *color table* or *paletted textures* to store a user defined 1-dimensional lookup table, which encodes the transfer function. During texture lookup this method replaces an 8bit texel with the color components at that 8 bit value's position in the lookup table. Dependent on the capabilities of the graphics accelerator lookups can be based on up to 12 bit texels. Also, the textures, whose values are to be replaced, might have to be in a specific internal format. As the transfer function is evaluated prior to interpolation this is referred to *pre-classification*. *Pre-classification* can cause significant visible artifacts in the final image, especially when there is a sharp peak in the transfer function.
The second method uses *dependent texture reads*. A dependent texture read is the process by which the interpolated color components from one texture fetch are used as texture coordinates to read from a second texture. In volume rendering applications, the first texture or set of textures is the volumetric data set itself and the second texture is the transfer function. Since the transfer function can be stored as a regular texture multi-dimensional transfer function can also be evaluated. Using dependent texture reads is also referred to as *post-classification* as the data values from the volume are interpolated prior to evaluation of the transfer function. *Post-classification* causes much less visible artifacts than *pre-classification*.
Evaluating transfer functions on-the-fly is an essential part of a volume rendering applications, first, due to the fact that it is inefficient to update the entire volume and reload it each time the transfer function changes. It is much faster to update the smaller lookup table and let the graphics hardware handle the transformation from data value to optical properties. Second, as this is equal to *pre-classification*, it can lead to visible artifacts in the rendering of the volume.

3.3 Implementation of a Texture-based Volume Renderer

This chapter presents an overview of the steps necessary for a texture-based volume renderer to visualize volumetric data from a implementation-centric point of view. It outlines where each individual component does fit in, and what their order of execution is. The steps presented here are based on separate portions of code that can be found in an actual implementation of a texture-based volume renderer for consumer graphics hardware. One of the implementation's requirements was to achieve cross-plattform compatibility. Therefore, OpenGL was used as the underlying 3D programming interface. The decision about the shader language used for implementation was taken in favor of Cg versus GLSL, since GLSL became widely available not until the latter stages of this project.

3.3.1 General Program Flow

In general, texture-based volume rendering techniques can be divided into three different stages:

1. **Initialization:** The initialization stage is usually performed only once at start-up. At the beginning of the application the volumetric data set is loaded into host memory. Depending on the bit alignment the volume is stored in, and the bit alignment that the host hardware uses, it might be necessary to realign the volume data during loading. If other computations are intended to further process the volume data and their results do not change over the course of rendering of the volume they can be performed after loading of the data. After loading and preprocessing of the volume data is finished, and the rendering context has been created, the volume has to be downloaded to one or several texture objects depending on the texture-based volume rendering technique exposed. Shader programs necessary to the rendering algorithms have to be created as well.

2. **Update:** After initialization and each time viewing parameters change, the proxy geometry has to be updated. If the rendering mode changes textures might have to be refreshed. If rendering also involves the application of a transfer function to the volumetric data set it has to be updated as well. Also, if the sampling rate changes, e.g. through increasing or decreasing the number of drawn slices, the opacity correction factor has to be recomputed.

3. **Rendering:** At the beginning of the rendering stage all rendering states need to be set up appropriately. This usually includes the disabling of lighting and culling, and setting up the correct parameters for alpha blending. In order to display opaque objects inside the volume, depth testing has to be enabled, but writing to the depth buffer has to be disabled. Before drawing the proxy geometry, textures that store the respective volume and transfer function have to be bound to texture units, so that a fragment shader can access these textures. After all the vertex shader's and fragment shader's input parameters have been specified, and the shaders have been bound as well, the proxy geometry can finally be drawn in back-to-front order. After drawing has finished all rendering states have to be restored, so that the volume renderer does not affect the display of other objects in the scene.

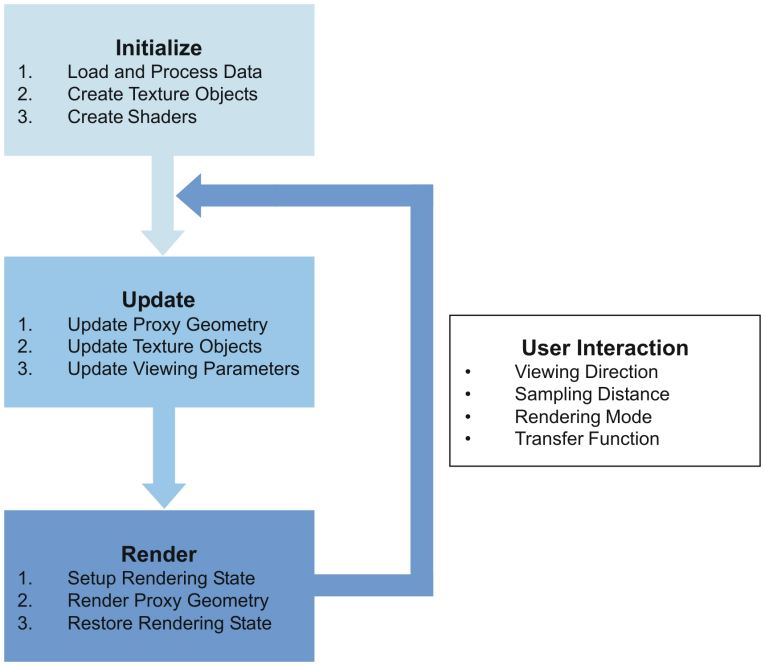


Figure 3.14: The steps of a typical texture-based volume renderer implementation.

3.3.2 Volumetric Data and Transfer Function Representation

The volumetric data set and the transfer function both have to be stored in host memory in a suitable format for rendering. Usually medical images generated by computer tomography use 12-bits per value, so they must be represented in a format capable of storing at least 12-bits in order not to lose any precision. This might require realigning the individual bits representing the data if the volume was generated on a machine with a different type of endian than the machine it is going to be displayed on. Although it is possible to compute some properties of the volume directly on the GPU, it is best to leave all computations to the CPU, whose results do not change during rendering. Finally, the preprocessed volumetric data set should be prepared for uploading it to the graphics accelerators's memory, thus storing the volume in one or several texture objects is preferable. The volume can either be stored in a single 3-dimensional texture, when view-aligned slices are used, or split up into three stacks of 2-dimensional textures, when object-aligned slices are used. In the case of object-aligned slices, data replication can be avoided by uploading the data set only once and reconstructing slices on-the-fly, though it decreases performance [44]. In the case, that the volume might have to be re-initialized or re-processed from the original data set during application execution it is useful to also leave a copy of the original volume in host memory. In order to describe all the informations needed for rendering, several volumes might have to be stored depending on the complexity of the rendering mode, the classification, and illumination technique used. Also, the actual storage format of each voxel contained in the volume depends on the rendering mode and the actual type of the volume. For example, a volume containing only density values requires only a single channel per voxel for storing its information, whereas a volume representing gradients at least needs three channels per voxel. If possible, conceptually different volumes, may also be combined into a single vol-

ume for rendering in order to lower memory requirements. For example, it is possible to store gradient and density values in a single RGBA texture as gradients need three channels per voxel and density only need one channel per voxel.

A transfer function can typically be represented by either a 1-dimensional or multi-dimensional color lookup table. All transfer functions must as well be stored in a format suitable for uploading to the graphics hardware's memory, thus preferable in a texture object. As opposed to volume data a copy of the actual transfer function should always be left in host memory as it might change often during the course of rendering. That way, it is possible to update certain entries in its corresponding color lookup table and to update the transfer function stored in a texture map easily afterward.

Usually the issue of volume data representation is part of the initialization stage, though in some cases new data may have to be generated on-the-fly when the rendering mode or certain parameters are changed. Transfer functions are also set up in the preprocessing stage but it is quite common that their values are going to be changed during the visualization of the volume data. Their alteration is usually triggered by the user of the volume rendering application.

3.3.3 Storing the Volume in a Texture

In order for the graphics hardware to be able to access the values stored inside the volume, the volume itself must be stored in one or several texture maps that are uploaded to the graphics hardware before rendering. During the generation of the texture the volume's values, stored in a specific data format, might have to be transformed in order to fit into the format of the texture. The transformation from "external texture format"to "internal texture format"is done by the graphics harware's driver during uploading texture to the graphics accelerator. The whole process of uploading volume data as textures is usually done only during initialization of the renderer, but it might also be necessary when some parameters change.

How and what textures containing the actual volume have to be uploaded to local graphics card memory exactly is dependent on the actual mode of rendering and type of classification (see section 3.2.7). Listing 3.1 shows a code fragment for creating and uploading a single 3-dimensional texture for rendering with view-aligned slices. The code fragment in listing 3.2 creates one of the three texture stacks containing $number\_of\_Slices\_in\_Stack$ texture objects and uploads them to the graphics hardware for rendering of object aligned slices. Line 1 and 1 in either listing, creates the specified number of texture object IDs, whereas line 3 and 5 respectively bind the texture object given by its respective ID to the current active texture unit. Lines 5 to 9 in listing 3.1 are responsible for setting some texture specific parameters for the bound 3-dimensional texture. The respective parameters for 2-dimensional textures can be set using lines 7 to 10 in listing 3.2. The **glTexImage3D**(...) command in line 11 of listing 3.1 uploads a volume containing only density information to one 3-dimensional texture that stores only a single channel. **glTexImage2D**(...) in line 12 of listing 3.2 uploads the same volume to several 2-dimensional textures respectively.

```
1  glGenTextures(1, pointer_To_TextureID);
2
3  glBindTexture(GL_TEXTURE_3D, pointer_To_TextureID[0]);
4
5  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, wrap_Mode_S);
6  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, wrap_Mode_T);
7  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, wrap_Mode_R);
8  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, minification_Filter);
9  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, magnification_Filter);
10
11 glTexImage3D( GL_TEXTURE_3D,
12               0,
13               GL_INTENSITY16,
14               texture_Width,
15               texture_Height,
16               texture_Depth,
17               0,
18               GL_LUMINANCE,
19               GL_UNSIGNED_SHORT,
20               pointer_To_Volume_Data);
```

Listing 3.1: Code sample that creates and uploads a texture for view-aligned volume rendering.

```
1  glGenTextures(number_of_Slices_in_Stack, pointer_To_TextureIDs_Stack);
2
3  for(int textureID = 0; textureID < number_of_Slices_in_Stack; i++)
4  {
5    glBindTexture(GL_TEXTURE_2D, pointer_To_TextureIDs_Stack[textureID]);
6
7    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_Mode_S);
8    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_Mode_T);
9    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minification_Filter);
10   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magnification_Filter);
11
12   glTexImage2D( GL_TEXTURE_2D,
13                 0,
14                 GL_INTENSITY16,
15                 width,
16                 height,
17                 0,
18                 GL_LUMINANCE,
19                 GL_UNSIGNED_SHORT,
20                 pointer_To_Volume_Data_Slice);
21 }
```

Listing 3.2: Code sample that creates and uploads one of the three texture stacks for object-aligned volume rendering.

3.3.4 Handling the Transfer Function

Except from initialization, uploading a transfer function is only necessary when the transfer function changes. Although not advisable, a transfer function might not be even necessary at all, if the transfer function has already been applied to the volume data itself. Depending on the type of classification used transfer function can be uploaded to the graphics hardware in one of two formats:

pre-classification: In this case, transfer functions are uploaded as texture palettes. Indexes to a texture palette are expanded on-the-fly to the stored color values before linear blending is performed. Listing 3.3 shows a code fragment for uploading a single texture palette. It is possible to use the density values stored in a volume as indexes for the lookup in the color table, thus making computing indexes obsolete. It is only necessary to configure the OpenGL texture unit to do so. Multi-dimensional transfer functions cannot be represented using color look-up tables. Another drawback of using texture palettes is that linear blending is not performed on the density values stored in a volume itself but on the indexed color values retrieved from the texture palette instead. This can lead to visible artifacts if the transfer function has a peak between extracted values.

```
1 glColorTable( GL_SHARED_TEXTURE_PALETTE,
2               GL_RGBA8,
3               lenght * channels,
4               GL_RGBA,
5               GL_UNSIGNED_BYTE,
6               pointer_To_Transfer_Function_Array);
```

Listing 3.3: Code sample that uploads a transfer function representation stored in an array into a color table for pre-classification.

post-classification: If post classification is used instead, the same lookup table can be used, but it has to be uploaded as either a 1-dimensional or multi-dimensional texture map. Basically, a 1-dimensional texture map can also be represented by a 2-dimensional texture map with height 1 or width 1. Listing 3.4 uploads the array that represents the transfer function to a 2-dimensional texture map with height 1. How an already specified transfer function is updated efficiently, is depicted in listing 3.5. An advantage of using dependent texture lookups over using texture palettes is that linear blending is performed on the density values stored in the volume before the interpolated color is used to index the final color in the transfer function lookup table.

```
1  if(transfer_Function_TextureID == NULL)
2  {
3    glGenTextures(1, &transfer_Function_TextureID);
4
5    glBindTexture(GL_TEXTURE_2D, transfer_Function_TextureID);
6
7    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap_Mode_S);
8    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap_Mode_T);
9    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minification_Filter);
10   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magnification_Filter);
11
12   glTexImage2D( GL_TEXTURE_2D,
13                 0,
14                 GL_RGBA16,
15                 transfer_Function_Width,
16                 transfer_Function_Height,
17                 0,
18                 GL_RGBA,
19                 GL_UNSIGNED_SHORT,
20                 pointer_To_Transfer_Function_Array);
21 }
```

Listing 3.4: Code sample that uploads a transfer function representation stored in an array into a 2-dimensional texture object for post-classification.

```
1  if(transfer_Function_TextureID != NULL)
2  {
3    glBindTexture(GL_TEXTURE_2D, transfer_Function_TextureID);
4
5    glTexSubImage2D( GL_TEXTURE_2D,
6                     0,
7                     x_offset,
8                     y_offset,
9                     transfer_Function_Width,
10                    transfer_Function_Height,
11                    GL_RGBA,
12                    GL_UNSIGNED_SHORT,
13                    pointer_To_Transfer_Function_Array);
14 }
```

Listing 3.5: Code sample that updates an already specified transfer function representation for post-classification. The **glTexSubImage2D**(...) command is used to update the contents of the currently bound 2-dimensional texture object.

3.3.5 Configuring the Programmable Shader Units

In order to harness the capabilities provided by both programmable shader units, their corresponding programs must be uploaded at first. Using Cg it is possible to write these programs in a C-like language as well as in an assembler like language. For the Cg runtime it does not matter if the actual program is stored in a separate file or in a string inside the main application as it provides support for loading a program either way. Furthermore, Cg also offers an on-line compiler that is capable of compiling programs written in Cg into their corresponding assembler notation during execution of the actual application. The Cg compiler offers different profiles for compiling to reflect the different capabilities of the underlying graphics processing units, i.e the use of profiles makes it clear if a certain shader's instructions are supported by corresponding GPUs, thus allowing its execution or not. As the Cg runtime allows to specify the entry function of a shader during its loading it is possible to store different versions of a shader in the same location.

Shaders written in Cg allow two different types of parameters as input, either *varying* or *uniform* parameters respectively. *Uniform* parameters are parameters that are constant during a shader's execution. Additionally, uniform parameters are the only parameters that can be set from an external environment. Therefore, for an application to change an uniform parameter the shader must first be unloaded. Uniform parameters are specified as such by using the `uniform` keyword before the declaration of the parameter. Usually, sampler parameters, i.e texture objects, are declared as uniform. Values of *varying* parameters typically vary for each invocation of a shader, that is either for each processed vertex or fragment. Cg introduces different *semantics*, always a colon followed by a special word after a parameters declaration. This *semantic* indicates which hardware resource feeds the respective member when a Cg program is executed, such as the position or color of each input vertex, positions of lights, or a fragment's color or depth value. Not all semantics are available in all profiles supported by the Cg runtime.

Semantics are also used to bind the computed results of a shader to certain hardware resources for further processing down the rendering pipeline. Though some input and output semantics may have the same name, they are associated with different hardware resources. For example, for an input parameter to a vertex program, the semantic **`POSITION`** refers to the application-specified position assigned by the application when it sends a vertex to the graphics hardware. However, an output parameter of the same vertex program associated with the **`POSITION`** semantic represents the clip-space position that is fed to the rasterization stage of the rendering pipeline. Both semantics are named **`POSITION`** and represent a position, but each of them refers to a position at different points along the rendering pipeline.

Listing 3.6 shows a fragment shader that evaluates a transfer function for each value interpolated from the volume by a dependent texture lookup. The result is the modulated by the fragment's interpolated vertex color prior to sending it to the pixel operations stage of the rendering pipeline. The instructions necessary to load this program from a file and create an actual fragment shader from it are depicted in listing 3.7. This has to be done only once during the application's initialization. cgCreateContext() is used to obtain the current context

for compiling and executing a shader program. The cgCreateProgramFromFile(...) and the cgGLLoadProgram() commands are necessary to load a shader program form an external file to the GL subsystem. For each uniform parameter of the shader, a cgGetNamedParameter(...) function call has to be performed so that the application is able to pass parameters to the shader. Listing 3.8 outlines the steps necessary to upload, i.e bind, this fragment shader to the programmable fragment processing unit. In order to enable a shader for execution, the shader program has to be bound to he GPU by calling cgGLBindProgram(...) followed by cgGLEnableProfile(...) command. When the shader is activated all necessary uniform parameters have to be set by cgGLSetParameter(...). The subsequent calling of cgGLEnableTextureParameter(...) is only essential for texture object parameters. All subsequent rendering calls between lines 8 and 12 are executed by the currently bound shader programs, i.e by the programmable rendering pipeline. To disable a bound shader program and hand over execution of rendering calls back to the fixed-function rendering pipeline it is sufficient to disable the currently active profile. cgGLDisableProfile(...). In addition, all enable texture parameters have to be disabled as well.

```
1  float4 main( uniform sampler3D volume_texture_3D,
2               uniform sampler2D transfer_function_texture_2D,
3               float3 texture_coordinates_3D : TEXCOORD0,
4               float4 vertex_color : COLOR0
5              ):COLOR
6  {
7    float4 color = tex3D(volume_texture_3D, texture_coordinates_3D);
8
9    color = tex2D(transfer_function_texture_2D, color.xy);
10
11   return color;
12 }
```

Listing 3.6: Fragment shader that evaluates a transfer function for an interpolated value from a 3-dimensional texture and modulates the result by the fragment's interpolated vertex color. The tex3D (...); and tex2D (...); commands respectively are responsible for retrieving the interpolated color values from the specified sampler objects, i.e texture objects.

3.3.6 Setting and Restoring the Rendering State

Setting up the rendering state usually involves enabling or disabling certain states of the rendering pipeline. For direct volume rendering using texture mapping techniques lighting has to be disabled as it would only affect the rendered proxy geometry, thus yielding wrong results. However, if lighting is required, it has to be done on a per-fragment basis using a programmable fragment shader. Culling either the front or the back faces of the geometry drawn could result in not rendering any geometry at all for certain views or at least make the computation of the proxy geometry more complex. Therefore culling should be disabled as well.

```
1  CGcontext cg_context = cgCreateContext();
2
3  CGprofile supported_profile = CG_PROFILE_ARBFP1;
4
5  CGprogram cg_program_handle =
6    cgCreateProgramFromFile(cg_context, CG_SOURCE, cg_program_filename_string,
7                            supported_profile, entry_point_string, NULL);
8
9  cgGLLoadProgram(cg_program_handle);
10
11 CGparameter volume_texture_3D_parameter =
12   cgGetNamedParameter(cg_program_handle, "volume_texture_3D");
13
14 CGparameter transfer_function_texture_2D_parameter =
15   cgGetNamedParameter(cg_program_handle, "transfer_function_texture_2D");
```

Listing 3.7: Instructions necessary to load the fragment program depicted in figure 3.6 from disc and create an executable shader from it

```
1  cgGLBindProgram(cg_program_handle);
2  cgGLEnableProfile(supported_profile);
3
4  cgGLSetTextureParameter(volume_texture_3D_parameter, volume_TextureID);
5  cgGLEnableTextureParameter(volume_texture_3D_parameter);
6
7  cgGLSetTextureParameter(transfer_function_texture_2D_parameter, transfer_Function_TextureID);
8  cgGLEnableTextureParameter(transfer_function_texture_2D_parameter);
9
10 renderSomething();
11
12 cgGLDisableTextureParameter(transfer_function_texture_2D_parameter);
13 cgGLDisableTextureParameter(volume_texture_3D_parameter);
14
15 cgGLDisableProfile(supported_profile);
```

Listing 3.8: Instructions necessary to bind/unbind the fragment program depicted in figure 3.6 to the programmable fragment processing unit

Furthermore, the necessary modelview and projection matrices have to be set for computing the final image. Also, configuring the blend function used to determine how a fragment is combined with its corresponding pixel in the frame buffer must be done once per frame (see section 3.3.7). Before any geometry is rendered at all, all mandatory shaders have to be created and enabled as well. Also the shaders' uniform parameters have to be specified (see section 3.3.5). This also includes updating of the transfer function texture if it has changed. If no programmable shaders are used, i.e the fixed-function rendering pipeline is

used instead, textures have to be bound to their corresponding texture units. As all slices are rendered in either front-to-back or back-to-front order depth testing and depth buffer updating could both be disabled. However, if opaque objects are required to be rendered inside the volumetric data, the opaque objects should be rendered first with depth testing and depth buffer writing enabled. Before the volumetric data set is rendered depth buffer writing is then disabled while still leaving depth testing enabled. That way, the volumetric data is only blended in front of the opaque object, i.e where the generated fragments pass the depth test, thus obscuring the object gradually with each rendered slice. Therefore, the opaque objects appears to be inside the volumetric data.
In order to allow for a smooth integration with other graphics applications, it is important to store all states, i.e parameters, that are going to be set during execution of the volume renderer. For this case, OpenGL provides the application developer with a set of instructions, like `glPushMatrix` or `glGet`. After rendering a single frame of the volume is done the rendering state has to be restored accordingly, so that the volume rendering application does not interfere with the display of subsequent objects in the scene.

3.3.7 Configuring the Blending Function

How a fragment is combined with its corresponding pixel, that already resides in the frame buffer, is determined by the current blending mode. In addition to the configuration of the *blending function*, *alpha testing* must be configured as well in this component, if it is used during rendering. For example, *alpha testing* can be used to discard fragments that do not correspond to the current iso-surface, i.e are greater or less than the desired iso-value. Although these two steps are performed at the very end of the rendering pipeline, they must be set up before actually rendering any geometry. As this configuration usually stays the same for an entire frame, it has to be executed only once per frame as it might be changed by subsequent rendering calls. There are a number of common blending functions used in direct volume visualization:

Over: Blending slices using the over operator approximates the flow of light through a colored, semi-transparent material. The transparency of each point in the volume is determined by its associated texel's alpha value. This way, texels with a higher alpha value tend to obscure texels behind them, and stand out through obscuring texels in front of them. It can be implemented in OpenGL by setting the blending function to

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

If the color values are stored as opacity-weighted [88] or associated [14] colors, i.e each color value pre-multiplied by its corresponding opacity value, the blend function must be set to

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Attenuate The attenuate operator can be used to simulate an x-ray of the material. In this case, a texel's alpha value appears to attenuate light shining through the material along the viewing direction toward the viewer, i.e a texel's alpha value models density inside the material. The final brightness at each pixel is attenuated by the total texel density along the viewing direction. Attenuation can be implemented in OpenGL by scaling each fragment's color by its alpha value, then summing the results by blending them into the frame buffer.

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Maximum Intensity Projection , or MIP for short, is a variant of direct volume rendering, where the maximum value encountered during image composition is used to determine the final color of the corresponding pixel. If volumetric data exhibits significants amounts of noise that can make it hard to extract meaningful iso-surfaces, or define transfer functions that aid their interpretation, maximum intensity projection can be used to visualize such data sets if the important data values are higher than the common data values. In other words, MIP is an contrast enhancing operator as it finds the highest alpha value from all generated fragments at each pixel position. MIP can be implemented using OpenGL if the **`GL_EXT_blend_minmax`** is supported by the underlying graphics hardware. Graphics accelerators that are fully OpenGL v1.2 compliant always support this extension. For MIP OpenGL has to be set up as seen in

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_ONE, GL_ONE);
3 glBlendEquationEXT(GL_MAX_EXT);
```

Non-polygonal iso-surfaces can be rendered by configuring the OpenGL alpha test [86] for selection of fragments that are greater, equal, or less than the iso-value corresponding to the desired iso-surface. As not many interpolated density values are exactly equal to a given reference value it is better to compare the incoming fragment's value with the reference value by the **`GL_GREATER`** or **`GL_LESS`** operator in order to achieve a smooth surface appearance. For rendering non-polygonal iso-surfaces alpha blending must be disabled as to only display the surface. However, in order to produce a visual appealing image a per-pixel illumination model has to be applied to shade the surface's pixels accordingly. The steps necessary for rendering non-polygonal iso-surfaces with OpenGL are depicted in

```
1 glDisable(GL_BLEND);
2 glEnable(GL_ALPHA_TEST);
3 glAlphaFunc(GL_GREATER, isovalue);
```

3.3.8 Rendering the Proxy Geometry

The last component of the execution sequence for rendering volumetric data on consumer graphics hardware is getting the graphics accelerator to render geometry. Rendering geometry is what actually causes the generation of fragments in the rasterization stage of the rendering pipeline. These fragments are then shaded and blended into the frame buffer, after resampling the volumetric data set accordingly. This component has to be executed once per slice, irrespective whether 2-dimensional textures or 3-dimensional textures are used for generating each frame. However, the steps necessary to generate a final image differ from using either 2-dimensional textures or 3-dimensional textures. In the first case, object-aligned slices of proxy geometry have to be sent to the graphics hardware as opposed to view-aligned slices in the second case.

3.3.8.1 Rendering Object-Aligned Slices using 2-dimensional Textures

In this case, the proxy geometry is composed of a number, dependent on the number of slices stored in the volumetric data, of quadrangles. As there are three stacks of 2-dimensional textures necessary to render a volume (see section 3.2.2), it is also important that the slices match the different extents of each texture stack, i.e that the width and height of each drawn slice resemblances the width and height of the textures stored in the currently selected texture stack. Therefore, in order to receive a correct final image, three sets of slices with different extents have to be rendered depending on the current viewing direction. The extents of these sets of slices can easily be precomputed during the initialization of the application. Listing 3.9 shows a code example that computes the dimensions of a single slice that is repeatedly drawn to render all textures from the current texture stack. Lines 1 to 14 compute the geometry of the respective object-aligned slices, whereas lines 16 to 29 specify their respective texture coordinates.

If the the volume data had to be enlarged in order to fit into the texture objects, the texture coordinates have to be adjusted by replacing $1.0f$ with

$$\frac{original\_volume\_dimension}{enlarged\_to\_power\_of\_two\_texture\_dimension}.$$

This way only fragments resampling actual volume data are generated, thus saving computation time. Otherwise, if *non-power-of-two texture objects* are used for storing the volumetric data instead the texture coordinates have to be replaced with their respective non-normalized texture coordinates, i.e the width or height of the texture to be sampled.

```
 1 slice_x_dir[0] = -1.0f; slice_x_dir[1]  = -dim[Y]/2.0f; slice_x_dir[2]  = -dim[Z]/2.0f;
 2 slice_x_dir[3] = -1.0f; slice_x_dir[4]  = -dim[Y]/2.0f; slice_x_dir[5]  = dim[Z]/2.0f;
 3 slice_x_dir[6] = -1.0f; slice_x_dir[7]  = dim[Y]/2.0f;  slice_x_dir[8]  = dim[Z]/2.0f;
 4 slice_x_dir[9] = -1.0f; slice_x_dir[10] = dim[Y]/2.0f;  slice_x_dir[11] = -dim[Z]/2.0f;
 5
 6 slice_y_dir[0] = -dim[X]/2.0f; slice_y_dir[1]  = -1.0f; slice_y_dir[2]  = -dim[Z]/2.0f;
 7 slice_y_dir[3] = dim[X]/2.0f;  slice_y_dir[4]  = -1.0f; slice_y_dir[5]  = -dim[Z]/2.0f;
 8 slice_y_dir[6] = dim[X]/2.0f;  slice_y_dir[7]  = -1.0f; slice_y_dir[8]  = dim[Z]/2.0f;
 9 slice_y_dir[9] = -dim[X]/2.0f; slice_y_dir[10] = -1.0f; slice_y_dir[11] = dim[Z]/2.0f;
10
11 slice_z_dir[0] = -dim[X]/2.0f; slice_z_dir[1]  = -dim[Y]/2.0f; slice_z_dir[2]  = -1.0f;
12 slice_z_dir[3] = dim[X]/2.0f;  slice_z_dir[4]  = -dim[Y]/2.0f; slice_z_dir[5]  = -1.0f;
13 slice_z_dir[6] = dim[X]/2.0f;  slice_z_dir[7]  = dim[Y]/2.0f;  slice_z_dir[8]  = -1.0f;
14 slice_z_dir[9] = -dim[X]/2.0f; slice_z_dir[10] = dim[Y]/2.0f;  slice_z_dir[11] = -1.0f;
15
16 tex_coords_x_dir[0] = 0.0f;     tex_coords_x_dir[1] = 0.0f;
17 tex_coords_x_dir[2] = 1.0f;     tex_coords_x_dir[3] = 0.0f;
18 tex_coords_x_dir[4] = 1.0f;     tex_coords_x_dir[5] = 1.0f;
19 tex_coords_x_dir[6] = 0.0f;     tex_coords_x_dir[7] = 1.0f;
20
21 tex_coords_y_dir[0] = 0.0f;     tex_coords_y_dir[1] = 0.0f;
22 tex_coords_y_dir[2] = 1.0f;     tex_coords_y_dir[3] = 0.0f;
23 tex_coords_y_dir[4] = 1.0f;     tex_coords_y_dir[5] = 1.0f;
24 tex_coords_y_dir[6] = 0.0f;     tex_coords_y_dir[7] = 1.0f;
25
26 tex_coords_z_dir[0] = 0.0f;     tex_coords_z_dir[1] = 0.0f;
27 tex_coords_z_dir[2] = 1.0f;     tex_coords_z_dir[3] = 0.0f;
28 tex_coords_z_dir[4] = 1.0f;     tex_coords_z_dir[5] = 1.0f;
29 tex_coords_z_dir[6] = 0.0f;     tex_coords_z_dir[7] = 1.0f;
```

Listing 3.9: For each major axis the extents of an object-aligned quadrangle are set to match the extents of the volumetric data set along this major axis. Also texture coordinates are set accordingly.

The first step in rendering object-aligned proxy geometry is to determine which stack to display as to avoid "see-through"through the space between each slice. Therefore, the stack most perpendicular to the current viewing direction, i.e the stack whose major axis is most parallel to the current viewing direction, has to be computed. In OpenGL, the camera is orientated so that it points down the negative z-axis, i.e the viewing direction is equal to

$$\vec{v} = \begin{pmatrix} 0.0f \\ 0.0f \\ -1.0f \\ 0.0f \end{pmatrix}.$$

In order to find the stack most perpendicular to this viewing direction the dot-product between the viewing direction and each of the volume's transformed axes has to be computed. Then the maximum of the absolute values of these dot-products determines the stack whose major axis is most parallel to the current viewing direction as the dot-product's absolute

value is 1 for $k\pi \ \wedge \ k \ \in \ Z$ for normalized vectors.

$$\begin{aligned} dot_x &= \vec{x} \circ \vec{v} \\ dot_y &= \vec{y} \circ \vec{v} \\ dot_z &= \vec{z} \circ \vec{v} \end{aligned}$$

$$stack \ = \ max(|dot_x|; \ |dot_y|; \ |dot_z|)$$

The transformed volume's x-, y-, and z-axis are the current modelview matrix's first, second, and third column respectively. After selecting the slice stack, it must be rendered in one of two directions, in order to guarantee actual back-to-front rendering. That is, if a stack is viewed from the back, with respect to the stack itself, it has to be rendered in reversed order, i.e front-to-back. The code fragment depicted in listing 3.10 shows how both of these decisions could be implemented.

```
1  float dot[3];
2
3  float view_dir[3] = {0.0f, 0.0f, -1.0f};
4
5  dot[X] = modelview[0]*view_dir[X] + modelview[1]*view_dir[Y] + modelview[2]*view_dir[Z];
6  dot[Y] = modelview[4]*view_dir[X] + modelview[5]*view_dir[Y] + modelview[6]*view_dir[Z];
7  dot[Z] = modelview[8]*view_dir[X] + modelview[9]*view_dir[Y] + modelview[10]*view_dir[Z];
8
9  int selected_stack = GetAbsoluteMaximum(dot);
10
11 bool back_to_front = (dot[selected_stack] > 0.0f) ? true : false;
```

Listing 3.10: First the stack whose major axis, with respect to the stack itself, is most parallel to the current viewing direction. Then decide if the selected stack is viewed from up front or behind, thus selecting the order it has to be rendered in.

After selecting the stack for rendering each individual slice, the volume is composited of, must be send to the graphics hardware for rasterization. In order to map a texture to each slice the following steps are necessary. First, the corresponding texture must be passed to the fragment shader or, when no fragment shader is available, bound to a texture unit. Then, for each vertex of the drawn quad 2-dimensional texture coordinates, s and t respectively, must be specified. Before actually drawing any geometry either the fragment shader or the texture target has to be enabled to affect rendering of the fragments corresponding to the quad. If it has not been done in a preprocessing stage of the application, the depth coordinate of each slice, i.e a slice's position on the major axis of its stack, must be updated before sending it down the rendering pipeline. Listing 3.11 gives an example how these steps can be implemented. First, the texture object corresponding to the current slice is passed to a fragment shader, as well as the current transfer function lookup table. After the fragment shader has been enabled, each vertex of the quad is drawn with its respective 2-dimensional texture coordinate set. In the example, the quads are rendered in back-to-front order.

```
1  glLoadIdentity();
2  glLoadMatrixf(modelview);
3
4  double current_depth = -dim[selected_stack]/2;
5
6  for (int layer = layers_in_stack[selected_stack]-1; layer >= 0; layer--)
7  {
8    SetupAndEnableCgFragmentShader();
9
10   glBegin(GL_QUAD);
11   for (int i = 0; i<4; i++)
12   {
13     glTexCoord2f(tex_coords[i*2],
14                  tex_coords[i*2 + 1]);
15
16     glVertex3f(slice[i*3],
17                slice[i*3 + 1],
18                slice[i*3 + 2]);
19   }
20   glEnd();
21
22 current_depth += (dim[selected_stack] / layers_in_stack[selected_stack]);
23 }
```

Listing 3.11: After passing each rendered quad's respective texture object and the current transfer function to a fragment shader, each of its vertices is sent to the graphics accelerator together with its 2-dimensional texture coordinate in back-to-front order.

3.3.8.2 Rendering Object-Aligned Slices using 2-dimensional Textures with trilinear Interpolation

If trilinear interpolation is to be achieved using only 2-dimensional textures and object-aligned slices, minor adjustments have to be made to the technique of rendering object-aligned slices as described in section 3.3.8.1. As the number of rendered slices is not dependent on the slices stored in the volume, it can be chosen arbitrarily instead. In order to determine the slices stored in the selected texture stack that enclose the current layer, it is necessary to keep track of the layer not only in the spatial dimensions of the volume but also in the texture stack dimensions. The closest integer values to the later value determine both texture objects from the stack of 2-dimensional objects that have to be linearly interpolated, thus have to be passed to the fragment shader. Furthermore, the fractional part of its value can be used to determine both interpolation factors. Figure 3.12 shows how these values are connected. The interpolation factor for the back slice can be passed to the rendering pipeline as the r component of a 3-dimensional texture coordinate, whereas the inverted fractional part can be computed by the fragment shader itself. Listing 3.13 shows a code fragment that incorporates all mentioned adjustments necessary for back-to-front rendering, whereas listing 3.12 depicts the adjusted fragment shader.

```
1  float4 main( uniform sampler2D front_texture_2D,
2               uniform sampler2D back_texture_2D,
3               uniform sampler2D transfer_function_texture_2D,
4               float3  texture_coordinates_3D : TEXCOORD0,
5               float4 vertex_color : COLOR0
6              ):COLOR
7  {
8    float4 front_color = tex2D(front_texture_2D, texture_coordinates_3D.xy);
9
10   float4 back_color = tex2D(back_texture_2D, texture_coordinates_3D.xy);
11
12   float4 final_color = front_color * (1.0 - texture_coordinates_3D.z)
13                           + back_color  * texture_coordinates_3D.z;
14
15         final_color = tex2D(transfer, final_color.xy);
16
17   return final_color;
18 }
```

Listing 3.12: This fragment shader fetches color values from two texture objects. Then these color values are linearly interpolated.

3.3.8.3 Rendering View-Aligned Slices using 3-dimensional Textures

When the volumetric data set is stored in a 3-dimensional texture, the proxy-geometry consists of a set of view-aligned slices. These slices must be clipped against the bounding box of the volume before sending them to the graphics accelerator for rendering. Additionally, 3-dimensional texture coordinates must be generated for each vertex processed by the GPU that correspond to locations inside the volume. Otherwise, the graphics hardware could not access the volume during texture mapping. Using OpenGL there are two different methods to do this. The first method utilizes additional clipping planes and automatic texture coordinates generation provided by OpenGL, thus leaving these complex computations to the graphics, i.e OpenGL, subsystem. It is also possible to compute the clipping against the bounding box of the volume as well as the corresponding texture coordinates completely on the CPU. Both methods are presented in detail.

3.3.8.4 Utilizing Additional Clipping Planes and Automatic Texture Coordinate Generation

In this case, the proxy geometry is composed of a set of quads drawn at equidistant spaces. The extends of these quads must be at least the distance between the two points of the volume that are most distant away from each other, i.e at least the length of the diagonal of the volume's bounding box.

In addition to the six clipping planes that define the viewing frustum, all OpenGL implementations support at least six extra clipping planes that can be used for arbitrary clipping of

```
1  double current_depth = -dim[selected_stack]/2;
2  double pos_in_vol = (double)(layers_in_stack[selected_stack] - 1);
3
4  for (int layer = layers_to_render-1; layer >= 0; layer--)
5  {
6    SetupAndEnableCgFragmentShader();
7
8    glBegin(GL_QUAD);
9    for (int i = 0; i<4; i++)
10   {
11     glTexCoord2f(tex_coords[i*2],
12                  tex_coords[i*2 + 1],
13                  pos_in_vol - floor(pos_in_vol) );
14
15     glVertex3f(slice[i*3],
16                slice[i*3 + 1],
17                slice[i*3 + 2]);
18   }
19   glEnd();
20
21   current_depth += (dim[selected_stack] / layers_to_render);
22
23   pos_in_vol -= (dim[selected_stack] / layers_to_render);
24 }
```

Listing 3.13: After passing each rendered quad's both enclosing texture objects and the current transfer function to the fragment shader, each of its vertices is sent to the graphics accelerator together with its 3-dimensional texture coordinate, where the r component specifies the factor for trilinear interpolation between the two slices.

geometry. Each of the clipping planes is specified by an array of four double-precision floating point values. These values are then interpreted as the coefficients of a plane equation in normalized form. When a clipping plane is specified the coefficients of its corresponding plane equation are transformed by the inverse of the current active modelview matrix and stored in the resulting eye coordinates. Subsequent changes of the modelview matrix have no effect on the stored plane equation components. Therefore, the modelview matrix used to transform the volume must be passed to the graphics hardware prior to specifying any clipping planes for clipping the proxy geometry. When a clipping plane is enabled all subsequent drawing commands are clipped against the viewing frustum and the enabled clipping plane. A vertex is *in*, i.e is drawn, with respect to the additional clipping plane, if the dot product of its eye coordinates with the stored clipping plane equation coefficients is positive or zero. Otherwise, it is referred to being *out* and thus discarded. Figure 3.14 shows a code fragment that specifies an additional clipping plane for each side of the volume's bounding box.

$$\vec{v}_{eye} \circ \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} = \begin{cases} \geq 0, & in \\ < 0, & out \end{cases}$$

```
 1 double clip0[]  = {-1.0,  0.0,  0.0, dimension[X]/2.0};
 2 double clip1[]  = { 1.0,  0.0,  0.0, dimension[X]/2.0};
 3 double clip2[]  = { 0.0, -1.0,  0.0, dimension[Y]/2.0};
 4 double clip3[]  = { 0.0,  1.0,  0.0, dimension[Y]/2.0};
 5 double clip4[]  = { 0.0,  0.0, -1.0, dimension[Z]/2.0};
 6 double clip5[]  = { 0.0,  0.0,  1.0, dimension[Z]/2.0};
 7
 8 glClipPlane(GL_CLIP_PLANE0, clip0);
 9 glClipPlane(GL_CLIP_PLANE1, clip1);
10 glClipPlane(GL_CLIP_PLANE2, clip2);
11 glClipPlane(GL_CLIP_PLANE3, clip3);
12 glClipPlane(GL_CLIP_PLANE4, clip4);
13 glClipPlane(GL_CLIP_PLANE5, clip5);
14
15 glEnable(GL_CLIP_PLANE0);
16 glEnable(GL_CLIP_PLANE1);
17 glEnable(GL_CLIP_PLANE2);
18 glEnable(GL_CLIP_PLANE3);
19 glEnable(GL_CLIP_PLANE4);
20 glEnable(GL_CLIP_PLANE5);
```

Listing 3.14: Specify an additional clipping plane for each side of the volume's bounding box. Each additional clipping plane is configured by a **glClipPlane**(...) command. Note: All coefficients used to specify a clipping plane are multiplied by the inverse of the active modelview matrix

In order to compute the 3-dimensional texture coordinates needed for mapping the 3-dimensional texture onto the set of view-aligned slices, it is possible to exploit the texture coordinate generation mechanisms provided by the OpenGL graphics subsystem. OpenGL can be configured to automatically generate one to four of the texture coordinates s, t, r, and q. For each of these coordinates OpenGL offers different modes, such as **GL_OBJECT_LINEAR** or **GL_EYE_LINEAR**, for computing a texture coordinate from a vertex's position. For generating the s-, t-, and r-texture coordinates for volume rendering using view-aligned slices the later mode - **GL_EYE_LINEAR** - has to be utilized as it generates texture coordinates based on a vertex's position in eye space, i.e inside the viewing frustum. Therefore, for each texture coordinate a reference plane has to be specified by passing the coefficients of its corresponding plane equation in normalized form to OpenGL. These parameters are then transformed using the inverse of the current set modelview matrix and stored in the resulting eye coordinates. Subsequent instructions that modify the active modelview matrix have no effect on the parameters stored for texture coordinate generation. The function

$$g = p'_1 x_{eye} + p'_2 y_{eye} + p'_3 z_{eye} + p'_4 w_{eye}$$

where

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ p'_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} M^{-1},$$

g is the value of the computed texture coordinate, x_{eye}, y_{eye}, z_{eye}, and w_{eye} are the coordinates of the vertex in eye coordinate space, i.e view coordinate space, p_1, p_2, p_3, and p_4 are the values of the reference plane, and M is the modelview matrix when the instructions to enable automatic texture coordinate generation are invoked. The code fragment depicted in listing 3.15 shows how automatic texture coordinate generation of the s-, t-, and r-coordinate for volume rendering is configured and enabled.

```
1  float x[4] = {1.0f/dimension[X], 0.0f, 0.0f, 0.5f};
2  float y[4] = {0.0f, 1.0f/dimension[Y], 0.0f, 0.5f};
3  float z[4] = {0.0f, 0.0f, 1.0f/dimension[Z], 0.5f};
4
5  glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
6  glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
7  glTexGenf(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
8
9  glTexGenfv(GL_S, GL_EYE_PLANE, x);
10 glTexGenfv(GL_T, GL_EYE_PLANE, y);
11 glTexGenfv(GL_R, GL_EYE_PLANE, z);
12
13 glEnable(GL_TEXTURE_GEN_S);
14 glEnable(GL_TEXTURE_GEN_T);
15 glEnable(GL_TEXTURE_GEN_R);
```

Listing 3.15: Configure and enable automatic texture coordinate generation based on each vertex position in eye coordinate space.

Prior to rendering the proxy geometry, i.e the view-aligned slices, and after all additional clipping planes are configured and enabled and the s-, t-, and r-texture coordinates are configured for automatic generation the modelview matrix has to be reset. Then only its translational part must be passed to the graphics accelerator, as no rotation must affect the rendering of the view-aligned slices. By loading the complete modelview matrix before setting up the clipping planes and automatic texture coordinates generation, their parameters still do get affected by rotation. Listing 3.16 illustrates the steps necessary to render a volumetric data set with view-aligned slices by using additional clipping planes and OpenGL's automatic texture coordinate generation mechanisms.

The drawback of using the additional clipping planes for clipping the proxy geometry to the bounding box of the volumetric data set is that it limits the application of further clipping planes for arbitrary clipping of the volume as most OpenGL implementations only support the minimum of six additional clipping planes.

3.3.8.5 Computating Clipping and Texture Coordinates on the CPU

As volume rendering applications on current generation graphics accelerators are usually fill rate limited, i.e the number of pixels that can be filled in a second is too low, it is possible

```
1  glLoadIdentity();
2  glLoadMatrixf(modelview);
3
4  SetupAndEnableAdditionalClippingPlanes();
5
6  SetupAndEnableAutomaticTextureCoordinateGeneration();
7
8  glLoadIdentity();
9  glTranslatef(modelview[12], modelview[13], modelview[14]);
10
11 SetupAndEnableCgFragmentShader();
12
13 if(back_to_front)
14 {
15   float z = -diagonal/2.0f;
16
17   for (int i = 0; i < number_of_layers; i++)
18   {
19     glBegin(GL_QUADS);
20     glVertex3f(-diagonal/2.0f, -diagonal/2.0f, z);
21     glVertex3f( diagonal/2.0f, -diagonal/2.0f, z);
22     glVertex3f( diagonal/2.0f,  diagonal/2.0f, z);
23     glVertex3f(-diagonal/2.0f,  diagonal/2.0f, z);
24     glEnd();
25
26     z += diagonal/number_of_layers;
27   }
28 }
```

Listing 3.16: Instructions necessary for rendering view-aligned slices using additional clipping planes and automatic generation of texture coordinates.

to ease the burden on the GPU by computing the clipping of the view-aligned slices to the bounding box of the volume as well as the generation of texture coordinates on the host CPU. Computing clipping on the host CPU also frees the additional clipping planes for using them for arbitrary clipping of the volumetric data set. In order to generate geometry that can be sent to the graphics hardware for rendering three steps a necessary to compute in each frame.

First, load the eight corners of the volume's bounding box. As the volume is centered around the origin of the coordinate space and therefore its bounding box, too, set each of the corners' coordinates to half of the volume's extends. Then transform them to view coordinate space but only using the rotation portion of the current modelview matrix. A code example is given in listing 3.17.

Then find the minimum and maximum z coordinate of the transformed bounding box corners. If the number of sampling planes is not set, compute it from the current sampling rate between these two values using equidistant spacings. The code fragment in listing 3.18 computes the minimum and maximum z coordinate of the bounding box.

```
1  bbox_coords[0][Y] = +dimension[Y]/2.0f;
2  bbox_coords[0][Z] = -dimension[Z]/2.0f;
3
4  LoadCubeCorners();
5
6  for(int i = 0; i < 8; i++)
7  {
8    float x = modelview[0] * bbox_coords[i][0] + modelview[4] * bbox_coords[i][1]
9              + modelview[ 8] * bbox_coords[i][2];
10   float y = modelview[1] * bbox_coords[i][0] + modelview[5] * bbox_coords[i][1]
11             + modelview[ 9] * bbox_coords[i][2];
12   float z = modelview[2] * bbox_coords[i][0] + modelview[6] * bbox_coords[i][1]
13             + modelview[10] * bbox_coords[i][2];
14
15   bbox_coords[i][0] = x;
16   bbox_coords[i][1] = y;
17   bbox_coords[i][2] = z;
18 }
```

Listing 3.17: Instructions for loading a volume's bounding box's corners. After the corners are loaded they are transformed using only the rotational part of the current modelview matrix.

```
1 float min_z = bbox_coords[0][2], max_z = bbox_coords[0][2];
2
3 for (int i = 1; i < 8; i++)
4 {
5   min_z = (bbox_coords[i][2] < min_z) ? bbox_coords[i][2] : min_z;
6   max_z = (bbox_coords[i][2] > max_z) ? bbox_coords[i][2] : max_z;
7 }
```

Listing 3.18: Compute the z range of the bounding box's current orientation.

When the z range has been found, compute for each view-aligned slice in back-to-front order or front-to-back order, i.e from the minimum z coordinate to the maximum z coordinate or vice versa respectively:

1. For each depth value test for intersection of the view-aligned plane with each edge of the bounding box. Compare the depth value of the current plane with the depth values of the two corners of the bounding box that make up for the edge. No intersection point is generated if the view-aligned plane is in front of the edge, behind the edge, or parallel to it. Otherwise, interpolate between the two corner points by their depth value's proportion. If an intersection point is found, add it to a temporary list. The maximum size of this list is fixed because only up to six intersection points can be

generated.

$$
\begin{aligned}
& z > \vec{a}_z \ \wedge \ z > \vec{b}_z && \Rightarrow \text{plane is in front} \\
& z < \vec{a}_z \ \wedge \ z < \vec{b}_z && \Rightarrow \text{plane is behind} \\
& \vec{a}_z \approx \vec{b}_z && \Rightarrow \text{plane is parallel} \\
\text{otherwise} & \\
& \vec{v}_x = \vec{a}_x + \mu(\vec{a}_x - \vec{b}_x) & \\
& \vec{v}_y = \vec{a}_y + \mu(\vec{a}_y - \vec{b}_y) && \text{, with } \mu = \frac{z - \vec{a}_z}{\vec{a}_z - \vec{b}_z} \\
& \vec{v}_z = z &
\end{aligned}
$$

2. If more then two intersection points are generated, i.e if the intersection points are the corners of at least a triangle, sort these points clockwise or counter-clockwise by projecting them onto the x-y plane. Find the projected point with the minimum y value and put it to the front of the list. Then use this point as a reference point and sort the other projected points by comparing their angle with respect to the reference point. In order to avoid trigonometric computations, the gradient of the straights between the reference point and two projected points and their location with respect to the reference point, can be used for sorting the intersection points [59]. Bubblesort can be used for sorting as the number of elements to be sorted is very small and therefore bubblesort is much faster than quicksort. Listing 3.19 shows a code fragment for setting up bubblesort, whereas listing 3.20 shows the steps necessary for comparing two intersection points based on their angle with respect to a given reference point. Figures 3.15 and 3.16 further illustrate the process of comparing the angles.

```
1 swapPoints(intersection[0], intersection[findMinYComponent(intersection, count)]);
2
3 for(int i = 1; i < count; i++)
4   for(int j = 1; j < (count-i); j++)
5     if(!compareGradient(a[j], a[j+1], reference_point))
6       swapPoints(a[j], a[j+1]);
```

Listing 3.19: Setting u the bubblesort algorithm for sorting the intersection points based on their angle with respect to the reference point (Note: For few elements bubblesort is actually faster than quicksort).

3. When the sorting is finished the resulting vertices can be sent down to the graphics hardware by using OpenGL's polygonal primitive **`GL_POLYGON`**. The polygon is automatically tessellated into triangles by the GPU during rendering.

When computing the geometry of the view-aligned slices on the host CPU, i.e computing the clipping in software, texture coordinates can be generated using either OpenGL's automatic texture coordinate generation mechanisms, the OpenGL texture matrix, a vertex shader, or the host CPU. Informations about OpenGL's automatic texture coordinate generation mechanisms can be found in section 3.3.8.4. On the host CPU each intersection point

```
1  float ma1 = (a1[Y]-reference_point[Y])/(a1[X]-reference_point[X]);
2  float ma2 = (a2[Y]-reference_point[Y])/(a2[X]-reference_point[X]);
3
4  if(approxEqual(a[X], reference_point[X]))
5    return ( (ma2 < 0.0f) || (approxEqual(ma2, 0.0f) && a2[X] < reference_point[X]) ); //(*)
6  if(approxEqual(a2[0], reference_point[X]))
7    return ( (ma1 > 0.0f) || (approxEqual(ma1, 0.0f) && a1[X] > reference_point[X]) ); //(**)
8
9  if(a1[X] < reference_point[X])
10   if(a2[X] < reference_point[X])
11     return (ma1 < ma2);
12   else
13     return false;
14 else
15   if(a2[X] < reference_point[X])
16     return true;
17   else
18     return (ma1 < ma2);
```

Listing 3.20: This code fragment compares two points based on their location with respect to a reference point and their gradients.

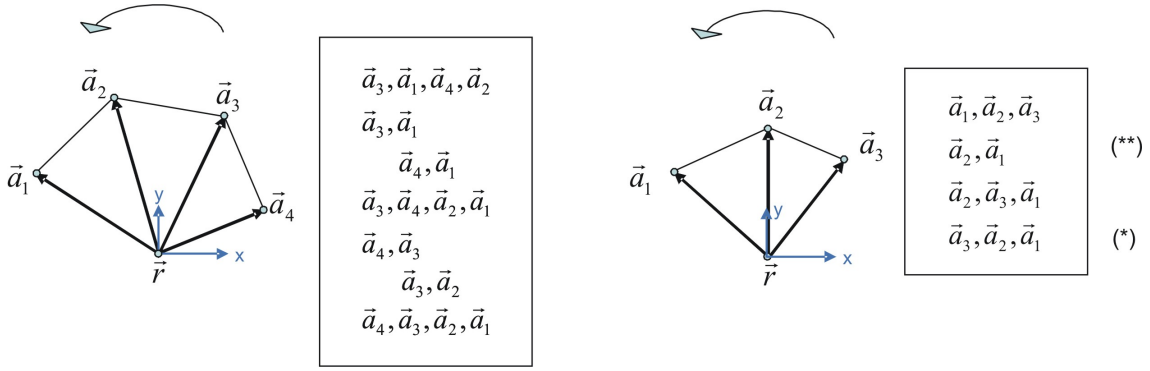


Figure 3.15: Illustrates the sorting of the intersection points.

of a view-aligned plane with the volume's bounding box computed above is first treated as a 3-dimensional texture coordinate. This texture coordinate must then be transformed back from view coordinate space to object coordinate space in order to get the texture coordinate's position inside the volume. Therefore, each point has to be transformed by the inverse rotation matrix, used to transform the volume's bounding box from object space to view space, and translated by $\left(\frac{dim_x}{2}; \frac{dim_y}{2}; \frac{dim_z}{2}\right)^T$. As texture coordinates are specified in the range of $[0;1]$ in OpenGL, each component of the resulting texture coordinate must be divided by the corresponding dimension of the 3-dimensional texture. If the volume had to be enlarged in order to fit into a 3-dimensional texture the texture coordinates can be scaled based on the ratio between each dimension of the volumetric data set and the corresponding dimension of the texture, prior to sending the texture coordinate to the graphics hardware along with

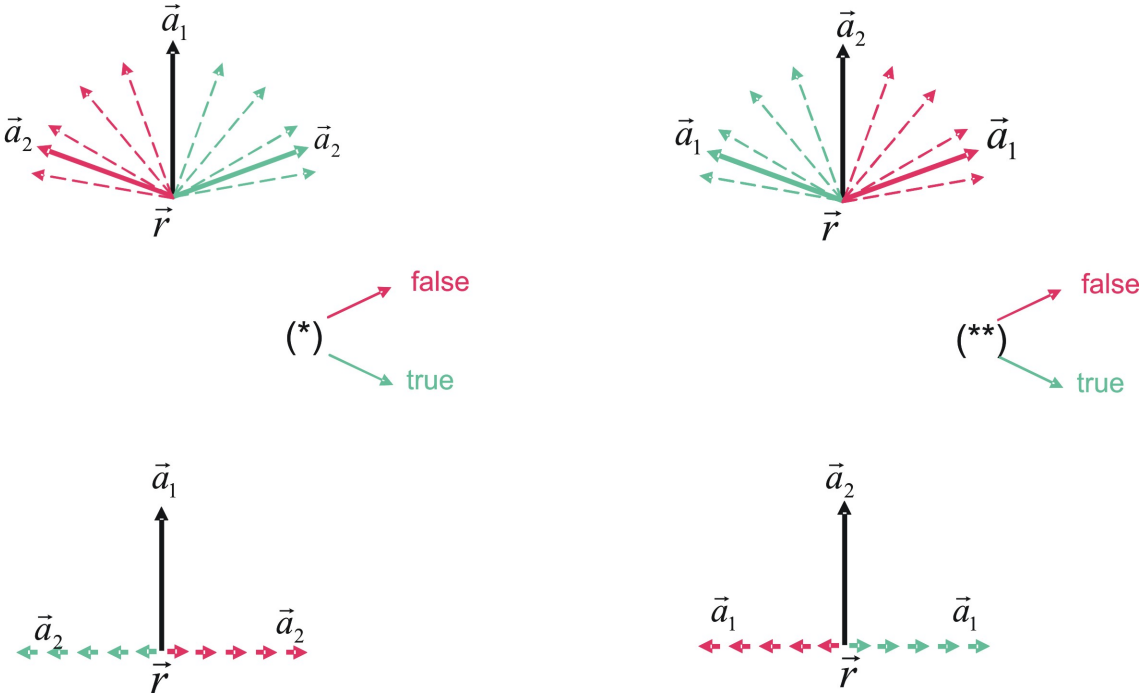


Figure 3.16: Illustrates the special cases, marked with $*$ and $**$ respectively, for comparing two intersection points.

its associated vertex. These same steps can also be computed using the OpenGL texture matrix or a vertex shader. Listing 3.21 shows a code fragment that transforms an intersection point into its corresponding 3-dimensional texture coordinate prior to sending both to the graphics hardware. All steps necessary to render a volumetric data set with view-aligned slices by computing both clipping and texture coordinate generation on the host CPU are illustrated in complete order in listing 3.22.

```
1 float texCoord[3] = {intersection[i][X], intersection[i][Y], intersection[i][Z]};
2
3 float x = inv_mv[0]*texCoord[X] + inv_mv[4]*texCoord[Y] + inv_mv[ 8]*texCoord[Z];
4 float y = inv_mv[1]*texCoord[X] + inv_mv[5]*texCoord[Y] + inv_mv[ 9]*texCoord[Z];
5 float z = inv_mv[2]*texCoord[X] + inv_mv[6]*texCoord[Y] + inv_mv[10]*texCoord[Z];
6
7 texCoord[X] = ( x + dimension[X]/2 ) / dimension[X];
8 texCoord[Y] = ( y + dimension[Y]/2 ) / dimension[Y];
9 texCoord[Z] = ( z + dimension[Z]/2 ) / dimension[Z];
```

Listing 3.21: In order to transform an intersection point into a 3-dimensional texture coordinate that can be used to map the volume onto the view-aligned slice, the texture coordinates have to be transformed into object space by multiplying them with the rotation part of the inverse modelview matrix used to transform the volume's bounding box. Then the texture coordinates have to be computed in the range $[0; 1]$.

```
1  SetupAndRotateCubeCorners();
2
3  computeZRange();
4
5  SetupAndEnableCgFragmentShader();
6
7  for(float depth = min_z; depth < max_z; depth += sampling_distance)
8  {
9    int count = 0;
10
11   computeViewAlignedLayer(depth, count, intersection);
12
13   if(count > 2)
14   {
15     glLoadIdentity();
16     glTranslatef(modelview[12], modelview[13], modelview[14]);
17
18     glBegin(GL_POLYGON);
19     for (int p = 0; p < count; p++)
20     {
21       computeTextureCordinate(intersection[p], texCoord);
22
23       glTexCoord3f(texCoord[X],
24                    texCoord[Y],
25                    texCoord[Z]);
26
27       glVertex3fv(intersection[p]);
28     }
29   }
30 }
```

Listing 3.22: Instructions necessary for rendering view-aligned slices computing both clipping and generation of texture coordinates on the host CPU.

4 CPU-based Techniques for Volume Rendering

Before using consumer graphics accelerators or even specifically tailored graphics hardware for rendering volumetric data sets, such data sets were visualized using general purpose processors, i.e CPUs. As mentioned in section 2.3, there are two distinct basic approaches for visualizing volumetric data sets, that is indirect volume rendering and direct volume rendering. Both can be adopted using general purpose processors. However, in order to visualize all characteristics contained in a volumetric data set, that is the principal goal of medical imaging applications, the whole information of a voxel is needed. Therefore, only direct volume rendering techniques are apt to visualize volumes containing medical data.
Three techniques have emerged as particularly popular in direct volume rendering on CPUs: *Ray-Casting* [46, 83], *Splatting* [87] and *Shear-Warp* [40]. Ray-Casting is an *image-order* approach as it computes the volume rendering integral for each pixel in the final image. In contrast, splatting evaluates the volume rendering integral by projecting each voxel contained inside the volume onto the image plane, thus it is deemed an *object-order* approach. Both techniques deliver the most accurate solution of the volume rendering integral, but not without a drawback: Early implementations of Ray-Casting and splatting are not capable of displaying volumes at interactive rates, due to their high computational costs. On the contrary, shear-warp offers high performance rendering, that can be used to illustrate the volume interactively, at the cost of an inferior image quality of the rendered image. Therefore, many researchers have worked independently on refining these techniques over the years.
These software-only approaches are still the subject of further research as they can be executed on every computer due to the fact, that they do not need dedicated hardware for rendering volumes.
In this chapter I will describe a basic approach for casting rays into a volumetric data set, as well as present a novel approach for accelerating Ray-Casting by pre-integrating rays inside the volume.

4.1 Ray-Casting - A straight-forward Approach to Volume Rendering

Ray-Casting is considered the most straight-forward approach to evaluate the volume rendering integral, when a simple optical model, such as absorption only, emission only, or absorption plus emission (section 2.4), is used as the underlying model for direct volume rendering. In addition to direct volume rendering, Ray-Casting can also be used to display non-ploygonal issurfaces using a special transfer function [46]. For direct volume rendering Ray-Casting allows for different types of classification, primarily pre-classification [46, 47]

or post-classification [11, 32, 82, 81].
For each pixel of the final image, a ray is cast from the ray source through this particular point of the image plane into the scene. If a ray intersects the volumetric data set, the volume is resampled along this ray via point sampling. Then these sampling points are reconstructed from the discrete volume using a particular reconstruction filter and their values are accumulated in the frame buffer at the current ray's corresponding pixel position. The whole process is illustrated in figure 4.1

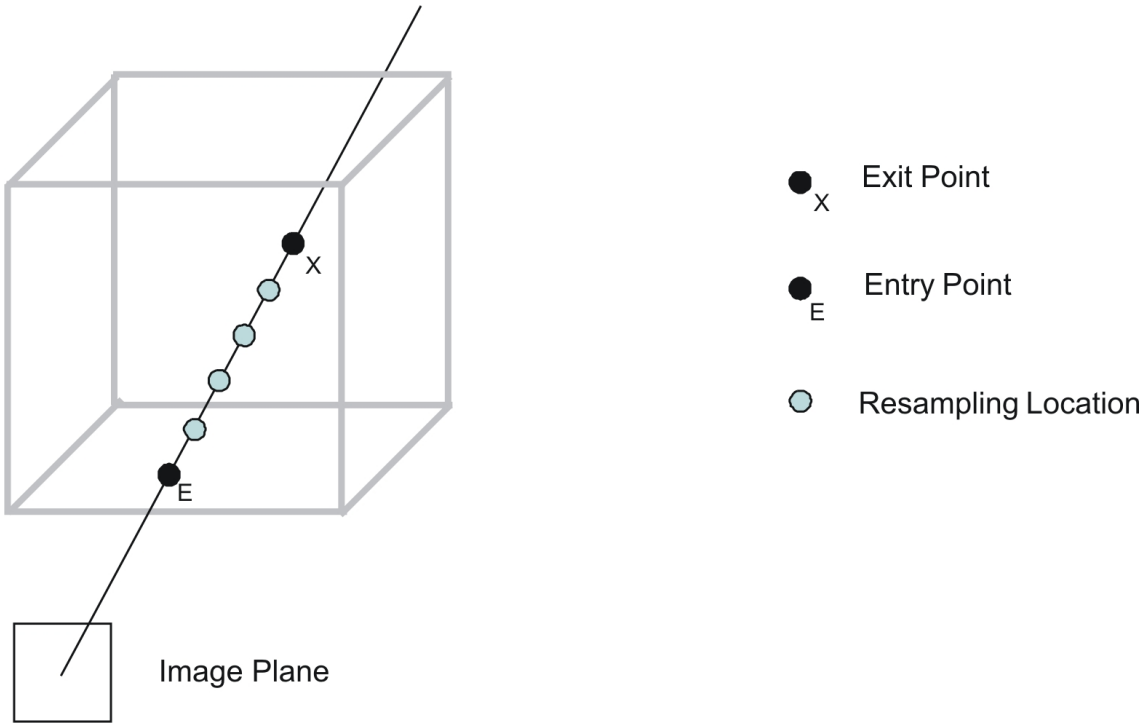


Figure 4.1: Ray-Casting performed for a single ray.

4.1.1 Computing the Entry- and Exit-Points of a Ray

A ray cast into the scene is defined by

$$\vec{r} = \vec{s} + \lambda \vec{d}\,, \tag{4.1}$$

where $\vec{s}$ denotes the source of all rays cast into the scene, i.e the camera center, and $\vec{d}$ describes the direction of the current. In order to traverse a ray inside the volumetric data set, first its entry point and exit point, i.e the corresponding values λ_{entry} and λ_{exit} of the ray equation 4.1, have to be computed respectively.

The ray equation 4.1 can be reformed to

$$\lambda_x = \frac{r_x - s_x}{d_x} \quad \lambda_y = \frac{r_y - s_y}{d_y} \quad \lambda_z = \frac{r_z - s_z}{d_z} \tag{4.2}$$

When the volumetric data set is assumed to be centered about the origin of the world coordinate system, its boundaries are denoted by $(w/2, h/2, d/2)^T$ and $(-w/2, -h/2, -d/2)^T$. Then λ_{entry} and λ_{exit} can be determined by evaluating the equations 4.2 with $\vec{r} = (w/2, h/2, d/2)^T$ and $\vec{r} = (-w/2, -h/2, -d/2)^T$ respectively. The results are compared by pairs and exchanged so that the possible candidates for λ_{entry} consist of the lower values, whereas the candidates for λ_{exit} consist of the higher ones. λ_{entry} and λ_{exit} are then determined by the maximum and minimum of the given candidates, respectively. An example for the 2-dimensional case is depicted in figure 4.2

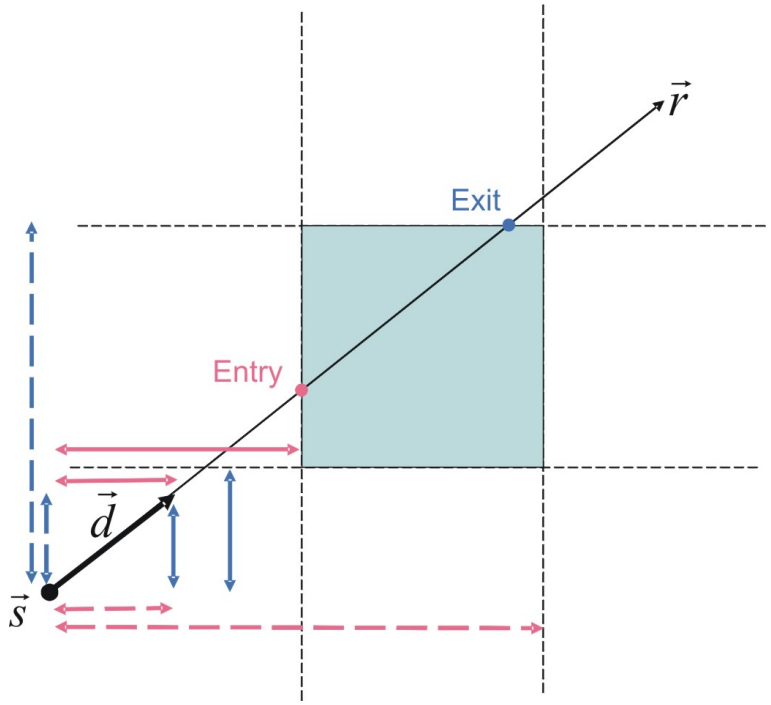


Figure 4.2: The entry point end exit point of a ray cast into the volume can be determined by computing the ratio between the distance to from the ray source to the boundaries of the volume and the direction of the ray in each dimension.

4.1.2 Reconstruction of the Volumetric Data Set

As most resampling locations do not match the exact position of an existing voxel of the volumetric data set, a reconstruction step has to be performed in order to obtain the continuous value at this particular position inside the volume. Any 3-dimensional interpolation technique can be used to reconstruct a value from the volume. Most commonly a trilinear filter is used for point sampling, as it represents a good trade-off between computation performance and resampling quality. The interpolated value is determined based on a weighted average of eight existing values in the nearest $2 \times 2 \times 2$ neighborhood of the desired resampling location. Trilinear interpolation of the different resampling locations denoted by their position

x, y, z can be computed using the following equation

$$\begin{aligned} trilinear(x,y,z) &= access(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor) * (1-x_f) * (1-y_f) * (1-z_f) \\ &+ access(\lceil x \rceil, \lfloor y \rfloor, \lfloor z \rfloor) * x_f * (1-y_f) * (1-z_f) \\ &+ access(\lfloor x \rfloor, \lceil y \rceil, \lfloor z \rfloor) * (1-x_f) * y_f * (1-z_f) \\ &+ access(\lfloor x \rfloor, \lfloor y \rfloor, \lceil z \rceil) * (1-x_f) * (1-y_f) * z_f \\ &+ access(\lceil x \rceil, \lfloor y \rfloor, \lceil z \rceil) * x_f * (1-y_f) * z_f \\ &+ access(\lfloor x \rfloor, \lceil y \rceil, \lceil z \rceil) * (1-x_f) * y_f * z_f \\ &+ access(\lceil x \rceil, \lceil y \rceil, \lfloor z \rfloor) * x_f * y_f * (1-z_f) \\ &+ access(\lceil x \rceil, \lceil y \rceil, \lceil z \rceil) * x_f * y_f * z_f \end{aligned} , \tag{4.3}$$

where x_f, y_f, z_f denote the fractional part of the considered resampling location's position defined by

$$\begin{aligned} x_f &= x - \lfloor x \rfloor \\ y_f &= y - \lfloor y \rfloor \\ z_f &= z - \lfloor z \rfloor \end{aligned} \tag{4.4}$$

Depending on the type of classification employed, the transfer function is evaluated either before or after resampling a location on the ray. Usually, the sampling locations along a ray are spaced apart in equal distances $\triangle s$, but some approaches exist that jitter the sampling locations to eliminate patterned sampling artifacts [19]. Then, each computed resampling value is subsequently composited in order to calculated the final value of the image plane's corresponding pixel, thus numerically evaluating the volume rendering integral.

4.1.3 Optimizing Ray-Casting

Ray-Casting yields images that are of very high quality, but, in its most basic implementations, takes several seconds to generate these on current x86 based CPUs. Therefore, many approaches have been made to speed up the rendering of volumetric data sets using Ray-Casting. Listing all of these is beyond the scope of this document, but a few examples should be mentioned:
Early Ray Termination stops compositing of further sample locations if the difference of subsequent compositing steps is considered marginal, i.e is below a certain threshold. Another technique uses spatial information about the contents of a volumetric data set to adapt the distance between two successive resampling locations along a ray based on this information. That way, the sampling distance is increased to not compute the reconstruction inside empty regions of the volumetric data set. Therefore, this technique is commonly referred to as *Empty Space Skipping* or *Empty Space Leaping* [90]. Both methods allow a speed-up of rendering times without sacrificing image quality. However, the rendering times can be decreased even further, if lesser image quality or lesser interaction, i.e fewer viewing directions, are deemed sufficient. One such approach uses a data structure called *transgraph*, thus its name *Volume Rendering using Transgraph* [42]. In a preprocessing step two coordinate planes are used to cast rays into the volume. The first coordinate plane can be considered a ray source plane due to the fact that each point in this plane corresponds to a certain source of rays used in the preprocessing step. For each of these points a complete Ray-Casting step is performed with the second coordinate plane used as the image plane. Indexed by the 2-dimensional coordinates of the current ray's source and the 2-dimensional coordinates of

its corresponding point in the image plane, the obtained values are then stored inside the transgraph. Therefore, the transgraph can be considered a 4-dimensional database of precomputed rays. During rendering of the volumetric data set the values in the transgraph are quadrilinearly interpolated in order to generate arbitrary viewing directions. In general, the precomputation is not limited to Ray-Casting, but can be done using all known direct volume rendering techniques. However, one drawback of this approach is, that only a limited number of viewing directions can be displayed, i.e only views whose projection center lies within the boundaries of the first coordinate plane can be reconstructed. Section 4.2 describes a novel approach to increase the rendering performance of a software based Ray-Casting volume renderer

Details on how to implement a basic implementation of a volume renderer application based on Ray-Casting can be found in section 4.3.2.

4.2 Fragmented Line Ray-Casting

In order to decrease rendering times of a volume rendering application, that is based on ray-casting, one has to understand where the high computational costs of ray-casting actually accrue from. To begin with, the recurring reconstruction that must be done to obtain a certain sampling location's value from the volumetric data set might not be too computational intensive but its frequency of occurrence along a single ray places quite a burden on the underlying processing unit. For example, the most commonly used resampling filter for volume rendering, trilinear interpolation, requires eight memory access operations, seven additions, and twenty four multiplications for a single invocation. This resampling function then has to be called $(|\vec{r}| / \triangle s)$ times, where $|\vec{r}|$ is the length of segment of the current ray that lies within the volumetric data set, and $\triangle s$ denotes the distance between two successive resampling locations. Therefore, to evaluate the discretized volume rendering integral (equation 2.16) for a single ray cast into the volume

$$\sum_{0}^{|\vec{r}|/\triangle s} n_{MEM} + n_{MUL} + n_{ADD}$$

operations have to be performed by the processing unit, where n_{MEM} denotes he number of memory access operations, n_{MUL} the number of multiplications, and n_{ADD} the number of additions necessary to reconstruct the volumetric data set at a single resampling location. These reconstruction steps have then to be performed for each ray that is cast into the scene. Therefore, in order to compute every ray that is part of the rendering process of a single frame,

$$\sum_{0}^{w*h} \sum_{0}^{|\vec{r}|/\triangle s} n_{MEM} + n_{MUL} + n_{ADD} \tag{4.5}$$

operations have to be performed, where w and h denote the width and height of the current image plane, i.e frame buffer, respectively. Though some rays may not intersect with the volume at all, this still places quite a burden on the processing unit.

Possible starting points for optimizing the rendering speed of ray-casting applications arise from equation 4.5. To lower the number of operations necessary for computing a single frame, first, the number of rays cast into a scene could be lowered. In order to obtain an image with the original resolution all missing pixels would have to be computed through interpolation of the existing ones. Another approach could consist of decreasing the number of operations necessary to successfully reconstruct the volumetric data set at each resampling location. In general, this would involve choosing a less complex reconstruction filter. Lastly, the distance between two successive resampling locations, i.e the sampling distance, could be increased, thereby decreasing the number of actual locations used for volume reconstruction. However, it is worth mentioning, that incorporating any of these optimization approaches usually tends to result in generated images of less quality compared to an unoptimized ray-casting volume renderer.

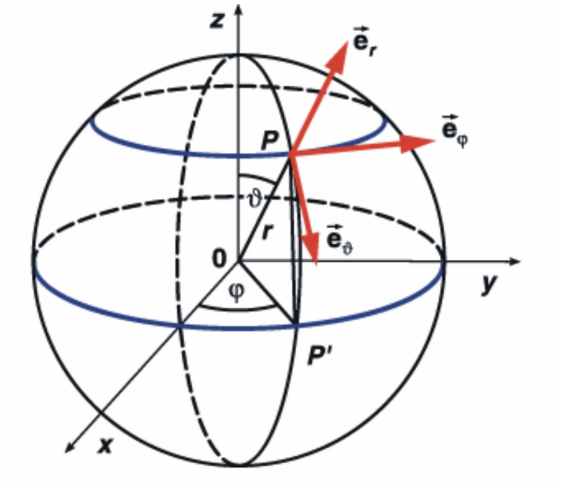
The novel approach described here tries to decrease the cost necessary for reconstruction of a volumetric data set. The speed-up is achieved by solving the discretized volume rendering integral (equation 2.16) for small ray segments inside the volume prior to its rendering, thereby offloading the major workload of reconstruction of the volumetric data set from the online rendering component to an offline precomputation step of the application. These fragmented ray segments are then used to approximate each ray that is cast into the volume during rendering. From now on, this technique is therefore referred to as *fragmented line ray-casting*. Strictly speaking, an implementation of this technique consists of two parts. The first part is a precomputation step, that is necessary to compute all fragmented ray segments inside the volume and then stores their computed values in an efficient data structure. The second part is the step performed during rendering of the volumetric data set, that reconstructs each ray from the precomputed fragmented ray segments.

4.2.1 Precomputing Fragmented Line Integrals

In order to evaluate the volume rendering integral for the fragmented ray segments, first, the fragmented ray segments have to be generated, i.e defined. Therefore, the volumetric data set is subdivided into regions of equal dimensions c_x, c_y, and c_z, the width, height, and depth respectively. Then multiple rays are cast through the center of each fragmented volume. In order to cover up for multiple viewing directions, the fragmented volumes have to be reconstructed using many possible ray directions. Therefore, discretized spherical coordinates, i.e the continuous domains of ϑ and φ are subdivided into res_ϑ and res_φ number of equally distanced angles, respectively, are used to determine the direction of each ray. An overview of spherical coordinates and their respective domains can be found in figure 4.3.

Due to the fact, that the reconstruction of the volumetric data is performed on a regular grid, i.e on Cartesian coordinates, the direction given in spherical coordinates has to be transformed into its corresponding Cartesian representation. Using equations 4.6 and equations 4.7 a vector $\vec{v}_{sphere}(r;\vartheta;\varphi)$ in spherical coordinates can be transformed into its corresponding vector $\vec{v}_{Cart}(x;y;z)$ in Cartesian coordinates and vice versa, respectively.

$$x = r\cos\varphi\sin\vartheta \quad y = r\sin\varphi\sin\vartheta \quad z = r\cos\vartheta \tag{4.6}$$



$$r \geq 0$$
$$0 \leq \vartheta \leq \pi$$
$$-\pi \leq \varphi < \pi$$

Figure 4.3: Spherical Coordinates and their respective domains

$$r = \sqrt{x^2 + y^2 + z^2} \quad \varphi = \arctan \frac{y}{x} \quad \vartheta = \arccos \frac{z}{r} \tag{4.7}$$

It is worth mentioning that it is sufficient to use only half of the domain of φ for precomputing the direction of the fragmented rays as the the other half can easily be obtained by rotating the direction vector.

For each fragmented volume the reconstruction of the volumetric data set is then performed for all rays. The emission point of a ray is the center of the fragmented volume, whereas its direction is determined by each of the discretized spherical coordinates as described above. For each ray the fragmented volume rendering integral

$$\int_{-l/2}^{l/2} c(f(\vec{x}(t)))\, e^{-\int_0^t \tau(f(\vec{x}(s)))ds}\, dt\ ,$$

where l denotes the length of the current ray inside the fragmented volume, is approximated by resampling the fragmented volume at equispaced locations. As this reconstruction is done offline, i.e not during rendering of the volume, a high-order reconstruction filter, like the tri-cubic filter, can be used instead of trilinear interpolation.

Figure 4.4 (1) illustrates the final precomputation result that is obtained when the reconstruction is performed as described above. However, it also points out a problem that can occur when the fragmented volumes used for computing the fragmented lines do not overlap. In this case a ray intersecting the volumetric data set along a border of fragmented volumes can not be reconstructed without an error. In order to lessen the impact of this error, the fragmented volumes have to overlap each other to a certain degree. Additionally, to decrease the error of reconstructing rays that intersect the volume near its border, the fragmented volumes should overlap the volume's border as well. Figure 4.4 (2) depicts fragmented volumes, each overlapping each other half their size in each dimension and the volume's border. It is important to note, that the number of fragmented volumes, that have

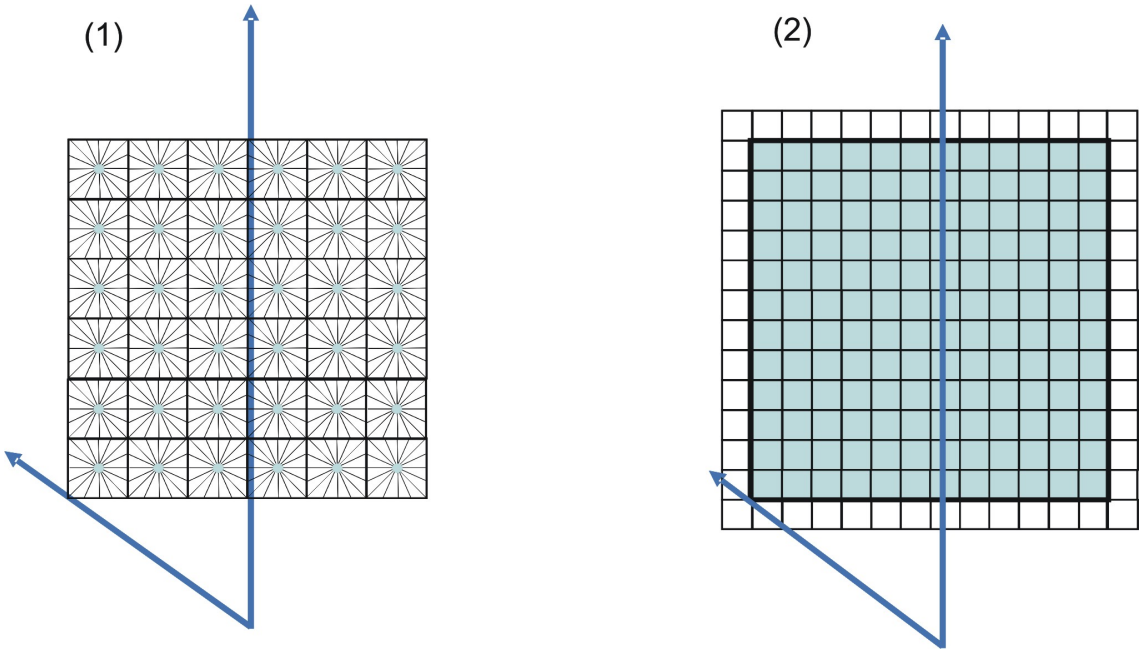


Figure 4.4: Precomputing fragmented lines through the each fragmented volume's center can lead to reconstruction problems when the fragmented volumes do not overlap (1). When the fragmented lines are computed with overlapping fragmented volumes the problems vanishes (2).

to be precomputed, rises with increasing overlap among them. The actual number of fragmented volumes in each dimension is determined by equations 4.8, 4.9, and 4.10 respectively

$$n_x = v_x/(c_x - o_x) + 1 \tag{4.8}$$
$$n_y = v_y/(c_y - o_y) + 1 \tag{4.9}$$
$$n_z = v_z/(c_z - o_z) + 1\,, \tag{4.10}$$

where v_i denotes the resolution of the volumetric data set in the specified direction, c_i denotes the dimension of a single fragmented volume used for precomputation, and o_i denotes the number of voxels that overlap between two neighboring fragmented volumes in the corresponding dimension. Each amount has to be increased by 1 in order to make sure that the fragmented volumes overlap the volume's border as well.

In order to make a prediction about the computational time this precomputation of fragmented rays may take, it is important to look at the number of operations that need to be executed for reconstructing the volumetric data set. This number can be estimated by the following equation

$$n_x * n_y * n_z * res_\vartheta * res_\varphi * avg\,(n_{OPS})\ ,$$

where n_i describes the number of fragmented volumes in each major axis, res_i denotes the number of precomputed directions within each fragmented volume, and $avg\,(n_{OPS})$ de-

scribes the average number of operations necessary to successfully reconstruct a particular fragmented line. For example, a volumetric data set with a size of $512 \times 512 \times 209$, a fragmented volume's size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$ needs about

$$257 * 257 * 105 * 90 * 90 * avg\,(n_{OPS}) \;=\; 56,174,674,500 * avg\,(n_{OPS})$$

operations, when precomputing 90 discrete angles for each spherical coordinate. Depending on the reconstruction filter used for traversing the fragmented lines, the necessary operations include a certain amount of memory access operations and numerical operations, respectively. Therefore, one can assume that the precomputation might take an awfully long time to finish.

In order to optimize precomputation speeds, it is possible to distribute the precomputation to several general processing units. Like ray-casting, the precomputation described here is highly parallel. However, as the precomputation is done offline, a high computational time is not necessarily considered a reason for refusal of a technique.

4.2.2 Storing Fragmented Line Integrals

If the precomputation is carried out like described in section 4.2.1, each fragmented ray can be uniquely identified by its fragmented volume's position the ray is derived from as well as its direction. When using spherical coordinates for describing the direction of a ray, only two degrees of freedom for the direction of the ray have to be stored, because the radius can be assumed equal to one. Additionally, a fragmented ray's position inside the volume has three degrees of freedom. Therefore, two different nested data structures could be used to store the precomputed fragmented rays.

position-first: In this case, the top-level data structure consists of a 3-dimensional array. The indices of this array correspond to the coordinates of each fragmented volume's position. Each element in this array is itself a pointer to a 2-dimensional array. The indices of these second-level arrays correspond to the precomputed fragmented ray's direction in spherical coordinates. Each element of a particular sub-array is a numerical value representing the accumulated attenuation and intensity along a particular fragmented line. In general, these are implemented linearized using the following equations

$$\begin{array}{rcl} idx_{pos} & = & (z * n_y + y) * n_x + x \\ idx_{dir} & = & \vartheta * res_{\varphi} + \varphi \end{array} \quad , \tag{4.11}$$

where x, y, z denote the position of the fragmented volume, ϑ, φ are the fragmented line's direction in spherical coordinates, and n_x, n_y are the number of precomputed fragmented volumes in x- and y-direction, respectively, for linearizing. In other words, this data structure stores the second-level array elements, i.e the precomputed values of the fragmented rays indexed by spherical coordinates, close to each other. Whereas accessing adjacent fragmented volumes is more costly as they are stored much farther apart. Therefore, this data structure is better suited if a ray cast into the volume is reconstructed using interpolation between the spherical coordinates.

direction-first: In this case, the top-level data structure consists of a 2-dimensional array. The indices of this array correspond to the spherical coordinates of each fragmented line's direction. Each element in this array is itself a pointer to a 3-dimensional array. The indices of these second-level arrays correspond to the precomputed fragmented volumes' position. Each element of a certain second-level array is a numerical value representing the accumulated attenuation and intensity along a particular fragmented line. Usually, all arrays are implemented linearized using the following equations

$$\begin{aligned} idx_{dir} &= \vartheta * res_{\varphi} + \varphi \\ idx_{pos} &= (z * n_y + y) * n_x + x \end{aligned} \quad , \tag{4.12}$$

where ϑ, φ are the fragmented line's direction in spherical coordinates, x, y, z denote the position of the fragmented volume, and n_x, n_y are the number of precomputed fragmented volumes in x- and y-direction, respectively. In this case, the values corresponding to a single direction of adjacent fragmented volumes are stored next to each other. Whereas adjacent spherical coordinates are stored much farther apart. Therefore, this data structure is better suited for reconstructing rays cast into the volume using interpolation between adjacent fragmented volumes.

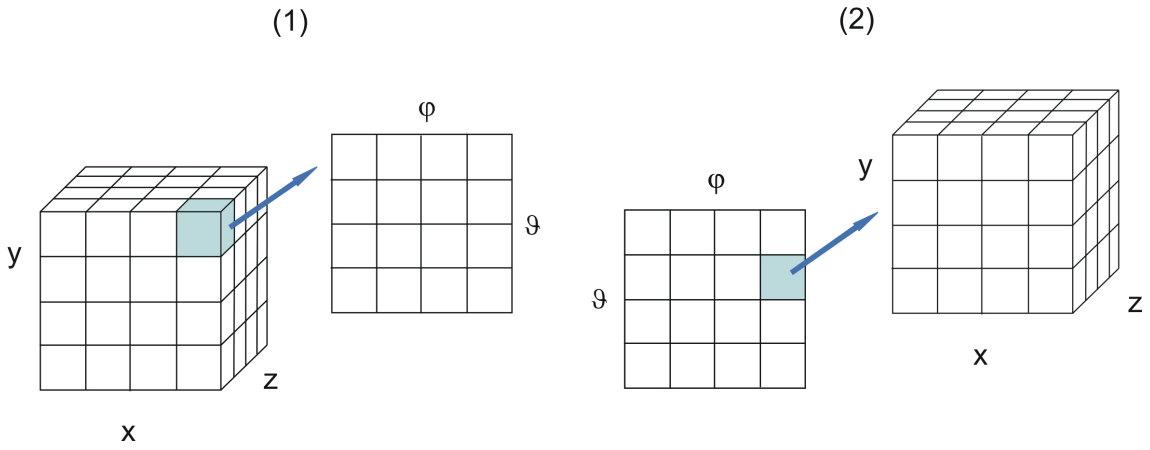


Figure 4.5: Illustration of the position-first (1) and the direction-first (2) approach for storing the precomputed values of each fragmented line.

Both approaches for storing the precomputed values of the fragmented lines are illustrated in figure 4.5. As a matter of fact, the position-first as well as the direction-first approach can be linearized completely, thus obsoleting the storage of pointers to each sub-level array, by using the following linearizing equations respectively.

$$\begin{aligned} idx &= (((z * n_y + y) * n_x + x) * res_{\vartheta} + \vartheta) * res_{\varphi} + \varphi \\ idx &= (((\vartheta * res_{\varphi} + \varphi) * n_z + z) * n_y + y) * n_x + x \end{aligned}$$

It is important to note, that regardless of the reconstruction filter being used, a direction-first data structures provides a better overall performance for reconstruction rays cast into the volume. This is accounted for by the fact that traversing along a particular ray is done by looking-up the precomputed values in one or several 3-dimensional second-level arrays,

i.e memory locations accessed during reconstruction are in close proximity opposed to a position-first data structure where the precomputed values corresponding to a particular direction are stored much farther apart.

The following equation determines the memory consumption that is necessary for storing the values of the precomputed fragmented lines:

$$n_x * n_y * n_z * res_\vartheta * res_\varphi * sizeof\,(type)$$

n_i describes the number of fragmented volumes in each major axis, res_i denotes the number of precomputed directions within each fragmented volume, and $sizeof\,(type)$ describes the number of bytes necessary to store the reconstruct value of a particular fragmented line without a loss in precision. The example given at the end of section 4.2.1, a volume with a resolution of $512 \times 512 \times 209$, a fragmented volume's size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$, and 90 discrete angles for each spherical coordinate, the fully precomputed data structure consumes about

$$257 * 257 * 105 * 90 * 90 * sizeof\,(float) \;\approx\; 214,289MB\;,$$

when the desired precision of the precomputed values is 32bit float. This clearly exceeds the available memory on even high-end workstations. Therefore, reducing the memory cost of the precomputed data has to be considered an essential part of the precomputation step of the application.

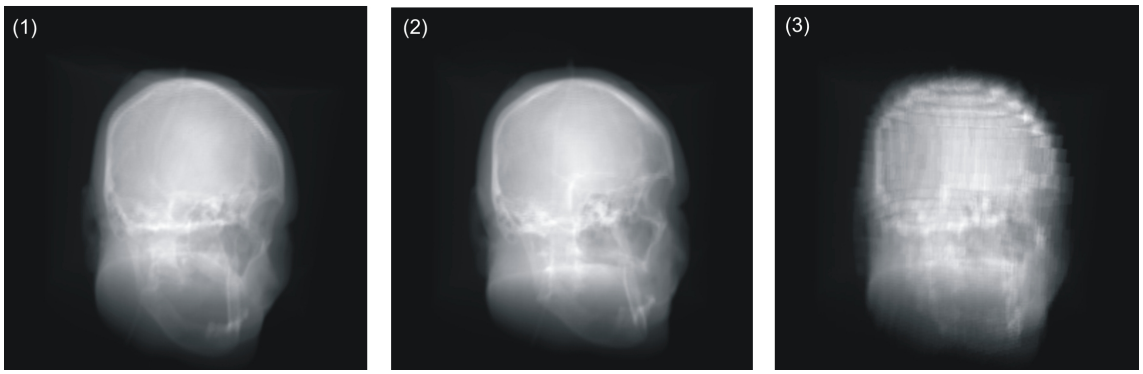


Figure 4.6: All pictures were generated using the same volume. (1) was further processed using 90 steps for parameterizing both spherical coordinates and a fragmented volume size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$. (2) used 64 steps for both spherical coordinates and a fragmented volume size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$ as parameters for the precomputation. (3) was precomputed using 90 steps as parameters for both spherical coordinates and a fragmented volume size of $20 \times 20 \times 20$ that overlap each other by $10 \times 10 \times 10$.

In order to circumvent this particular problem, it is possible to either enlarge the fragmented volumes or decrease the number of possible direction. However, as figure 4.6 clearly points out, this approach is noticeable in the quality of the rendered image. Therefore, compressing the precomputed data structure has to be the option of choice. When using

a small size for the fragmented volumes, it can be assumed that the standard deviation of the precomputed fragmented lines' values is quite low. This fact can be used to develop a simple, but efficient lossy-compression for the precomputed data. Basically, a slightly modified version of the position-first data structure, as described above, is used, to store the precomputed fragmented line integrals. The top-level data structure still consists of a 3-dimensional array. Instead of storing only a pointer to a particular second-level array in each element, the mean value of the fragmented volume's corresponding line integrals as well as a pointer to its corresponding second-level array is stored in each element. At the first glance this increases the memory demands of the data structure even more, as a particular value has to be stored in addition to the pointer to a sub-level 2-dimensional array. At closer inspection the combination of a stored mean value and a pointer aids in decreasing the memory consumption of the whole data structure, albeit less precise data. The modified data structure is depicted in figure 4.7.

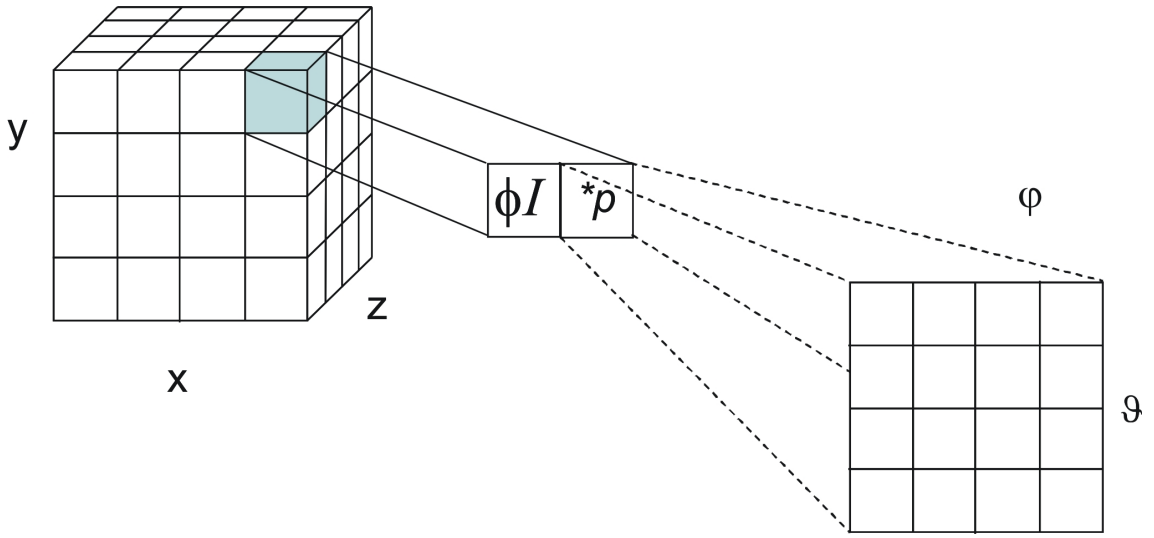


Figure 4.7: Illustrates the modified position-first data structure. For each element of the top-level 3-dimensional array, the mean value of the precomputed fragmented lines or a pointer to the corresponding second-level 2-dimensional array is stored.

For each fragmented volume the fragmented line integrals are evaluated normally in each direction. Then, the standard deviation of the fragmented line integrals is computed using equation 4.13. If the standard deviation is below a certain threshold, the mean value of the fragmented line integrals is computed using equation 4.14 and stored in the top-level array. Additionally, the second-level array is discarded and its respective pointer is set to `NULL`. In case that the standard deviation is greater or equal to the given threshold the precomputed values of the fragmented volume is retained and their corresponding memory address is stored in the pointer of the top-level array. The mean value is set to an unlikely value, e.g -1.

$$s^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2 \tag{4.13}$$

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.14}$$

When reconstructing a particular ray, the mean value is used instead of interpolation along the ray, if the address stored in the pointer is equal to `NULL` independent of its direction. When the pointer points to an actual memory location, the ray's direction in spherical coordinates is used to determine the exact value stored in the second-level array.

4.2.3 Reconstructing Fragmented Line Integrals

Actual rendering of a volumetric data set using fragmented line ray-casting proceeds as rendering a set using unmodified ray-casting. First a ray is determined by the ray source and the pixel of the image plane it corresponds to. Then the entry and exit points, respectively, where the ray intersects the volume, are computed. It is worth mentioning that this computation is not based on the resolution of the original volumetric data set anymore, but instead on the number of precomputed fragmented volumes stored in the fragmented line data structure. The last stage of evaluating the volume rendering integral along a particular ray is composed of the recurring execution of the utilized reconstruction filter. However, this step differs greatly from the resampling performed by ray-casting because of the respective underlying data structures, i.e ray-casting is executed on volumes that store one or several numerical values in their elements, whereas fragmented line ray-casting is performed on volumes that store multiple values in each of their elements each corresponding to a certain direction in space. Therefore, the first step to be performed for reconstructing fragmented line integral volumes is to determine a rays direction in spherical coordinates. These spherical coordinates can then be used to access the values of each fragmented volume that correspond to this particular direction. The actual operations necessary for reconstructing a ray can be divided into either *inter-ray interpolation* or *inter-volume interpolation*. Methods from both interpolation techniques can be combined arbitrarily. However, one should know, that their order of execution is dependent on how the data is actually stored in the fragmented line data structure, i.e inter-volume interpolation has to be executed before inter-ray interpolation when the data is stored in position-first order and vice versa. In the following descriptions of the different applicable techniques for resampling the stored data, the $access()$ function is used to access the value that is stored at the respective memory location denoted by its arguments.

4.2.3.1 Techniques for Inter-Ray Interpolation

Inter-ray interpolation deals with the issue of reconstructing a ray cast into the volumetric data set whose direction has not been precomputed in the preprocessing stage. As the different directions are stored in a 2-dimensional array, it is possible to do either nearest-neighbor interpolation, bilinear interpolation, or any other 2-dimensional interpolation technique.

Both methods can then be combined with any of the interpolation methods performed when traversing along the ray, i.e the methods that are used to interpolate between the precomputed fragmented volumes.

Nearest-Neighbor Interpolation is the simplest method and basically makes the elements stored in an array to appear bigger. The interpolated value of a resampling location is the value of the nearest value of the original 2-dimensional array in both dimensions.

$$nearest(\vartheta, \varphi) = access(\lfloor \vartheta + 0.5 \rfloor, \lfloor \varphi + 0.5 \rfloor)$$

Bilinear Interpolation determines the interpolated value based on a weighted average of the four existing values in the nearest 2×2 neighborhood of the desired resampling location in the direction array. This averaging has an anti-aliasing effect. Bilinear interpolation of the different ray directions denoted by spherical coordinates can be computed using the following equation

$$\begin{aligned} bilinear(\vartheta, \varphi) &= access(\lfloor \vartheta \rfloor, \lfloor \varphi \rfloor) * (1 - \vartheta_f) * (1 - \varphi_f) \\ &+ access(\lceil \vartheta \rceil, \lfloor \varphi \rfloor) * \vartheta_f * (1 - \varphi_f) \\ &+ access(\lfloor \vartheta \rfloor, \lceil \varphi \rceil) * (1 - \vartheta_f) * \varphi_f \\ &+ access(\lceil \vartheta \rceil, \lceil \varphi \rceil) * \vartheta_f * \varphi_f \end{aligned},$$

where ϑ_f and φ_f denote the fractional part of the considered ray's direction defined by

$$\begin{aligned} \vartheta_f &= \vartheta - \lfloor \vartheta \rfloor \\ \varphi_f &= \varphi - \lfloor \varphi \rfloor \end{aligned}$$

4.2.3.2 Techniques for Inter-Volume Interpolation

Inter-volume interpolation deals with the issue of accumulating the fragmented line integral's values along a certain ray cast into the volumetric data set. The ray is traversed by resampling the ray at equidistant resampling locations. Resampling locations that have not been precomputed have to be interpolated from the existing fragmented volumes. As the different fragmented volumes are stored in a 3-dimensional array, it is possible to use either nearest-neighbor interpolation, trilinear interpolation, bilinear interpolation along a major axis, or any other 3-dimensional interpolation technique. All three methods can then be combined with any of the interpolation techniques performed when interpolating the direction of a considered ray.

Nearest-Neighbor Interpolation is the simplest method for reconstructing the fragmented volumes. The interpolated value of a particular resampling location is the value of the nearest value of the original 3-dimensional array in all three dimensions.

$$nearest(x, y, z) = access(\lfloor x + 0.5 \rfloor, \lfloor y + 0.5 \rfloor, \lfloor z + 0.5 \rfloor)$$

Trilinear Interpolation determines the interpolated value based on a weighted average of the eight existing values in the nearest $2 \times 2 \times 2$ neighborhood of the desired resampling location in the fragmented volumes array. Trilinear interpolation of the different

resampling locations denoted by their position x, y, z can be computed using the following equation

$$\begin{aligned} trilinear(x,y,z) &= access(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor) * (1-x_f) * (1-y_f) * (1-z_f) \\ &+ access(\lceil x \rceil, \lfloor y \rfloor, \lfloor z \rfloor) * x_f * (1-y_f) * (1-z_f) \\ &+ access(\lfloor x \rfloor, \lceil y \rceil, \lfloor z \rfloor) * (1-x_f) * y_f * (1-z_f) \\ &+ access(\lfloor x \rfloor, \lfloor y \rfloor, \lceil z \rceil) * (1-x_f) * (1-y_f) * z_f \\ &+ access(\lceil x \rceil, \lfloor y \rfloor, \lceil z \rceil) * x_f * (1-y_f) * z_f \\ &+ access(\lfloor x \rfloor, \lceil y \rceil, \lceil z \rceil) * (1-x_f) * y_f * z_f \\ &+ access(\lceil x \rceil, \lceil y \rceil, \lfloor z \rfloor) * x_f * y_f * (1-z_f) \\ &+ access(\lceil x \rceil, \lceil y \rceil, \lceil z \rceil) * x_f * y_f * z_f \end{aligned} ,$$

where x_f, y_f, z_f denote the fractional part of the considered resampling location's position defined by

$$\begin{aligned} x_f &= x - \lfloor x \rfloor \\ y_f &= y - \lfloor y \rfloor \\ z_f &= z - \lfloor z \rfloor \end{aligned}$$

Bilinear Interpolation along a Major Axis determines the interpolated value based on a weighted average of the four existing values in the nearest $2 \times 2 \times 1$ neighborhood of the desired resampling location. This interpolation can be used when trilinear interpolation is deemed computationally too expensive. The major axis of the volumetric data set that is most parallel to the current viewing direction is denoted r. Interpolation along this axis r is performed using nearest-neighbor interpolation. Bilinear interpolation is then performed using the four neighbors of the resampling location in both remaining major axes, denoted s and t respectively. In order to compute the major axes that is most parallel to the current viewing direction, the dot product between the viewing direction and each major axis of the transformed volume coordinate system is computed. Then the maximum of these dot products corresponds to the major axis that is most parallel to the viewing direction. Bilinear interpolation of the different fragmented volumes along the major axis r can be computed using the following equation

$$\begin{aligned} bilinear(s,t) &= access(\lfloor s \rfloor, \lfloor t \rfloor, \lfloor r+0.5 \rfloor) * (1-s_f) * (1-t_f) \\ &+ access(\lceil s \rceil, \lfloor t \rfloor, \lfloor r+0.5 \rfloor) * s_f * (1-t_f) \\ &+ access(\lfloor s \rfloor, \lceil t \rceil, \lfloor r+0.5 \rfloor) * (1-s_f) * t_f \\ &+ access(\lceil s \rceil, \lceil t \rceil, \lfloor r+0.5 \rfloor) * s_f * t_f \end{aligned} ,$$

where s_f and t_f denote the fractional part of the considered sampling location's position defined by

$$\begin{aligned} s_f &= s - \lfloor s \rfloor \\ t_f &= t - \lfloor t \rfloor \end{aligned}$$

4.3 Implementation of CPU-based Techniques for Volume Rendering

This chapter presents an overview of the steps necessary for implementing a ray-casting based volume rendering application. It outlines the order of execution of the individual components. The steps presented here are based on separate portions of code that can be found in an actual implementation of a ray-casting based volume renderer application.

4.3.1 General Program Flow

In general, volume rendering applications can be divided into three different stages:

1. **Initialization:** The initialization stage is usually performed only once at start-up of the application. It is responsible for loading the volumetric data set into host memory. It might be necessary to realign the volume data during loading, when the bit alignment the volume is stored in and the bit alignment that the host hardware uses differ. Other computations to further process the volume data can be performed here as well, if their results do not change over the course of rendering.
   If the fragmented line integral rendering technique is used for rendering, the necessary data elements have to be precomputed from the volumetric data set as well unless they have been precomputed before.

2. **Ray-Casting:** In this stage, all computation necessary for casting a ray into the scene are performed. This includes the transformation of the camera center, of the principal point of the image plane, and of the image plane's right-vector. Then, each ray's direction is computed by the camera center and a point on the image plane. The last step involves determining the intersection points of a particular ray and the volume. This step does not differ between unmodified ray-casting and fragmented line integral rendering.

3. **Reconstruction:** For ray-casting this stage is about traversing a particular ray between its entry point and exit point. Usually, equidistant sampling points are used to traverse the ray. Then a value corresponding to a certain resampling location is reconstructed from the discrete volumetric data set using a certain reconstruction filter. This value is then added to the already accumulated values.
   For fragmented line integral rendering, a ray's direction in spherical coordinates is computed bfore the ray is traversed. Then, two reconstruction step are performed at each resampling location to obtain a precomputed fragmented line integral's value.

4.3.2 Implementation of a Volume Renderer based on Ray-Casting

4.3.2.1 Data Representation

In general, all data is stored in arrays of adequate dimensions. Therefore, the volumetric data set is stored in a 3-dimensional array, whereas the transfer function lookup table is stored as

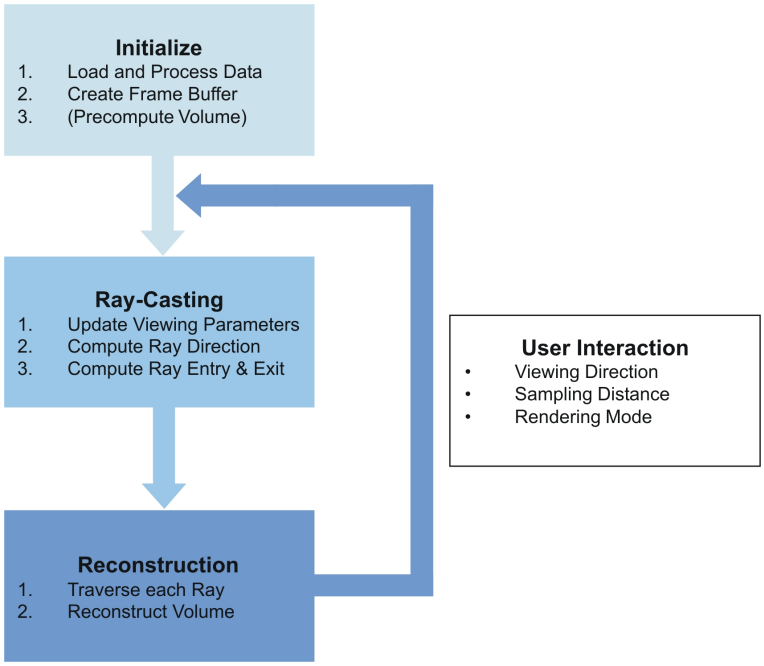


Figure 4.8: The steps of a typical ray-casting based volume renderer implementation.

an array of one or multiple dimensions. The final image is also described as a 2-dimensional array. The data format is dependent on the desired precision.

4.3.2.2 Computing Ray Directions

Before actual rays are cast into the scene, all points describing the camera, such as the camera center or ray source $\vec{s}$, the image plane's principal point $\vec{p}$, and the image plane's right-vector or $\vec{u}$, have to be transformed by multiplying them with the corresponding transformation matrix. Then, the normal of the image plane has to be computed. By evaluating the cross product of the image plane's normal and $\vec{u}$, its up-vector or $\vec{v}$ can be obtained. Lastly, both image plane vectors, $\vec{u}$ and $\vec{v}$ respectively, have to be normalized. The following code sample illustrate these steps.

It is important to note, that the camera points have to be rotated and translated in the opposite direction as the volume would be, i.e the camera points have to be transformed using the inverse modelview matrix, that is used to transform the volume. If the matrix only consist of rotation and translation, its inverse can easily computed using the following equation

$$\begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0\ 0\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{R^T} & \mathbf{-R^T t} \\ 0\ 0\ 0 & 1 \end{pmatrix} \tag{4.15}$$

```
1  MatrixVectorProduct(matrix, ray_source);
2  MatrixVectorProduct(matrix, image_plane_center);
3  MatrixVectorProduct(matrix, image_plane_right);
4
5  for(int i = 0; i<3; i++)
6  {
7    image_plane_u[i]      = image_plane_right[i] - image_plane_center[i];
8    image_plane_normal[i] = image_plane_center[i] - ray_source[i];
9  }
10
11 CrossProduct(image_plane_normal, image_plane_u, image_plane_v);
12
13 Normalize(image_plane_u);
14 Normalize(image_plane_v);
```

Listing 4.1: Code sample that transforms the necessary camera points. Then, $\vec{u}$ and $\vec{v}$ of the image plane are computed.

After the camera points have been transformed, a ray is cast through each pixel of the image plane, i.e a ray is cast through each element of the 2-dimensional image array. This is depicted in listing 4.2. The coordinates of the pixel's location in the final image are used to determine the direction of the ray. Therefore, the normalized $\vec{u}$ and $\vec{v}$ vectors are multiplied by the appropriate coordinates of the pixel. In order to obtain the pixel's representation in the world space coordinate system both scaled vectors have to be combined with the image plane's center point. By subtracting the the camera center, i.e the ray source, from the pixel's representation, the direction of the ray can be obtained. Before the direction is used for further processing of the ray, it has to be normalized as well. The sample code depicted in listing 4.4 computes the steps described here.

```
1 for(int v=0; v<image_height; v++)
2 {
3   for(int u=0; u<image_width; u++)
4   {
5     frame_buffer[v * image_width + u] =
6       castRay(u - image_width/2, v - image_height/2);
7   }
8 }
```

Listing 4.2: For each pixel in the frame buffer, a ray is cast into the scene.

```
1 for(int i=0; i<3; i++)
2   pixel[i] = image_plane_center[i] + u * image_plane_u[i] - v * image_plane_v[i];
3
4 for(int i=0; i<3; i++)
5   ray_direction[i] = pixel[i] - ray_source;
6
7 Normalize(ray_direction);
```

Listing 4.3: Code sample that determines the coordinates of a particular pixel of the image plane in world space coordinate system. Then, the direction of the ray is computed by subtracting the ray source's coordinates from the pixel's coordinates.

4.3.2.3 Computing the Entry Point and Exit Point

Both λ_{entry} and λ_{exit} corresponding to the entry point and the exit point respectively can be computed by evaluating equations 4.2. That way, the ratio between the distance from each component of the ray source's coordinates to each boundary of the volumetric data set and the ray's direction respective coordinate is computed. Then, by comparing the ratio between opposite boundaries, the computed ratios can be subdivided into candidates for the entry point or exit point respectively, whereas the lower ratio of opposite boundaries is considered a candidate for the entry point and vice versa. λ_{entry} corresponding to the entry point of the ray into the volume can be determined by choosing the maximum from the possible list of candidates. On the other hand, λ_{exit} corresponding to the exit point can be determined by choosing the minimum of the appropriate candidates. It is important to note, that all computations must be performed in object space. The following code fragment illustrates how λ_{entry} and λ_{exit} can be computed.

```
1  for(int i=0; i<3; i++)
2  {
3    lambda1 = (volume_dimension[i] - 1) - ray_source[i]) / ray_direction[i];
4    lambda2 = (-ray_source[i]) / ray_direction[i];
5
6    if(lambda2 < lambda1)
7      swap(lambda1, lambda2);
8
9    if(lambda1 > lambda_entry)
10     lambda_entry = lambda1;
11
12   if(lambda2 < lambda_exit)
13     lambda_exit = lambda2;
14 }
```

Listing 4.4: Code sample that determines λ_{entry} and λ_{exit} of a particular ray.

4.3.2.4 Reconstructing the Volumetric Data Set

The resampling points needed for reconstructing the volumetric data set are generated by traversing the ray between the entry point and the exit point in equisized steps. Then, at each resampling location a reconstruction filter is evaluated to obtain its corresponding value from the discrete volumetric data set. As the recurring evaluation of the reconstruction filter is the computational most demanding task of ray-casting, a reconstruction filter must present a good trade-off between rendering performance and image quality. In general, trilinear interpolation is chosen from the available reconstruction filters as it satisfactorily fulfills the aforementioned condition. The reconstructed values are then accumulated and stored at the ray's corresponding pixel location. It is important to note, that traversing a ray is only performed if λ_{exit} is greater than λ_{entry}. Listing 4.5 illustrates the steps for traversing a particular ray. Trilinear interpolation can be performed by implementing equations 4.3 and 4.4.

```
1  if(lambda_entry < lambda_exit)
2  {
3    for(double l = lambda_entry; l < lambda_exit; l += sampling_distance)
4    {
5      value = TrilinearInterpolation(ray_source[0] + l * ray_direction[0],
6                                      ray_source[1] + l * ray_direction[1],
7                                      ray_source[2] + l * ray_direction[2]);
8
9      sum += value;
10   }
11 }
```

Listing 4.5: Traversing along a particular ray.

4.3.3 Implementation of a Volume Renderer based on Fragmented Line Integrals Rendering

As described in section 4.2, fragmented line integral rendering consists of two separate steps. The first step, done prior to any actual rendering, precomputes all fragmented line integrals inside a volume as described in section 4.2.1 and stores them in the compressed data structure mentioned at the end of section 4.2.2. Rendering of the precomputed data set is very similar to common volume ray-casting. However, the actual reconstruction of the data set is different. As this technique uses both spherical coordinates as well as Cartesian coordinates, the function displayed in the code fragment 4.6 can be used to transform a representation into the other.

4.3.3.1 Implementing the Precomputation of Fragmented Line Integrals

Prior to precomputation the number of fragmented volumes, that have to be computed, has to be determined. This value is obtained, as depicted in the code fragment in listing 4.7,

```
1  void toSphereCoord(float * cart, float * sphere)
2  {
3    sphere[RADIUS] = sqrt(cart[X]*cart[X] + cart[Y]*cart[Y] + cart[Z]*cart[Z]);
4          sphere[THETA]  = acos(cart[Z] / sphere[RADIUS]);
5          sphere[PHI]    = atan2(cart[Y] , cart[X] );
6  }
7
8  void toCartCoord(float * sphere, float * cart)
9  {
10         cart[X] = sin(sphere[THETA]) * cos(sphere[PHI]) * sphere[RADIUS];
11         cart[Y] = sin(sphere[THETA]) * sin(sphere[PHI]) * sphere[RADIUS];
12         cart[Z] = cos(sphere[THETA]) * sphere[RADIUS];
13 }
```

Listing 4.6: Transformation of Cartesian Coordinates to spherical coordinates and vice versa.

by dividing the number of voxels in the volumetric data set by the distance between two successive fragmented volumes in each dimension.

```
1  for(int i=0; i<3; i++)
2  {
3    frag_volumes_distance[i] = frag_volumes_dim[i] - frag_volumes_overlap[i]
4    frag_volumes_number[i]   = (volume_dim[i] / frag_volumes_distance[i]) + 1;
5  }
```

Listing 4.7: Computing the number of fragmented volume, that have to be precomputed, is performed here for every dimension.

The next step is to loop over all fragmented volumes, that have been generated, and precompute all considered ray direction in each of them. Therefore, each fragmented volume is loaded into a separate memory location. Then for each considered ray direction a ray is cast through the center of the fragmented volume and its accumulated value is stored in a 2-dimensional array. Each accumulated value is also used to compute both the variance and the mean value of the cast rays. Depending on the variance, the computed fragmented line integrals are either added to the fragmented line data structure, or discarded. When the fragmented line integral values are discarded, their mean value is stored in the fragmented line data structure instead. The associated sample code can be found in listing 4.8.

The actual ray-casting is quite similar to the ray-casting technique described in section 4.3.2. However, a few differences should be pointed out:

```
1  float mean_value = 0.0f;
2  float variance   = 0.0f;
3
4  float step_size_phi   = PI / resolution_phi;
5  float step_size_theta = PI / resolution_theta;
6
7  LoadFragmentedVolume(x, y, z);
8
9  ray_direction[THETA] = 0.0f;
10
11 for(theta = 0; theta < resolution_theta; theta++)
12 {
13         ray_direction[PHI] = 0.0f;
14   for(phi = 0; phi < resolution_phi; phi++)
15   {
16           result[theta][phi] = castRay(ray_direction);
17
18           mean _value += result[theta][phi];
19           variance    += result[theta][phi]*result[theta][phi];
20
21           ray_direction[PHI] += step_size_phi,
22   }
23   ray_direction[THETA] += step_size_theta;
24 }
25
26 mean_value /= (resolution_phi * resolution_theta);
27 variance    = (variance /(resolution_phi * resolution_theta) )
28             - (mean_value * mean_value);
29
30 if(variance > threshold)
31   RetainResult();
32 else
33   DiscardResult();
```

Listing 4.8: $res_\vartheta * res_\varphi$ number of rays have to be precomputed inside each fragmented volume.

Instead of determining the direction of a ray based on the coordinates of its associated pixel and its source, the direction is computed by transforming its spherical coordinates representation into a Cartesian coordinate representation using equations 4.6. The appropriate code fragment can be found in listing 4.6. The remaining steps of normalizing the ray's direction and is coordinate transformation into object space are performed as described in section 4.3.2.2.

The next difference results from the fact that the emission point of a ray is not outside the volume but inside. To be more precise, the ray's emission point during precomputation is the exact center of the fragmented volume. Therefore, the computation of λ_{entry} and λ_{exit} has to be modified accordingly. Instead of computing the ratio between the distance from each component of the ray source's coordinates to each boundary of the volumetric data set and the ray's direction respective coordinate, the ratio between half of each component of the fragmented volume's dimension and the ray's direction corresponding coordinate is

computed. The remaining steps of determining λ_{entry} and λ_{exit} from the possible list of candidates is performed as described in section 4.3.2.3.

The actual reconstruction along a ray is done exactly like in ray-casting. However, since this reconstruction is done offline, a high-order reconstruction filter could be used instead of trilinear interpolation.

4.3.3.2 Rendering Fragmented Line Integrals

As described in section 4.2.3 rendering fragmented line intgrals is quite similar to unmodified ray-casting. Determining the direction of a ray cast into the scene does not differ from the method described for ray-casting in section 4.3.2.2. Additionally, the computation of the entry point and the exit point is not changed from the implementation described in section 4.3.2.3. However, it is important to note, that the dimensions used to calculate λ_{entry} and λ_{exit} respectively have to resemble the number of fragmented volume in each direction instead of the size of the original volume. After entry points and exit points have been acquired, the utilized reconstruction filter is executed repeatedly. As noted in section 4.2.3 the reconstruction is performed in two steps. The first step reconstructs a resampling location from the 3-dimensional top-level array, that stores either a pointer to a second-level array or the mean value of the precomputed fragmented line integrals. The second step is only performed if the top-level array stores a pointer to a second-level array. Its task is to reconstruct a particular ray direction from the precomputed fragmented line integrals. Section 4.2.3 refers to both steps as *inter-ray interpolation* and *inter-volume interpolation* respectively. Methods from both interpolation techniques can be combined arbitrarily. All methods are presented in detail in section 4.2.3.1 and section 4.2.3.2 respectively.

5 Results

5.1 Texture-based Volume Rendering Techniques Performance Evaluation

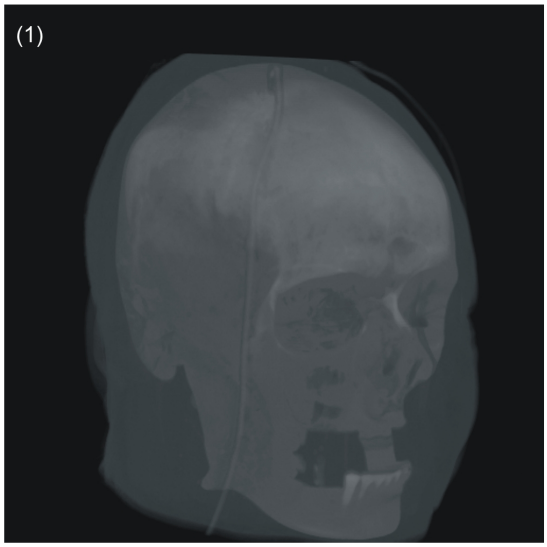
The following results were obtained on a Intel Pentium IV processor. The core speed was 2.60 GHz, whereas the front side bus speed was set to 200 MHz. The processor had 512 KBytes of level 2 cache. The underlying mainboard model was an ASUS P4C800-Deluxe based on the Intel i875P chipset. The amount of random access memory was 2560 MBytes of DDR-SDRAM. Hyperthreading was enabled for all test cases. The OS used for obtaining the following values was Microsoft Windows XP Professional with service pack 1 installed.

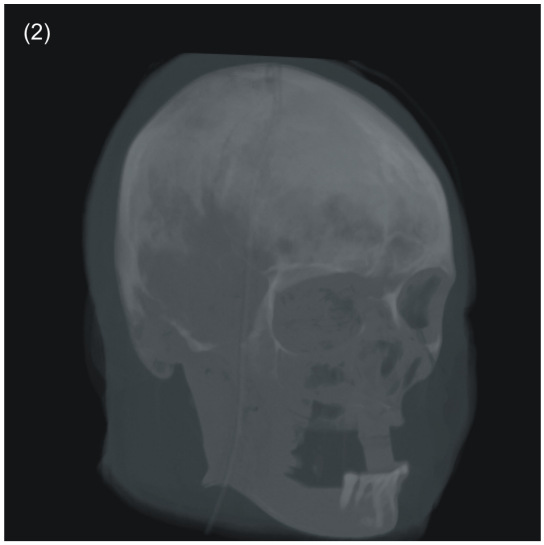
The graphics subsystem used for evaluation of the implemented texture-ased volume rendering techniques was a NVIDIA GeForce 6600GT graphics accelerator. The GeForce 6600GT has a core clock speed of 500 MHz and a memory clock speed equaling 1000 MHz. The bandwidth for accessing the on-board memory is only 128bit and the amount of memory on this particular model was only 128MBytes of GDDR3 memory. Therefore, the memory bandwidth's upper limit is 15259 MBytes per second. A GeForce 6600GT has eight pixel pipelines, each capable of single unique texture lookup, and three vertex pipelines. The graphics card is capable of transforming 375 million vertices per second and its multi-texturing fill-rate peaks at 4000 million pixels per second. The installed driver was the NVIDIA reference driver, revision 66.93. For all tests vertical sync was disabled by default.

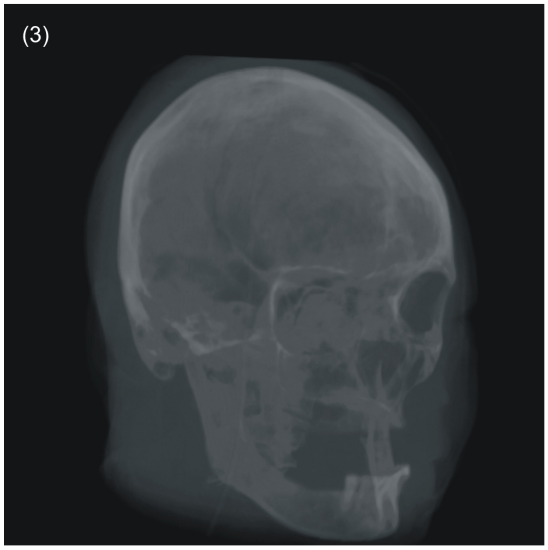
The volumetric data set used had a resolution of $512 \times 512 \times 209$. As the volume consisted of 12bit values it was stored in one or several 16bit texture objects respectively. The transfer function was stored in a 8bit RGBA 2-dimensional texture object with a width of 2048 and a height of 1. The texture wrap mode was set to **`GL_CLAMP_TO_EDGE`** and the minification filter and the magnification filter was set to **`GL_LINEAR`** for all texture objects. The field-of-view was 60 degrees and the distance of the camera from the world coordinate system origin was 304.

5.1.1 View-aligned Slices using 3-dimensional Texture Mapping without Transfer Function Evaluation

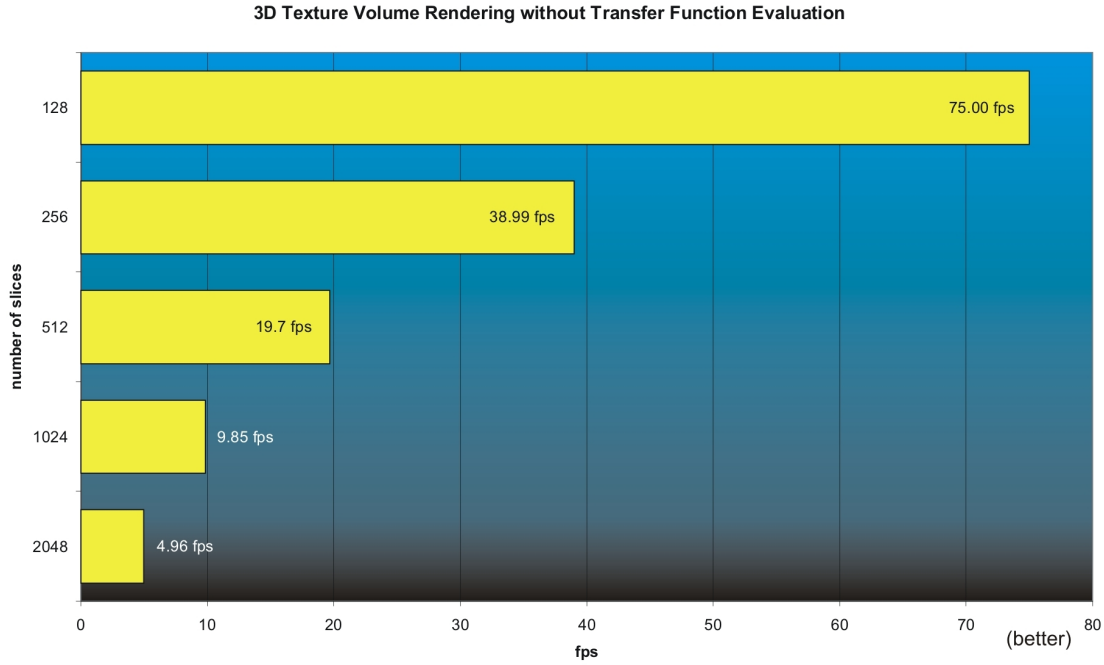
The following examples were generated with view-aligned slices used as proxy geometry with 3-dimensional texture mapping. Image (1) was generated using 1024 slices. (2) was rendered with 512 slices. (3) and (4) were rendered using 256 and 128 slices respectively. No transfer function was applied to the density values stored in the volume.


(1)


(2)


(3)

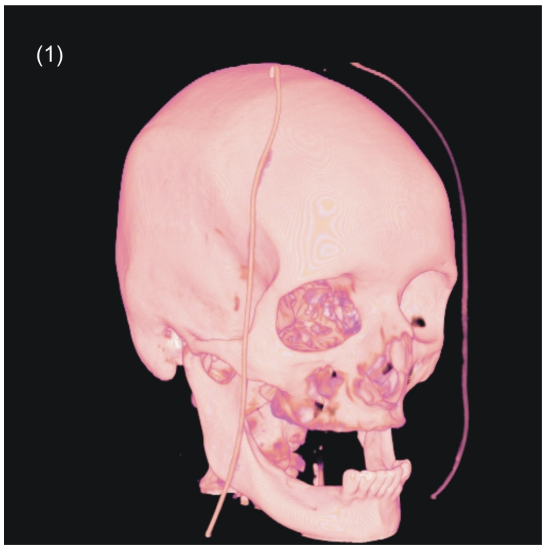

(4)

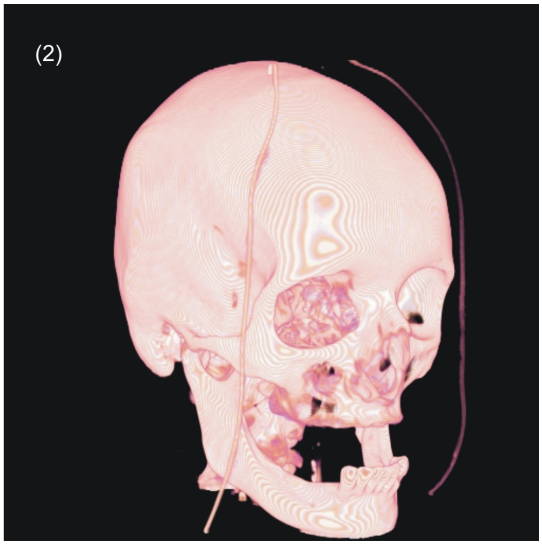


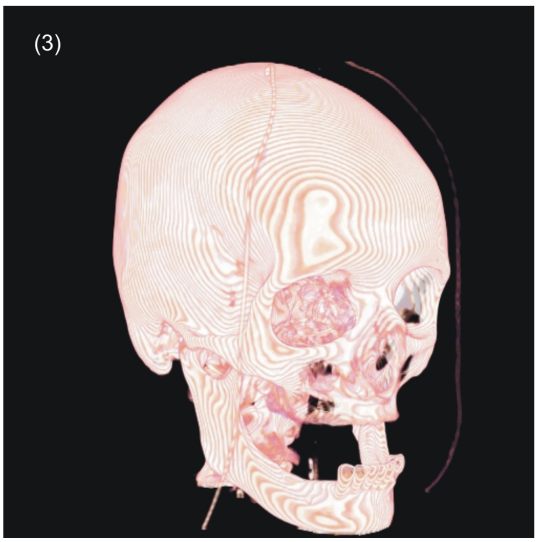
It is obvious that the actual rendering performance of a view-aligned texture-based volume rendering application inversely proportional to the number of slices drawn to reconstruct the volumetric data set. The best trade-off between rendering speed and image quality can be reached when rendering as many slices as the maximum resolution of the visualized volumetric data set.

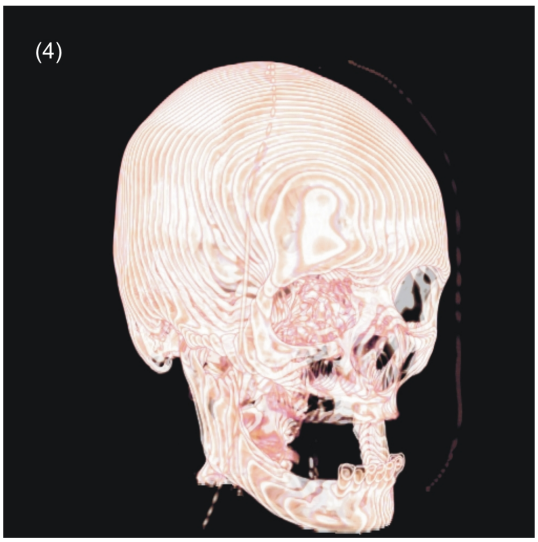
5.1.2 View-aligned Slices using 3-dimensional Texture Mapping with Transfer Function Evaluation

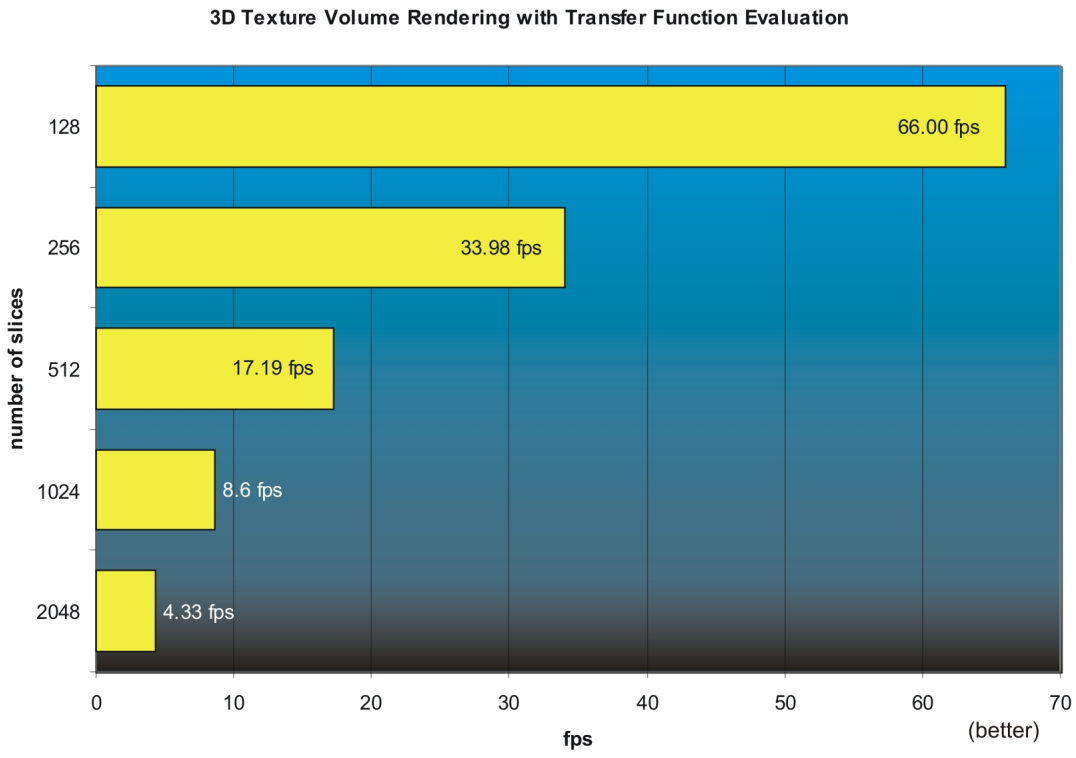
View-aligned slices were used as proxy geometry to render the following examples. Image (1) was generated using 1024 slices. (2) was rendered with 512 slices. (3) and (4) were rendered using 256 and 128 slices respectively. A transfer function was evaluated for every drawn pixel.


(1)


(2)


(3)

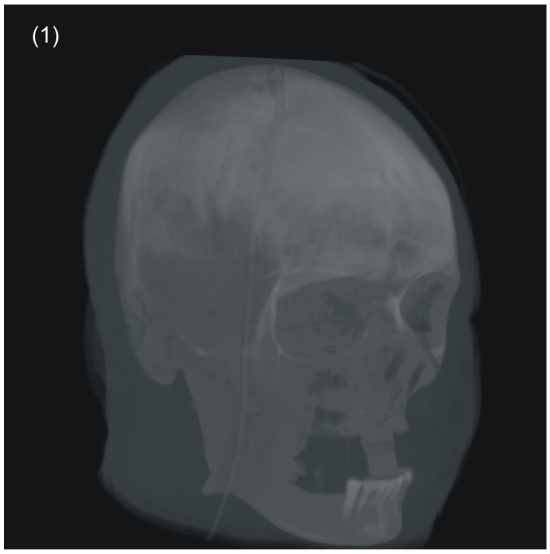

(4)

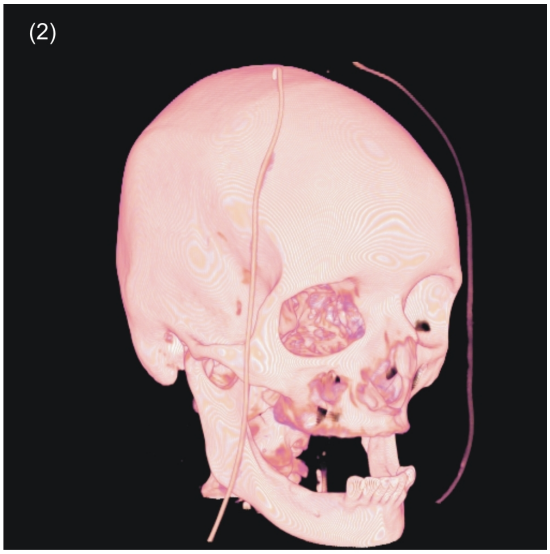


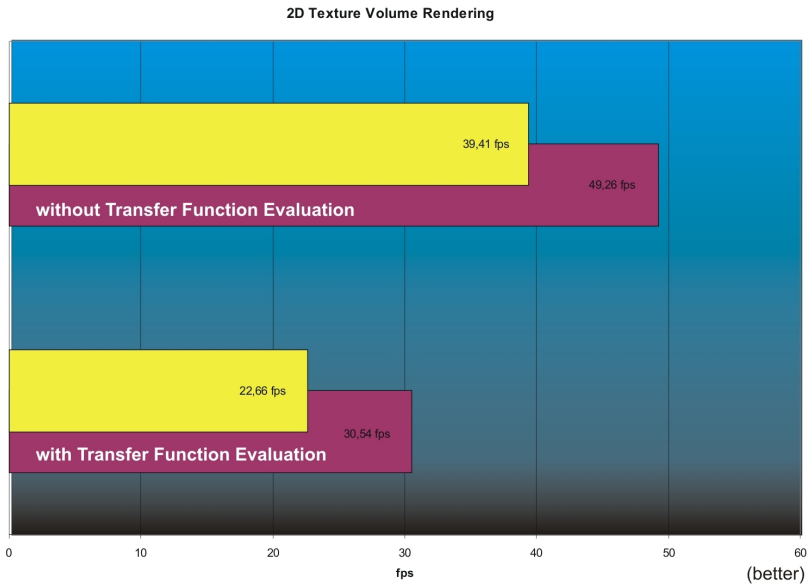
Expectedly, the dependent texture lookup used to evaluate the transfer function for an interpolated density value, places a big burden on the graphics hardware. In this case, the performance of a texture-based volume rendering application using view-aligned slices drops about 12.7%. Still, the performance left for visualizing the volume is astonishing. The best trade-off between rendering performance and image quality can be achieved when rendering as many slices as the volume's maximum resolution component.

5.1.3 Object-aligned Slices using 2-dimensional Texture Mapping

(1) displays a volume rendered using object-aligned slices and 2-dimensional texture mapping. No transfer function was evaluated in this case. (2) represents the same data set with a transfer function applied. The violet bars represent the frames per second when looking down the axis with the lowest resolution.



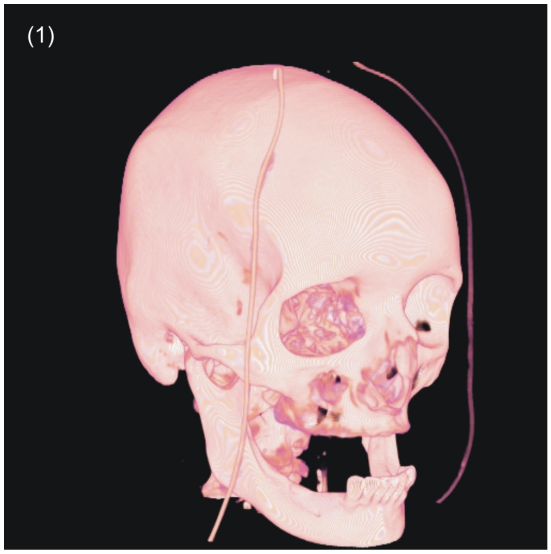


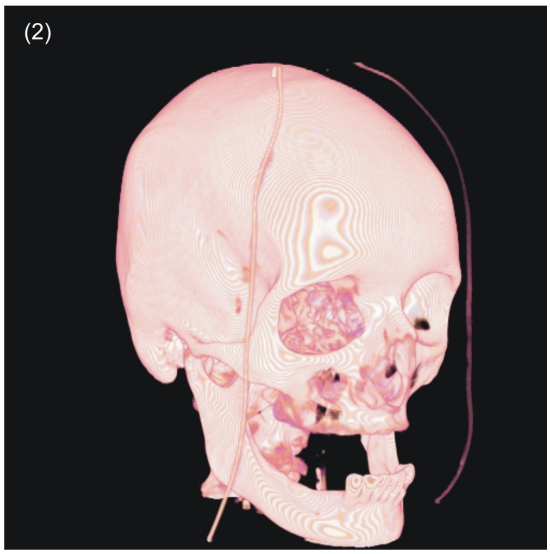


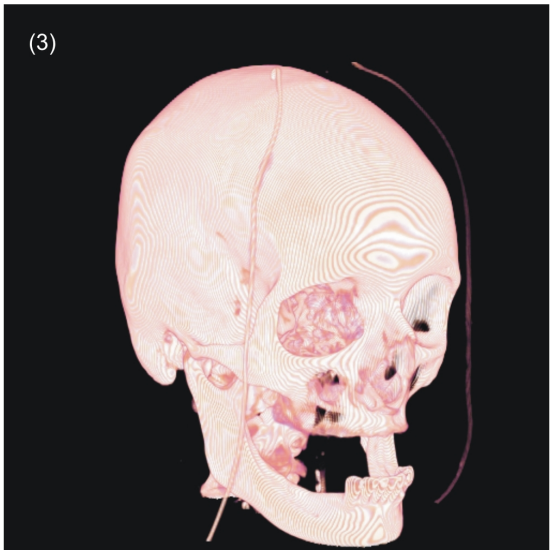
In the case of object aligned slices the achievable frames per second are higher due to the reconstruction of the volumetric data set using only bilinear interpolation as opposed to trilinear interpolation when using 3-dimensional textures. One can also note a difference in performance for different viewing directions. This is due to the fact, that the number of actual slices is fixed by the resolution of the volumetric data set.

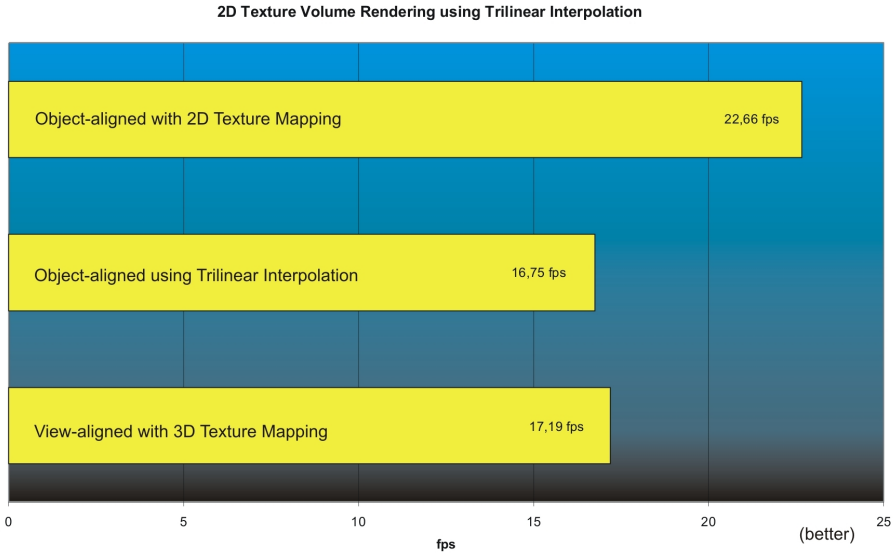
5.1.4 Object-aligned Slices using 2-dimensional Texture Mapping with Trilinear Interpolation

Image (1) of the following examples was generated using object-aligned slices using bilinear interpolation. View-aligned slices using 3-dimensional texture mapping was used to render (2). (3) was visualized using object-aligned slices using trilinear interpolation.


(1)


(2)

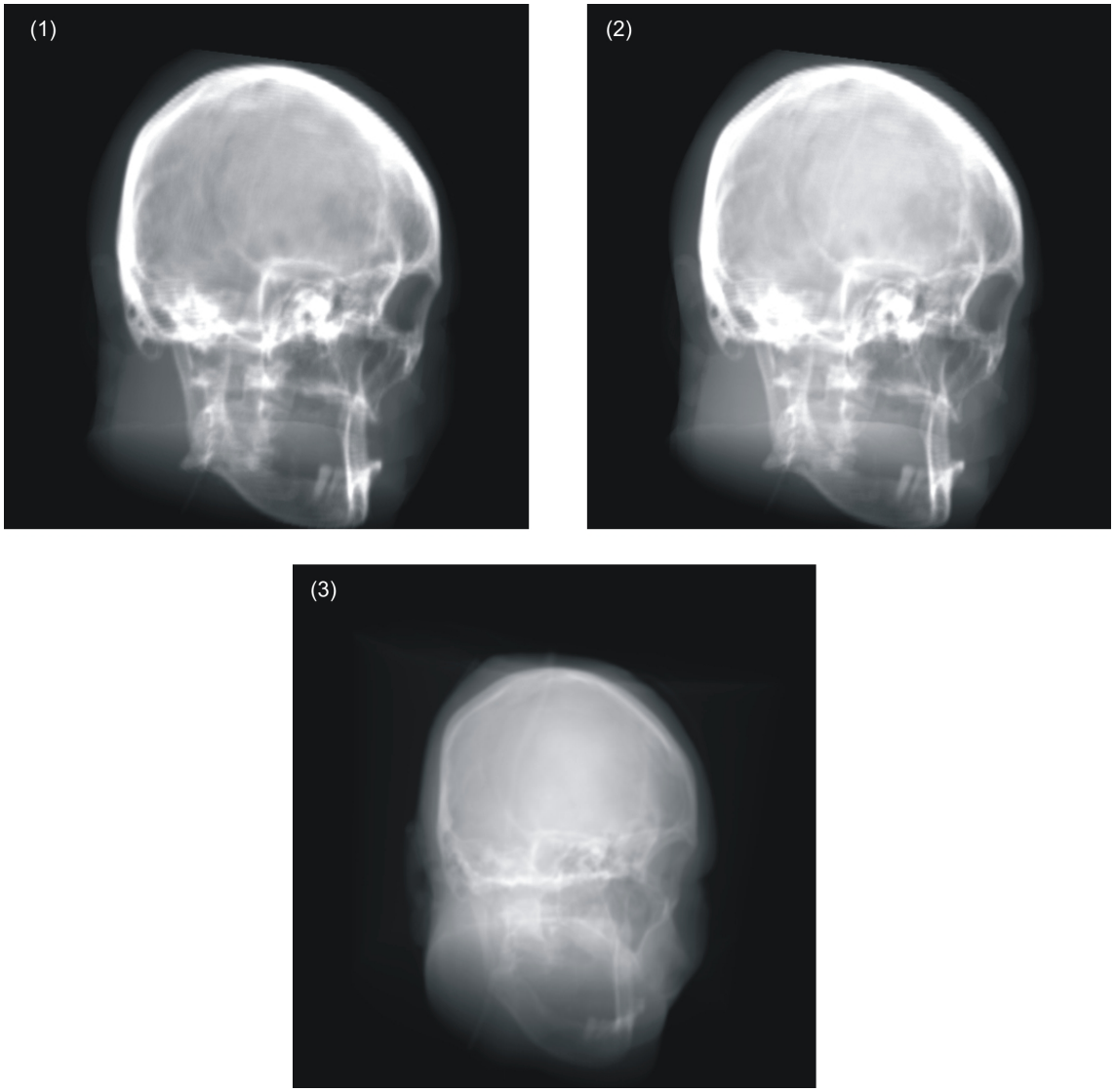

(3)

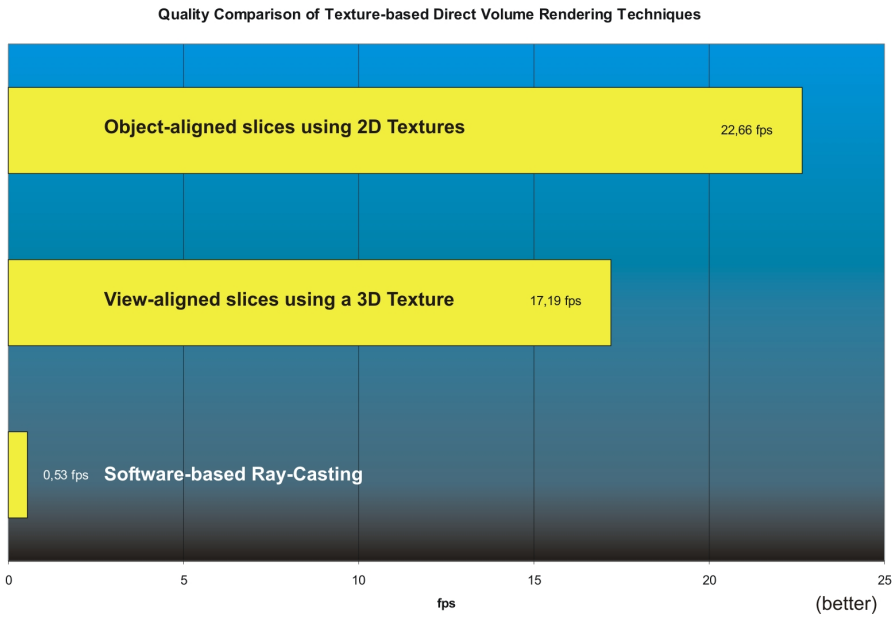


The rendering performance for computing the trilinear interpolation of the volumetric data set inside a fragment shader has no performance gain on using view-aligned slices with 3-dimensional texture mapping.

5.1.5 Quality Comparison of Texture-based Volume Rendering Techniques

The following examples illustrate the difference in quality of both texture-based methods for volume rendering. Image (1) was generated using 256 view-aligned slices each reconstruction the volumetric data set by trilinear interpolation of a 3-dimensional texture object. For rendering image (2) object-aligned slices were used to display the volumetric data set. Therefore, the volume was stored in several 2-dimensional texture objects, thus allowing only for bilinear interpolation within each texture object for reconstruction of the volume. In addition, image (3) was rendered using an unoptimized software-based ray-casting algorithm, i.e no optimization techniques like empty space skipping or early ray termination were employed for rendering. The resolution of the frame buffer was 512×512 for all test cases.


(1)
(2)
(3)



Although software-based ray-casting delivers the most accurate results, its slow rendering speed does not make it sufficient for neither interactive visualization, nor image registration. On the contrary, rendering view-aligned slices with 3-dimensional textures yield results that are comparable to those of software-based ray-casting but at much faster rendering speeds. Even the image quality of images generated by drawing object aligned-slices using several 2-dimensional texture objects is sufficient for most visualization tasks.

Essentially, texture-based methods for direct volume rendering deliver a comparable image quality to software-based ray-casting while achieving very high frame rates. However, their major drawback in terms of image quality is, that on current generation graphics hardware, the blending operation used to accumulate the resampled values is only performed on 8bit precision. In addition, the frame buffer of the graphics hardware is only of 8bit precision per channel as well. It is important to note, that the current top-product from NVIDIA, the GeForce 6800 line of GPUs, offer blending operations to an offscreen buffer to be performed on a half precision, i.e 16bit, format. Still, for displaying the result on screen the rendered image has to be transfered to the 8bit per channel precision frame buffer.

5.2 CPU-based Volume Rendering Techniques Performance Evaluation

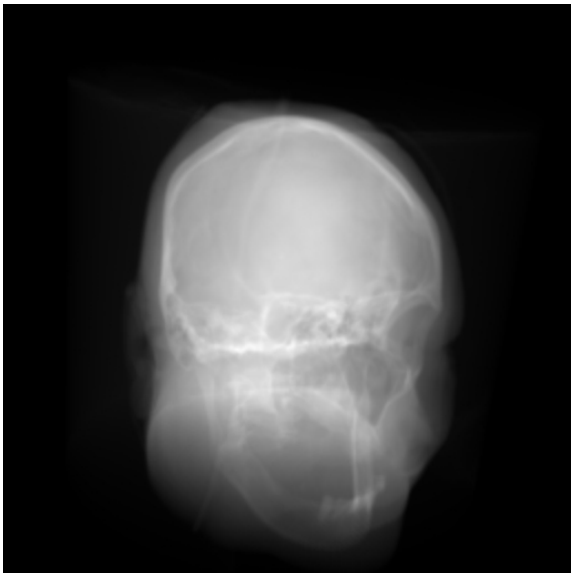
The results in this section were obtained on a Intel Pentium IV processor running at 2.60 GHz. The front side bus speed was set to 200 MHz. The processor had 512 KBytes of level 2 cache. The underlying mainboard model was an ASUS P4C800-Deluxe based on the Intel i875P chipset. The amount of memory was 2560 MBytes of DDR-SDRAM. Hyperthreading was enabled for all test cases. The OS used for obtaining the following values was Microsoft Windows XP Professional with service pack 1 installed.

The graphics subsystem used for displaying the computed image of the implemented CPU-based volume rendering techniques was a NVIDIA GeForce 6600GT graphics accelerator. The installed driver was the NVIDIA reference driver, revision 66.93. For displaying vertical sync was disabled by default. However, the time determined for rendering did not include the time necessary to display the final image on screen.

The volumetric data set used had a resolution of $512 \times 512 \times 209$. The volume was stored in a array of floats matching the volume's original resolution. The frame buffer used to store each computed ray's value was a 2-dimensional floating point array matching 256×256 in resolution. No transfer function evaluated during rendering of the volumetric data set. The field-of-view was 60 degrees and the distance of the camera from the world coordinate system origin was 304.

5.2.1 Unoptimized Ray-Casting

The following example was rendered using an unoptimized, i.e for this test no acceleration techniques like empty space skipping or early ray termination were employed, ray-casting algorithm.

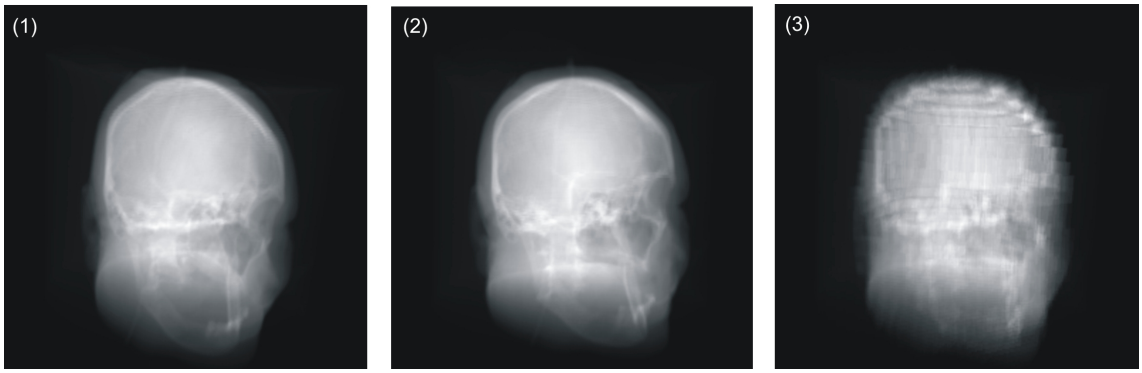


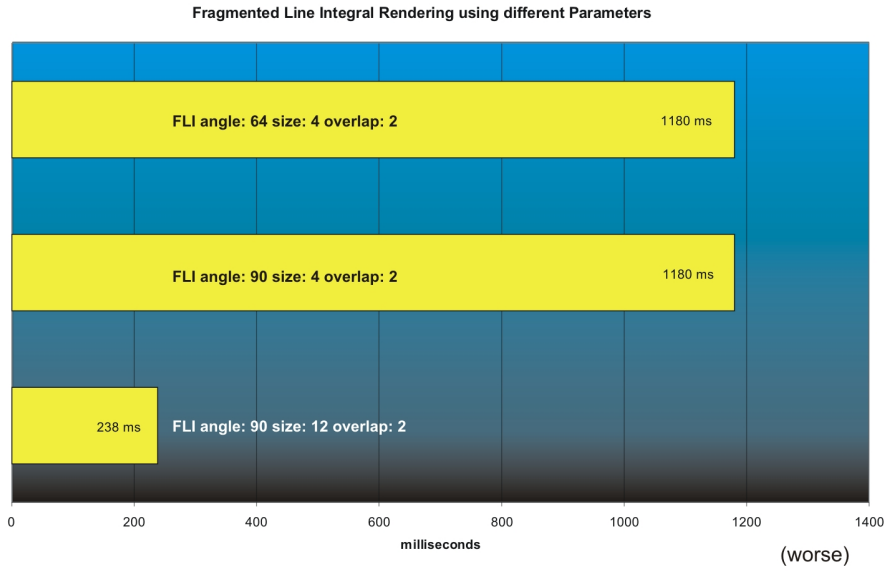
The image took 1859 milliseconds to render.

5.2.2 Fragmented Line Integral Rendering

The following example was rendered using the proposed fragmented line integral rendering technique. All pictures were generated using the same volumetric data set. Image (1) was preprocessed using 90 steps for parameterizing both spherical coordinates and a fragmented volume size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$. In image (2) steps for both spherical coordinates were reduced to 64, whereas the fragmented volume size and their

overlap remained unchanged. The last image (3) was precomputed using again 90 steps as parameters for both spherical coordinates and a fragmented volume size of $12 \times 12 \times 12$ that overlap each other by $2 \times 2 \times 2$. The optimized data structure described in section 4.2.2 was used to store the precomputed fragmented line integrals. All precomputed fragmented line integrals were rendered using nearest-neighbor interpolation for both the ray direction and the position of the fragmented volume.





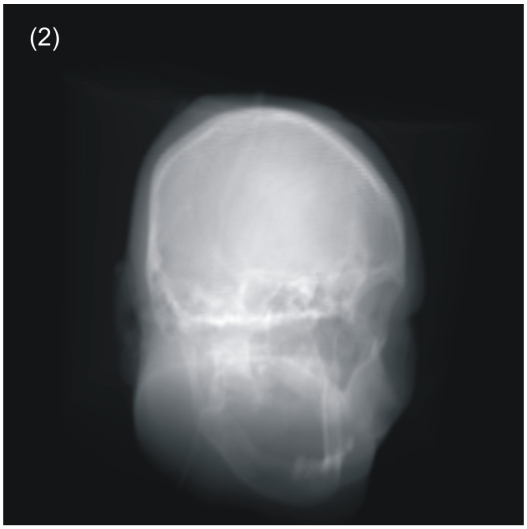
Obviously, the quality of the final image is dependent on the size of the fragmented volumes used for precomputation. However, this leaves the rendering application with an additional number of fragmented volumes to resample during reconstruction of each ray, thus slowing down the rendering performance. When one takes a closer look to the parameters used for precomputation only each second degree got precomputed in the case of an angle resolution of 90. In contrast, the difference in degrees of two adjacent pixels on the image plane is 0.23 when considering a field-of-view of 60 degrees and a frame buffer resolution of 256×256. Therefore, the blocky look of image (3) could also be explained by the insufficient number of precomputed angles, i.e about eight neighboring pixels are reconstructed using the same precomputed ray direction. I was not able to clarify this assumption as I had not

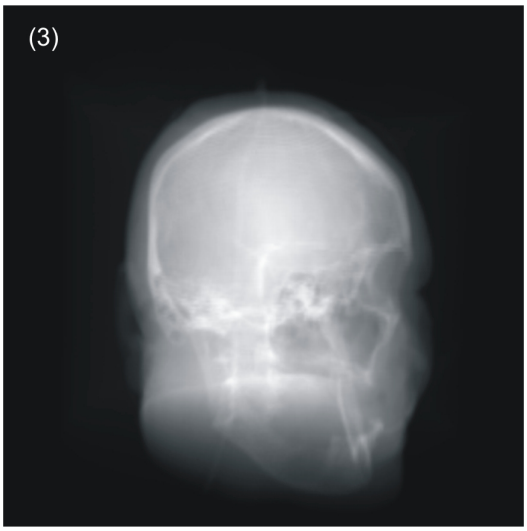
been able to precompute the necessary amount of ray directions due to the increasing memory demands imposed by the underlying data structure used for storing the precomputed fragmented line integrals.

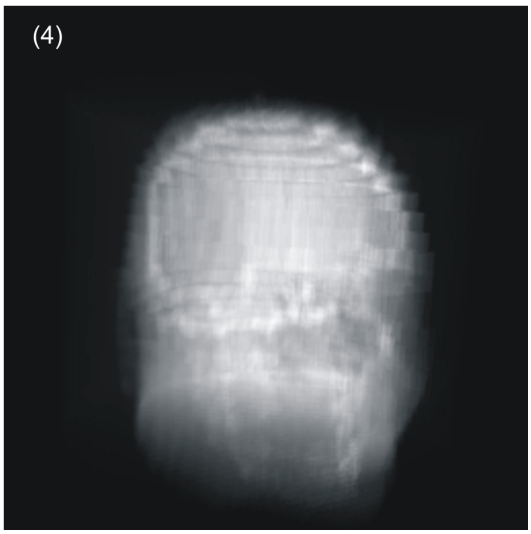
5.2.3 Quality Comparison of CPU-based Volume Rendering Techniques

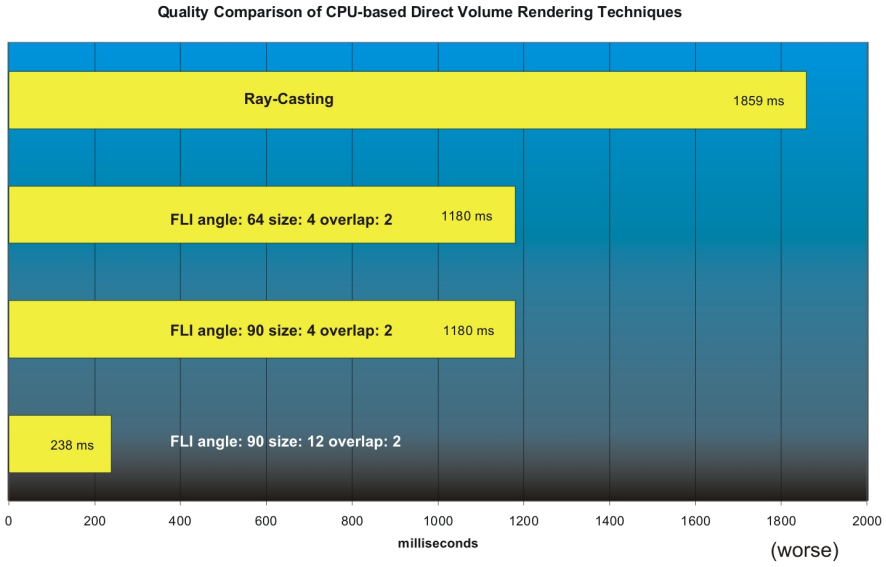
The following examples illustrate the difference in quality of both ray-casting and fragmented line integral rendering techniques. Image (1) was rendered using an unoptimized software-based ray-casting algorithm as described in section 4.3.2. Image (2) was precomputed using 90 steps for parameterizing both spherical coordinates and a fragmented volume size of $4 \times 4 \times 4$ that overlap each other by $2 \times 2 \times 2$. In image (3) steps for both spherical coordinates were reduced to 64, whereas the fragmented volume size and their overlap remained unchanged. The last image (4) was precomputed using again 90 steps as parameters for both spherical coordinates and a fragmented volume size of $12 \times 12 \times 12$ that overlap each other by $2 \times 2 \times 2$. The optimized data structure described in section 4.2.2 was used to store the precomputed fragmented line integrals. The precomputed fragmented line integrals were rendered using nearest-neighbor interpolation for both the ray direction and the position of the fragmented volume. The resolution of the frame buffer was 256×256 for all test cases.


(1)


(2)


(3)


(4)



Still, ray-casting delivers the most accurate rendered images. On the contrary, the fragmented line integrals rendering technique can significantly decrease rendering times compared to an unoptimized ray-casting implementation. The actual amount of speedup is dependent on the parameters used for preprocessing the volumetric data set. However, sets of parameters that result in a large increase in rendering performance yield images of much inferior image quality compared to the ray-casting implementation. The lesser image quality results in part of the number of angles that have been precomputed. Using the aforementioned compression scheme the current implementation is only capable of precomputing integer degrees. However, for most field-of-views the difference between ray directions corresponding to adjacent pixels is only fractional, thus rendering results in a blocky looking image, because the same ray direction is used for reconstructing a particular neighborhood of pixels.

6 Conclusion

6.1 Texture-based Volume Rendering

In this thesis I presented an overview on different texture-based methods for visualization of volumetric data sets. I worked out the characteristics of consumer graphics hardware and their effects on such hardware's use for visualization. For example, as consumer graphics hardware was intentionally designed for accelerating the rendering of polygonal objects, these graphic chips lack a native support for volumetric primitives, i.e voxels. I illustrated basic approaches for dealing with this specific problem by rendering textured proxy geometry. These approaches covered different types of available texture objects as well as their respective proxy geometries. Further functionality was added to the described volume rendering applications: Different blend modes, maximum intensity projection, and non-polygonal isosurfaces were supported using hardware-amenable blending capabilities and pixel operations respectively. In addition, support for classification of the volumetric data was enabled by employing lookup tables for transfer functions. These lookups were performed using the programmable rendering pipeline offered by current generation graphics hardware, thus allowing for transfer function evaluation after the interpolation of the volume, i.e post-classification.

Completing the topic on texture-based techniques for volume visualization, an extensive overview on the details of an actual implementation of these direct volume rendering techniques is presented, that can easily be used as a starting point for further implementations.

It was also shown that the performance achievable by hardware-amenable volume rendering clearly surpasses that of traditional CPU-based ray-casting volume rendering applications.

Refining hardware-amenable volume rendering techniques are still a very active field in today's related research. On the contrary to software-based methods, the major goal is to increase the quality of the final image instead of decreasing their rendering time. Therefore, many different approaches have been proposed, such as the application of hardware accelerated high-quality filtering [28, 29], or pre-integrated volume rendering [22]. In addition, well-known software based volume rendering techniques have been refined by porting them to the GPU [38] as well as new ones have been proposed. It is possible to assume that rendering times of GPU-based volume visualization techniques will decrease even further with the introduction of each next generation of commodity graphics hardware. For example, it is rumored that next generation products will certainly increase the amount of actual rendering

pipelines along with the number of on-board vertex shaders and fragment shaders respectively. In addition, the added capabilities introduced with each future generation might make even more visualization techniques possible. However, one drawback of current generation hardware that has to be addressed, if GPU-based volume rendering techniques want to rival or even surpass the quality and the precision of software-based methods, is the lack of high-precision value transfer across all stages of the rendering pipeline.

6.2 CPU-based Volume Rendering

First, I illustrated the components of a volume rendering application based on the ray-casting technique. The advantages of ray-casting includes that ray-casting can be performed completely in software, thus omitting the need for dedicated graphics hardware. Additionally, ray-casting is able to deliver the most precise results of any volume rendering technique. However, its sub-par rendering performance is its major drawback. Therefore, we proposed a novel approach for accelerating ray-casting, by precomputing fragmented rays inside the volumetric data set prior to visualization. As this approach is based on traditional ray-casting it can be executed on any platform. In addition, a compression scheme based on the variance of the fragmented line integrals within a particular fragmented volume was presented as a solution to circumvent one of this approach's obvious drawbacks, the high memory requirements of storing the recomputed data.

Furthermore, I give an extensive amount of details on the implementation of both software-based volume rendering techniques.

Concluding the chapter on CPU-based accelerated volume rendering techniques, the performance of the presented fragmented line integral method is compared to a basic ray-casting implementation. Obviously, both the quality of the rendered image as well as the rendering performance of the fragmented line integral rendering is dependent on the parameters used during precomputation. In order to achieve a high image quality the size of the fragmented volumes has to be quite small, whereas the number of precomputed angles has to be quite large, thus resulting in high precomputation times. However, these high precomputation times are unbearable when considering the lesser quality of the final image. The lesser image quality results in part of the number of angles that have been precomputed. For example, a suitable precomputation using the aforementioned compression scheme is only capable of precomputing integer degrees. However, a small field-of-view is usually sufficient when simulating a x-ray scanner. The difference between ray directions corresponding to adjacent pixels is therefore marginal, thus resulting in a blocky looking image, because the same ray direction is used for reconstructing a particular neighborhood of pixels.

In principle, the approach of precomputing the volume rendering integral for certain rays inside the volume presents a good starting-point for accelerating CPU-based volume rendering applications. However, without further refining the proposed method neither the rendering performance nor the quality of the resulting image is superior compared to other

accelerated software-based volume rendering techniques. A possible starting-point for increasing the quality of the rendered image might be the utilization of a more sophisticated data structure. Such a data structure can be deemed superior to the data structure proposed in this thesis, if it offers better compression of the precomputed fragmented line integrals. This would allow us to precompute and store even fractional degrees for a ray's direction, thus lowering the aforementioned negative effects of commonly used small field-of-views. In addition, the compression should also be less lossy than the one proposed in order not to lose the gained increase in precision. Although, this might increase the quality of the computed image, it might also result in increased rendering times. Even if this is not the case, I doubt that the actual precomputation cannot be accelerated big time. In contrast, precomputation times may increase even further when fractional steps between subsequent ray directions are used. However, CPU-based rendering methods are still an active field of research as some of these methods clearly yield the best possible image quality for rendering a volumetric data set. In addition, such methods are not in need of dedicated hardware, thus enabling their execution on a large amount of computers.

Bibliography

[1] *Cg Developer Website*. http://developer.nvidia.com/cg.

[2] *OpenGL Website*. http://www.opengl.org.

[3] *Sampling Theory 101*. http://graphics.cs.ucdavis.edu/ okreylos/PhDStudies/Winter2000/SamplingTheory.html,
2000.

[4] *DirectX SDK*. http://msdn.microsoft.com/directx, 2001.

[5] *Cg Toolkit - User's Manual*. http://developer.nvidia.com/cg, 2004.

[6] *NVIDIA GPU Programming Guide*. http://developer.nvidia.com/cg, 2004.

[7] K. AKELEY, *The Silicon Graphics 4D/240GTX Superworkstation*, IEEE Computer Graphics and Applications, 9 (1989), pp. 71–83.

[8] T. AKENINE-MÖLLER and E. HAINES, *Real-Time Rendering, Second Edition*, A K Peters, Ltd., 2002.

[9] A. APODACA and L. GRITZ, eds., *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufmann Publishers, Inc., 1999.

[10] ATI TECHNOLOGIES, INC., *Radeon X850 Graphics Technology*.
http://www.ati.com/products/radeonx850/.

[11] R. AVILA, T. HE, L. HONG, A. KAUFMAN, H. PFISTER, C. SILVA, L. SOBIERAJSKI, and S. WANG, *VolVis: a diversified volume visualization system*, Visualization, 1994.

[12] D. BARTZ and M. MEIŠNER, *Voxels versus plygons: A comparative approach for volume graphics*, Proceedings on Visualization, 1999.

[13] BIONICFX, *BionicFX Announces Audio Processing on NVIDIA GPU*. www.bionicfx.com, 2004.

[14] J. F. BLINN, *Jim Blinn's Corner: Image compositing-theory*, IEEE Computer Graphics and Applications, 14 (1994).

[15] J. F. BLINN and M. E. NEWELL, *Texture and reflection in computer generated images*, Communications of the ACM, 19 (1976), pp. 542–547.

[16] B. CABRAL, N. CAM, and J. FORAN, *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, IEEE Symposium on Volume Visualization, 1994.

[17] R. COOK, *Shade Trees*, SIGGRAPH, 1984, pp. 223–231.

[18] N. CORPORATION, *GeForce 6 Series*. http://www.nvidia.com/page/geforce6.html.

[19] J. DANSKIN and P. HANRAHAN, *Fast algorithms for volume raytracing*, Workshop on Volume Visualization, 1992.

[20] P. DIEFENBACH and N. BADLER, *Multi-Pass Pipeline Rendering: Realism for Dynamic Environments*, Proceedings 1997 Symposium on Interactive 3D Graphics, 1997.

[21] D. EBERT, F. K. MUSGRAVE, D. PEACHEY, K. PERLIN, and S. WORLEY, *Texturing and Modeling: A Procedural Approach*, Academic Press, 1998.

[22] K. ENGEL, M. KRAUS, and T. ERTL, *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*, Graphics Hardware, 2001.

[23] W. ENGEL, ed., *ShaderX*, Wordware, 2002. http://www.shaderx.com.

[24] R. FERNANDO and M. J. KILGARD, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addision-Wesley, 2003.

[25] H. GOURAUD, *Continuous shading of curved surfaces*, IEEE Transactions on Computers, C-20 (1971), pp. 623–629.

[26] N. K. GOVINDARAJU, B. LLOYD, W. WANG, M. LIN, and D. MANOCHA, *Fast Computation of Database Operators using Graphic Processors*, SIGMOD, 2004.

[27] M. HADWIGER, J. M. KNISS, K. ENGEL, and C. REZK-SALAMA, *High-Quality Volume Graphics on Consumer PC Hardware*, SIGGRAPH, 2002.

[28] M. HADWIGER, T. THEUŠL, H. HAUSER, and E. GRÖLLER, *Hardware-accelerated high-quality filtering on PC hardware*, Proceedings of Vision, Modeling, and Visualization, 2001.

[29] M. HADWIGER, I. VIOLA, and H. HAUSER, *Fast convolution with high-resolution filters*. Technical Report of the VRVis Research Center for Virtual Reality and Visualization, 2002.

[30] M. J. HARRIS, *General purpose computation using graphics hardware*. http://www.gpgpu.org.

[31] M. J. HARRIS and A. LASTRA, *Real-time cloud rendering*, Eurographics, 2001.

[32] K. H. HÖHNE, B. PFIESSER, A. POMMERT, M. RIEMER, T. SCHIEMANN, R. SCHUBERT, and U. TIEDE, *A virtual body model for surgical education and rehearsal*, vol. 29, IEEE Computer, 1996.

[33] J. HUANG, R. CRAWFIS, and D. STREDNEY, *Edge preservation in volume rendering using splatting*, Symposium on Volume Visualization, 1998.

[34] J. KAJIYA and B. V. HERZEN, *Ray casting volume densities*, vol. 18, Computer Graphics, 1984.

[35] K. KANEDA, T. OKAMOTO, E. NAKAMAE, and T. NISHITA, *Highly realistic visual simulation of outdoor scenes under various atmospheric conditions*, Proceedings on Computer Graphics, 1990.

[36] D. B. J. KESSENRICH and R. ROST, *OpenGL 2.0 Shading Language*, Addison-Wesley, 2003.

[37] R. G. KEYS, *Cubic convolution interpolation for digital image processing*, IEEE Transactions on Acoustics, Speech, and Signal Processing, 1981.

[38] J. KRÜGER and R. WESTERMANN, *Acceleration Techniques for GPU-based Volume Rendering*, IEEE Visualization, 2003.

[39] J. KRÜGER and R. WESTERMANN, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*, SIGGRAPH, 2003.

[40] P. LACROUTE and M. LEVOY, *Fast volume rendering using a shear-warp factorization of the viewing transform*, SIGGRAPH, 1994.

[41] E. LAMAR, B. HAMANN, and K. JOY, *Multiresolution Techniques for Interactive Texture-based Volume Visualization*, In Proceedings of IEEE Visualization, 1999.

[42] D. A. LAROSE, *Iterative X-ray/CT Registration Using Accelerated Volume Rendering*, PhD thesis, Carnegie Mellon University, 2001.

[43] A. LASTRA, S. MOLNAR, M. OLANO, and Y. WANG, *Real-Time Programmable Shading*, Proceedings 1995 Symposium on Interactive 3D Graphics, 1995.

[44] A. LEFOHN, J. M. KNISS, C. HANSEN, and R. WHITAKER, *A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets*, IEEE Transactions on Visualization and Computer Graphics, 2004.

[45] J. LENGYEL, M. REICHERT, B. R. DONALD, and D. GREENBERG, *Real-time robot motion planning using rasterizing computer graphics hardware*, SIGGRAPH, 1990.

[46] M. LEVOY, *Display of surfaces from volume data*, IEEE Computer Graphics and Applications, 1988.

[47] M. LEVOY, *Efficient ray tracing of volume data*, vol. 9, ACM Transactions on Computer Graphics, 1990.

[48] E. LINDHOLM, M. KILGARD, and H. MORETON, *A User-Programmable Vertex Engine*, SIGGRAPH, 2001, pp. 149–158.

[49] W. E. LORENSEN and H. E. CLINE, *Marching Cubes: A high resolution 3D surface construction algorithm*, SIGGRAPH, 1987.

[50] S. R. MARSCHNER and R. J. LOBB, *An Evaluation of Reconstruction Filters for Volume Rendering*, Visualization, 1994.

[51] C. MAUGHAN and M. WLOKA, *Vertex Shader Introduction*. http://developer.nvidia.com, 2001.

[52] N. MAX, *Atmospheric illumination and shadows*, vol. 20, Computer Graphics, 1986.

[53] N. MAX, *Light diffusion through clouds and haze*, vol. 33, Computer Vision, Graphics, and Image Processing, 1986.

[54] N. MAX, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics, 1995.

[55] M. D. MCCOOL, J. ANG, and A. AHMAD, *Homomorphic Factorization of BRDFs for High-Performance Rendering*, SIGGRAPH, 2001.

[56] M. MEIŠNER, J. HUANG, D. BARTZ, K. MUELLER, and R. CRAWFIS, *A Practical Evaluation of Popular Volume Rendering Algorithms*, Volume Visualization, 2000.

[57] D. P. MITCHELL and A. N. NETRAVALI, *Reconstruction filters in computer graphics*, SIGGRAPH, 1988.

[58] K. MORELAND and E. ANGEL, *The FFT on a GPU*, SIGGRAPH, 2003.

[59] B. MORET and H. SHAPIRO, *Algorithms from P to NP*, 1991.

[60] K. MUELLER, T. MOELLER, and R. CRAWFIS, *Splatting without the blur*, Visualization, 1999.

[61] G. NIELSON and J. TVEDT, *Comparing methods of interpolation for scattered volumetric data*. State of the Art in Computer Graphics - Aspects of Visualization, 1994.

[62] T. NISHITA, Y. MIYAWAKI, and E. NAKAMAE, *A shading model for atmospheric scattering considering luminous intensity of light sources*, vol. 21, Computer Graphics, 1987.

[63] M. OLANO and A. LASTRA, *A Shading Language on Graphics Hardware: The PixelFlow Shading System*, SIGGRAPH, 1998.

[64] OPENGL ARCHITECTURE REVIEW BOARD:, *OpenGL 2.0 Shading Language*, Addison-Wesley, 2003.

[65] OPENGL ARCHITECTURE REVIEW BOARD: M. WOO, J. NEIDER, T. DAVIS, and D. SHREINER, *OpenGL Programming Guide, Third Edition*, Addison-Wesley, 1999.

[66] A. V. OPPENHEIM and R. W. SCHAFER, *Digital Signal Processing*, Prentice Hall, Englewood Cliffs, 1975.

[67] K. PALLISTER, *Rendering to Texture Surfaces Using DirectX7*. http://www.gamasutra.com/features/19991112/pallister_01.htm.

[68] S. PARKER, P. SHIRLEY, Y. LIVNAT, C. HANSEN, and P. SLOAN, *Interactive ray tracing for isosurface rendering*, Proceedings on Visualization, 1998.

[69] M. PEERCY, M. OLANO, J. AIREY, and J. UNGAR, *Interactive Multi-Pass Programmable Shading*, SIGGRAPH, 2000.

[70] H. PFISTER, J. HARDENBERGH, J. KNITTEL, H. LAUER, and L. SEILER, *The VolumePro real-time ray-casting system*, Proceedings of the 26th annual conference on Computer graphics and interactive techniques, 1999.

[71] B. T. PHONG, *Illumination for Computer Generated Pictures*, Communications of the ACM, 18 (1975), pp. 311–317.

[72] PIXAR, *The RenderMan Interface*, 2000.

[73] J. PROAKIS and D. MANOLAKIS, *Digital Signal Processing : Principles, Algorithms, and Applications*, Macmillan Publishing Company, 1992.

[74] K. PROUDFOOT, W. R. MARK, S. TZVETKOV, and P. HANRAHAN, *A Real-Time Procedural Shading System for Programmable Graphics Hardware*, SIGGRAPH, 2001.

[75] C. REZK-SALAMA, K. ENGEL, M. BAUER, G. GREINER, and T. ERTL, *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Texturing and Multi-Stage Rasterization*, SIGGRAPH, 2000.

[76] M. RUMPF and R. STRZODKA, *Level segmentation in graphics hardware*, vol. 3, ICIP, 2001, pp. 1103–1106.

[77] J. SCHNEIDER and R. WESTERMANN, *Compression Domain Volume Rendering*, IEEE Visualization 2003, 2003.

[78] P. SHIRLEY and A. TUCHMAN, *A polygonal approximation to direct scalar volume rendering*, Symposium on Volume Visualization, 1990.

[79] I. SILICON GRAPHICS, *Silicon Graphics, Inc. Visualization Products Page*.

[80] J. STEWART, *Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes*, Graphics Interface, 2001.

[81] U. TIEDE, , T. SCHIEMANN, and K. H. HÖHNE, *High quality rendering of attributed volume data*, Visualization, 1998.

[82] U. TIEDE, K. H. HÖHNE, M. BOMANS, A. POMMERT, M. RIEMER, and G. WIEBECKE, *Investigation of medical 3D-rendering algorithms*, vol. 10, IEEE Computer Graphics & Applications, 1990.

[83] H. TUY and L. TUY, *Direct 2D display of 3D objects*, vol. 4, IEEE Computer Graphics & Applications, 1984.

[84] S. UPSTILL, ed., *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, 1990.

[85] D. WEISKOPF, M. HOPF, and T. ERTL, *Hardwareaccelerated visualization of time-varying 2d and 3d vector fields by texture advection via programmable per-pixel operations*, VMV, 2001.

[86] R. WESTERMANN and T. ERTL, *Efficiently using graphics hardware in volume rendering applications*, SIGGRAPH, 1998.

[87] L. WESTOVER, *Footprint evaluation for volume rendering*, SIGGRAPH, 1990.

[88] C. M. WITTENBRINK, T. MALZBENDER, and M. E. GOSS, *Opacity-weighted color interpolation for volume sampling*, In Proceedings of IEEE Symposium on Volume Visualization, 1998, pp. 135–142.

[89] M. WLOKA, *Implementation of 'Missing' Vertx Shader Instructions*. http://developer.nvidia.com, 2000.

[90] R. YAGEL and Z. SHI, *Accelerating volume animation by space-leaping*, Visualization, 1993.