Real-Time Volume Graphics

Klaus Engel[†] Markus Hadwiger[‡] Joe M. Kniss[§] Christof Rezk-Salama[¶]

Siemens Corporate Research[†] VRVis Research Center[‡] University of Utah[§] University of Siegen[¶]

Abstract

This full-day tutorial covers high-quality real-time volume rendering techniques for consumer graphics hardware. In addition to the traditional field of scientific visualization, the interest in applying these techniques for visual arts and real-time rendering is steadily growing. This tutorial covers applications for science, visual arts and entertainment, such as medical visualization, visual effects and computer games. Participants will learn techniques for harnessing the power of consumer graphics hardware and high-level shading languages for real-time rendering of volumetric data and effects. Beginning with a short theoretical part, the basic texture-based approaches are explained. These basic algorithms are improved and expanded incrementally throughout the tutorial. Special attention is paid to latest developments in GPU ray casting.
We will cover local and global illumination, scattering, and participating media. GPU optimization techniques are explained in detail, such as pre-integration, space leaping, occlusion queries, early ray termination and level-of-detail. We will show efficient techniques for clipping and voxelization, and for rendering implicit surfaces. Participants will learn to deal with large volume data, segmented volumes and to apply higher-order filtering, and non-photorealistic techniques to improve image quality. Further presentations cover multi-dimensional classification and transfer function design, as well as techniques for volumetric modeling, animation and deformation. Participants are provided with code samples covering important implementation details usually omitted in publications.

1. Prerequisites

Participants should have a working knowledge of computer graphics and some background in graphics programming APIs such as OpenGL or DirectX. Familiarity with GPU shading languages is helpful, but not necessarily required.

2. Level of Difficulty

Intermediate.

3. Intended Audience

The basic modules will be of value for all people who want to learn more about real-time volume graphics. The advanced topics are intended for scientists, who want to visualize large data, and for graphics and game programmers who want to generate convincing visual effects and render participating media.

4. Syllabus

This section gives a detailed structure of the tutorial.

1 Theoretical Background [15 min]

- Physical Model of Light Transport
- Volume Rendering Integral

2 GPU Programming [15 min]

- The Graphics Pipeline
- Vertex and Fragment Processing
- The High-level Shading Language Cg

[†] klaus.engel@scr.siemens.com
[‡] msh@vrvis.at
[§] jmk@cs.utah.edu
[¶] rezk@fb12.uni-siegen.de

3 Basic GPU-Based Volume Rendering [30 min]

- 2D Texture-Based Volume Rendering
- 3D Texture-Based Approach
- 2D Multi-Textures-Based Approach
- Vertex Programs

4 GPU-Based Ray-Casting [30 min]

- Basic Structure of Ray-Casting
- Performance Aspects and Acceleration Methods
- Object-Order Empty Space Skipping
- Isosurface Ray-Casting
- Ray-Casting of Unstructured Grids

COFFEE BREAK

5 Transfer Functions [30 min]

- Classification
- Pre- versus Post-Classification
- Pre-Integrated Transfer Functions

6 Local Volume Illumination [30 min]

- Gradient-Based Illumination
- Local Illumination Models
- Pre-Computed Gradients
- On-the-fly Gradients
- Environment Mapping

7 Global Volume Illumination [30 min]

- Volumetric Shadows
- Phase Functions
- Translucent Volume Lighting

LUNCH BREAK

8 Improving Performance [20 min]

- Swizzling of volume data
- Asynchronous Data Upload
- Empty Space Leaping
- Occlusion Culling
- Early Ray-Termination
- Deferred Shading
- Image Downscaling

9 Improving Image Quality [20 min]

- Sampling Artifacts
- Filtering Artifacts
- Classification Artifacts
- Shading Artifacts
- Blending Artifacts

10 Advanced Transfer Functions [20 min]

- Image Data Versus Scalar Field
- Multi-Dimensional Transfer Functions
- Engineering Multi-Dimensional Transfer Functions
- Transfer Function User Interfaces

11 Game Developer's Guide to Volume Graphics [30 min]

- Volume Graphics in Games
- Differences From Stand-Alone Volume Rendering
- Integrating Volumes With Scene Geometry
- A Simple Volume Ray-Caster for Games
- Volumetric Effects and Simulation
- Integrating Volumes With Scene Shadowing and Lighting

COFFEE BREAK

12 Volume Modeling, Deformation and Animation [30 min]

- Rendering into a 3D Texture
- Voxelization
- Procedural Modeling
- Compositing and Image Processing
- Deformation in Model Space
- Deformation in Texture Space
- Deformation and Illumination
- Animation Techniques

13 Non-Photorealistic and Illustrative Techniques [30 min]

- Basic NPR Shading Models
- Contour Rendering
- Surface and Isosurface Curvature
- Deferred Shading of Isosurfaces
- Curvature-Based Isosurface Illustration

14 Large Volume Data [30 min]

- Memory Performance Considerations
- Bricking
- Multi-Resolution Volume Rendering
- Build-in Texture Compression
- Wavelet Compression
- Packing Techniques
- Vector Quantization

5. Course History

We have presented a course with a subset of these topics before at SIGGRAPH 2002 (with a narrow focus on scientific visualization) and at SIGGRAPH 2004. The course notes

have evolved into a book that was published by A K Peters, Ltd. at SIGGRAPH 2006. For 2006, the course has been redesigned from a didactic point of view and the scope has been again broadened to keep it state-of-the-art and take into account the growing interest of game developers and the visual arts communities.

6. Course Presenter Information

Klaus Engel,
Siemens Corporate Research, Princeton, USA,
klaus.engel@scr.siemens.com

Klaus Engel is a researcher for Siemens Corporate Research, Inc. in Princeton/NJ. He received a PhD from the University of Stuttgart in 2002 and a Diplom (Masters) of computer science from the University of Erlangen in 1997. He has presented the results of his research at international conferences and in journals, including IEEE Visualization, Visualization Symposium, IEEE Transactions on Visualization and Computer Graphics and Graphics Hardware. In 2000 and 2001, his papers *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization* and "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading" have won the best paper awards at the SIGGRAPH/Eurographics Workshop on Graphics Hardware. Klaus has regularly taught courses and seminars on computer graphics, visualization and computer games algorithms. In his PhD thesis he investigated "Strategies and Algorithms for Distributed Volume-Visualization on Different Graphics-Hardware Architectures".

Markus Hadwiger,
VRVis Research Center, Vienna, Austria
msh@vrvis.at

Markus Hadwiger is a senior researcher in the Medical Visualization department at the VRVis Research Center in Vienna, Austria. He received a PhD degree in computer science from the Vienna University of Technology in 2004, concentrating on high quality real-time volume rendering and texture filtering with graphics hardware. He is regularly teaching courses and seminars on computer graphics, visualization, and game programming, including courses at the annual SIGGRAPH conference, and tutorials at IEEE Visualization and Eurographics. Before concentrating on scientific visualization, he was working in the area of computer games and interactive entertainment.

Joe M. Kniss,
University of Utah, USA
jmk@cs.utah.edu

Joe recently finished his Ph.D. in computer science at the Unversity of Utah's School of Computing. As a member of the Scientific Computing and Imaging Institute, he has done research in the areas of volume rendering, volume light transport, human-computer interaction, and image processing. His Ph.D. work was supported by the Department of Energy High-Performance Computer Science Graduate Fellowship. In his free time, Joe enjoys snowboarding, skateboarding, music, carpentry, and art.

Christof Rezk-Salama,
University of Siegen, Germany
rezk@fb12.uni-siegen.de

Christof Rezk Salama is an assistant professor at the Computer Graphics and Multimedia Group of the University of Siegen, Germany. Before that he was a research engineer at Siemens Medical Solutions. He has received a PhD at the Computer Graphics Group in Erlangen in 2002 as a scholarship holder at the graduate college "3D Image Analysis and Synthesis". His research interests include scientific visualization, GPU programming, real-time rendering, and computer animation. He is regularly holding lectures and teaching courses and seminars on computer graphics, scientific visualization, character animation and graphics programming. He has gained practical experience in applying computer graphics to several scientific projects in medicine, geology and archaeology. He is member of ACM SIGGRAPH, and the Gesellschaft für Informatik.

7. Organizer contact information

Dr. Markus Hadwiger
VRVis Research Center
Donau City Strasse 1
A-1220 Vienna, Austria
mailto:msh@vrvis.at
phone: +43 1 20501 30701
fax: +43 1 20501 30900

Eurographics 2006 Tutorial Notes T7

Real-Time Volume Graphics

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

Real-Time Volume Graphics

Abstract This tutorial covers high-quality real-time volume rendering techniques for consumer graphics hardware. In addition to the traditional field of scientific visualization, the interest in applying these techniques for visual arts and real-time rendering is steadily growing. This tutorial covers applications for science, visual arts and entertainment, such as medical visualization, visual effects and computer games. Participants will learn techniques for harnessing the power of consumer graphics hardware and high-level shading languages for real-time rendering of volumetric data and effects. Beginning with a short theoretical part, the basic texture-based approaches are explained. These basic algorithms are improved and expanded incrementally throughout the tutorial. Special attention is paid to latest developments in GPU ray casting.

We will cover local and global illumination, scattering, and participating media. GPU optimization techniques are explained in detail, such as pre-integration, space leaping, occlusion queries, early ray termination and levelof- detail. We will show efficient techniques for clipping and voxelization, and for rendering implicit surfaces. Participants will learn to deal with large volume data, segmented volumes and to apply higher-order filtering, and non-photorealistic techniques to improve image quality. Further presentations cover multi-dimensional classification and transfer function design, as well as techniques for volumetric modeling, animation and deformation. Participants are provided with code samples covering important implementation details usually omitted in publications.

Prerequisites Participants should have a working knowledge of computer graphics and some background in graphics programming APIs such as OpenGL or DirectX. Familiarity with GPU shading languages is helpful, but not necessarily required.

Level of Difficulty Intermediate.

Lecturers

Klaus Engel
Siemens Corporate Research
Princeton, USA
email: klaus.engel@scr.siemens.com

Klaus Engel is a researcher for Siemens Corporate Research, Inc. in Princeton/NJ. He received a PhD from the University of Stuttgart in 2002 and a Diplom (Masters) of computer science from the University of Erlangen in 1997. He has presented the results of his research at international conferences and in journals, including IEEE Visualization, Visualization Symposium, IEEE Transactions on Visualization and Computer Graphics and Graphics Hardware. In 2000 and 2001, his papers *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization* and *High- Quality Pre-Integrated Volume Rendering Using Hardware- Accelerated Pixel Shading* have won the best paper awards at the SIGGRAPH/Eurographics Workshop on Graphics Hardware. Klaus has regularly taught courses and seminars on computer graphics, visualization and computer games algorithms. In his PhD thesis he investigated *Strategies and Algorithms for Distributed Volume-Visualization on Different Graphics-Hardware Architectures.*

Markus Hadwiger
VRVis Research Center
Donau-City-Strasse 1, A-1220 Vienna, Austria
email: msh@vrvis.at

Markus Hadwiger is a senior researcher in the Medical Visualization department at the VRVis Research Center in Vienna, Austria. He received a PhD degree in computer science from the Vienna University of Technology in 2004, concentrating on high quality real-time volume rendering and texture filtering with graphics hardware. He is regularly teaching courses and seminars on computer graphics, visualization, and game programming, including courses at the annual SIGGRAPH conference, and tutorials at IEEE Visualization and Eurographics. Before concentrating on scientific visualization, he was working in the area of computer games and interactive entertainment.

Joe Michael Kniss
Scientific Computing and Imaging Institute
University of Utah
50 S. Central Campus Dr. #3490, Salt Lake City, UT 84112
email: jmk@cs.utah.edu

Joe recently finished his Ph.D. in computer science at the Unversity of Utahs School of Computing. As a member of the Scientific Computing and Imaging Institute, he has done research in the areas of volume rendering, volume light transport, human-computer interaction, and image processing. His Ph.D. work was supported by the Department of Energy High-Performance Computer Science Graduate Fellowship. In his free time, Joe enjoys snowboarding, skateboarding, music, carpentry, and art.

Christof Rezk Salama
Computergraphik und Multimediasysteme
University of Siegen
Hölderlinstr. 3, 57068 Siegen, Germany
email: rezk@fb12.uni-siegen.de

Christof Rezk Salama is an assistant professor at the Computer Graphics and Multimedia Group of the University of Siegen, Germany. Before that he was a research engineer at Siemens Medical Solutions. He has received a PhD at the Computer Graphics Group in Erlangen in 2002 as a scholarship holder at the graduate college "3D Image Analysis and Synthesis". His research interests include scientific visualization, GPU programming, real-time rendering, and computer animation. He is regularly holding lectures and teaching courses and seminars on computer graphics, scientific visualization, character animation and graphics programming. He has gained practical experience in applying computer graphics to several scientific projects in medicine, geology and archaeology. He is member of ACM SIGGRAPH, and the Gesellschaft für Informatik.

Course Syllabus

MORNING

Theoretical Background [Ch. Rezk Salama] **9:00 – 9:15**
- Physical Model of Light Transport
- Volume Rendering Integral

GPU Programming [Ch. Rezk Salama] **9:15 – 9:30**
- The Graphics Pipeline
- Vertex and Fragment Processing
- The High-Level Shading Language Cg

Basic GPU-Based Volume Rendering [Ch. Rezk Salama] **9:30 – 10:00**
- 2D Texture-Based Volume Rendering
- 3D Texture-Based Approach
- 2D Multi-Textures-Based Approach
- Vertex Programs

GPU-Based Ray-Casting [M. Hadwiger] **10:00-10:30**
- Basic Structure of Ray-Casting
- Performance Aspects and Acceleration Methods
- Object-Order Empty Space Skipping
- Isosurface Ray-Casting
- Ray-Casting of Unstructured Grids

COFFEE BREAK **10:30-11:00**

11:00-11:30 **Transfer Functions** [K. Engel]
- Classification
- Pre- versus Post-Classification
- Pre-Integrated Transfer Functions

11:30-12:00 **Local Volume Illumination** [Ch. Rezk Salama]
- Gradient-Based Illumination
- Local Illumination Models
- Pre-Computed Gradients
- On-the-fly Gradients
- Environment Mapping

12:00-12:30 **Global Volume Illumination** [J. Kniss]
- Volumetric Shadows
- Phase Functions
- Translucent Volume Lighting

12:30-14:00 **LUNCH BREAK**

AFTERNOON

14:00-14:20 **Improving Performance** [K. Engel]
- Swizzling of Volume Data
- Asynchronous Data Upload
- Empty Space Leaping
- Occlusion Culling
- Early Ray-Termination
- Deferred Shading
- Image Downscaling

14:20-14:40 **Improving Image Quality** [K. Engel]
- Sampling Artifacts
- Filtering Artifacts
- Classification Artifacts
- Shading Artifacts
- Blending Artifacts

Advanced Transfer Functions [J. Kniss] **14:40-15:00**
- Image Data Versus Scalar Field
- Multi-Dimensional Transfer Functions
- Engineering Multi-Dimensional Transfer Functions
- Transfer Function User Interfaces

Game Developer's Guide to Volume Graphics [M. Hadwiger] **15:00-15:30**
- Volume Graphics in Games
- Differences From Stand-Alone Volume Rendering
- A Simple Volume Ray-Caster for Games
- Volumetric Effects and Simulation
- Integrating Volumes with Scene Shadowing and Lighting

COFFEE BREAK **15:30-16:00**

Volume Modeling, Deformation and Animation [Ch. Rezk Salama] **16:00-16:30**
- Rendering into a 3D Texture
- Voxelization
- Procedural Modeling
- Compositing and Image Processing
- Deformation in Model Space
- Deformation in Texture Space
- Deformation and Illumination
- Animation Techniques

Non-Photorealistic and Illustrative Techniques [M. Hadwiger] **16:30-17:00**
- Basic NPR Shading Models
- Contour Rendering
- Surface and Isosurface Curvature
- Deferred Shading of Isosurfaces
- Curvature-Based Isosurface Illustration

Large Volume Data [K. Engel] **17:00-17:30**
- Memory Performance Considerations
- Bricking
- Multi-Resolution Volume Rendering
- Built-in Texture Compression
- Wavelet Compression
- Packing Techniques
- Vector Quantization

Contents

Course Notes T7

Real-Time Volume Graphics

Introduction

Klaus Engel

Siemens Corporate Research, Princeton, USA

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Joe M. Kniss

SCI Institute, University of Utah, USA

Christof Rezk Salama

University of Siegen, Germany

Volume Rendering

In traditional modeling, 3D objects are created using surface representations such as polygonal meshes, NURBS patches or subdivision surfaces. In the traditional modeling paradigm, visual properties of surfaces, such as color, roughness and reflectance, are modeled by means of a shading algorithm, which might be as simple as the Phong model or as complex as a fully-featured shift-variant anisotropic BRDF. Since light transport is evaluated only at points on the surface, these methods usually lack the ability to account for light interaction which is taking place in the atmosphere or in the interior of an object.

Contrary to surface rendering, volume rendering [28, 9] describes a wide range of techniques for generating images from three-dimensional scalar data. These techniques are originally motivated by scientific visualization, where volume data is acquired by measurement or numerical simulation of natural phenomena. Typical examples are medical data of the interior of the human body obtained by computed tomography (CT) or magnetic resonance imaging (MRI). Other examples are computational fluid dynamics (CFD), geological and seismic data, as well as abstract mathematical data such as 3D probability distributions of pseudo random numbers.

With the evolution of efficient volume rendering techniques, volumetric data is becoming more and more important also for visual arts and computer games. Volume data is ideal to describe fuzzy objects, such as fluids, gases and natural phenomena like clouds, fog, and fire. Many artists and researchers have generated volume data synthetically to supplement surface models, i.e., procedurally [11], which is especially useful for rendering high-quality special effects.

Although volumetric data are more difficult to visualize than surfaces, it is both worthwhile and rewarding to render them as truly three-dimensional entities without falling back to 2D subsets.

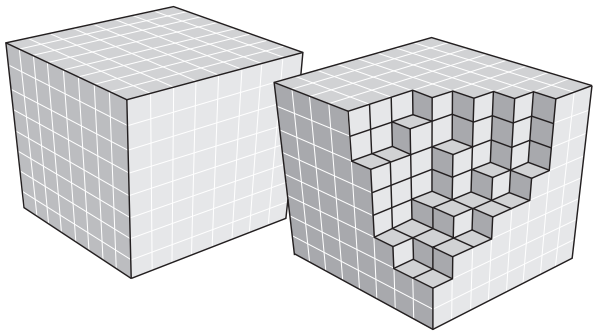


Figure 1.1: Voxels constituting a volumetric object after it has been discretized.

1.1 Volume Data

Discrete volume data set can be thought of as a simple three-dimensional array of cubic elements (voxels[1]) [22], each representing a unit of space (Figure 1.1).

Although imagining voxels as tiny cubes is easy and might help to visualize the immediate vicinity of individual voxels, it is more appropriate to identify each voxel with a sample obtained at a single infinitesimally small point from a continuous three-dimensional signal

$$f(\vec{x}) \in \mathbb{R} \quad \text{with} \quad \vec{x} \in \mathbb{R}^3. \tag{1.1}$$

Provided that the continuous signal is band-limited with a cut-off-frequency ν_s, sampling theory allows the exact reconstruction, if the signal is evenly sampled at more than twice the cut-off-frequency (Nyquist rate). However, there are two major problems which prohibit the ideal reconstruction of sampled volume data in practise.

- Ideal reconstruction according to sampling theory requires the convolution of the sample points with a *sinc* function (Figure 1.2a) in the spacial domain. For the one-dimensional case, the sinc function reads

  $$sinc(x) \;=\; \frac{\sin(\pi x)}{\pi x}. \tag{1.2}$$

  The three-dimensional version of this function is simply obtained by tensor-product. Note that this function has infinite extent. Thus, for an exact reconstruction of the original signal at an arbitrary position *all* the sampling points must be considered, not only

[1] volume elements

those in a local neighborhood. This turns out to be computationally intractable in practise.

- Real-life data in general does not represent a band-limited signal. Any sharp boundary between different materials represents a step function which has infinite extent in the frequency domain. Sampling and reconstruction of a signal which is not band-limited will produce aliasing artifacts.

In order to reconstruct a continuous signal from an array of voxels in practise the ideal 3D *sinc* filter is usually replaced by either a box filter (Figure 1.2a) or a tent filter (Figure 1.2b). The box filter calculates nearest-neighbor interpolation, which results in sharp discontinuities between neighboring cells and a rather blocky appearance. Trilinear interpolation, which is achieved by convolution with a 3D tent filter, represents a good trade-off between computational cost and smoothness of the output signal.

In Part 7 of these course notes, we will investigate higher-order reconstruction methods for GPU-based real-time volume rendering [17, 18].

1.2 Direct Volume Rendering

In comparison to the indirect methods, which try to extract a surface description from the volume data in a preprocessing step, direct methods display the voxel data by evaluating an *optical model* which describes how the volume emits, reflects, scatters, absorbs and occludes light [30]. The scalar value is virtually mapped to physical quantities which describe light interaction at the respective point in 3D-space. This mapping is

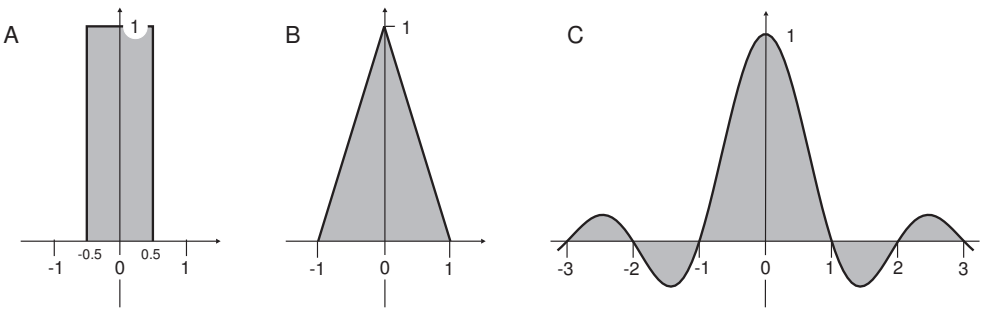


Figure 1.2: Reconstruction filters for one-dimensional signals. In practise, box filter *(A)* and tent filter *(B)* are used instead of the ideal *sinc*-filter *(C)*.

termed *classification* (see Part 4 of the course notes) and is usually performed by means of a transfer function. The physical quantities are then used for images synthesis. Different optical models for direct volume rendering are described in section 1.2.1.

During image synthesis the light propagation is computed by integrating light interaction effects along viewing rays based on the optical model. The corresponding integral is known as the *volume rendering integral*, which is described in section 1.2.2. Naturally, under real-world conditions this integral is solved numerically. Optionally, the volume can be shaded according to the *illumination* from external light sources, which is the topic of Part 3.

1.2.1 Optical Models

Almost every direct volume rendering algorithms regards the volume as a distribution of light-emitting particles of a certain density. These densities are more or less directly mapped to RGBA quadruplets for compositing along the viewing ray. This procedure, however, is motivated by a physically-based optical model.

The most important optical models for direct volume rendering are described in a survey paper by Nelson Max [30], and we only briefly summarize these models here:

- **Absorption only.** The volume is assumed to consist of cold, perfectly black particles that absorb all the light that impinges on them. They do not emit, or scatter light.

- **Emission only.** The volume is assumed to consist of particles that only emit light, but do not absorb any, since the absorption is negligible.

- **Absorption plus emission.** This optical model is the most common one in direct volume rendering. Particles emit light, and occlude, i.e., absorb, incoming light. However, there is no scattering or indirect illumination.

- **Scattering and shading/shadowing.** This model includes scattering of illumination that is external to a voxel. Light that is scattered can either be assumed to impinge unimpeded from a distant light source, or it can be shadowed by particles between the light and the voxel under consideration.

- **Multiple scattering.** This sophisticated model includes support for incident light that has already been scattered by multiple particles.

The optical model used in all further considerations will be the one of particles simultaneously emitting and absorbing light. The *volume rendering integral* described in the following section also assumes this particular optical model. More sophisticated models account for scattering of light among particles of the volume itself, and also include shadowing and self-shadowing effects.

1.2.2 The Volume Rendering Integral

Every physically-based volume rendering algorithms evaluates the volume rendering integral in one way or the other, even if viewing rays are not employed explicitly by the algorithm. The most basic volume rendering algorithm is ray-casting, covered in Section 1.2.3. It might be considered as the "most direct" numerical method for evaluating this integral. More details are covered below, but for this section it suffices to view ray-casting as a process that, for each pixel in the image to render, casts a single ray from the eye through the pixel's center into the volume, and integrates the optical properties obtained from the encountered volume densities along the ray.

Note that this general description assumes both the volume and the mapping to optical properties to be continuous. In practice, of course, the volume data is discrete and the evaluation of the integral is approximated numerically. In combination with several additional simplifications, the integral is usually substituted by a Riemann sum.

We denote a ray cast into the volume by $\vec{x}(t)$, and parameterize it by the distance t from the eye. The scalar value corresponding to a position along the ray is denoted by $s\big(\vec{x}(t)\big)$. If we employ the emission-absorption model, the volume rendering equation integrates *absorption coefficients* $\kappa(s)$ (accounting for the absorption of light), and *emissive colors* $c(s)$ (accounting for radiant energy actively emitted) along a ray. To keep the equations simple, we denote emission c and absorption coefficients κ as function of the eye distance t instead of the scalar value s:

$$c(t) := c\Big(s\big(\vec{x}(t)\big)\Big) \quad \text{and} \quad \kappa(t) := \kappa\Big(s\big(\vec{x}(t)\big)\Big) \tag{1.3}$$

Figure 1.3 illustrates the idea of emission and absorption. An amount of radiant energy, which is emitted at a distance $t = d$ along the viewing

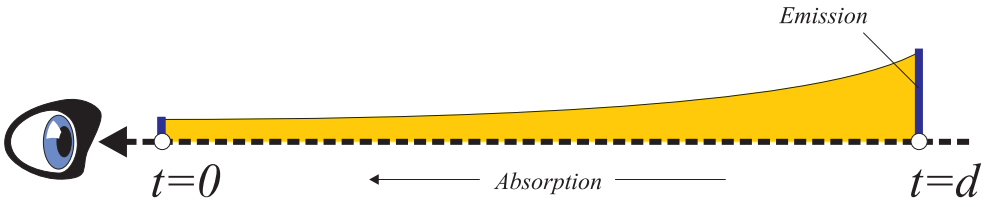


Figure 1.3: An amount of radiant energy emitted at $t = d$ is partially absorbed along the distance d.

ray is continuously absorbed along the distance d until it reaches the eye. This means that only a portion c' of the original radiant energy c emitted at $t = d$ will eventually reach the eye. If there is a constant absorption $\kappa = const$ along the ray, c' amounts to

$$c' = c \cdot e^{-\kappa d} \quad . \tag{1.4}$$

However, if absorption κ is not constant along the ray, but itself depending on the position, the amount of radiant energy c' reaching the eye must be computed by integrating the absorption coefficient along the distance d

$$c' = c \cdot e^{-\int_0^d \kappa(\hat{t})\, d\hat{t}} \quad . \tag{1.5}$$

The integral over the absorption coefficients in the exponent,

$$\tau(d_1, d_2) = \int_{d_1}^{d_2} \kappa(\hat{t})\, d\hat{t} \tag{1.6}$$

is also called the *optical depth*. In this simple example, however, light was only emitted at a single point along the ray. If we want to determine the total amount of radiant energy C reaching the eye from this direction, we must take into account the emitted radiant energy from all possible positions t along the ray:

$$C = \int_0^\infty c(t) \cdot e^{-\tau(0,t)}\, dt \tag{1.7}$$

In practice, this integral is evaluated numerically through either back-to-front or front-to-back compositing (i.e., alpha blending) of samples along the ray, which is most easily illustrated in the method of *ray-casting*.

1.2.3 Ray-Casting

Ray-casting [28] is an image-order direct volume rendering algorithm, which uses straight-forward numerical evaluation of the volume rendering integral (Equation 1.7). For each pixel of the image, a single ray[2] is cast into the scene. At equi-spaced intervals along the ray the discrete volume data is resampled, usually using tri-linear interpolation as reconstruction filter. That is, for each resampling location, the scalar values of eight neighboring voxels are weighted according to their distance to the actual location for which a data value is needed. After resampling, the scalar data value is mapped to optical properties via a lookup table, which yields an RGBA quadruplet that subsumes the corresponding emission and absorption coefficients [28] for this location. The solution of the volume rendering integral is then approximated via alpha blending in either back-to-front or front-to-back order.

The optical depth τ (Equation 1.6), which is the cumulative absorption up to a certain position $\vec{x}(t)$ along the ray, can be approximated by a Riemann sum

$$\tau(0,\, t) \;\approx\; \tilde{\tau}(0,\, t) \;=\; \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t \tag{1.8}$$

with Δt denoting the distance between successive resampling locations. The summation in the exponent can immediately be substituted by a multiplication of exponentiation terms:

$$e^{-\tilde{\tau}(0,\, t)} \;=\; \prod_{i=0}^{\lfloor t/\Delta t \rfloor} e^{-\kappa(i \cdot \Delta t)\, \Delta t} \tag{1.9}$$

Now, we can introduce *opacity* A, well-known from alpha blending, by defining

$$A_i \;=\; 1 - e^{-\kappa(i \cdot \Delta t)\, \Delta t} \tag{1.10}$$

and rewriting equation 1.9 as:

$$e^{-\tilde{\tau}(0,\, t)} \;=\; \prod_{i=0}^{\lfloor t/d \rfloor} (1 - A_j) \tag{1.11}$$

This allows opcaity A_i to be used as an approximation for the absorption of the i-th ray segment, instead of absorption at a single point.

[2] assuming super-sampling is not used for anti-aliasing

Similarly, the emitted color of the i-th ray segment can be approximated by:

$$C_i \;=\; c(i \cdot \Delta t)\,\Delta t \tag{1.12}$$

Having approximated both the emissions and absorptions along a ray, we can now state the approximate evaluation of the volume rendering integral as (denoting the number of samples by $n = \lfloor T/\delta t \rfloor$):

$$\tilde{C} \;=\; \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{1.13}$$

Equation 1.13 can be evaluated iteratively by *alpha blending* in either back-to-front, or front-to-back order.

1.2.4 Alpha Blending

Equation 1.13 can be computed iteratively in back-to-front order by stepping i from $n-1$ to 0:

$$C'_i = C_i + (1 - A_i)C'_{i+1} \tag{1.14}$$

A new value C'_i is calculated from the color C_i and opacity A_i at the current location i, and the composite color C'_{i+1} from the previous location $i+1$. The starting condition is $C'_n = 0$.

Note that in all blending equations, we are using *opacity-weighted colors* [40], which are also known as *associated colors* [6]. An opacity-weighted color is a color that has been pre-multiplied by its associated opacity. This is a very convenient notation, and especially important for interpolation purposes. It can be shown that interpolating color and opacity separately leads to artifacts, whereas interpolating opacity-weighted colors achieves correct results [40].

The following alternative iterative formulation evaluates equation 1.13 in front-to-back order by stepping i from 1 to n:

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{1.15}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{1.16}$$

New values C'_i and A'_i are calculated from the color C_i and opacity A_i at the current location i, and the composited color C'_{i-1} and opacity A'_{i-1} from the previous location $i-1$. The starting condition is $C'_0 = 0$ and $A'_0 = 0$.

Note that front-to-back compositing requires tracking alpha values, whereas back-to-front compositing does not. In a hardware implementation, this means that *destination alpha* must be supported by the frame buffer (i.e., an alpha valued must be stored in the frame buffer, and it must be possible to use it as multiplication factor in blending operations), when front-to-back compositing is used. However, since the major advantage of front-to-back compositing is an optimization commonly called *early ray termination*, where the progression along a ray is terminated as soon as the cumulative alpha value reaches 1.0, and this is difficult to perform in hardware, GPU-based volume rendering usually uses back-to-front compositing.

1.2.5 The Shear-Warp Algorithm

The shear-warp algorithm [26] is a very fast approach for evaluating the volume rendering integral. In contrast to ray-casting, no rays are cast back into the volume, but the volume itself is projected slice by slice onto the image plane. This projection uses bi-linear interpolation within two-dimensional slices, instead of the tri-linear interpolation used by ray-casting.

The basic idea of shear-warp is illustrated in figure 1.4 for the case of orthogonal projection. The projection does not take place directly on the final image plane, but on an intermediate image plane, called the *base plane*, which is aligned with the volume instead of the viewport. Furthermore, the volume itself is *sheared* in order to turn the oblique projection direction into a direction that is perpendicular to the base plane, which allows for an extremely fast implementation of this projection. In such a setup, an entire slice can be projected by simple two-dimensional image resampling. Finally, the base plane image has to be *warped* to the final

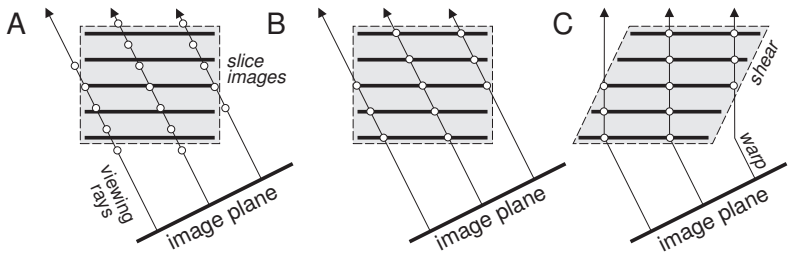


Figure 1.4: The shear-warp algorithm for orthogonal projection.

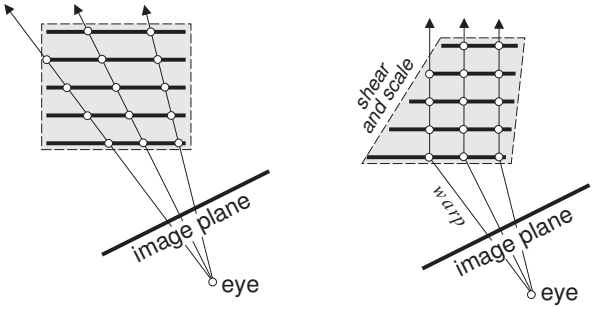


Figure 1.5: The shear-warp algorithm for perspective projection.

image plane. Note that this warp is only necessary once per generated image, not once per slice. Perspective projection can be accommodated similarly, by scaling the volume slices, in addition to shearing them, as depicted in figure 1.5.

The clever approach outlined above, together with additional optimizations, like run-length encoding the volume data, is what makes the shear-warp algorithm probably the fastest software method for volume rendering. Although originally developed for software rendering, we will encounter a principle similar to shear-warp in hardware volume rendering, specifically in the chapter on 2D-texture based hardware volume rendering (3.2). When 2D textures are used to store slices of the volume data, and a stack of such slices is texture-mapped and blended in hardware, bi-linear interpolation is also substituted for tri-linear interpolation, similarly to shear-warp. This is once again possible, because this hardware method also employs object-aligned slices. Also, both shear-warp and 2D-texture based hardware volume rendering require three slice stacks to be stored, and switched according to the current viewing direction. Further details are provided in chapter 3.2.

1.3 Maximum Intensity Projection

Maximum intensity projection (MIP) is a variant of direct volume rendering, where, instead of compositing optical properties, the maximum value encountered along a ray is used to determine the color of the corresponding pixel. An important application area of such a rendering mode, are medical data sets obtained by MRI (magnetic resonance imaging) scanners. Such data sets usually exhibit a significant amount of noise

that can make it hard to extract meaningful iso-surfaces, or define transfer functions that aid the interpretation. When MIP is used, however, the fact that within angiography data sets the data values of vascular structures are higher than the values of the surrounding tissue, can be exploited easily for visualizing them.

In graphics hardware, MIP can be implemented by using a maximum operator when blending into the frame buffer, instead of standard alpha blending. Figure 1.6 shows a comparison of direct volume rendering and MIP used with the same data set.

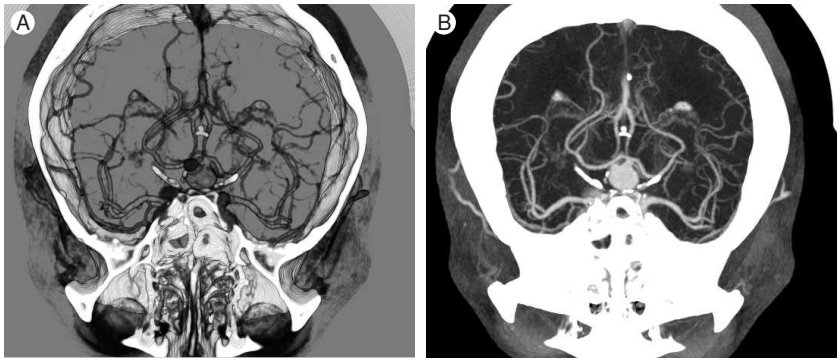


Figure 1.6: A comparison of direct volume rendering (A), and maximum intensity projection (B).

Graphics Hardware

For hardware accelerated rendering, a virtual scene is modeled by the use of planar polygons. The process of converting such a set of polygon into a raster image is called *display traversal*. The majority of 3D graphics hardware implement the display traversal as a fixed sequence of processing stages [15]. The ordering of operations is usually described as a graphics pipeline displayed in Figure 2.1. The input of such a pipeline is a stream of vertices, which are initially generated from the description of a virtual scene by decomposing complex objects into planar polygons *(tessellation)*. The output is the raster image of the virtual scene, that can be displayed on the screen.

The last couple of years have seen a breathtaking evolution of consumer graphics hardware from traditional *fixed-function* architectures (up to 1998) over *configurable* pipelines to *fully programmable* floating-point graphics processors with more than 100 million transistors in 2002. With forthcoming graphics chips, there is still a clear trend towards higher programmability and increasing parallelism.

2.1 The Graphics Pipeline

For a coarse overview the graphics pipeline can be divided into three basic tiers.

Geometry Processing computes linear transformations of the incoming vertices in the 3D spacial domain such as rotation, translation and scaling. Groups of vertices from the stream are finally joined together to form *geometric primitives* (points, lines, triangles and polygons).

Rasterization decomposes the geometric primitives into *fragments*. Each fragment corresponds to a single pixel on the screen. Rasterization also comprises the application of *texture mapping*.

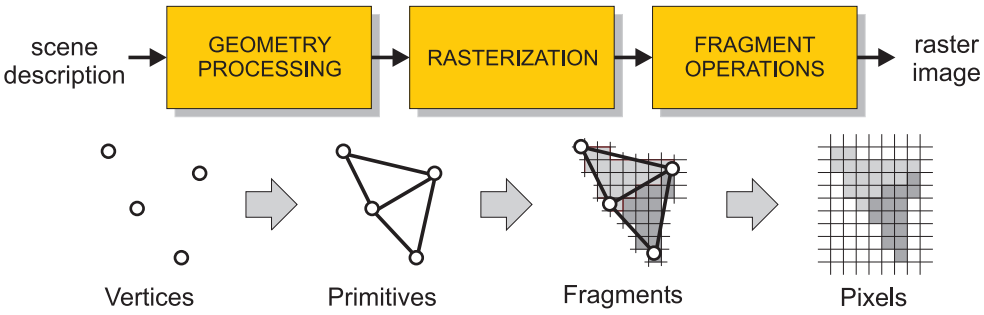


Figure 2.1: The standard graphics pipeline for display traversal.

Fragment Operations are performed subsequently to modify the fragment's attributes, such as color and transparency. Several tests are applied that finally decide whether the incoming fragment is discarded or displayed on the screen.

For the understanding of the new algorithms that have been developed within the scope of this thesis, it is important to exactly know the ordering of operations in this graphics pipeline. In the following sections, we will have a closer look at the different stages.

2.1.1 Geometry Processing

The geometry processing unit performs so-called *per-vertex operations*, i.e operations that modify the incoming stream of vertices. The geometry engine computes linear transformations, such as translation, rotation and projection of the vertices. Local illumination models are also evaluated on a per-vertex basis at this stage of the pipeline. This is the reason why geometry processing is often referred to as *transform & light* unit (T&L). For a detailed description the geometry engine can be further divided into several subunits, as displayed in Figure 2.2.

Modeling Transformation: Transformations which are used to arrange objects and specify their placement within the virtual scene are called *modeling transformations*. They are specified as a 4×4 matrix using homogenous coordinates.

Viewing Transformation: A transformation that is used to specify the camera position and viewing direction is termed *viewing transformation*. This transformation is also specified as a 4×4 matrix.

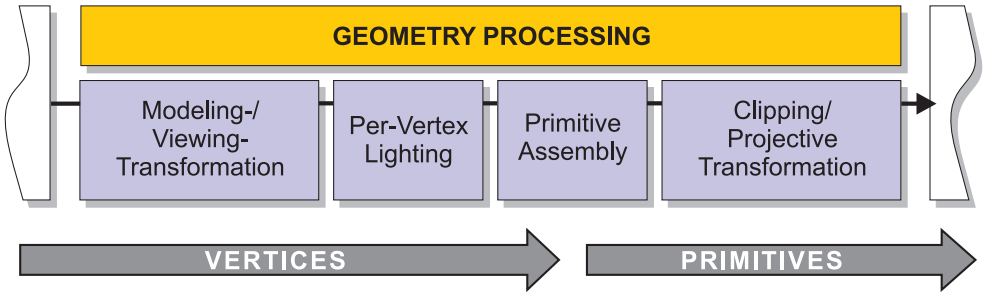


Figure 2.2: Geometry processing as part of the standard graphics pipeline.

Modeling and viewing matrices can be pre-multiplied to form a single *modelview* matrix.

Lighting/Vertex Shading: After the vertices are correctly placed within the virtual scene, the Phong model [33] for local illumination is calculated for each vertex by default. On a programmable GPU, an alternative illumination model can be implemented using a vertex shader. Since illumination requires information about normal vectors and the final viewing direction, it must be performed after modeling and viewing transformation.

Primitive Assembly: Rendering primitives are generated from the incoming vertex stream. Vertices are connected to lines, lines are joined together to form polygons. Arbitrary polygons are usually tessellated into triangles to ensure planarity and to enable interpolation in barycentric coordinates.

Clipping: Polygon and line clipping is applied after primitive assembly to remove those portions of geometry which are is not displayed on the screen.

Perspective Transformation: Perspective transformation computes the projection of the geometric primitive onto the image plane.

Perspective transformation is the final step of the geometry processing stage. All operations that are located after the projection step are performed within the two-dimensional space of the image plane.

2.1.2 Rasterization

Rasterization is the conversion of geometric data into *fragments*. Each fragment corresponds to a square pixel in the resulting image. The

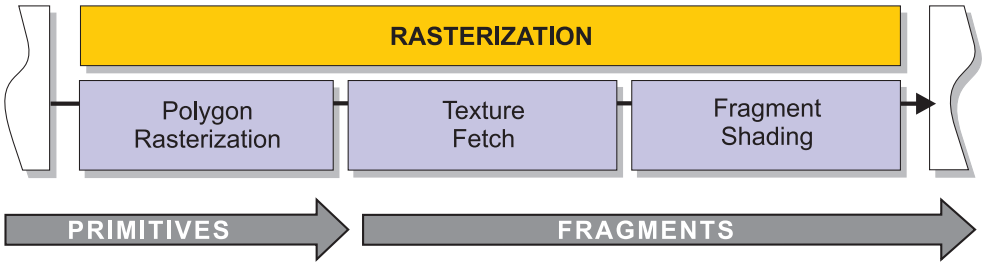


Figure 2.3: Rasterization as part of the standard graphics pipeline.

process of rasterization can be further divided into three different subtasks as displayed in Figure 2.3.

Polygon rasterization: In order to display filled polygons, *rasterization* determines the set of pixels that lie in the interior of the polygon. This also comprises the interpolation of visual attributes such as color, illumination terms and texture coordinates given at the vertices.

Texture Fetch: Textures are two-dimensional raster images, that are mapped onto the polygon according to texture coordinates specified at the vertices. For each fragment these texture coordinates must be interpolated and a texture lookup is performed at the resulting coordinate. This process generates a so-called *texel*, which refers to an interpolated color value sampled from the texture map. For maximum efficiency it is also important to take into account that most hardware implementations maintain a texture cache.

Fragment Shading: If texture mapping is enabled, the obtained texel is combined with the interpolated primary color of the fragment in a user-specified way. After the texture application step the color and opacity values of a fragment are final.

2.1.3 Fragment Operations

The fragments produced by rasterization are written into the *frame buffer*, which is a set of pixels arranged as a two-dimensional array. The frame buffer also contains the portion of memory that is finally displayed on the screen. When a fragment is written, it modifies the values already contained in the frame buffer according to a number of parameters and

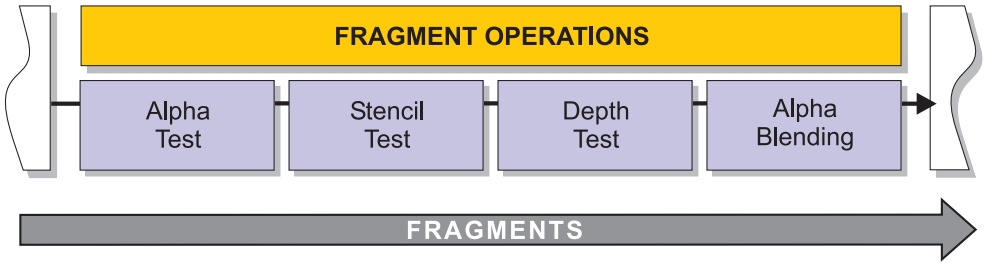


Figure 2.4: Fragment operations as part of the standard graphics pipeline.

conditions. The sequence of tests and modifications is termed *fragment operations* and is displayed in Figure 2.4.

Alpha Test: The alpha test allows the discarding of a fragment conditional on the outcome of a comparison between the fragments opacity α and a specified reference value.

Stencil Test: The stencil test allows the application of a pixel stencil to the visible frame buffer. This pixel stencil is contained in a so-called *stencil-buffer*, which is also a part of the frame buffer. The stencil test conditionally discards a fragment, if the stencil buffer is set for the corresponding pixel.

Depth Test: Since primitives are generated in arbitrary sequence, the depth test provides a mechanism for correct depth ordering of partially occluded objects. The depth value of a fragment is therefore stored in a so-called *depth buffer*. The depth test decides whether an incoming fragment is occluded by a fragment that has been previously written by comparing the incoming depth value to the value in the depth buffer. This allows the discarding of occluded fragments.

Alpha Blending: To allow for semi-transparent objects, *alpha blending* combines the color of the incoming fragment with the color of the corresponding pixel currently stored in the frame buffer.

After the scene description has completely passed through the graphics pipeline, the resulting raster image contained in the frame buffer can be displayed on the screen or written to a file. Further details on the rendering pipeline can be found in [36, 15]. Different hardware architectures ranging from expensive high-end workstations to consumer PC graphics

boards provide different implementations of this graphics pipeline. Thus, consistent access to multiple hardware architectures requires a level of abstraction, that is provided by an additional software layer called *application programming interface (API)*. We are using OpenGL [36] as API and Cg as shading language throughout these course notes, although every described algorithm might be as well implemented using DirectX and any high-level shading language.

2.2 Programmable GPUs

The first step towards a fully programmable GPU was the introduction of configurable rasterization and vertex processing in late 1999. Prominent examples are NVidia's *register combiners* or ATI's *fragment shader* OpenGL extensions. Unfortunately, it was not easy to access these vendor-specific features in a uniform way, back then.

The major innovation provided by today's graphics processors is the introduction of true programmability. This means that user-specified micro-programs can be uploaded to graphics memory and executed directly by the geometry stage (*vertex shaders*) and the rasterization unit (*fragment or pixel shaders*). Such programs must be written in an assembler-like language with the limited instruction set understood by the graphics processor (`MOV`, `MAD`, `LERP` and so on). However, high-level shading languages which provide an additional layer of abstraction were introduced quickly to access the capabilities of different graphics chips in an almost uniform way. Popular examples are Cg introduced by NVidia, which is derived from the *Stanford Shading Language*. The high-level shading language (HLSL) provided by Microsoft's DirectX 8.0 uses a similar syntax. The terms *vertex shader* and *vertex program*, and also *fragment shader* and *fragment program* have the same meaning, respectively.

2.2.1 Vertex Shaders

Vertex shaders are user-written programs which substitute major parts of the fixed-function computation of the geometry processing unit. They allow customization of the vertex transformation and the local illumination model. The vertex program is executed *once per vertex*: Every time a vertex enters the pipeline, the vertex processor receives an amount of data, executes the vertex program and writes the attributes for exactly one vertex. The vertex shader cannot create vertices from scratch or

remove incoming vertices from the pipeline.

The programmable vertex processor is outlined in Figure 2.5. For each vertex the vertex program stored in the instruction memory is executed once. In the loop outlined in the diagram, an instruction is first fetched and decoded. The operands for the instruction are then read from input registers which contain the original vertex attributes or from temporary registers. All instruction are vector operations, which are performed on `xyzw`-components for homogenous coordinates or `RGBA`-quadruplets for colors. Mapping allows the programmer to specify, duplicate and exchange the indices of the vector components (a process known as *swizzling*) and also to negate the respective values. If all the operands are correctly mapped the instruction is eventually executed and the result is written to temporary or output registers. At the end of the loop the vertex processor checks whether or not there are more instructions to be executed, and decides to reenter the loop or terminate the program by emitting the output registers to the next stage in the pipeline.

A simple example of a vertex shader is shown in the following code snippet. Note that in this example the vertex position is passed as a 2D coordinate in screen space and no transformations are applied. The vertex color is simply set to white.

```
// A simple vertex shader
struct myVertex {
    float4 position :  POSITION;
    float4 color    :  COLOR;
};

myVertex main (float2 pos :  POSITION)
{
    myVertex result;

    result.position = float4(pos,0,1);

    result.color = float4(1, 1, 1, 1);

    return result;
}
```

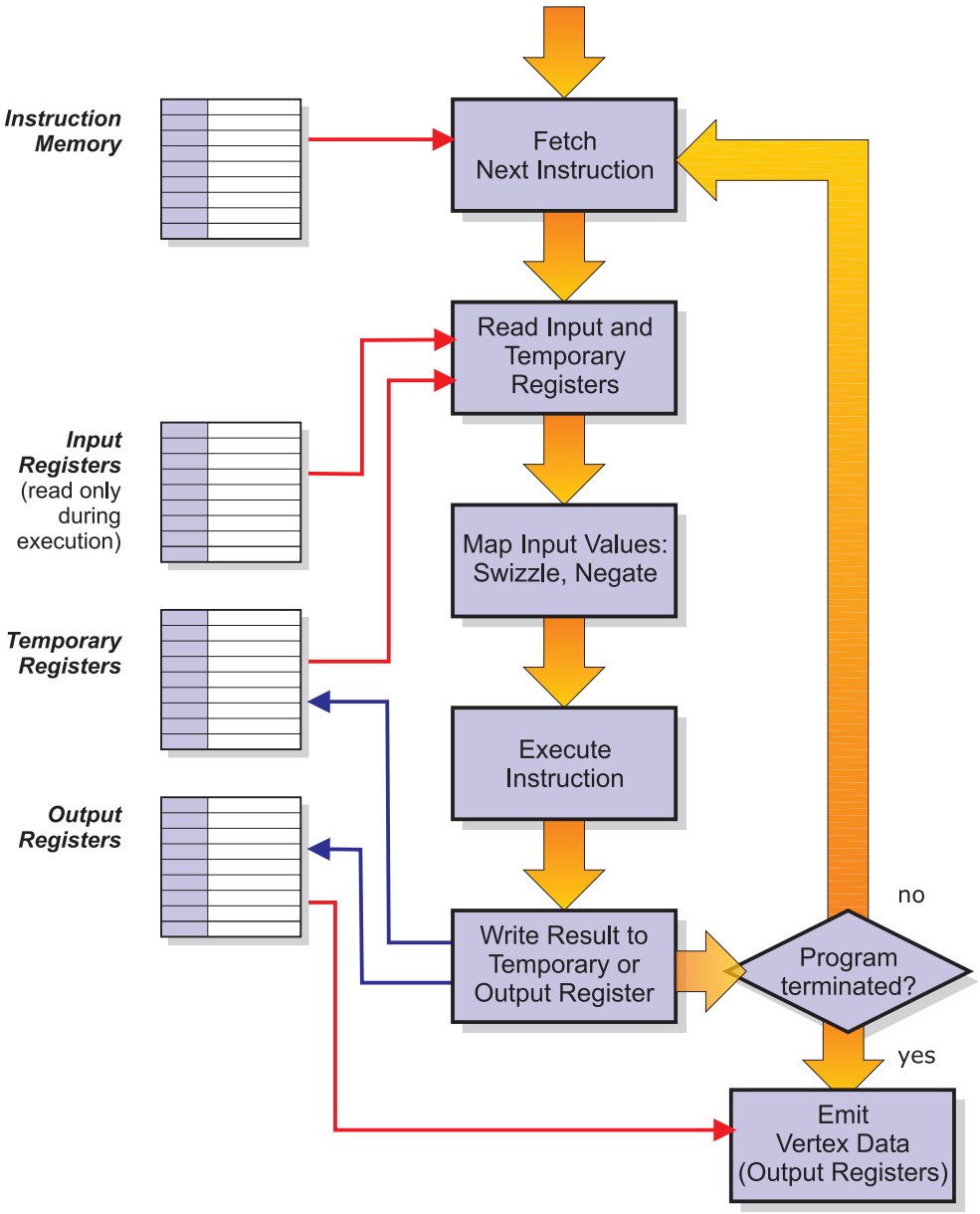


Figure 2.5: The programmable vertex processing unit executes a vertex program stored in local video memory. During the execution a limited set of input-, output- and temporary registers is accessed.

2.2.2 Fragment Shaders

Pixel shaders refer to programs, which are executed by the rasterization unit. They are used to compute the final color and depth values of a

fragment. The fragment program is executed *once per fragment*: Every time that polygon rasterization creates a fragment, the fragment processor receives a fixed set of attributes, such as colors, normal vectors or texture coordinates, executes the fragment program and writes the final color and z-value of the fragment to the output registers.

The diagram for the programmable fragment processor is shown in Figure 2.6. For each fragment the fragment program stored in instruction memory is executed once. The instruction loop of the fragment processor is similar to the vertex processor, with a separate path for texture fetch instructions. At first an instruction is first fetched and decoded. The operands for the instruction are read from the input registers which contain the fragments attributes or from temporary registers. The mapping step again computes the component swizzling and negation.

If the current instruction is a texture fetch instruction, the fragment processor computes the texture address with respect to texture coordinates and level of detail. Afterwards, the texture unit fetches all the texels which are required to interpolate a texture sample at the give coordinates. These texels are finally filtered to interpolate the final texture color value, which is then written to an output or temporary register.

If the current instruction is not a texture fetch instruction, it is executed with the specified operands and the result is written to the respective registers. At the end of the loop the fragment processor checks whether or not there are more instructions to be executed, and decides to reenter the loop or terminate the program by emitting the output registers to the fragment processing stage. As an example, the most simple fragment shader is displayed in the following code snippet:

```
// The most simple fragment shader
struct myOutput {
    float4 color :  COLOR;
};

myOutput main (float4 col :  COLOR)
{
    myOutput result;

    result.color = col;

    return result;

}
```

For more information on the programmable vertex and fragment processors, please refer to the Cg programming guide [14]

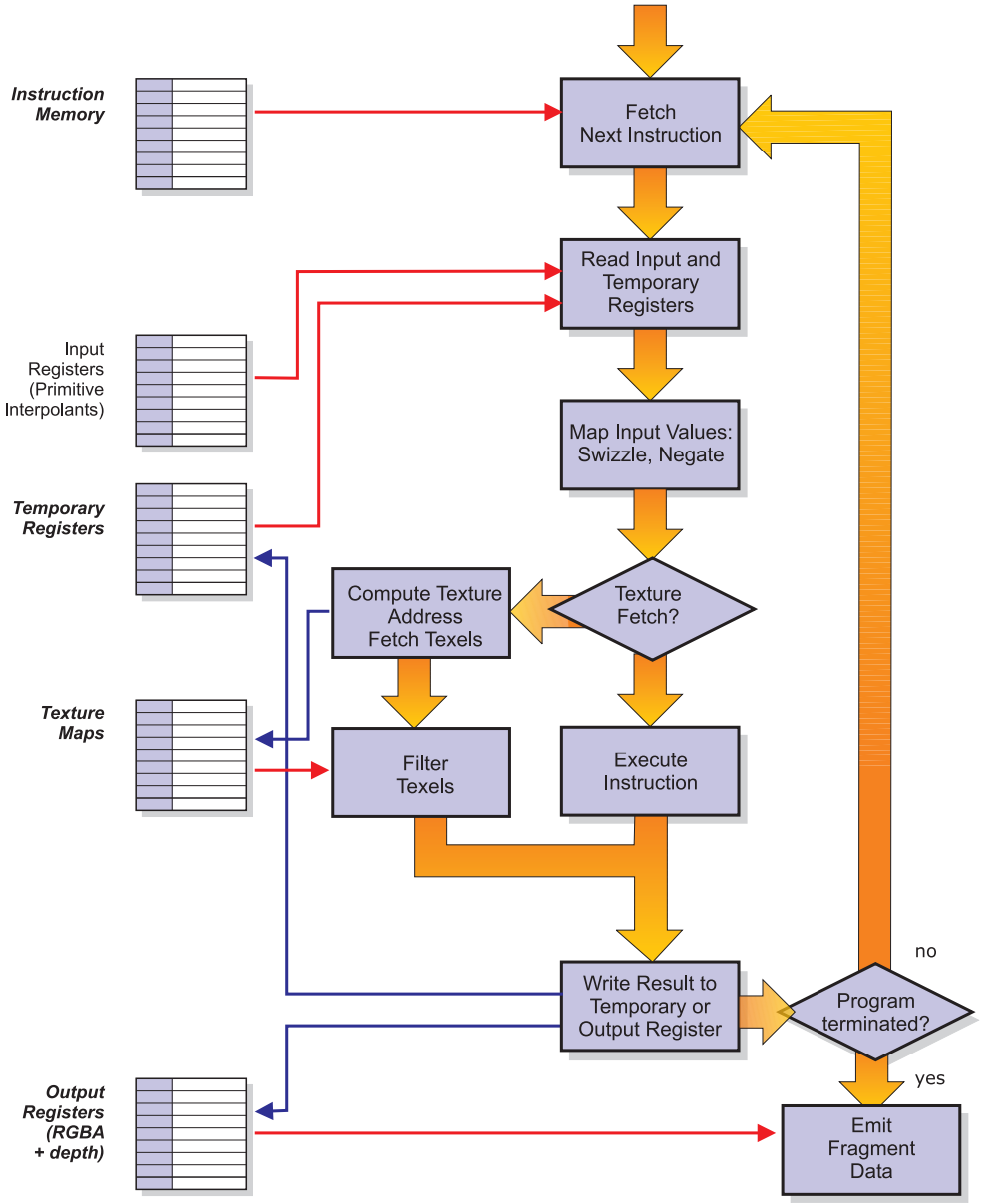


Figure 2.6: For each fragment, the programmable fragment processor executes a micro-program. In addition to reading the input and temporary registers, the fragment processor is able to generate filtered texture samples from the texture images stored in vide memory.

Course Notes T7
Real-Time Volume Graphics

GPU-Based Volume Rendering

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

Sampling a Volume Via Texture Mapping

As illustrated in the introduction to these course notes, the most fundamental operation in volume rendering is sampling the volumetric data (Section 1.1). Since this data is already discrete, the sampling task performed during rendering is actually a *resampling*, which means that the continuous signal must reconstructed approximately as necessary to sampling it again in screen space. The ray casting approach, that we have examined in the previous part is a classical *image-order approach*, because it divides the resulting image into pixels and then computes the contribution of the entire volume to each pixel.

Image-order approaches, however, are contrary to the way rasterization hardware generates images. Graphics hardware usually uses an *object-order* approach, which divides the object into primitives and then calculates which set of pixels are influenced by a primitive.

As we have seen in the introductory part, the two major operations related to volume rendering are *interpolation* and *compositing*, both of

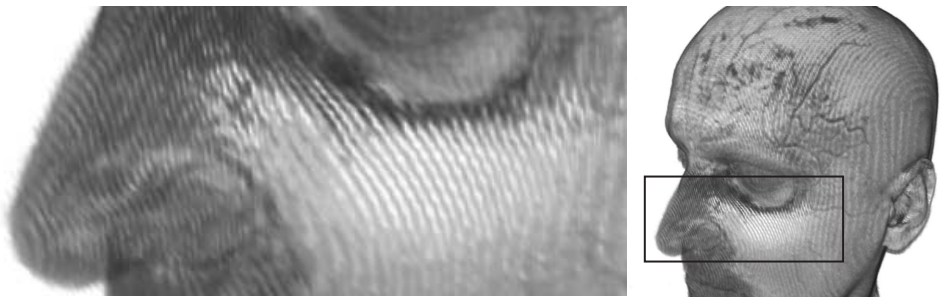


Figure 3.1: Rendering a volume by compositing a stack of 2D texture-mapped slices in back-to-front order. If the number of slices is too low, they become visible as artifacts.

which can efficiently be performed on modern graphics hardware. Texture mapping operations basically interpolate a texture image to obtain color samples at locations that do not coincide with the original grid. Texture mapping hardware is thus an ideal candidate for performing repetitive resampling tasks. Compositing individual samples can easily be done by exploiting fragment operations in hardware. The major question with regard to hardware-accelerated volume rendering is how to achieve the same – or a sufficiently similar – result as the ray-casting algorithm.

In order to perform volume rendering in an *object-order approach*, the resampling locations are generated by rendering a *proxy geometry* with interpolated texture coordinates (usually comprised of slices rendered as texture-mapped quads), and compositing all the parts (slices) of this proxy geometry from back to front via alpha blending. The volume data itself is stored in 2D- or 3D-texture images. If only a density volume is required, it can be stored in a single 3D texture with each texel corresponding to a single voxel. If the volume is too large to fit into texture memory, it must be split onto several 3D textures. Alternatively, volume data can be stored in a stack of 2D textures, each of which corresponds to an axis-aligned slice through the volume.

There are several texture-based approaches which mainly differ in the way the proxy geometry is computed.

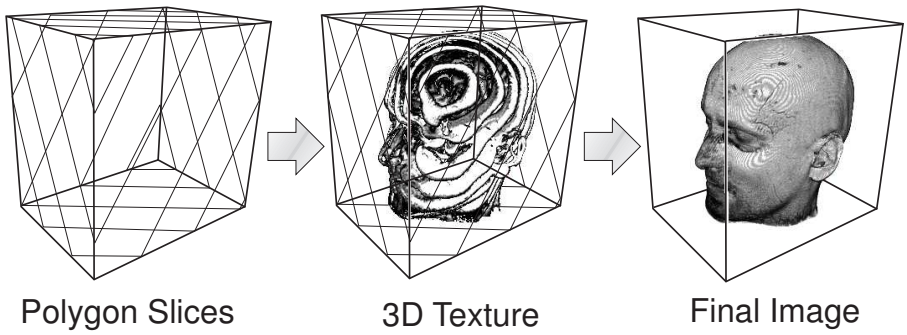


Figure 3.2: View-aligned slices used as proxy geometry with 3D texture mapping.

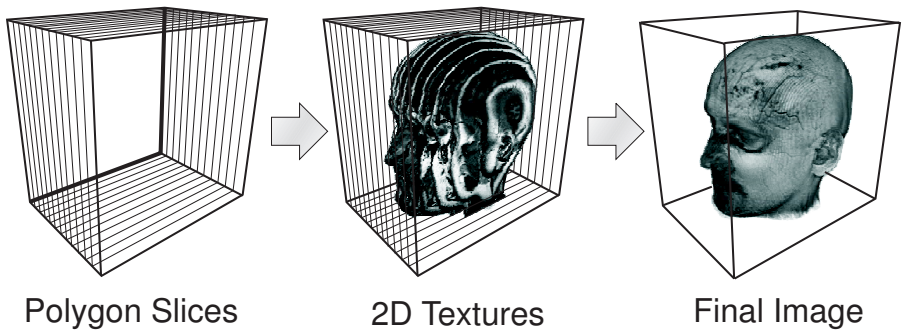


Figure 3.3: Object-aligned slices used as proxy geometry with 2D texture mapping.

3.1 Proxy Geometry

The first thing we notice if we want to perform volume rendering with rasterization hardware is, that hardware does not support any volumetric rendering primitives. Supported primitives comprise points, lines and planar polygons. In consequence, if we want to utilize rasterization hardware for volume rendering, we have to convert our volumetric representation into rendering primitives supported by hardware. A set of hardware primitives representing out volumetric object is called a proxy geometry. Ideally, with respect to the traditional modeling paradigm of separating *shape* from *appearance*, the shape of the proxy geometry should not have any influence on the final image, because only the appearance, i.e. the texture, is important.

The conceptually simplest example of proxy geometry is a set of *view-aligned slices* (quads that are parallel to the viewport, usually also clipped against the bounding box of the volume, see Figure 3.2), with 3D texture coordinates that are interpolated over the interior of these slices, and ultimately used to sample a single 3D texture map at the corresponding locations. 3D textures, however, often incur a performance penalty in comparison to 2D textures. This penalty is mostly due texture caches which are optimized for 2D textures.

One of the most important things to remember about the proxy geometry is that it is intimately related to the type of texture (2D or 3D) used. When the orientation of slices with respect to the original volume data

(i.e., the texture) can be arbitrary, 3D texture mapping is mandatory, since a single slice would have to fetch data from several different 2D textures. If, however, the proxy geometry is aligned with the original volume data, texture fetch operations for a single slice can be guaranteed to stay within the same 2D texture. In this case, the proxy geometry is comprised of a set of *object-aligned slices* (see Figure 3.3), for which 2D texture mapping capabilities suffice. The following sections describe different kinds of proxy geometry and the corresponding resampling approaches in more detail.

3.2 2D-Textured Object-Aligned Slices

If only 2D texture mapping capabilities are used, the volume data must be stored in several two-dimensional texture maps. A major implication of the use of 2D textures is that the hardware is only able to resample two-dimensional subsets of the original volumetric data.

The proxy geometry in this case is a stack of planar slices, all of which are required to be aligned with one of the major axes of the volume (either the x, y, or z axis), mapped with 2D textures, which in turn are resampled by the hardware-native bi-linear interpolation [7]. The reason for the requirement that slices must be aligned with a major axis is that each time a slice is rendered, only two dimensions are available for texture coordinates, and the third coordinate must therefore be constant. Now, instead of being used as an actual texture coordinate, the third coordinate selects the texture to use from the stack of slices, and the

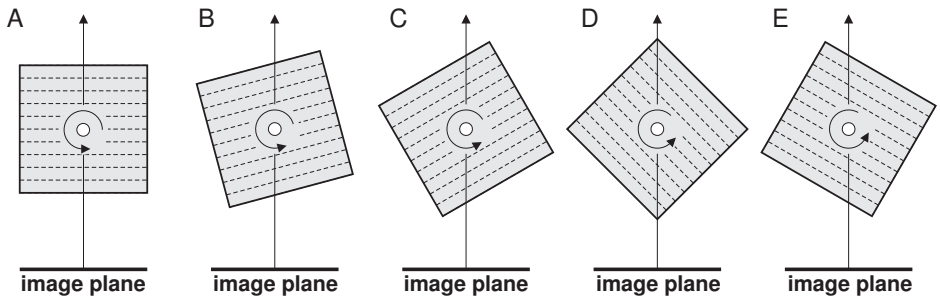


Figure 3.4: Switching the slice stack of object-aligned slices according to the viewing direction. Between image (C) and (D) the slice stack used for rendering has been switched.

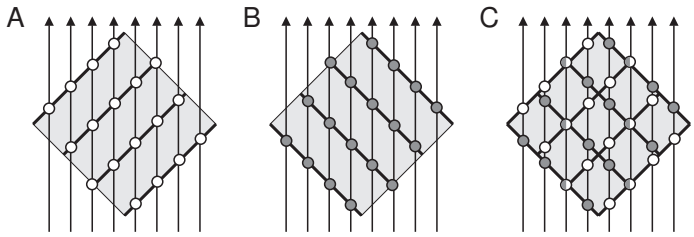


Figure 3.5: The location of sampling points changes abruptly (C), when switching from one slice stack (A), to the next (B).

other two coordinates become the actual 2D texture coordinates used for rendering the slice. Rendering proceeds from back to front, blending one slice on top of the other (see Figure 3.3).

Although a single stack of 2D slices can stores the entire volume, one slice stack does not suffice for rendering. When the viewpoint is rotated about the object, it would be possible that imaginary viewing rays pass through the object without intersecting any slices polygons. This cannot be prevented with only one slice stack. The solution for this problem is to actually store three slice stacks, one for each of the major axes. During rendering, the stack with slices most parallel to the viewing direction is chosen (see Figure 3.4).

Under-sampling typically occurs most visibly along the major axis of the slice stack currently in use, which can be seen in Figure 3.1. Additional artifacts become visible when the slice stack in use is switched from one stack to the next. The reason for this is that the actual locations of sampling points change abruptly when the stacks are switched, which is illustrated in Figure 3.5. To summarize, an obvious drawback of using object-aligned 2D slices is the requirement for three slice stacks, which consume three times the texture memory a single 3D texture would consume. When choosing a stack for rendering, an additional consideration must also be taken into account: 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, to achieve the desired result.

The following code fragment (continued on the next page) shows how both of these decisions, depending on the current viewing direction with respect to the volume, could be implemented:

```
GLfloat modelview_matrix[16];
GLfloat modelview_rotation_matrix[16];

// obtain the current viewing transformation
// from the OpenGL state
glGet( GL_MODELVIEW_MATRIX, modelview_matrix );
// extract the rotation from the matrix
GetRotation( modelview_matrix, modelview_rotation_matrix );

// rotate the initial viewing direction
GLfloat view_vector[3] = {0.0f, 0.0f, -1.0f};
MatVecMultiply( modelview_rotation_matrix, view_vector );
// find the largest absolute vector component
int max_component = FindAbsMaximum( view_vector );

// render slice stack according to viewing direction
switch ( max_component ) {
    case X:
        if ( view_vector[X] > 0.0f )
            DrawSliceStack_PositiveX();
        else
            DrawSliceStack_NegativeX();
        break;
    case Y:
        if ( view_vector[Y] > 0.0f )
            DrawSliceStack_PositiveY();
        else
            DrawSliceStack_NegativeY();
        break;
    case Z:
        if ( view_vector[Z] > 0.0f )
            DrawSliceStack_PositiveZ();
        else
            DrawSliceStack_NegativeZ();
        break;
}
```

Opacity Correction

In texture-based volume rendering, alpha blending is used to compute the compositing of samples along a ray. As we have seen in Section 1.2.4, this alpha blending operation is actually numerical approximation to the volume rendering integral. The distance Δt (see Equation 1.8) between successive resampling locations most of all depends on the distance between adjacent slices.

The sampling distance Δt is easiest to account for if it is constant for all "rays" (i.e., pixels). In this case, it can be incorporated into the numerical integration in a preprocess, which is usually done by simply adjusting the transfer function lookup table accordingly.

In the case of view-aligned slices the slice distance is equal to the sampling distance, which is also equal for all "rays" (i.e., pixels). Thus, it can be accounted for in a preprocess.

When 2D-textured slices are used, however, Δt not only depends on the slice distance, but also on the viewing direction. This is shown in Figure 3.6 for two adjacent slices. The sampling distance is only equal to the slice distance when the stack is viewed perpendicularly to its major axis (d_3). When the view is rotated, the sampling distance increases. For this reason, the lookup table for numerical integration (the transfer function table, see Part 5) has to be updated whenever the viewing direction changes. The correct opacity $\tilde{\alpha}$ for a sampling distance Δt amounts to

$$\tilde{\alpha} \;=\; 1-(1-\alpha)^{\frac{\Delta t}{\Delta s}} \tag{3.1}$$

with α referring to the opacity at the original sampling rate Δs which is accounted for in the transfer function.

This opacity correction is usually done in an approximate manner, by simply multiplying the stored opacities by the reciprocal of the cosine between the viewing vector and the stack direction vector:

```
// determine cosine via dot-product
// vectors must be normalized!
float cor_cos = DotProduct3( view_vector, slice_normal);
// determine correction factor
float opac_cor_factor =
    ( cor_cos != 0.0f ) ?  (1.0f / cor_cos) :  1.0f;
```

Note that although this correction factor is used for correcting opacity

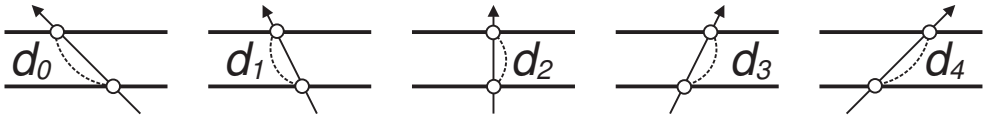


Figure 3.6: The distance between adjacent sampling points depends on the viewing angle.

values, it must also be applied to the respective RGB colors, if these are stored as opacity-weighted colors, which usually is the case [40].

Discussion

The biggest advantage of using object-aligned slices and 2D textures for volume rendering is that 2D textures and the corresponding bi-linear interpolation are a standard feature of all 3D graphics hardware architectures, and therefore this approach can practically be implemented anywhere. Also, the rendering performance is extremely high, since bi-linear interpolation requires only a lookup and weighting of four texels for each resampling operation.

The major disadvantages of this approach are the high memory requirements, due to the three slice stacks that are required, and the restriction to using two-dimensional, i.e., usually bi-linear, interpolation for texture reconstruction. The use of object-aligned slice stacks also leads to sampling and stack switching artifacts, as well as inconsistent sampling rates for different viewing directions. A brief summary is contained in table 3.1.

2D-Textured Object-Aligned Slices	
Pros	Cons
⊕ very high performance ⊕ high availability	⊖ high memory requirements ⊖ bi-linear interpolation only ⊖ sampling and switching artifacts ⊖ inconsistent sampling rate

Table 3.1: Summary of volume rendering with object-aligned slices and 2D textures.

3.3 2D Slice Interpolation

Figure 3.1 shows a fundamental problem of using 2D texture-mapped slices as proxy geometry for volume rendering. In contrast to view-aligned 3D texture-mapped slices (section 3.4), the number of slices cannot be changed easily, because each slice corresponds to exactly one slice from the slice stack. Furthermore, no interpolation between slices is performed at all, since only bi-linear interpolation is used within each slice. Because of these two properties of that algorithm, artifacts can become visible when there are too few slices, and thus the sampling frequency is too low with respect to frequencies contained in the volume and the transfer function.

In order to increase the sampling frequency without enlarging the volume itself (e.g., by generating additional interpolated slices before downloading them to the graphics hardware), inter-slice interpolation has to be performed on-the-fly by the graphics hardware itself. On the graphics boards hardware which support multi-texturing, this can be achieved by binding two textures simultaneously instead of just one when rendering the slice, and performing linear interpolation between these two textures [34].

In order to do this, we have to specify fractional slice positions, where the integers correspond to slices that actually exist in the source slice stack, and the fractional part determines the position between two adjacent slices. The number of rendered slices is now independent of the number of slices contained in the volume, and can be adjusted arbitrarily.

For each slice to be rendered, two textures are activated, which correspond to the two neighboring original slices from the source slice stack. The fractional position between these slices is used as weight for the inter-slice interpolation. This method actually performs tri-linear interpolation within the volume. Standard bi-linear interpolation is employed for each of the two neighboring slices, and the interpolation between the two obtained results altogether achieves tri-linear interpolation.

On-the-fly interpolation of intermediate slices can be implemented as a simple fragment shader in Cg, as shown in the following code snippet. The two source slices that enclose the position of the slice to be rendered are configured as `texture0` and `texture1`, respectively. The two calls to the function `tex2D` interpolate both texture images bi-linearly, using the `x-` and `y-` components of the texture coordinate. The third linear interpolation step which is necessary for trilinear interpolation is performed subsequently using the `lerp` function and the `z-` component of the texture coordinate. The final fragment contains the linearly interpolated

result corresponding to the specified fractional slice position.

```
// Cg fragment shader for 2D slice interpolation
  half4 main (float3 texcoords :  TEXCOORD0,
              uniform sampler2D texture0,
              uniform sampler2D texture1) : COLOR0
  {
     float4 t0 = tex2D(texture0,texcoords.xy);
     float4 t1 = tex2D(texture1,texcoords.xy);

     return (half4) lerp(t0, t1, texcoords.z);
  }
```

Discussion

The biggest advantage of using object-aligned slices together with on-the-fly interpolation between two 2D textures for volume rendering is that this method combines the advantages of using only 2D textures with the capability of arbitrarily controlling the sampling rate, i.e., the number of slices. Although not entirely comparable to tri-linear interpolation in a 3D texture, the combination of bi-linear interpolation and a second linear interpolation step ultimately allows tri-linear interpolation in the volume. The necessary features of consumer hardware, i.e., multi-texturing with at least two simultaneous textures, and the ability to interpolate between them, are widely available on consumer graphics hardware.

Disadvantages inherent to the use of object-aligned slice stacks still apply, though. For example, the undesired visible effects when switching slice stacks, and the memory consumption of the three slice stacks. A

2D Slice Interpolation	
Pros	Cons
⊕ high performance ⊕ tri-linear interpolation ⊕ available on consumer hardware	⊖ high memory requirements ⊖ switching effects ⊖ inconsistent sampling rate for perspective projection

Table 3.2: Summary of 2D slice interpolation volume rendering.

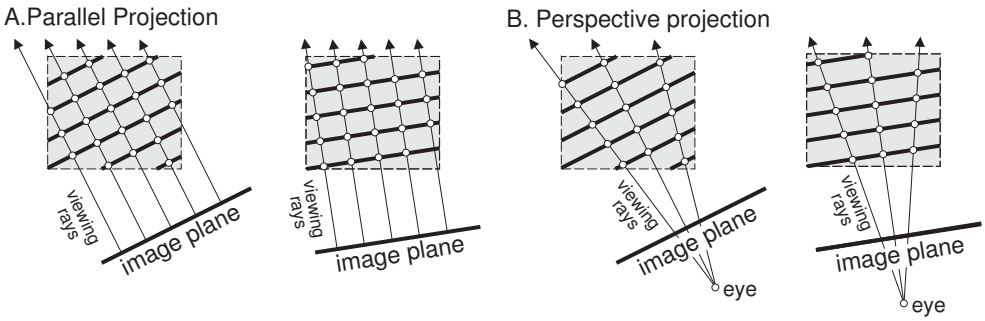


Figure 3.7: Sampling locations on view-aligned slices for parallel (A), and perspective projection (B), respectively.

brief summary is contained in table 3.2.

3.4 3D-Textured View-Aligned Slices

In many respects, 3D-textured view-aligned slices are the simplest type of proxy geometry (see Figure 3.2). In this case, the volume is stored in a single 3D texture map, and 3D texture coordinates are interpolated over the interior of the proxy geometry polygons. These texture coordinates are then used directly for indexing the 3D texture map at the corresponding location, and thus resampling the volume.

The big advantage of 3D texture mapping is that it allows slices to be oriented arbitrarily within the 3D texture domain, i.e., the volume itself. Thus, it is natural to use slices aligned with the viewport, since such slices closely mimic the sampling used by the ray-casting algorithm. They offer constant distance between samples for orthogonal projection and all viewing directions, as outlined in Figure 3.7(A). Since the graphics hardware is already performing completely general tri-linear interpolation within the volume for each resampling location, proxy slices are not bound to original slices at all. The number of slices can easily be adjusted on-the-fly and without any restrictions. In case of perspective projection, the distance between successive samples is different for adjacent pixels, however, which is depicted in Figure 3.7(B). Artifacts caused by a not entirely accurate opacity compensation, however, are only noticeable for a large field-of-view angle of the viewing frustum. If this is the case, spherical shells must be employed instead of planar slices as

described in Section 3.5.

Discussion

The biggest advantage of using view-aligned slices and 3D textures for volume rendering is that hardware-accelerated tri-linear interpolation is employed for resampling the volume at arbitrary locations. Apart from better image quality compared to bi-linear interpolation, this allows the rendering of slices with arbitrary orientation with respect to the volume, making it possible to maintain a constant sampling rate for all pixels and viewing directions. Additionally, a single 3D texture suffices for storing the entire volume, if there is enough texture memory available.

The major disadvantage of this approach is that tri-linear interpolation is significantly slower than bi-linear interpolation, due to the requirement for using eight texels for every single output sample, and texture fetch patterns that decrease the efficiency of texture caches. A brief summary is contained in table 3.3.

3.5 3D-Textured Spherical Shells

All types of proxy geometry that use planar slices (irrespective of whether they are object-aligned, or view-aligned), share the basic problem that the distance between successive samples used to determine the color of a single pixel is different from one pixel to the next in the case of perspective projection. This fact is illustrated in Figure 3.7(B).

When incorporating the sampling distance in the numerical approximation of the volume rendering integral, this pixel-to-pixel difference cannot easily be accounted for. A possible solution to this problem is the use of spherical shells instead of planar slices [27]. In order to attain a constant sampling distance for all pixels using perspective projection, the proxy geometry has to be spherical, i.e., be comprised of concentric

3D-Textured View-Aligned Slices	
Pros	Cons
⊕ high performance ⊕ tri-linear interpolation	⊖ availability still limited ⊖ inconsistent sampling rate for perspective projection

Table 3.3: Summary of 3D-texture based volume rendering.

spherical shells. In practice, these shells are generated by clipping tessellated spheres against both the viewing frustum and the bounding box of the volume data.

The major drawback of using spherical shells as proxy geometry is that they are more complicated to setup than planar slice stacks, and they also require more geometry to be rendered, i.e., parts of tessellated spheres.

This kind of proxy geometry is only useful when perspective projection is used, and can only be used in conjunction with 3D texture mapping. Furthermore, the artifacts of pixel-to-pixel differences in sampling distance are often hardly noticeable, and planar slice stacks usually suffice even when perspective projection is used.

3.6 Slices vs. Slabs

An inherent problem of using slices as proxy geometry is that the number of slices directly determines the (re)sampling frequency, and thus the quality of the rendered result. Especially when high frequencies ("sharp edges") are contained in the employed transfer functions, the required number of slices can become very high. Even though the majority of texture based implementations allow the number of slices to be increased on demand via interpolation done entirely on the graphics hardware, the fill rate demands increase dramatically.

A very elegant solution to this problem arrives with the use of *slabs* instead of slices, together with pre-integrated classification [12], which is described in more detail in Part 7 of these course notes.. A slab is not a new geometrical primitive, but simply the space between two adjacent slices. During rendering, the solution of the integral of ray segments which intersect this space is properly accounted for by looking up a pre-computed solution. This solution is a function of the scalar values of both the back slice to the front slice. It is obtained from a pre-integrated lookup table stored as a 2D texture. Geometrically, a slab can be rendered as a slice with its immediately neighboring slice (either in the back, or in front) projected onto it. For further details on pre-integrated volume rendering, we refer you to Part 7.

Components of a Hardware Volume Renderer

This chapter presents an overview of the major components of a texture-based hardware volume renderer from an implementation-centric point of view. The goal of this chapter is to convey a feeling of where the individual components of such a renderer fit in, and in which order they are executed. Details are covered in subsequent chapters. The component structure presented here is modeled after separate portions of code that can be found in an actual implementation of a volume renderer for consumer graphics cards. They are listed in the order in which they are executed by the application code, which is not necessarily the same as they are "executed" by the graphics hardware itself!

4.1 Volume Data Representation

Volume data has to be stored in memory in a suitable format, usually already prepared for download to the graphics hardware as textures. Depending on the type of proxy geometry used, the volume can either be stored in a single block, when view-aligned slices together with a single 3D texture are used, or split up into three stacks of 2D slices, when object-aligned slices together with multiple 2D textures are used. Usually, it is convenient to store the volume only in a single 3D array, which can be downloaded as a single 3D texture, and extract data for 2D textures on demand.

Depending on the complexity of the rendering mode, classification, and illumination, there may even be several volumes containing all the information needed. Likewise, the actual storage format of voxels depends on the rendering mode and the type of volume, e.g., whether the volume stores densities, gradients, gradient magnitudes, and so on. Conceptually different volumes may also be combined into the same actual volume, if possible. For example, combining gradient and density data in RGBA voxels.

Although it is often the case that the data representation issue is part of a preprocessing step, this is not necessarily so, since new data may have to be generated on-the-fly when the rendering mode or specific parameters are changed.

This component is usually executed only once at startup, or only executed when the rendering mode changes.

4.2 Volume Textures

In order for the graphics hardware to be able to access all the required volume information, the volume data must be downloaded and stored in textures. At this stage, a translation from data format (external format) to texture format (internal format) might take place, if the two are not identical.

This component is usually executed only once at startup, or only executed when the rendering mode changes.

How and what textures containing the actual volume data have to be downloaded to the graphics hardware depends on a number of factors, most of all the rendering mode and type of classification, and whether 2D or 3D textures are used.

The following example code fragment downloads a single 3D texture. The internal format is set to GL_INTENSITY8, which means that a single 8 bit value is stored for each texel. For uploading 2D textures instead of 3D textures , similar commands have to be used to create all the slices of all three slice stacks.

```
// bind 3D texture target
glBindTexture( GL_TEXTURE_3D, volume_texture_name_3d );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );

// download 3D volume texture for pre-classification
glTexImage3D( GL_TEXTURE_3D, 0, GL_INTENSITY8,
   size_x, size_y, size_z,
   GL_COLOR_INDEX, GL_UNSIGNED_BYTE, volume_data_3d );
```

4.3 Transfer Function Tables

Transfer functions are usually represented by color lookup tables. They can be one-dimensional or multi-dimensional, and are usually stored as simple arrays.

Transfer functions may be downloaded to the hardware in basically one of two formats: In the case of pre-classification, transfer functions are downloaded as texture palettes for on-the-fly expansion of palette indexes to RGBA colors. If post-classification is used, transfer functions are downloaded as 1D, 2D, or even 3D textures (the latter two for multi-dimensional transfer functions). If pre-integration is used, the transfer function is only used to calculate a pre-integration table, but not downloaded to the hardware itself. Then, this pre-integration table is downloaded instead. This component might even not be used at all, which is the case when the transfer function has already been applied to the volume textures themselves, and they are already in RGBA format.

This component is usually only executed when the transfer function or rendering mode changes.

The following code fragment demonstrated the use of the OpenGL extensions `GL_ext_paletted_texture` und `GL_ext_shared_texture_palette` for pre-classification. It uploads a single texture palette that can be used in conjunction with an indexed volume texture for pre-classification. The respective texture must have an internal format of `GL_COLOR_INDEX8_EXT`. The same code can be used for rendering with either 2D, or 3D slices, respectively:

```
// download color table for pre-classification
glColorTableEXT(
   GL_SHARED_TEXTURE_PALETTE_EXT,
   GL_RGBA8,
   256 * 4,
   GL_RGBA,
   GL_UNSIGNED_BYTE,
   opacity_corrected_palette );
```

If post-classification is used instead, the same transfer function table can be used, but it must be uploaded as a 1D texture instead:

```
// bind 1D texture target
glBindTexture( GL_TEXTURE_1D, palette_texture_name );
glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );

// download 1D transfer function texture for post-classification
glTexImage1D( GL_TEXTURE_1D, 0, GL_RGBA8, 256 * 4, 0,
   GL_RGBA, GL_UNSIGNED_BYTE, opacity_corrected_palette );
```

If pre-integration is used, a pre-integration texture is uploaded as a 2D texture instead of the transfer function table (see Part 7).

4.4 Fragment Shader Configuration

Before the volume can be rendered using a specific rendering mode, the fragment shader has to be configured accordingly. How textures are stored and what they contain is crucial for the fragment shader. Likewise, the format of the shaded fragment has to correspond to what is expected by the alpha blending stage (section 4.5).

This component is usually executed once per frame, i.e., the entire volume can be rendered with the same fragment shader configuration.

The code that determines the operation of the fragment shader is highly dependent on the underlying hardware architecture (section 2.2.2). The code for configuring the fragment shader also depends on the shading language that is used. The following code snippet shows how to create a fragment shader using Cg with the multivendor `ARB_fragment_program` OpenGL profile:

```
// create the fragment shader
CGprogram myShader = cgCreateProgram(context,
                                     CG_SOURCE,
                                     programString,
                                     CG_PROFILE_ARBFP1,
                                     "main",
                                     args);

cgGLLoadProgram(myShader);
```

If the fragment shader is loaded to the graphics board, it can be bound and enabled as many times as necessary:

```
// bind the fragment shader
cgGLBindProgram(myShader);
cgGLEnableProfile(CG_PROFILE_ARBFP1);

// use the shader

cgGLDisableProfile(CG_PROFILE_ARBFP1);
```

4.5 Blending Mode Configuration

The blending mode determines how a fragment is combined with the corresponding pixel in the frame buffer. In addition to the configuration of *alpha blending*, we also configure *alpha testing* in this component, if it is required for discarding fragments to display non-polygonal iso-surfaces. The configuration of the blending stage and the alpha test usually stays the same for an entire frame.

This component is usually executed once per frame, i.e., the entire volume can be rendered with the same blending mode configuration.

For **direct volume rendering**, the blending mode is more or less standard alpha blending. Since color values are usually pre-multiplied by the corresponding opacity (also known as *opacity-weighted* [40], or *associated* [6] colors), the factor for multiplication with the source color is one:

```
// enable blending
glEnable( GL_BLEND );
// set blend function
glBlendFunc( GL_ONE, GL_ONE_MINUS_SRC_ALPHA );
```

For **non-polygonal iso-surfaces**, alpha testing has to be configured for selection of fragments corresponding to the desired iso-values. The comparison operator for comparing a fragment's density value with the reference value is usually `GL_GREATER`, or `GL_LESS`, since using `GL_EQUAL` is not well suited to producing a smooth surface appearance (not many interpolated density values are exactly equal to a given reference value). Alpha blending must be disabled for this rendering mode. More de-

tails about rendering non-polygonal iso-surfaces, especially with regard to illumination, can be found in Part 4

```
// disable blending
glDisable(GL_BLEND );
// enable alpha testing
glEnable(GL_ALPHA_TEST );
// configure alpha test function
glAlphaFunc(GL_GREATER, isovalue );
```

For **maximum intensity projection**, an alpha blending equation of `GL_MAX_EXT` must be supported, which is either a part of the imaging subset, or the separate `GL_EXT_blend_minmax` extension. On consumer graphics hardware, querying for the latter extension is the best way to determine availability of the maximum operator.

```
// enable blending
glEnable(GL_BLEND );
// set blend function to identity (not really necessary)
glBlendFunc(GL_ONE,GL_ONE );
// set blend equation to max
glBlendEquationEXT(GL_MAX_EXT );
```

4.6 Texture Unit Configuration

The use of texture units corresponds to the inputs required by the fragment shader. Before rendering any geometry, the corresponding textures have to be bound. When 3D textures are used, the entire configuration of texture units usually stays the same for an entire frame. In the case of 2D textures, the textures that are bound change for each slice.

This component is usually executed once per frame, or once per slice, depending on whether 3D, or 2D textures are used.

The following code fragment shows an example for configuring two texture units for interpolation of two neighboring 2D slices from the z slice stack (section 3.3):

```
// configure texture unit 1
glActiveTextureARB(GL_TEXTURE1_ARB );
glBindTexture( GL_TEXTURE_2D, tex_names_stack_z[sliceid1]);
glEnable( GL_TEXTURE_2D );
// configure texture unit 0
glActiveTextureARB( GL_TEXTURE0_ARB );
glBindTexture( GL_TEXTURE_2D, tex_names_stack_z[sliceid0]);
glEnable( GL_TEXTURE_2D );
```

4.7 Proxy Geometry Rendering

The last component of the execution sequence outlined in this chapter, is getting the graphics hardware to render geometry. This is what actually causes the generation of fragments to be shaded and blended into the frame buffer, after resampling the volume data accordingly.

This component is executed once per slice, irrespective of whether 3D or 2D textures are used.

Explicit texture coordinates are usually only specified when rendering 2D texture-mapped, object-aligned slices. In the case of view-aligned slices, texture coordinates can easily be generated automatically, by exploiting OpenGL's texture coordinate generation mechanism, which has to be configured before the actual geometry is rendered:

```
// configure texture coordinate generation
// for view-aligned slices
float plane_x[]  = { 1.0f, 0.0f, 0.0f, 0.0f };
float plane_y[]  = { 0.0f, 1.0f, 0.0f, 0.0f };
float plane_z[]  = { 0.0f, 0.0f, 1.0f, 0.0f };

glTexGenfv( GL_S, GL_OBJECT_PLANE, plane_x );
glTexGenfv( GL_T, GL_OBJECT_PLANE, plane_y );
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_GEN_R );
```

The following code fragment shows an example of rendering a single slice as an OpenGL quad. Texture coordinates are specified explicitly, since this code fragment is intended for rendering a slice from a stack of

object-aligned slices with z as its major axis:

```
// render a single slice as quad (four vertices)
glBegin( GL_QUADS );
   glTexCoord2f( 0.0f, 0.0f );
   glVertex3f( 0.0f, 0.0f, axis_pos_z );
   glTexCoord2f( 0.0f, 1.0f );
   glVertex3f( 0.0f, 1.0f, axis_pos_z );
   glTexCoord2f( 1.0f, 1.0f );
   glVertex3f( 1.0f, 1.0f, axis_pos_z );
   glTexCoord2f( 1.0f, 0.0f );
   glVertex3f( 1.0f, 0.0f, axis_pos_z );
glEnd();
```

Vertex coordinates are specified in object-space, and transformed to view-space using the modelview matrix. In the case of object-aligned slices, all the `glTexCoord2f()` commands can simply be left out. If multi-texturing is used, a simple vertex program can be exploited for generating the texture coordinates for the additional units, instead of downloading the same texture coordinates to multiple units. On the Radeon 8500 it is also possible to use the texture coordinates from unit zero for texture fetch operations at any of the other units, which solves the problem of duplicate texture coordinates in a very simple way, without requiring a vertex shader or wasting bandwidth.

Course Notes T7

Real-Time Volume Graphics

Transfer Functions

Klaus Engel

Siemens Corporate Research, Princeton, USA

Markus Hadwiger

VRVis Research Center, Vienna, Austria

Joe M. Kniss

SCI Institute, University of Utah, USA

Christof Rezk Salama

University of Siegen, Germany

Introduction

The volume rendering of abstract data values requires the assignment of optical properties to the data values to create a meaningful image. It is the role of the transfer function to assign these optical properties, the result of which has profound effect on the quality of the rendered image. While the transformation from data values to optical properties is typically a simple table lookup, the creation of a good transfer function to create the table can be a difficult task.

In order to make the discussion of transfer function design more understandable, we dissect it into two distinct parts; Classification and Optical Properties. The first chapter focuses on the conceptual role of the transfer function as a feature classifier. The following chapter covers the second half of the transfer function story; how optical properties are assigned to the classified features for image synthesis.

Classification and Feature Extraction

Classification in the context of volume graphics is defined as "the process of identifying features of interest based on abstract data values". In typical volume graphics applications, especially volume visualization, this is effectively a pattern recognition problem in which patterns found in raw data are assigned to specific categories. The field of pattern recognition is mature and widely varying, and an overview of general theory and popular methods can be found in the classic text by Duda, Hart, and Stork [10]. Traditionally, the transfer function is not thought of as a feature classifier. Rather, it is simply viewed as a function that takes the domain of the input data and transforms it to the range of red, green, blue, and alpha. However, transfer functions are used to assign specific patterns to ranges of values in the source data that correspond to features of interest, for instance bone and soft tissue in CT data, whose unique visual quality in the final image can be used to identify these regions.

Why do we need a transfer function anyway? Why not store the optical properties in the volume directly? There are at least two good answers to these questions. First, it is inefficient to update the entire volume and reload it each time the transfer function changes. It is much faster to load the smaller lookup table and let the hardware handle the transformation from data value to optical properties. Second, evaluating the transfer function (assigning optical properties) at each sample prior to interpolation can cause visual artifacts. This approach is referred to as pre-classification and can cause significant artifacts in the final rendering, especially when there is a sharp peak in the transfer function. An example of pre-classification can be seen on the left side of Figure 6.1 while post-classification (interpolating the data first, then assigning optical properties) using the exact same data and transfer function is seen on the right.

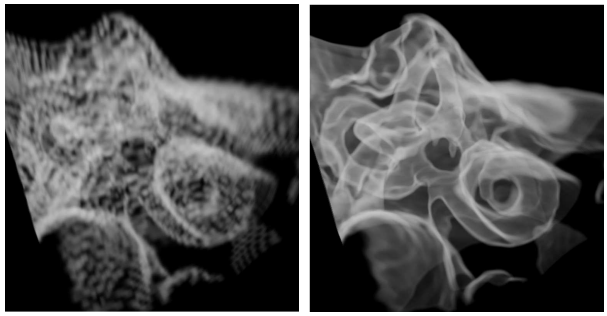


Figure 6.1: Pre-classification (left) verses post-classification (right)

6.1 The Transfer Function as a Feature Classifier

Often, the domain of the transfer function is 1D, representing a scalar data value such as radio-opacity in CT data. A single scalar data value, however, need not be the only quantity used to identify the difference between materials in a transfer function. For instance, Levoy's volume rendering model [28] includes a 2D transfer function, where the domain is scalar data value cross gradient magnitude. In this case, data value and gradient magnitude are the axes of a multi-dimensional transfer function. Adding the gradient magnitude of a scalar dataset to the transfer function can improve the ability of the transfer function to distinguish materials from boundaries. Figures 6.2(c) and 6.2(d) show how this kind of 2D transfer function can help isolate the leaf material from the bark material of the Bonsai Tree CT dataset. It is important to consider any and all data values or derived quantities that may aid in identifying key features of interest. Other derived quantities, such as curvature or shape metrics [23], can help define important landmarks for generating technical illustration-like renderings as seen in Figure 6.3. See [24] for examples of more general multi-dimensional transfer functions applied to multivariate data. Examples of visualizations generated using a transfer function based on multiple independent (not derived) scalar data values can be seen in Figure 6.4.

6.2 Guidance

While the transfer function itself is simply a function taking the input data domain to the rgba range, the proper mapping to spectral space

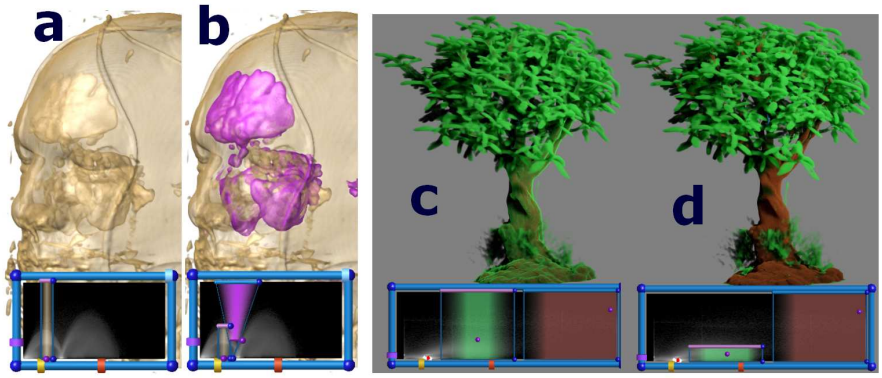


Figure 6.2: 1D (a and c) verses 2D (b and d) transfer functions.

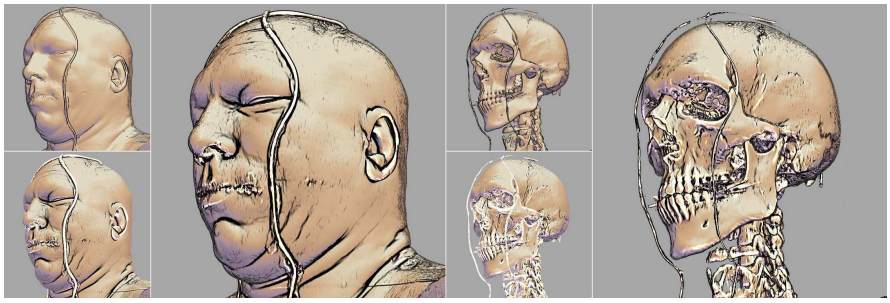


Figure 6.3: Using curvature and shape measures as axes of the transfer function. The upper small images show silhouette edges, the lower small images show ridge valley emphasis, and the large images show these combined into the final illustration. Images courtesy of Gordon Kindlman, use by permission.

(optical properties) is not intuitive and varies based on the range of values of the source data as well as the desired goal of the final volume rendering. Typically, the user is presented with a transfer function editor that visually demonstrates changes to the transfer function. A naive transfer function editor may simply give the user access to all of the optical properties directly as a series of control points that define piecewise linear (or higher order) ramps. This can be seen in Figure 6.5. This approach can make specifying a transfer function a tedious trial and error process. Naturally, adding dimensions to the transfer function can further complicate a user interface.

The effectiveness of a transfer function editor can be enhanced with features that guide the user with data specific information. He *et al.* [20]

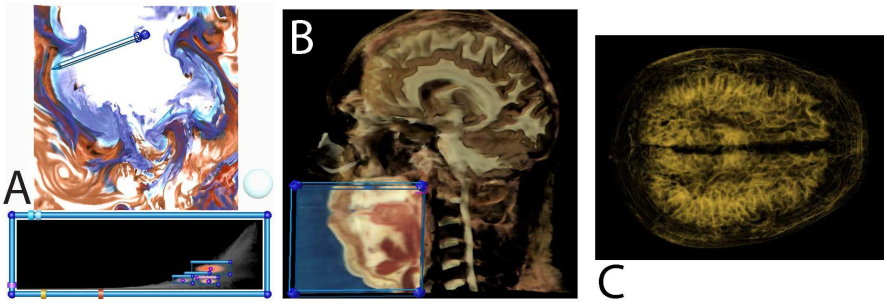


Figure 6.4: General multi-dimensional transfer functions, using multiple primary data values. A is a numerical weather simulation using temperature, humidity, and pressure. B is a color cryosection dataset using red, green, and blue visible light. C is a MRI scan using proton density, T1, and T2 pulse sequences.

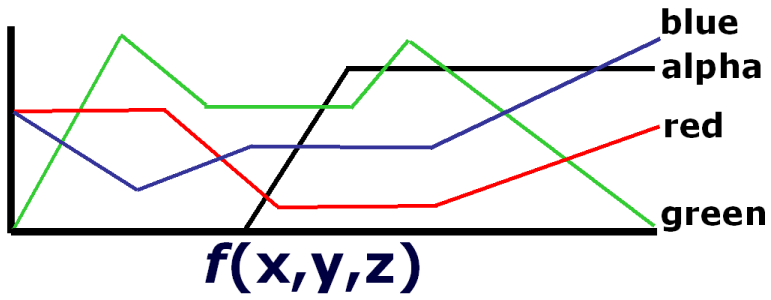


Figure 6.5: An arbitrary transfer function showing how red, green, blue, and alpha vary as a function of data value **f(x,y,z)** .

generated transfer functions with genetic algorithms driven either by user selection of thumbnail renderings, or some objective image fitness function. The purpose of this interface is to suggest an appropriate transfer function to the user based on how well the user feels the rendered images capture the important features.

The Design Gallery [29] creates an intuitive interface to the entire space of all possible transfer functions based on automated analysis and layout of rendered images. This approach parameterizes the space of all possible transfer functions. The space is stochastically sampled and a volume rendering is created. The images are then grouped based on similarity. While this can be a time consuming process, it is fully automated. Figure 6.6 shows an example of this user interface.

A more data-centric approach is the Contour Spectrum [2], which

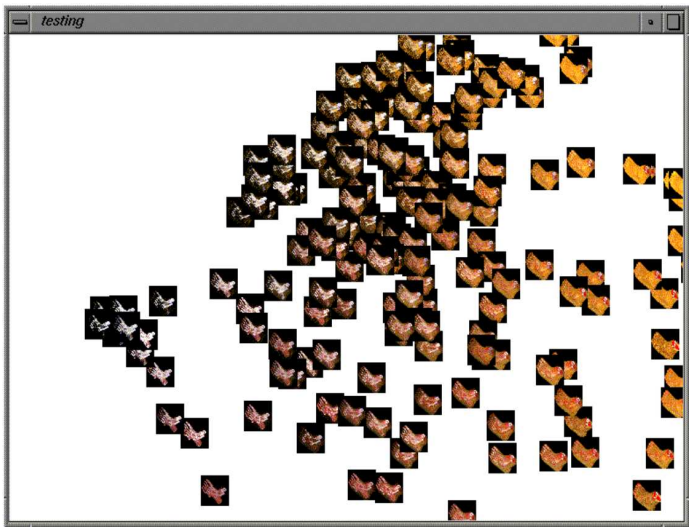


Figure 6.6: The Design Gallery transfer function interface.

visually summarizes the space of isosurfaces in terms of data metrics like surface area and mean gradient magnitude. This guides the choice of isovalue for isosurfacing, and also provides information useful for transfer function generation. Another recent paper [1] presents a novel transfer function interface in which small thumbnail renderings are arranged according to their relationship with the spaces of data values, color, and opacity. This kind of editor can be seen in Figure 6.2.

One of the most simple and effective features that a transfer function interface can include is a histogram. A histogram shows a user the behavior of data values in the transfer function domain. In time, a user can learn to read the histogram and quickly identify features. Figure 6.8(b) shows a 2D joint histogram of the Chapel Hill CT dataset. The arches identify material boundaries and the dark blobs located at the bottom identify the materials themselves.

Volume probing is another way to help the user identify features. This approach gives the user a mechanism for pointing at a feature in the spatial domain. The values at this point are then presented graphically in the transfer function interface, indicating to the user the ranges of data values which identify the feature. This approach can be tied to a mechanism that automatically sets the transfer function based on the data values at the being feature pointed at. This technique is called dual-domain interaction [24]. The action of this process can be seen in Figure 6.9.

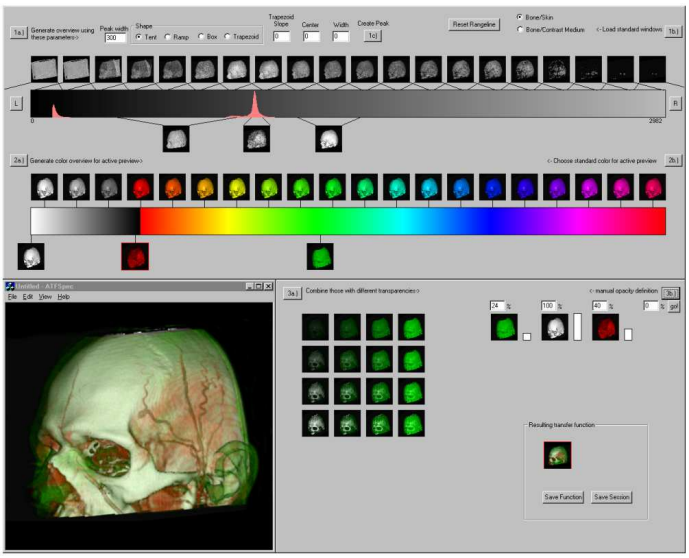
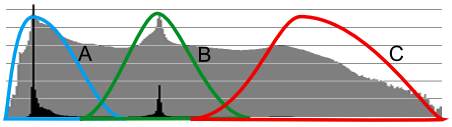


Figure 6.7: A thumbnail transfer function interface.

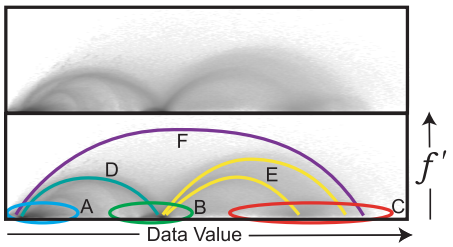
Often it is helpful to identify discrete regions in the transfer function domain that correspond to individual features. Figure 6.10 shows an integrated 2D transfer function interface. This type of interface constructs a transfer function using direct manipulation widgets. Classified regions are modified by manipulating control points. These control points change high level parameters such as position, width, and optical properties. The widgets define a specific type of classification function such as a Gaussian ellipsoid, inverted triangle, or linear ramp. This approach is advantageous because it allows the user to focus more on feature identification and less on the shape of the classification function. We have also found it useful to allow the user the ability to paint directly into the transfer function domain.

6.3 Summary

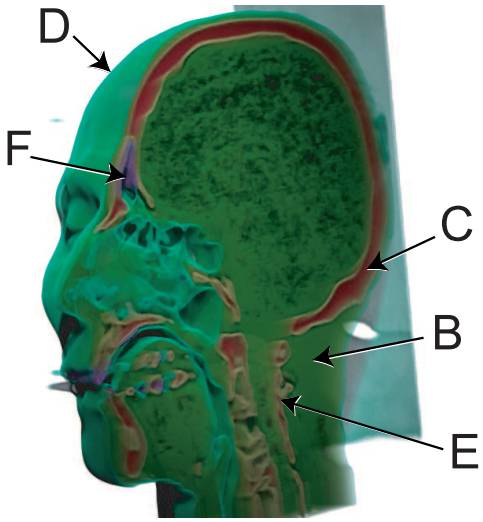
In all, our experience has shown that the best transfer functions are specified using an iterative process. When a volume is first encountered, it is important to get an immediate sense of the structures contained in the data. In many cases, a default transfer function can achieve this. By assigning higher opacity to higher gradient magnitudes and varying color based on data value, as seen in Figure 6.11, most of the important features of the datasets are visualized. The process of volume probing



(a) A 1D histogram. The black region represents the number of data value occurrences on a linear scale, the grey is on a log scale. The colored regions (A,B,C) identify basic materials.



(b) A log-scale 2D joint histogram. The lower image shows the location of materials (A,B,C), and material boundaries (D,E,F).



(c) A volume rendering showing all of the materials and boundaries identified above, except air (A), using a 2D transfer function.

Figure 6.8: Material and boundary identification of the Chapel Hill CT Head with data value alone(a) and data value and gradient magnitude (f')(b). The basic materials captured by CT, air (A), soft tissue (B), and bone (C) can be identified using a 1D transfer function as seen in (a). 1D transfer functions, however, cannot capture the complex combinations of material boundaries; air and soft tissue boundary (D), soft tissue and bone boundary (E), and air and bone boundary (F) as seen in (b) and (c).

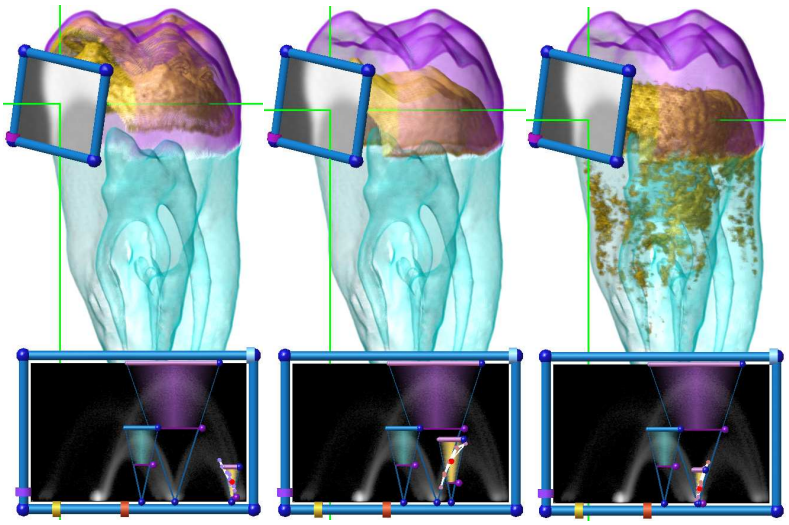


Figure 6.9: Probing and dual-domain interaction.

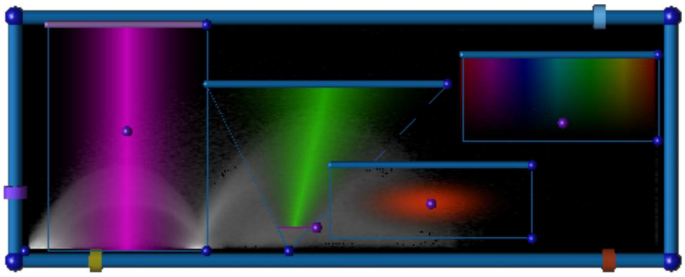


Figure 6.10: Classification widgets

allows the user to identify the location of data values in the transfer function domain that correspond to specific features. Dual-domain interaction allows the user to set the transfer function by simply pointing at a feature. By having simple control points on discrete classification widgets the user can manipulate the transfer function directly to expose a feature in the best way possible. By iterating through this process of exploration, specification, and refinement, a user can efficiently specify a transfer function that produces a high quality visualization.

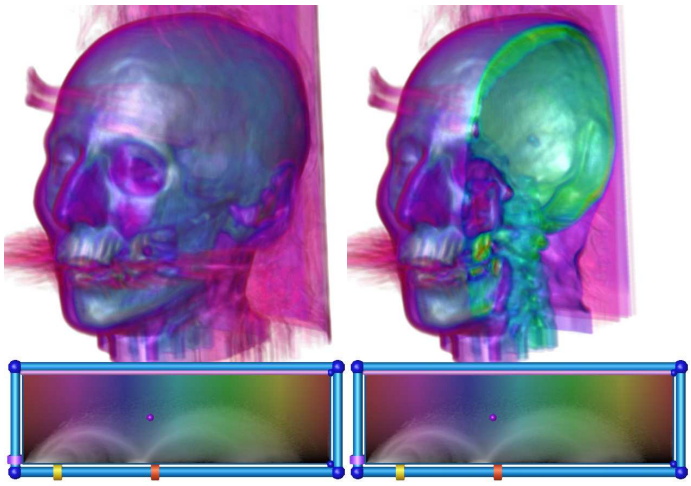


Figure 6.11: The “default” transfer function.

Implementation

The evaluation of the transfer function is computationally expensive and time consuming, and when implemented on the CPU the rate of display is limited. Moving the transfer function evaluation onto graphics hardware allows volume rendering to occur at interactive rates. Evaluating a transfer function using graphics hardware effectively amounts to an arbitrary function evaluation of data value via a table lookup. This can be accomplished in two ways, using a user defined lookup table and using dependent texture reads.

The first method uses the **glColorTable()** to store a user defined 1D lookup table, which encodes the transfer function. When **GL_COLOR_TABLE** is enabled, this function replaces an 8 bit texel with the RGBA components at that 8 bit value's position in the lookup table. Some high end graphics cards permit lookups based on 12 bit texels. On some commodity graphics cards, such as the NVIDIA GeForce, the color table is an extension known as **paletted texture**. On these platforms, the use of the color table requires that the data texture have an internal format of **GL_COLOR_INDEX*_EXT**, where * is the number of bits of precision that the data texture will have (1,2,4,or 8). Other platforms may require that the data texture's internal format be **GL_INTENSITY8**.

The second method uses **dependent texture reads**. A dependent texture read is the process by which the color components from one texture are converted to texture coordinates and used to read from a second texture. In volume rendering, the first texture is the data texture and the second is the transfer function. The GL extensions and function calls that enable this feature vary depending on the hardware, but their functionality is equivalent. On older GeForce3 and GeForce4, this functionality is part of the **Texture Shader** extensions. On the ATI Radeon 8500, dependent texture reads are part of the **Fragment Shader** extension. Fortunately, modern hardware platforms, GeforceFX and Radeon 9700 and later, provide a much more intuitive and simple mechanism to perform dependent texture reads via the **ARB_Fragment_Program**

extension. While dependent texture reads can be slower than using a color table, they are much more flexible. Dependent texture reads can be used to evaluate multi-dimensional transfer functions, or they can be used for pre-integrated transfer function evaluations. Since the transfer function can be stored as a regular texture, dependent texture reads also permit transfer functions that define more than four optical properties, achieved by using multiple transfer function textures. When dealing with transfer function domains with greater than two dimensions, it is simplest to decompose the transfer function domain into a separable product of multiple 1D or 2D transfer functions, or designing the transfer function as a procedural function based on simple mathematical primitives [25].

User Interface Tips

Figure 6.5 shows an example of an arbitrary transfer function. While this figure shows RGBα varying as piece-wise linear ramps, the transfer function can also be created using more continuous segments. The goal in specifying a transfer function is to isolate the ranges of data values, in the transfer function domain, that correspond to features, in the spatial domain. Figure 8.1 shows an example transfer function that isolates the bone in the Visible Male's skull. On the left, we see the transfer function. The alpha ramp is responsible for making the bone visible, whereas the color is constant for all of the bone. The problem with this type of visualization is that the shape and structure is not readily visible, as seen on the right side of Figure 8.1.One solution to this problem involves a simple modification of the transfer function, called *Faux shading*. By forcing the color to ramp to black proportionally to the alpha ramping to zero, we can effectively create silhouette edges in the resulting volume rendering, as seen in Figure 8.2. On the left, we see the modified transfer function. In the center, we see the resulting volume rendered image. Notice how much more clear the features are in this image. This approach

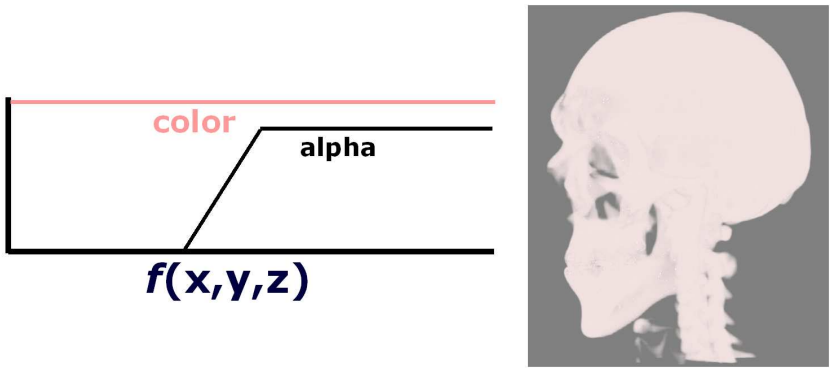


Figure 8.1: An example transfer function for the bone of the Visible Male (left), and the resulting rendering (right).

works because the darker colors are only applied at low opacities. This means that they will only accumulate enough to be visible when a viewing ray grazes a classified feature, as seen on the right side of Figure 8.2. While this approach may not produce images as compelling as surface shaded or shadowed renderings as seen in Figure 8.3, it is advantageous because it doesn't require any extra computation in the rendering phase.

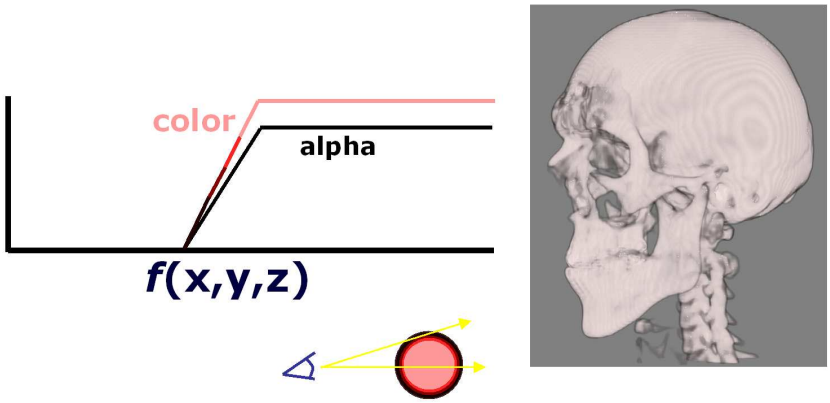


Figure 8.2: Faux shading. Modify the transfer function to create silhouette edges.

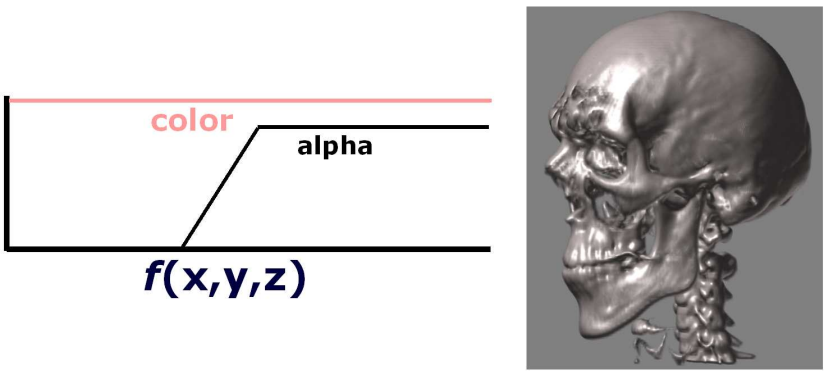


Figure 8.3: Surface shading.

Course Notes T7

Real-Time Volume Graphics

Local Volume Illumination

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

Basic Local Illumination

Local illumination models allow the approximation of the light intensity reflected from a point on the surface of an object. This intensity is evaluated as a function of the (local) orientation of the surface with respect to the position of a point light source and some material properties. In comparison to global illumination models indirect light, shadows and caustics are not taken into account. Local illumination models are simple, easy to evaluate and do not require the computational complexity of global illumination. The most popular local illumination model is the Phong model [33, 4], which computes the lighting as a linear combination of three different terms, an *ambient*, a *diffuse* and a *specular* term,

$$I_{\text{Phong}} \quad = \quad I_{\text{ambient}} \; + \; I_{\text{diffuse}} \; + \; I_{\text{specular}}.$$

Ambient illumination is modeled by a constant term,

$$I_{\text{ambient}} \; = \; k_a \; = \; const.$$

Without the ambient term parts of the geometry that are not directly lit would be completely black. In the real world such indirect illumination effects are caused by light intensity which is reflected from other surfaces.

Diffuse reflection refers to light which is reflected with equal intensity in all directions (*Lambertian* reflection). The brightness of a dull, matte surface is independent of the viewing direction and depends only on the *angle of incidence* φ between the direction $\vec{l}$ of the light source and the surface normal $\vec{n}$. The diffuse illumination term is written as

$$I_{\text{diffuse}} \; = \; I_p \; k_d \; \cos\varphi \; = \; I_p \; k_d \; (\vec{l} \bullet \vec{n}).$$

I_p is the intensity emitted from the light source. The surface property k_d is a constant between 0 and 1 specifying the amount of diffuse reflection as a material specific constant.

Specular reflection is exhibited by every shiny surface and causes so-called highlights. The specular lighting term incorporates the vector $\vec{v}$ that runs from the object to the viewer's eye into the lighting computation. Light is reflected in the direction of reflection $\vec{r}$ which is the direction of light $\vec{l}$ mirrored about the surface normal $\vec{n}$. For efficiency the reflection vector $\vec{r}$ can be replaced by the halfway vector $\vec{h}$,

$$I_{\text{specular}} \;=\; I_p \; k_s \; \cos^n \alpha \;=\; I_p \; k_s \; (\vec{h} \bullet \vec{n})^n.$$

The material property k_s determines the amount of specular reflection. The exponent n is called the *shininess* of the surface and is used to control the size of the highlights.

Basic Gradient Estimation

The Phong illumination models uses the normal vector to describe the local shape of an object and is primarily used for lighting of polygonal surfaces. To include the Phong illumination model into direct volume rendering, the local shape of the volumetric data set must be described by an appropriate type of vector.

For scalar fields, the gradient vector is an appropriate substitute for the surface normal as it represents the normal vector of the isosurface for each point. The gradient vector is the first order derivative of a scalar field $f(x, y, z)$, defined as

$$\nabla f \;=\; (f_x,\, f_y,\, f_z) \;=\; (\frac{\delta}{\delta x} f,\, \frac{\delta}{\delta y} f,\, \frac{\delta}{\delta z} f), \tag{9.1}$$

using the partial derivatives of f in x-, y- and z-direction, respectively. The scalar magnitude of the gradient measures the local variation of intensity quantitatively. It is computed as the absolute value of the vector,

$$||\nabla f|| \;=\; \sqrt{{f_x}^2 + {f_y}^2 + {f_z}^2}. \tag{9.2}$$

For illumination purposes only the direction of the gradient vector is of interest.

There are several approaches to estimate the directional derivatives for discrete voxel data. One common technique based on the first terms from a Taylor expansion is the *central differences method*. According to this, the directional derivative in x-direction is calculated as

$$f_x(x,y,z) \;=\; f(x+1,y,z) \;-\; f(x-1,y,z) \quad \text{with} \quad x,y,z \in \mathbb{N}. \tag{9.3}$$

Derivatives in the other directions are computed analogously. Central differences are usually the method of choice for gradient pre-computation. There also exist some gradient-less shading techniques which do not require the explicit knowledge of the gradient vectors. Such techniques usually approximate the dot product with the light direction by a forward difference in direction of the light source.

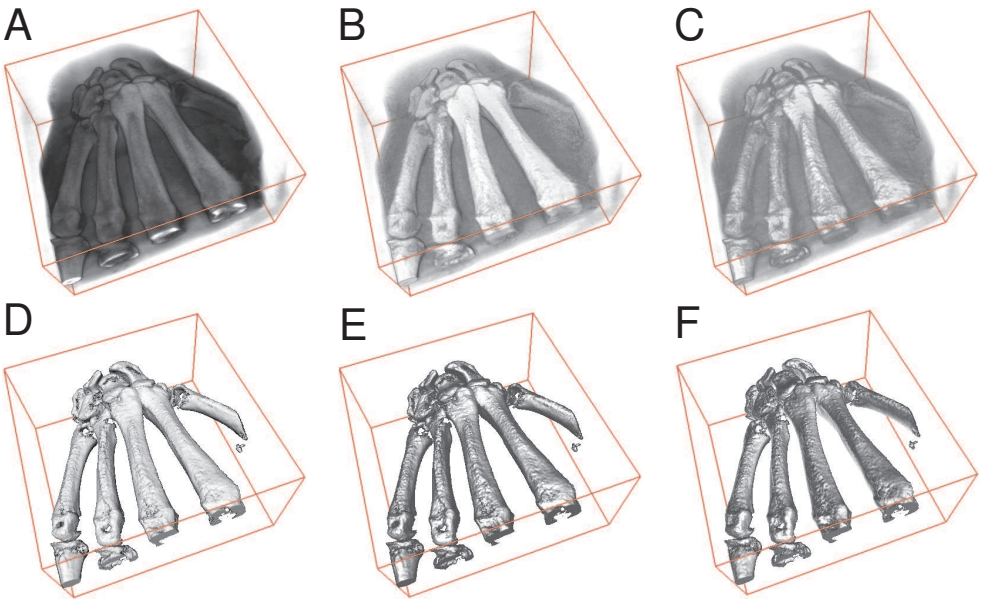


Figure 9.1: CT data of a human hand without illumination (A), with diffuse illumination (B) and with specular illumination (C). Non-polygonal isosurfaces with diffuse (D), specular (C) and diffuse and specular (E) illumination.

Simple Per-Pixel Illumination

The integration of the Phong illumination model into a single-pass volume rendering procedure requires a mechanism that allows the computation of dot products and component-wise products in hardware. This mechanism is provided by the pixel shaders functionality of modern consumer graphics boards. For each voxel, the x-, y- and z-components of the (normalized) gradient vector is pre-computed and stored as color

components in an RGB texture. The dot product calculations are directly performed within the texture unit during rasterization.

A simple mechanism that supports dot product calculation is provided by the standard OpenGL extension `EXT_texture_env_dot3`. This extension to the OpenGL texture environment defines a new way to combine the color and texture values during texture applications. As shown in the code sample, the extension is activated by setting the texture environment mode to `GL_COMBINE_EXT`. The dot product computation must be enabled by selecting `GL_DOT3_RGB_EXT` as combination mode. In the sample code the RGBA quadruplets (`GL_SRC_COLOR`) of the primary color and the texel color are used as arguments.

```
// enable the extension
 glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_COMBINE_EXT);

 // preserve the alpha value
 glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT,
GL_REPLACE);

 // enable dot product computation
 glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,
GL_DOT3_RGB_EXT);

 // first argument:  light direction stored in primary
color
 glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT,
GL_PRIMARY_COLOR_EXT);
 glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT,
GL_SRC_COLOR);

 // second argument:  voxel gradient stored in RGB
texture
 glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_TEXTURE);
 glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT,
GL_SRC_COLOR);
```

This simple implementation does neither account for the specular illumination term, nor for multiple light sources. More flexible illumination effects with multiple light sources can be achieved using newer

OpenGL extensions or high-level shading languages.

Advanced Per-Pixel Illumination

The drawback of the simple implementation described above is its restriction to a single diffuse light source. Using the fragment shading capabilities of current hardware via OpenGL extensions or high-level shading languages, additional light sources and more sophisticated lighting models and effects can be incorporated into volume shading easily. See Figure 9.1 for example images.

The following sections outline more sophisticated approaches to local illumination in volume rendering.

Non-Polygonal Isosurfaces

Rendering a volume data set with opacity values of only 0 and 1, will result in an isosurface or an isovolume. Without illumination, however, the resulting image will show nothing but the silhouette of the object as displayed in Figure 10.1 (*left*). It is obvious, that illumination techniques are required to display the surface structures (*middle* and *right*).

In a pre-processing step the gradient vector is computed for each voxel using the central differences method or any other gradient estimation scheme. The three components of the normalized gradient vector together with the original scalar value of the data set are stored as RGBA quadruplet in a 3D-texture:

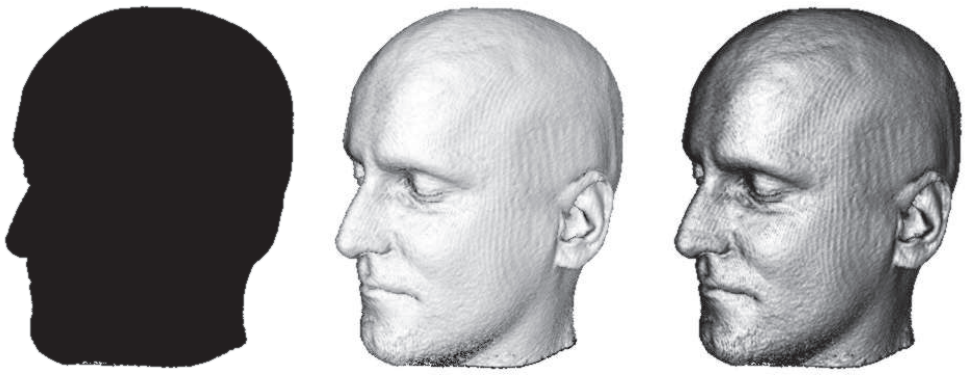


Figure 10.1: Non-polygonal isosurface without illumination (left), with diffuse illumination (middle) and with specular light (right)

$$\nabla I = \begin{pmatrix} I_x \\ I_y \\ I_z \end{pmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{matrix} \begin{matrix} \mathtt{R} \\ \mathtt{G} \\ \mathtt{B} \end{matrix}$$

$$I \longrightarrow \mathtt{A}$$

The vector components must be normalized, scaled and biased to adjust their signed range $[-1, 1]$ to the unsigned range $[0, 1]$ of the color components. In our case the alpha channel contains the scalar intensity value and the OpenGL alpha test is used to discard all fragments that do not belong to the isosurface specified by the reference alpha value. The setup for the OpenGL alpha test is displayed in the following code sample. In this case, the number of slices must be increased extremely to obtain satisfying images. Alternatively the alpha test can be set up to check for `GL_GREATER` or `GL_LESS` instead of `GL_EQUAL`, allowing a considerable reduction of the sampling rate.

```
glDisable(GL_BLEND); // Disable alpha blending

glEnable(GL_ALPHA_TEST); // Alpha test for isosurfacing
glAlphaFunc(GL_EQUAL, fIsoValue);
```

What is still missing now is the calculation the Phong illumination model. Current graphics hardware provides functionality for dot product computation in the texture application step which is performed during rasterization. Several different OpenGL extensions have been proposed by different manufacturers, two of which will be outlined in the following.

The original implementation of non-polygonal isosurfaces was presented by Westermann and Ertl [39]. The algorithm was expanded to volume shading my Meissner et al [31]. Efficient implementations on PC hardware are described in [34].

Reflection Maps

If the illumination computation becomes too complex for on-the-fly computation, alternative lighting techniques such as reflection mapping come into play. The idea of reflection mapping originates from 3D computer games and represents a method to pre-compute complex illumination scenarios. The usefulness of this approach derives from its ability to realize local illumination with an arbitrary number of light sources and different illumination parameters at low computational cost. A reflection map caches the incident illumination from all directions at a single point in space.

The idea of reflection mapping has been first suggested by Blinn [5]. The term *environment mapping* was coined by Greene [16] in 1986. Closely related to the diffuse and specular terms of the Phong illumination model, reflection mapping can be performed with diffuse maps or reflective *environment* maps. The indices into a diffuse reflection map are directly computed from the normal vector, whereas the coordinates

Figure 11.1: Example of a environment cube map.

for an environment map are a function of both the normal vector and the viewing direction. Reflection maps in general assume that the illuminated object is small with respect to the environment that contains it.

A special parameterization of the normal direction is used in order to construct a *cube map* as displayed in Figure 11.1. In this case the environment is projected onto the six sides of a surrounding cube. The largest component of the reflection vector indicates the appropriate side of the cube and the remaining vector components are used as coordinates for the corresponding texture map. Cubic mapping is popular because the required reflection maps can easily be constructed using conventional rendering systems and photography.

Since the reflection map is generated in the world coordinate space, accurate application of a normal map requires to account for the local transformation represented by the current modeling matrix. For reflective maps the viewing direction must also be taken into account. See figure 11.2 for example images of isosurface rendering with reflection mapping.

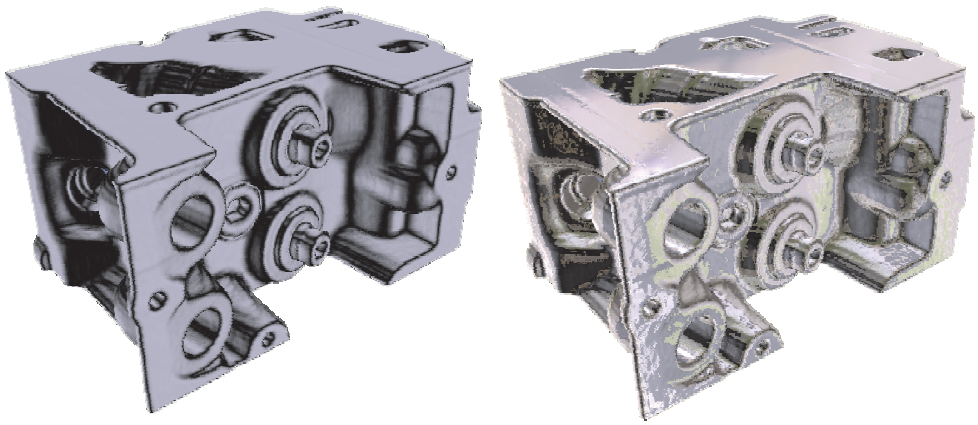


Figure 11.2: Isosurface of the engine block with diffuse reflection map (left) and specular environment map (right).

Course Notes T7

Real-Time Volume Graphics

Global Volume Illumination

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

Introduction

In the chapter on transfer functions we discussed various techniques for classifying patterns, specifically ranges of data value, that identify key features of interest in volumetric data. These techniques are most applicable to visualization tasks, where the data is acquired via scanning devices or numerical simulation. This chapter focuses on the application of optical properties, based on the classification, to generate meaningful images. In general, the discussion in this chapter applies to nearly all volume graphics application, whether they be visualization or general entertainment applications of volume rendering. Just as the domain of the transfer function isn't limited to scalar data, the range of the transfer function isn't limited to red, green, blue, and alpha color values.

Light Transport

The traditional volume rendering equation proposed by Levoy [28] is a simplified approximation of a more general volumetric light transport equation first used in computer graphics by Kajiya [21]. The rendering equation describes the interaction of light and matter as a series of scattering and absorption events of small particles. Accurate, analytic, solutions to the rendering equation however, are difficult and very time consuming. A survey of this problem in the context of volume rendering can be found in [30]. The optical properties required to describe the interaction of light with a material are spectral, *i.e.* each wavelength of light may interact with the material differently. The most commonly used optical properties are absorption, scattering, and phase function. Other important optical properties are index of refraction and emission. Volume rendering models that take into account scattering effects are complicated by the fact that each element in the volume can potentially contribute light to each other element. This is similar to other global illumination problems in computer graphics. For this reason, the traditional volume rendering equation ignores scattering effects and focuses on emission and absorption only. In this section, our discussion of optical properties and volume rendering equations will begin with simplified approximations and progressively add complexity. Figure 13.1 illustrates the geometric setup common to each of the approximations.

13.1 Traditional volume rendering

The classic volume rendering model originally proposed by Levoy [28] deals with direct lighting only with no shadowing. If we parameterize a ray in terms of a distance from the background point x_0 in direction $\vec{\omega}$ we have:

$$x(s) = x_0 + s\vec{\omega} \tag{13.1}$$

The classic volume rendering model is then written as:

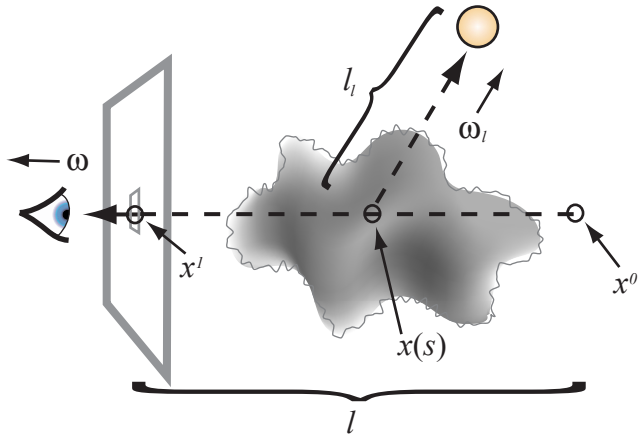


Figure 13.1: The geometric setup for light transport equations.

$$L(x_1, \vec{\omega}) \quad = \quad T(0,l)L(x_0, \vec{\omega}) \; + \; \int_0^l T(s,l)R(x(s))f_s(x(s))L_l ds \quad (13.2)$$

where R is the surface reflectivity color, f_s is the Blinn-Phong surface shading model evaluated using the normalized gradient of the scalar data field at $x(s)$, and L_l is the intensity of a point light source. $L(x_0, \vec{\omega})$ is the background intensity and T the amount the light is attenuated between two points in the volume:

$$T(s,l) = \exp\left(-\int_s^l \tau(s')ds'\right) \quad (13.3)$$

and $\tau(s')$ is the attenuation coefficient at the sample s'. This volume shading model assumes external illumination from a point light source that arrives at each sample unimpeded by the intervening volume. The only optical properties required for this model are an achromatic attenuation term and the surface reflectivity color, $R(x)$. Naturally, this model is well-suited for rendering surface-like structures in the volume, but performs poorly when attempting to render more translucent materials such as clouds and smoke. Often, the surface lighting terms are dropped and the surface reflectivity color, R, is replaced with the emission term, E:

$$L(x_1, \vec{\omega}) = T(0,l)L(x_0, \vec{\omega}) + \int_0^l T(s,l)E(x(s))ds \quad (13.4)$$

This is often referred to as the emission/absorption model. As with the classical volume rendering model, the emission/absorption model only requires two optical properties, α and E. In general, R, from the classical model, and E, from the emission/absorption model, are used interchangeably. This model also ignores inscattering. This means that although volume elements are emitting light in all directions, we only need to consider how this emitted light is attenuated on its path toward the eye. This model is well suited for rendering phenomena such as flame.

13.2 The Surface Scalar

While surface shading can dramatically enhance the visual quality of the rendering, it cannot adequately light homogeneous regions. Since the normalized gradient of the scalar field is used as the surface normal for shading, problems can arise when shading regions where the normal cannot be measured. The gradient nears zero in homogeneous regions where there is little or no local change in data value, making the normal undefined. In practice, data sets contain noise that further complicates the use of the gradient as a normal. This problem can be easily handled, however, by introducing a *surface scalar* term $S(s)$ to the rendering equation. The role of this term is to interpolate between shaded and unshaded. Here we modify the R term from the traditional rendering equation:

$$R'(s) = R(s)\left((1 - S(s)) + f_s(s)S(s)\right) \tag{13.5}$$

$S(s)$ can be acquired in a variety of ways. If the gradient magnitude is available at each sample, we can use it to compute $S(s)$. This usage implies that only regions with a high enough gradient magnitudes should be shaded. This is reasonable since homogeneous regions should have a very low gradient magnitude. This term loosely correlates to the index of refraction. In practice we use:

$$S(s) = 1 - (1 - \|\nabla f(s)\|)^2 \tag{13.6}$$

Figure 13.2 demonstrates the use of the surface scalar ($S(s)$). The image on the left is a volume rendering of the visible male with the soft tissue (a relatively homogeneous material) surface shaded, illustrating how this region is poorly illuminated. On the right, only samples with high gradient magnitudes are surface shaded.

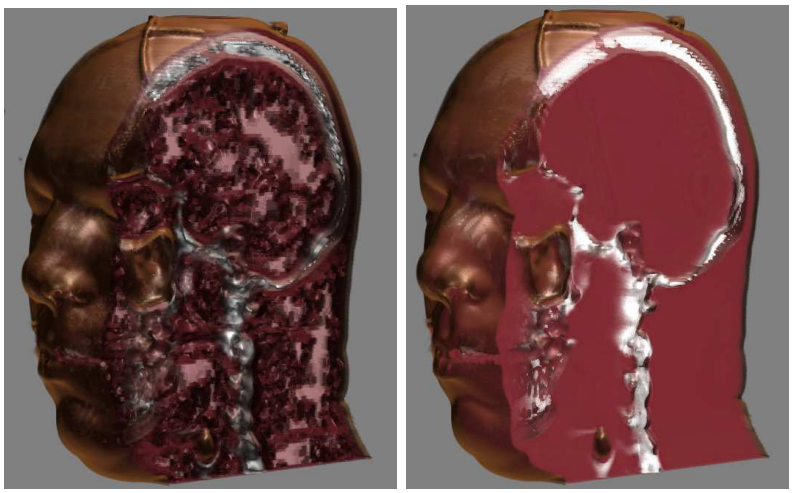


Figure 13.2: Surface shading without (left) and with (right) the surface scalar.

13.3 Shadows

Surface shading improves the visual quality of volume renderings. However, the lighting model is unrealistic because it assumes that light arrives at a sample without interacting with the portions of the volume between the sample and the light source. To model such interaction, volumetric shadows can be added to the volume rendering equation:

$$I_{eye} = I_B * T_e(0) + \int_0^{eye} T_e(s) * R(s) * f_s(s) * I_l(s) ds \tag{13.7}$$

$$I_l(s) = I_l(0) * exp\left(- \int_s^{light} \tau(x) dx \right) \tag{13.8}$$

where $I_l(0)$ is the light intensity, and $I_l(s)$ is the light intensity at sample s. Notice that $I_l(s)$ is similar to $T_e(s)$ except that the integral is evaluated from the sample toward the light rather than the eye, computing the light intensity that arrives at the sample from the light source.

A hardware model for computing shadows was first presented by Behrens and Ratering [3]. This model computes a second volume, the volumetric shadow map, for storing the amount of light arriving at each sample. At each sample, values from the second volume are multiplied by the colors from the original volume after the transfer function has been evaluated. This approach, however, suffers from an artifact referred to as attenuation leakage. The attenuation at a given sample point is blurred

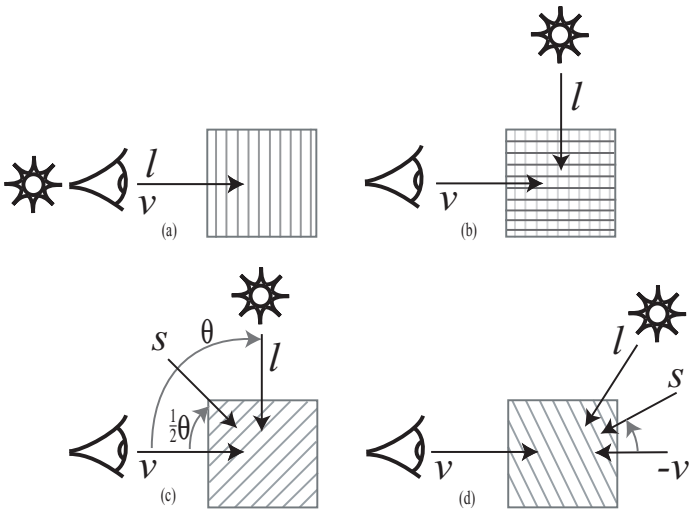


Figure 13.3: Modified slice axis for light transport.

when light intensity is stored at a coarse resolution and interpolated during the observer rendering phase. The visual consequences are blurry shadows, and surfaces that appear too dark due to the image space high frequencies introduced by the transfer function.

A simple and efficient alternative was proposed in [24]. First, rather than creating a volumetric shadow map, an off screen render buffer is utilized to accumulate the amount of light attenuated from the light's point of view. Second, the slice axis is modified to be the direction halfway between the view and light directions. This allows the same slice to be rendered from point of view of both the eye and light. Figure 13.3(a) demonstrates computing shadows when the view and light directions are the same. Since the slices for both the eye and light have a one to one correspondence, it is not necessary to pre-compute a volumetric shadow map. The amount of light arriving at a particular slice is equal to one minus the accumulated opacity of the slices rendered before it. Naturally if the projection matrices for the eye and light differ, we need to maintain a separate buffer for the attenuation from the light's point of view. When the eye and light directions differ, the volume is sliced along each direction independently. The worst case scenario is when the view and light directions are perpendicular, as seen in Figure 13.3(b). In the case, it would seem necessary to save a full volumetric shadow map which can be re-sliced with the data volume from the eye's point of view providing shadows. This approach also suffers from attenuation leakage resulting

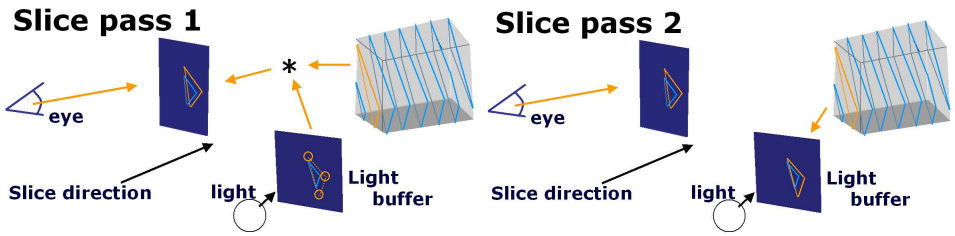


Figure 13.4: Two-pass shadows. Step 1 (left) render a slice for the eye, multiplying it by the attenuation in the light buffer. Step 2 (right) render the slice into the light buffer to update the attenuation for the next pass.

in blurry shadows and dark surfaces.

Rather than slice along the vector defined by the view or the light directions, we can modify the slice axis to allow the same slice to be rendered from both points of view. When the dot product of the light and view directions is positive, we slice along the vector halfway between the light and view directions, as demonstrated in Figure 13.3(c). In this case, the volume is rendered in front to back order with respect to the observer. When the dot product is negative, we slice along the vector halfway between the light and the inverted view directions, as in Figure 13.3(d). In this case, the volume is rendered in back to front order with respect to the observer. In both cases the volume is rendered in front to back order with respect to the light. Care must be taken to insure that the slice spacings along the view and light directions are maintained when the light or eye positions change. If the desired slice spacing along the view direction is d_v and the angle between v and l is θ then the slice spacing along the slice direction is

$$d_s = \cos(\frac{\theta}{2})d_v. \tag{13.9}$$

This is a multi-pass approach. Each slice is rendered first from the observer's point of view using the results of the previous pass from the light's point of view, which modulates the brightness of samples in the current slice. The same slice is then rendered from the light's point of view to calculate the intensity of the light arriving at the next layer.

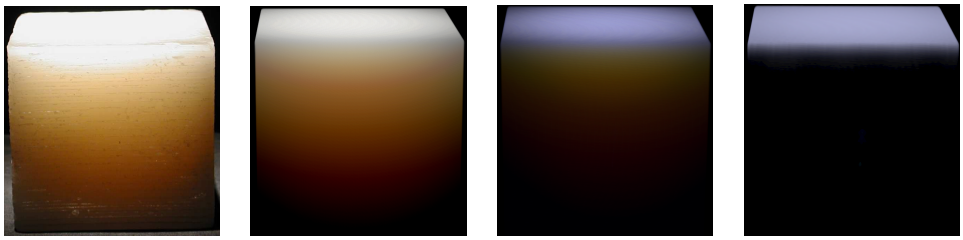
Since we must keep track of the amount of light attenuated at each slice, we utilize an off screen render buffer, known as the *pixel buffer*. This buffer is initialized to $1 - light\,intensity$. It can also be initialized using an arbitrary image to create effects such as spotlights. The projection matrix for the light's point of view need not be orthographic; a

perspective projection matrix can be used for point light sources. However, the entire volume must fit in the light's view frustum, so that light is transported through the entire volume. Light is attenuated by simply accumulating the opacity for each sample using the over operator. The results are then copied to a texture which is multiplied with the next slice from the eye's point of view before it is blended into the frame buffer. While this copy to texture operation has been highly optimized on the current generation of graphics hardware, we have achieved a dramatic increase in performance using a hardware extension known as *render to texture*. This extension allows us to directly bind a pixel buffer as a texture, avoiding the unnecessary copy operation. The two pass process is illustrated in Figure 13.4.

13.4 Translucency

Shadows can add a valuable depth queue as well as dramatic effects to a volume rendered scene. Even if the technique for rendering shadows can avoid attenuation leakage, the images can still appear too dark. This is not an artifact, it is an accurate rendering of materials which only absorb light and do not scatter it. Volume rendering models that account for scattering effects are too computationally expensive for interactive hardware based approaches. This means that approximations are needed to capture some of the effects of scattering. One such visual consequence of scattering in volumes is translucency. Translucency is the effect of light propagating deep into a material even though objects occluded by it cannot be clearly distinguished. Figure 13.5(a) shows a common translucent object, wax. Other translucent objects are skin, smoke, and clouds. Several simplified optical models for hardware based rendering of clouds have been proposed [19, 8]. These models are capable of producing realistic images of clouds, but do not easily extend to general volume rendering applications.

The previously presented model for computing shadows can easily be extended to achieve the effect of translucency. Two modifications are required. First, a second alpha value (α_i) is added which represents the amount of indirect attenuation. This value should be less than or equal to the alpha value for the direct attenuation. Second, an additional light buffer is needed for blurring the indirect attenuation. Thetranslucent volume rendering model then becomes:



(a) Wax (b) Translucent rendering (c) Different reflective color (d) Just shadows

Figure 13.5: Translucent volume shading. (a) is a photograph of wax block illuminated from above with a focused flashlight. (b) is a volume rendering with a white reflective color and a desaturated orange transport color (1− *indirect attenuation*). (c) has a bright blue reflective color and the same transport color as the upper right image. (d) shows the effect of light transport that only takes into account direct attenuation.

$$I_{eye} = I_0 * T_e(0) + \int_0^{eye} T_e(s) * C(s) * I_l(s) ds \tag{13.10}$$

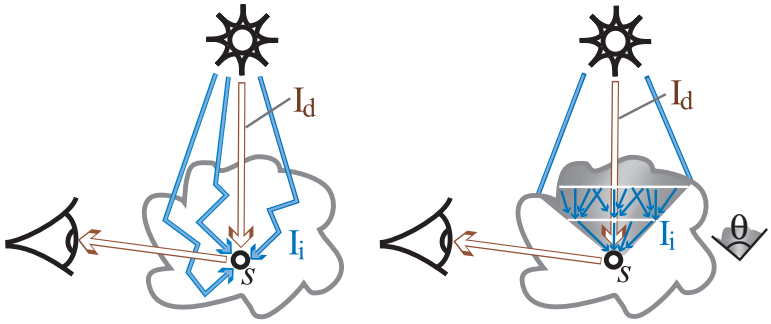
$$\begin{aligned} I_l(s) &= I_l(0) * exp\left(-\int_s^{light} \tau(x)dx\right) + \\ & \quad I_l(0) * exp\left(-\int_s^{light} \tau_i(x)dx\right) \mathbf{Blur}(\theta) \end{aligned} \tag{13.11}$$

where $\tau_i(s)$ is the indirect light extinction term, $C(s)$ is the reflective color at the sample s, $S(s)$ is a surface shading parameter, and I_l is the sum of the direct and indirect light contributions.

The indirect extinction term is spectral, meaning that it describes the indirect attenuation of light for each of the R, G, and B color components. Similar to the direct extinction, the indirect attenuation can be specified in terms of an indirect alpha:

$$\alpha_i = exp(-\tau_i(x)). \tag{13.12}$$

While this is useful for computing the attenuation, it is non-intuitive for user specification. Instead, specifying a *transport color* which is $1 - \alpha_i$ is more intuitive since the transport color is the color the indirect light will become as it is attenuated by the material.



(a) General Light Transport

(b) Translucency Approximation

Figure 13.6: On the left is the general case of direct illumination I_d and scattered indirect illumination I_i. On the right is a translucent shading model which includes the direct illumination I_d and approximates the indirect, I_i, by blurring within the shaded region. Theta is the angle indicated by the shaded region.

In general, light transport in participating media must take into account the incoming light from all directions, as seen in Figure 13.6(a). However, the net effect of multiple scattering in volumes is a blurring of light. The diffusion approximation [38, 13] models the light transport in multiple scattering media as a random walk. This results in light being diffused within the volume. The **Blur**(θ) operation in Equation 13.11 averages the incoming light within the cone with an apex angle θ in the direction of the light (Figure 13.6(b)). The indirect lighting at a particular sample is only dependent on a local neighborhood of samples computed in the previous iteration and shown as the arrows between slices in (b). This operation models light diffusion by convolving several random sampling points with a Gaussian filter.

The process of rendering using translucency is essentially the same as rendering shadows. In the first pass, a slice is rendered from the point of view of the light. However, rather than simply multiplying the sample's color by one minus the direct attenuation, one minus the direct and one minus the indirect attenuation is summed to compute the light intensity at the sample. In the second pass, a slice is rendered into the **next** light buffer from the light's point of view to compute the lighting for the next iteration. Two light buffers are maintained to accommodate the blur operation required for the indirect attenuation, **next** is the buffer being rendered to and **current** is the buffer bound as a texture.

Rather than blending slices using the standard OpenGL blend operation, the blend is explicitly computed in the fragment shading stage. The **current** light buffer is sampled once in the first pass, for the observer, and multiple times in the second pass, for the light, using the *render to texture* OpenGL extension. Whereas, the **next** light buffer, is rendered into only in the second pass. This relationship changes after the second pass so that the **next** buffer becomes the **current** and *vice versa*. We call this approach *ping pong blending*. In the fragment shading stage, the texture coordinates for the **current** light buffer, in all but one texture unit, are modified per-pixel using a random noise texture. The number of samples used for the computation of the indirect light is limited by the number of texture units. Randomizing the sample offsets masks some artifacts caused by this coarse sampling. The amount of this offset is bounded based on a user defined blur angle (θ) and the sample distance (d):

$$offset \leq d \tan(\frac{\theta}{2}) \tag{13.13}$$

The **current** light buffer is then read using the new texture coordinates. These values are weighted and summed to compute the blurred inward flux at the sample. The transfer function is evaluated for the incoming slice data to obtain the indirect attenuation (α_i) and direct attenuation (α) values for the current slice. The blurred inward flux is attenuated using α_i and written to the RGB components of the **next** light buffer. The alpha value from the **current** light buffer with the unmodified texture coordinates is blended with the α value from the transfer function to compute the direct attenuation and stored in the alpha component of the **next** light buffer.

This process is enumerated below:

1. Clear color buffer.
2. Initialize pixel buffer with 1-light color (or light map).
3. Set slice direction to the halfway between light and observer view directions.
4. For each slice:
   (a) Determine the locations of slice vertices in the light buffer.
   (b) Convert these light buffer vertex positions to texture coordinates.

(c) Bind the light buffer as a texture using these texture coordinates.

(d) In the Per-fragment blend stage:

i. Evaluate the transfer function for the Reflective color and direct attenuation.

ii. Evaluate surface shading model if desired (this replaces the Reflective color).

iii. Evaluate the phase function, using a lookup of the dot of the viewing and light directions.

iv. Multiply the reflective color by the 1-direct attenuation from the light buffer.

v. Multiply the reflective*direct color by the phase function.

vi. Multiply the Reflective color by 1-(indirect) from the light buffer.

vii. Sum the direct*reflective*phase and indirect*reflective to get the final sample color.

viii. The alpha value is the direct attenuation from the transfer function.

(e) Render and blend the slice into the frame buffer for the observer's point of view.

(f) Render the slice (from the light's point of view) to the position in the light buffer used for the observer slice.

(g) In the Per-fragment blend stage:

i. Evaluate the transfer function for the direct and indirect attenuation.

ii. Sample the light buffer at multiple locations.

iii. Weight and sum the samples to compute the blurred indirect attenuation. The weight is the blur kernel.

iv. Blend the blurred indirect and un-blurred direct attenuation with the values from the transfer function.

(h) Render the slice into the correct light buffer.

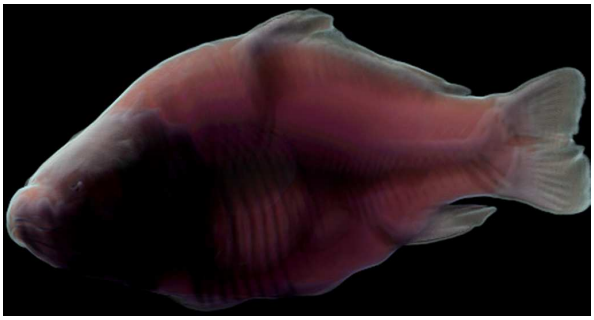
While this process my seem quite complicated, it is straightforward to implement. The *render to texture* extension is part of the **WGL_ARB_render_texture** OpenGL extensions. The key functions are **wglBindTexImageARB()** which binds a *P-Buffer* as a texture, and **wglReleaseTexImageARB()** which releases a bound *P-Buffer* so

that it may be rendered to again. The texture coordinates of a slice's light intensities from a light buffer are the 2D positions that the slice's vertices project to in the light buffer scaled and biased so that they are in the range zero to one.

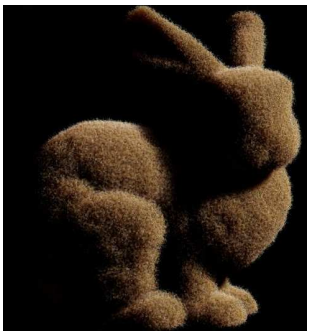
Computing volumetric light transport in screen space is advantageous because the resolution of these calculations and the resolution of the volume rendering can match. This means that the resolution of the light transport is decoupled from that of the data volume's grid, permitting procedural volumetric texturing.

13.5 Summary

Rendering and shading techniques are important for volume graphics, but they would not be useful unless we had a way to transform interpolated data into optical properties. While the traditional volume rendering model only takes into account a few basic optical properties, it is important to consider additional optical properties. Even if these optical properties imply a much more complicated rendering model than is possible with current rendering techniques, adequate approximations can be developed which add considerably to the visual quality. We anticipate that the development of multiple scattering volume shading models will be an active area of research in the future.



(a) Carp CT



(b) Stanford Bunny

(c) Joseph the Convicted

Figure 13.7: Example volume renderings using an extended transfer function.

Course Notes T7

Real-Time Volume Graphics

High-Quality Volume Rendering

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

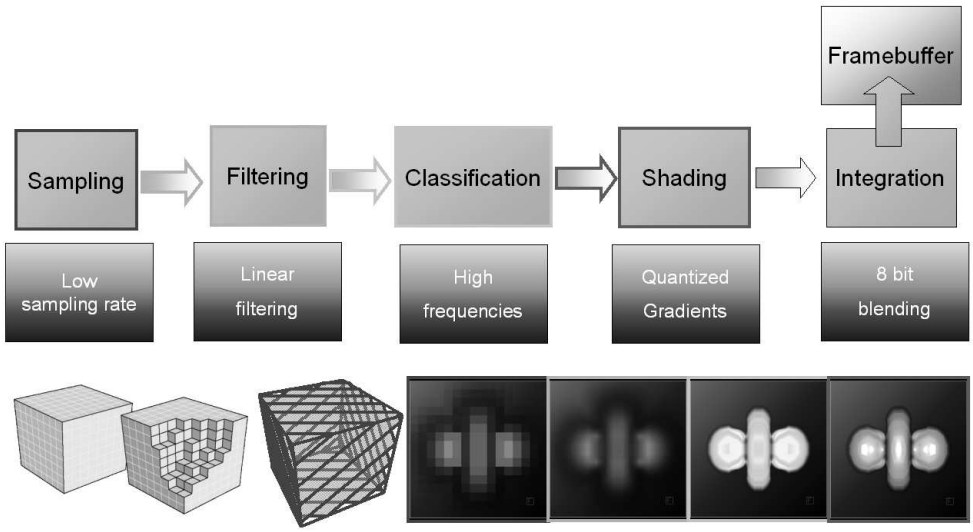


Figure 13.8: Volume rendering pipeline. Each step in this pipeline can introduce artifacts.

Today's GPUs support high-quality volume rendering (see figures ??). However, a careful examination of the results of various visualization packages reveals unpleasant artifacts in volumetric renderings. Especially in the context of real-time performance, which requires certain compromises to achieve high frame rates, errors seem to be inevitable.

To synthesize high-quality volume visualization results it is necessary to identify possible sources of artifacts. Those artifacts are introduced in various stages of the volume rendering pipeline. Generally speaking, the volume rendering pipeline consists of five stages (see figure 13.8): First of all, a sampling stage, which accesses the volume data along straight rays through the volume. Secondly, a filtering stage, that interpolates the voxel values. Thirdly, a classification step, which maps scalar values from the volume to emission and absorption coefficients. The fourth stage in this pipeline is optional and is only applied if external light sources are taken into account for computing the shading of the volume data. Finally, the integration of the volume data is performed. This is achieved in graphics hardware by blending emission colors with their associated alpha values into the frame buffer. This pipeline is repeated until all samples along the rays through the volume have been processed. Each of the stages of pipeline can be the source of artifacts.

Note that sampling and filtering are actually done in the same step in graphics hardware, i.e., during volume rendering we set sample position using texture coordinates of slice polygons or by computing texture coordinates explicitly using ray-marching in a fragment program for ray-casting-based approaches. The hardware automatically performs filtering as soon as the volume is accessed with a texture fetch with a position identifies using the corresponding a texture coordinate. The type of filtering performed by the graphics hardware is specified by setting the appropriate OpenGL state. Current graphics hardware only supports nearest neighbor and linear filtering, i.e., linear, bilinear and trilinear filtering. However, we will treat sampling and filtering as two steps, because they become two separate operations once we implement our own filtering method.

The goal of this chapter is to remove or at least suppress artifacts that occur during volume rendering while maintaining real-time performance. For this purpose, all proposed optimizations will be performed on the GPU in order to avoid expensive readback of data from the GPU memory. We will review the stages of the volume rendering pipeline step-by-step, identify possible sources of errors introduced in the corresponding stage and explain techniques to remove or suppress those errors while ensuring interactive frame rates.

Sampling Artifacts

The first stage in the process of volume rendering consists of sampling the discrete voxel data. Current GPU-based techniques employ explicit proxy geometry to trace a large number of rays in parallel through the volume (slice-based volume rendering) or directly sample the volume along rays (ray-casting). The distance of those sampling points influences how accurately we represent the data. A large distance between those sampling points, i.e., a low sampling rate, will result in severe artifacts (see figure 14.1). This effect is often referred to as under-sampling and the associated artifacts are often referred to as wood grain artifacts.

The critical question is: How many samples do we have to take along rays in the volume to accurately represent the volume data? The answer to this question lies in the so-called Nyquist-Shannon sampling theorem of information theory.

The theorem is one of the most important rules of sampling ([32, 37]). It states that, when converting analog signals to digital, the sampling frequency must be greater than twice the highest frequency of the input signal to be able to later reconstruct the original signal from the sampled version perfectly. Otherwise the signal will be aliased, i.e. lower frequencies will be incorrectly reconstructed from the discrete signal. An analog signal can contain arbitrary high frequencies, therefore an analog low-pass filter is often applied before sampling the signal to ensure that the input signal does not have those high frequencies. Such a signal is called band-limited. For an audio signal this means, that if we want to want to sample the audio signal with 22 kHz as the highest frequency, we must at least sample the signal with twice the sampling rates; i.e., with more than 44 kHz. As already stated, this rule applies if we want to discretize contiguous signals. But want does this rule mean for sampling an already discretized signal? Well, in volume rendering we assume that the data represents samples taken from a contiguous band-limited volumetric field. During sampling we might already have lost some information due to a too low sampling rate. This is certainly something we cannot fix during rendering. However, the highest frequency in a discrete signal

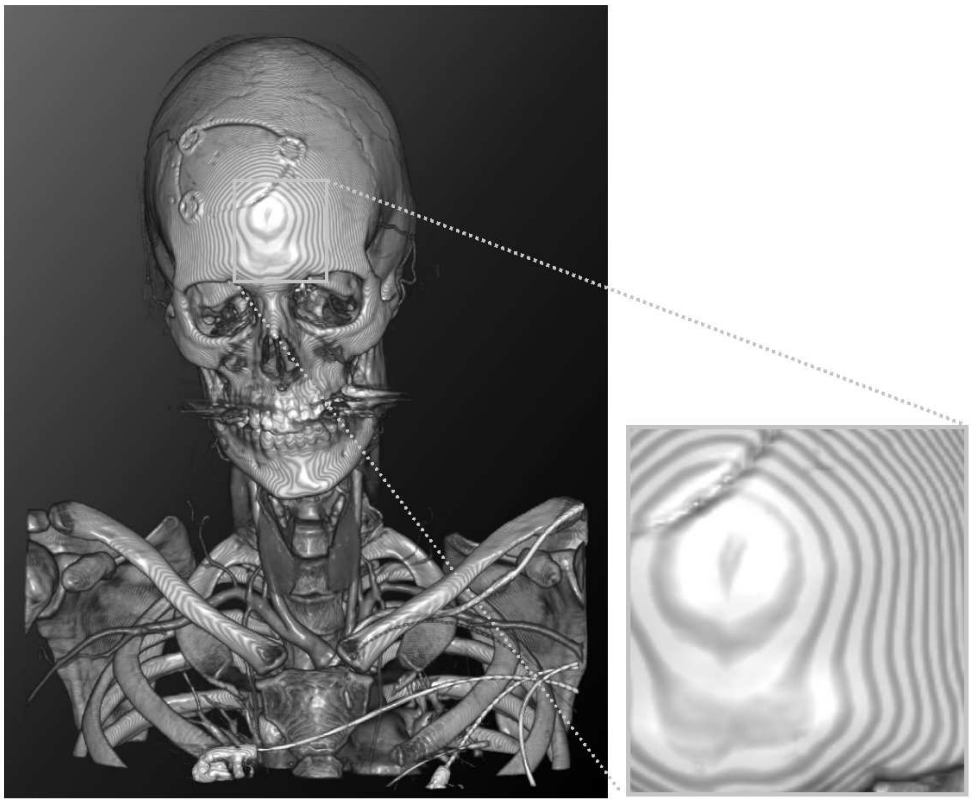


Figure 14.1: Wood grain artifact caused by a low sampling rate.

that is assumed to be contiguous is an abrupt change in the data from one sampling position to an adjacent one. This means, that the highest frequency is one divided by the distance between voxels of the volume data. Thus, in order to accurately reconstruct the original signal from the discrete data we need to take at least two samples per voxel.

There is actually no way to get around this theorem. We have to take two samples per voxel to avoid artifacts. However, taking a lot of samples along rays inside the volume has a direct impact on the performance. We achieve this high sampling rate by either increasing the number of slice polygons or by reducing the sampling distance during ray-casting. Taking twice the number of samples inside the volume will typically reduce the frame rate by a factor of two. However, volumetric data often does not consist alone of regions with high variations in the data values. In fact, volume data can be very homogeneous in certain regions while other regions contain a lot of detail and thus high frequencies. We can exploit this fact by using a technique called adaptive sampling.

Adaptive sampling techniques causes more samples to be taken in inhomogeneous regions of the volume as in homogeneous regions. In order to know if we are inside a homogeneous or inhomogeneous region of the volume during integration along our rays through the volume, we can use a 3D texture containing the sampling rate for each region. This texture will be called the importance-volume and must be computed in a preprocessing step and can have smaller spacial dimensions than our volume data texture. For volume ray-casting on the GPU it is easy to adapt the sampling rate to the sampling rate obtained from this texture because sampling positions are generated in the fragment stage. Slice-based volume rendering however, is more complicated because the sampling rate is directly set by the number of slice polygons. This means that the sampling rate is set in the vertex stage, while the sampling rate from our importance-volume is obtained in the fragment stage. The texture coordinates for sampling the volume interpolated on the slice polygons can be considered as samples for a base sampling rate. We can take additional samples along the ray direction at those sampling positions in a fragment program, thus sampling higher in regions where the data set is inhomogeneous. Note that such an implementation requires dynamic branching in a fragment program because we have to adapt the number of samples in the fragment program to the desired sampling rate at this position. Such dynamic branching is available on NVIDIA Nv40 hardware. Alternatively, computational masking using early-z or stencil culls can be employed to accelerate the rendering for regions with lower sampling rate. The slice polygon is rendered multiple times with different fragment programs for the different sampling rates, and rays (pixels) are selected by masking the corresponding pixels using the stencil- or z-buffer.

Changing the sampling rate globally or locally requires opacity correction; which can be implemented globally by changing the alpha values in the transfer function, or locally by adapting the alpha values before blending in a fragment program. The corrected opacity is function of the stored opacity α_{stored} and the sample spacing ratio $\Delta x/\Delta x_0$: $\alpha_{corrected} = 1 - [1 - \alpha_{stored}]^{\Delta x/\Delta x_0}$

We can successfully remove artifacts in volume rendering (see figure 14.2) using adaptive sampling and sampling the volume at the Nyquist frequency. However, this comes at the cost of high sampling rates that can significantly reduce performance. Even worse, in most volumetric renderings a non-linear transfer function is applied in the classification stage. This can introduce high-frequencies into the sampled data, thus increasing the required sampling rate well beyond the Nyquist frequency

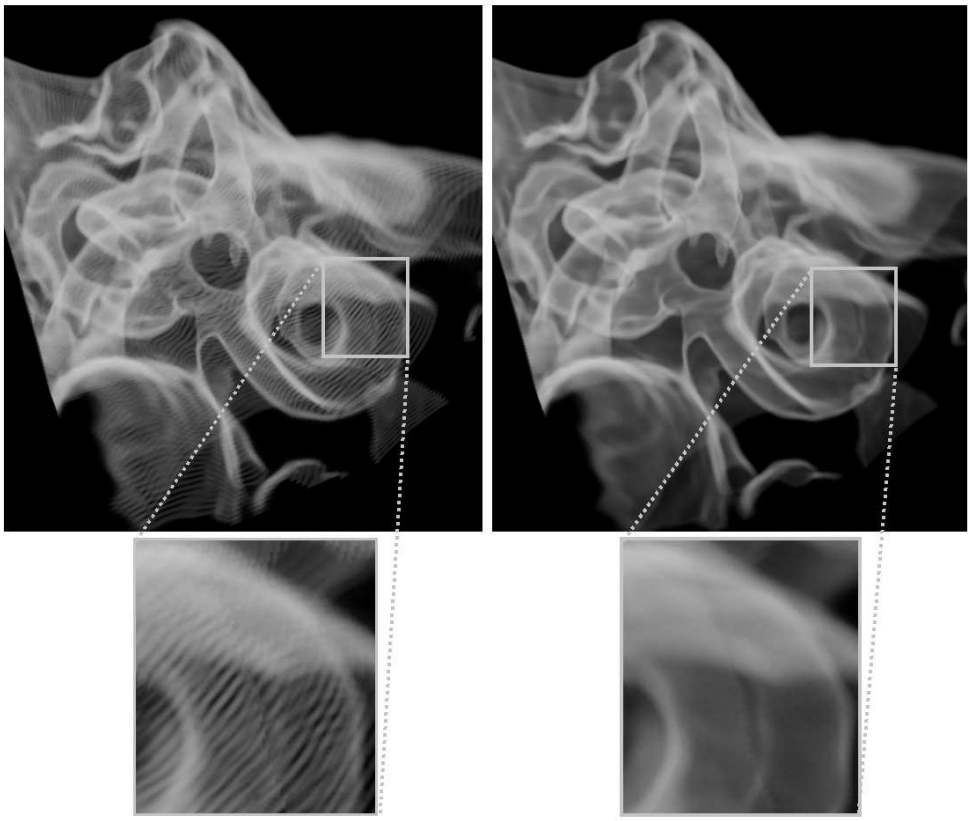


Figure 14.2: Comparison of a visualization of the inner ear with low (left) and high (right) sampling rate.

of the volume data. We will discuss this effect in detail in chapter 16 and provide a solution to the problems by using a technique that separates those high frequencies from classification in a pre-processing step.

Filtering Artifacts

The next possible source for artifacts in volumetric computer graphics is introduced during the filtering of the volume data. Basically, this phase converts the discrete volume data back to a continuous signal. To reconstruct the original continuous signal from the voxels, a reconstruction filter is applied that calculates a scalar value for the continuous three-dimensional domain (R^3) by performing a convolution of the discrete function with a filter kernel. It has been proven, that the perfect, or ideal reconstruction kernel is provided by the sinc filter.

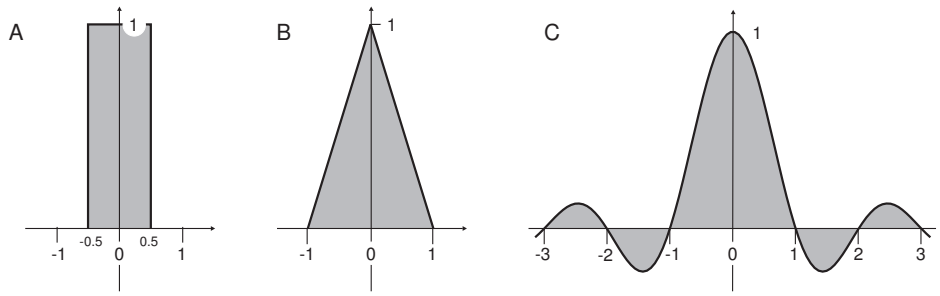


Figure 15.1: Three reconstruction filters: (a) box, (b) tent and (c) sinc filters.

Unfortunatelty, the sinc filter has an unlimited extent. Therefore, in practice simpler reconstruction filters like tent or box filters are applied (see Figure ...). Current graphics hardware supports pre-filtering mechanisms like mip-mapping and anisotropic filtering for minification and linear, bilinear, and tri-linear filters for magnification. The internal precision of the filtering on current graphics hardware is dependent on the precision of the input texture; i.e., 8 bit textures will internally only be filtered with 8 bit precision. To achieve higher quality filtering results with the built-in filtering techniques of GPUs we can use a higher-precision internal texture format when defining textures (i.e., the LUMINANCE16 and HILO texture formats). Note that floating point texture formats often do not support filtering.

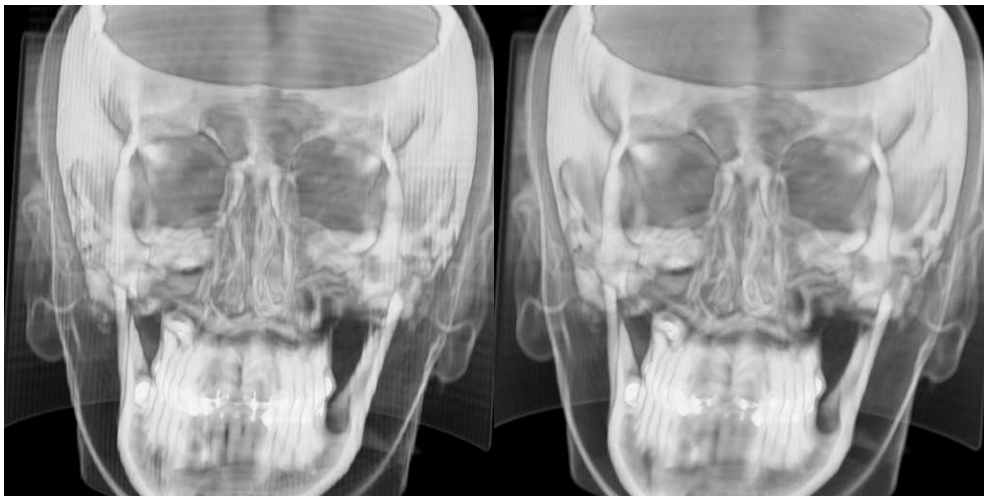


Figure 15.2: Comparison between trilinear filtering and cubic B-spline filtering.

However, the use of higher internal precision for filtering cannot on its own provide satisfactory results with built-in linear reconstruction filters (see left image in figure 15.2). Hadwiger et al.[17] have shown that multi-textures and flexible rasterization hardware can be used to evaluate arbitrary filter kernels during rendering.

The filtering of a signal can be described as the convolution of the signal function s with a filter kernel function h:

$$g(t) = (s * h)(t) = \int_{-\infty}^{\infty} s(t - t') \cdot h(t')dt' \tag{15.1}$$

The discretized form is:

$$g_t = \sum_{i=-I}^{+I} s_{t-i} h_i \tag{15.2}$$

where the half width of the filter kernel is denoted by I. The implication is, that we have to collect the contribution of neighboring input samples multiplied by the corresponding filter values to get a new filtered output sample. Instead of this *gathering* approach, Hadwiger et al. advocate a *distributing* approach for a hardware-accelerated implementation. That is, the contribution of an input sample is *distributed* to its neighboring samples, instead of the other way. The order was chosen, since this allows to collect the contribution of a single relative input sample for all output samples simultaneously. The term *relative input*

sample denotes the relative offset of an input sample to the position of an output sample. The final result is obtained by adding the result of multiple rendering passes, whereby the number of input samples that contribute to an output sample determine the number of passes.

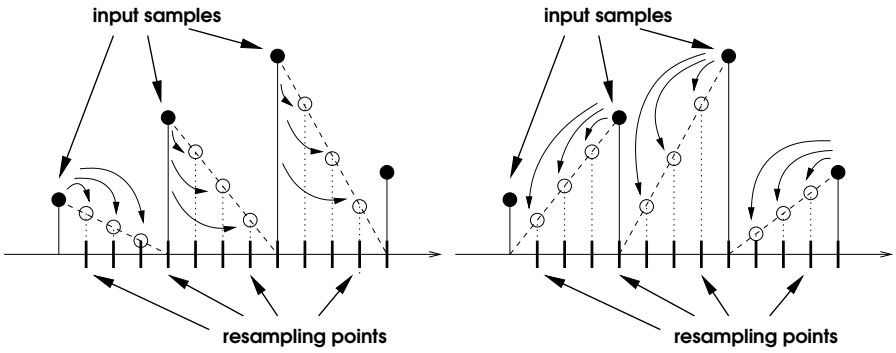


Figure 15.3: Distributing the contributions of all "left-hand" (a), and all "right-hand" (b) neighbors, when using a tent filter.

Figure 15.3 demonstrates this in the example of a one-dimensional tent filter. As one left-handed and one right-handed neighbor input sample contribute to each output sample, a two-pass approach is necessary. In the first pass, the input samples are shifted right half a voxel distance by means of texture coordinates. The input samples are stored in a texture-map that uses nearest-neighbor interpolation and is bound to the first texture stage of the multi-texture unit (see Figure 15.4). Nearest-neighbor interpolation is needed to access the original input samples over the complete half extent of the filter kernel. The filter kernel is divided into two tiles. One filter tile is stored in a second texture map, mirrored and repeated via the `GL_REPEAT` texture environment. This texture is bound to the second stage of the multi-texture unit. During rasterization the values fetched by the first multi-texture unit are multiplied with the result of the second multi-texture unit. The result is added into the frame buffer. In the second pass, the input samples are shifted left half a voxel distance by means of texture coordinates. The same unmirrored filter tile is reused for a symmetric filter . The result is again added to the frame buffer to obtain the final result.

The number of required passes can be reduced by n for hardware architectures supporting $2n$ multi-textures. That is, two multi-texture units calculate the result of a single pass. The method outlined above does not consider area-averaging filters, since it is assumed that magni-

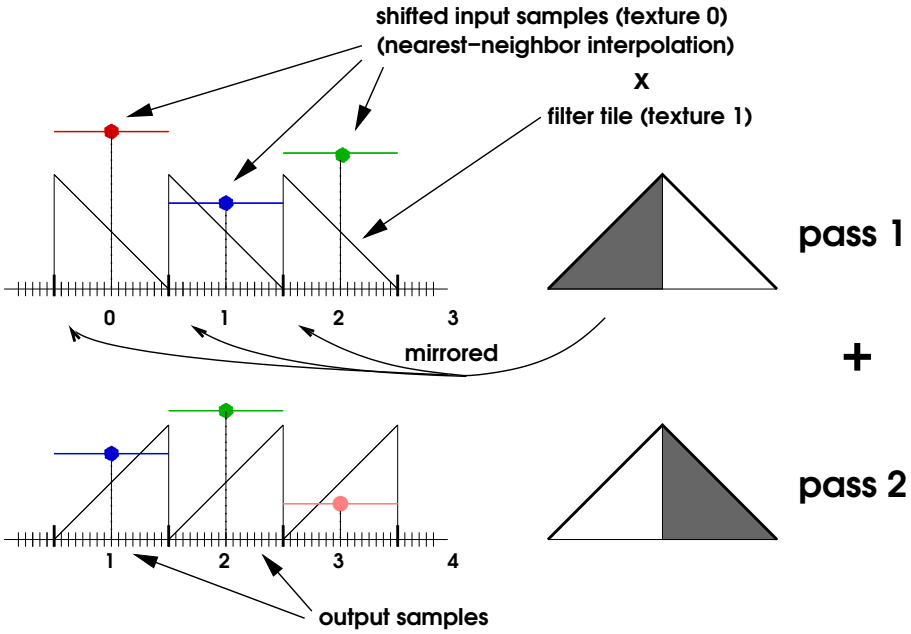


Figure 15.4: Tent filter (width two) used for reconstruction of a one-dimensional function in two passes. Imagine the values of the output samples added together from top to bottom.

fication is desired instead of minification. For minification, pre-filtering approaches like mip-mapping are advantageous. Figure 15.2 demonstrates the benefit of bi-cubic filtering using a B-spline filter kernel over a standard bi-linear interpolation.

High quality filters implemented in fragment programs can considerably improve image quality. However, it must be noted, that performing higher quality filtering in fragment programs on current graphics hardware is expensive. I.e., frame rates drop considerably. We recommend higher quality filters only for final image quality renderings. During interaction with volume data or during animations it is probably better to use build-in reconstruction filters, as artifacts will not be too apparent in motion. To prevent unnecessary calculations in transparent or occluded regions of the volume, the optimizations techniques presented in chapter ?? should be applied.

Classification Artifacts

Classification is the next crucial phase in the volume rendering pipeline and yet another possible source of artifacts. Classification employs transfer functions for color densities $\tilde{c}(s)$ and extinction densities $\tau(s)$, which map scalar values $s = s(\mathbf{x})$ to colors and extinction coefficients. The order of classification and filtering strongly influences the resulting images, as demonstrated in Figure 16.1. The image shows the results of pre- and post-classification for a 16^3 voxel hydrogen orbital volume and a high frequency transfer function for the green color channel.

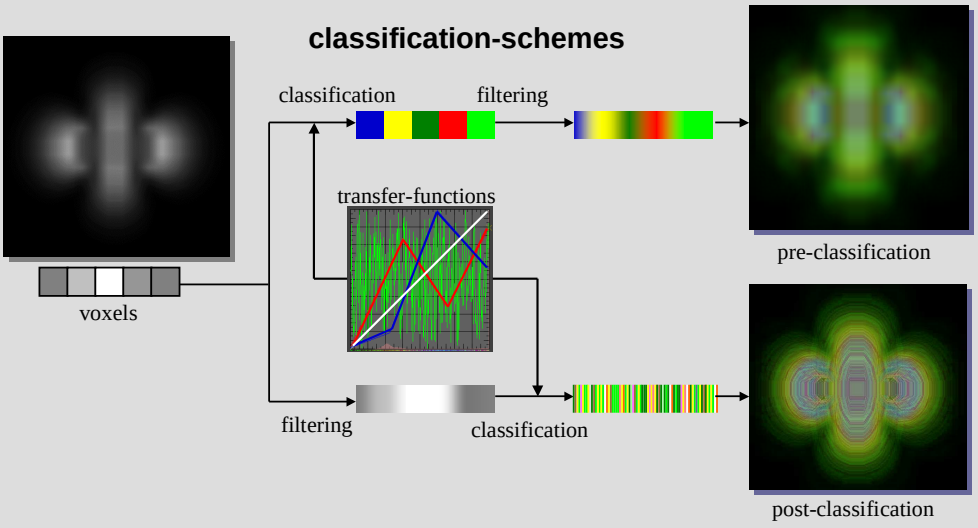


Figure 16.1: Comparison of pre-classification and post-classification. Alternate orders of classification and filtering lead to completely different results. For clarification a random transfer function is used for the green color channel. Piecewise linear transfer functions are employed for the other color channels. Note, in contrast to pre-classification, post-classification reproduces the high frequencies contained within in the transfer function.

It can be observed that pre-classification, i.e. classification before filtering, does not reproduce high-frequencies in the transfer function.

In contrast to this, post-classification, i.e. classification after filtering, reproduces high frequencies in the transfer function. However, high frequencies (e.g., iso-surface spikes) may not be reproduced in between two adjacent sampling points along a ray through the volume. To capture those details, oversampling (i.e., additional slice polygons or sampling points) must be added which directly decreases performance. Furthermore, very high frequencies in the transfer function require very high sampling rates to captured those details. It should be noted, that a high frequency transfer function does not necessarily mean a random transfer function. We only used random transfer functions to demonstrate the differences between the classification methods. A high frequency in the transfer function is easily introduced by using a simple step transfer function with steep slope. Such transfer function are very common in many application domains.

In order to overcome the limitations discussed above, the approximation of the volume rendering integral has to be improved. In fact, many improvements have been proposed, e.g., higher-order integration schemes, adaptive sampling, etc. However, these methods do not explicitly address the problem of high Nyquist frequencies of the color after the classification $\tilde{c}\big(s(\mathbf{x})\big)$ and an extinction coefficients after the classification $\tau\big(s(\mathbf{x})\big)$ resulting from non-linear transfer functions. On the other hand, the goal of *pre-integrated classification*[35] is to split the numerical integration into two integrations: one for the continuous scalar field $s(\mathbf{x})$ and one for each of the transfer functions $\tilde{c}(s)$ and $\tau(s)$ in order to avoid the problematic product of Nyquist frequencies.

The first step is the sampling of the continuous scalar field $s(\mathbf{x})$ along a viewing ray. Note that the Nyquist frequency for this sampling is not affected by the transfer functions. For the purpose of pre-integrated classification, the sampled values define a one-dimensional, piecewise linear scalar field. The volume rendering integral for this piecewise linear scalar field is efficiently computed by one table lookup for each linear segment. The three arguments of the table lookup are the scalar value at the start (front) of the segment $s_f := s\big(\mathbf{x}(id)\big)$, the scalar value the end (back) of the segment $s_b := s\big(\mathbf{x}((i+1)d)\big)$, and the length of the segment d. (See Figure 16.2.) More precisely spoken, the opacity α_i of the i-th segment is approximated by

$$\begin{aligned} \alpha_i &= 1 - \exp\left(-\int_{i\,d}^{(i+1)d} \tau\Big(s\big(\mathbf{x}(\lambda)\big)\Big)\mathrm{d}\lambda\right) \\ &\approx 1 - \exp\left(-\int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big)d\;\mathrm{d}\omega\right). \end{aligned} \tag{16.1}$$

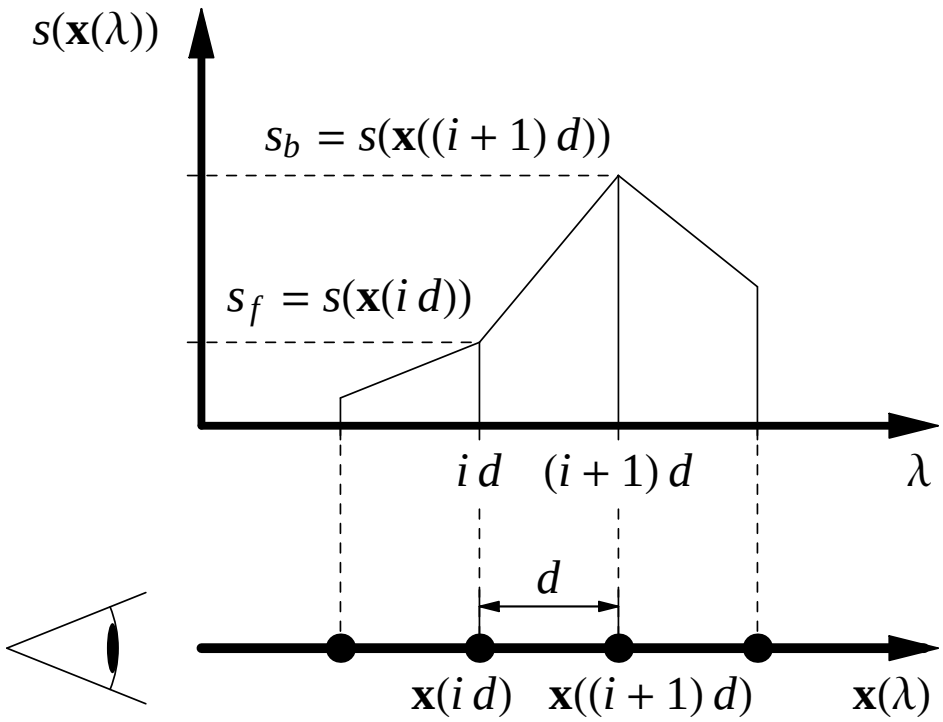


Figure 16.2: Scheme for determining the color and opacity of the i-th ray segment.

Thus, α_i is a function of s_f, s_b, and d. (Or of s_f and s_b, if the lengths of the segments are equal.) The (associated) colors $\tilde{C}_i$ are approximated correspondingly:

$$\begin{aligned}\tilde{C}_i \;\approx\; & \int_0^1 \tilde{c}\big((1-\omega)s_f + \omega s_b\big) \\ & \times \exp\Big(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big) d \,\mathrm{d}\omega'\Big) d \,\mathrm{d}\omega. \qquad (16.2)\end{aligned}$$

Analogous to α_i, $\tilde{C}_i$ is a function of s_f, s_b, and d. Thus, pre-integrated classification approximates the volume rendering integral by evaluating the following Equation:

$$I \approx \sum_{i=0}^{n} \tilde{C}_i \prod_{j=0}^{i-1} (1 - \alpha_j)$$

with colors $\tilde{C}_i$ pre-computed according to Equation (16.2) and opacities α_i pre-computed according to Equation (16.1). For non-associated color

transfer function, i.e., when substituting $\tilde{c}(s)$ by $\tau(s)c(s)$, we will also employ Equation (16.1) for the approximation of α_i and the following approximation of the associated color $\tilde{C}_i^\tau$:

$$\tilde{C}_i^\tau \approx \int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) c\big((1-\omega)s_f + \omega s_b\big) \times \exp\Big(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big) d \, \mathrm{d}\omega'\Big) d \, \mathrm{d}\omega. \quad (16.3)$$

Note that pre-integrated classification always computes associated colors, whether a transfer function for associated colors $\tilde{c}(s)$ or for non-associated colors $c(s)$ is employed.

In either case, pre-integrated classification allows us to sample a continuous scalar field $s(\mathbf{x})$ without increasing the sampling rate for any non-linear transfer function. Therefore, pre-integrated classification has the potential to improve the accuracy (less undersampling) and the performance (fewer samples) of a volume renderer at the same time.

One of the major disadvantages of the pre-integrated classification is the need to integrate a large number of ray-segments for each new transfer function dependent on the front and back scalar value and the ray-segment length. Consequently, an interactive modification of the transfer function is not possible. Therefore several modifications to the computation of the ray-segments were proposed[12], that lead to an enormous speedup of the integration calculations. However, this requires neglecting the attenuation within a ray segment. Yet, it is a common approximation for post-classified volume rendering and well justified for small products $\tau(s)d$. The dimensionality of the lookup table can easily be reduced by assuming constant ray segment lengths d. This assumption is correct for orthogonal projections and view-aligned proxy geometry. It is a good approximation for perspective projections and view-aligned proxy geometry, as long as extreme perspectives are avoided. This assumption is correct for perspective projections and shell-based proxy geometry. In the following hardware-accelerated implementation, two-dimensional lookup tables for the pre-integrated ray-segments are employed, thus a constant ray segment length is assumed.

For a hardware implementation of pre-integrated volume rendering, texture coordinates for two adjacent sampling points along rays through the volume must be computed. The following Cg vertex program computes the second texture coordinates for s_b from the texture coordinates given for s_f:

```
vertout main(vertexIn IN,
```

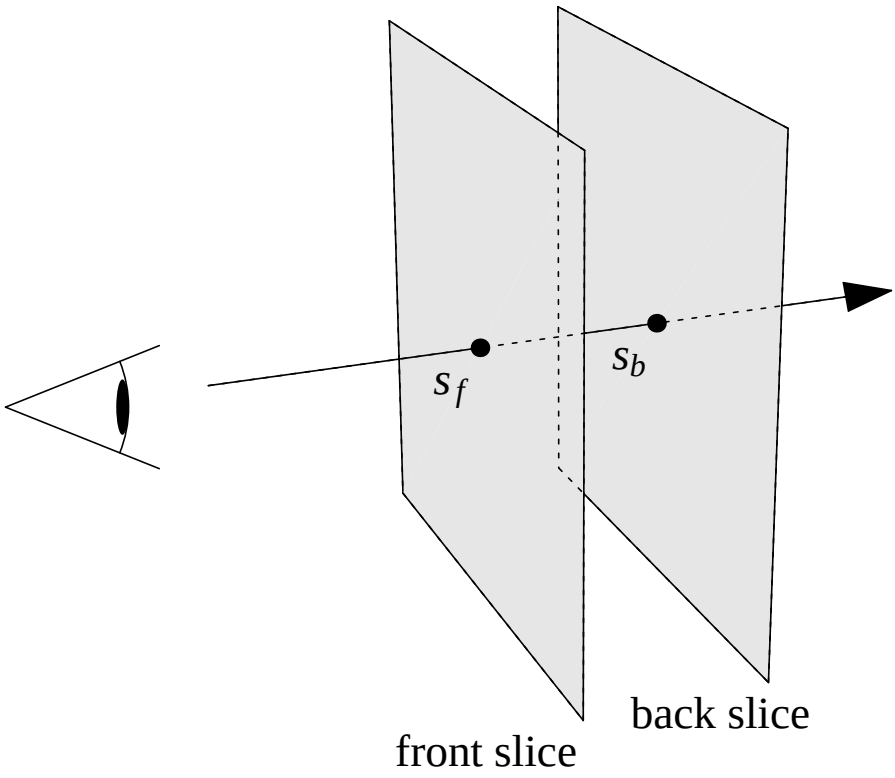


Figure 16.3: A slab of the volume between two slices. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b).

```
            uniform float SliceDistance,
            uniform float4x4 ModelViewProj,
            uniform float4x4 ModelViewI,
            uniform float4x4 TexMatrix)
{
  vertexOut OUT;

  // transform texture coordinate for s_f
  OUT.TCoords0 = mul(TexMatrix, IN.TCoords0);

  // transform view pos vec and view dir to obj space
  float4 vPosition = mul(ModelViewI,
```

```
                        float4(0,0,0,1));

  // compute view direction
  float4 vDir = normalize(mul(ModelViewI, float4(0.f,0.f,-1.f,1.f)));
  // compute vector from eye to vertex
  float3 eyeToVert = normalize( IN.Position.xyz - vPosition.xyz);
  // compute position of s_b
  float4 backVert = {1,1,1,1};
  backVert.xyz = IN.Position.xyz +
                 eyeToVert * (SliceDistance / dot(vDir.xyz,eyeToVert));

  //compute texture coordinates for s_b
  OUT.TCoords1 = mul(TexMatrix, backVert);

  // transform vertex position into homogenous clip-space
  OUT.HPosition = mul(ModelViewProj, IN.Position);

  return OUT;
}
```

In the fragment stage, the texture coordinates for s_f and s_b are used to lookup two adjacent samples along a ray. Those two samples are then used as texture coordinates for a dependent texture lookup into a 2D texture containing the pre-integration table, as demonstrated in the following Cg fragment shader code:

```
struct v2f_simple {
    float3 TexCoord0 : TEXCOORD0;
    float3 TexCoord1 : TEXCOORD1;
};
float4 main(v2f_simple IN, uniform sampler3D Volume,
             uniform sampler2D PreIntegrationTable) : COLOR
{
    fixed4 lookup;
    //sample front scalar
    lookup.x = tex3D(Volume, IN.TexCoord0.xyz).x;
    //sample back scalar
    lookup.y = tex3D(Volume, IN.TexCoord1.xyz).x;
    //lookup and return pre-integrated value
    return tex2D(PreIntegrationTable, lookup.yx);
}
```

A comparison of the results of pre-classification, post-classification and pre-integrated classification is shown in Figure 16.4. Obviously, pre-integration produces the visually most pleasant results. However, this comes at the cost of looking up an additional filtered sample from the volume for each sampling position. This considerably reduces performance due to the fact that memory access is always expensive. However, using pre-integration, a substantially smaller sampling rate is required when rendering volume with high frequency transfer functions. Another advantage is that pre-integration can be performed as a pre-processing step with the full precision of the CPU. This reduces artifacts introduced during blending for a large number of integration steps (see section 18).

To overcome the problem of the additional sample that has to be considered, we need a means of caching the sample from the previous sampling position. The problem can be reduced by computing multiple steps integration at once , i.e. if we compute five integrations at once we need six samples from the volume instead of ten compared to a single integration step. Current graphics hardware allows to perform the complete integration along a ray in a single pass. In this case, pre-integration does not introduce an significant performance loss compared to the standard integration using post-classification.

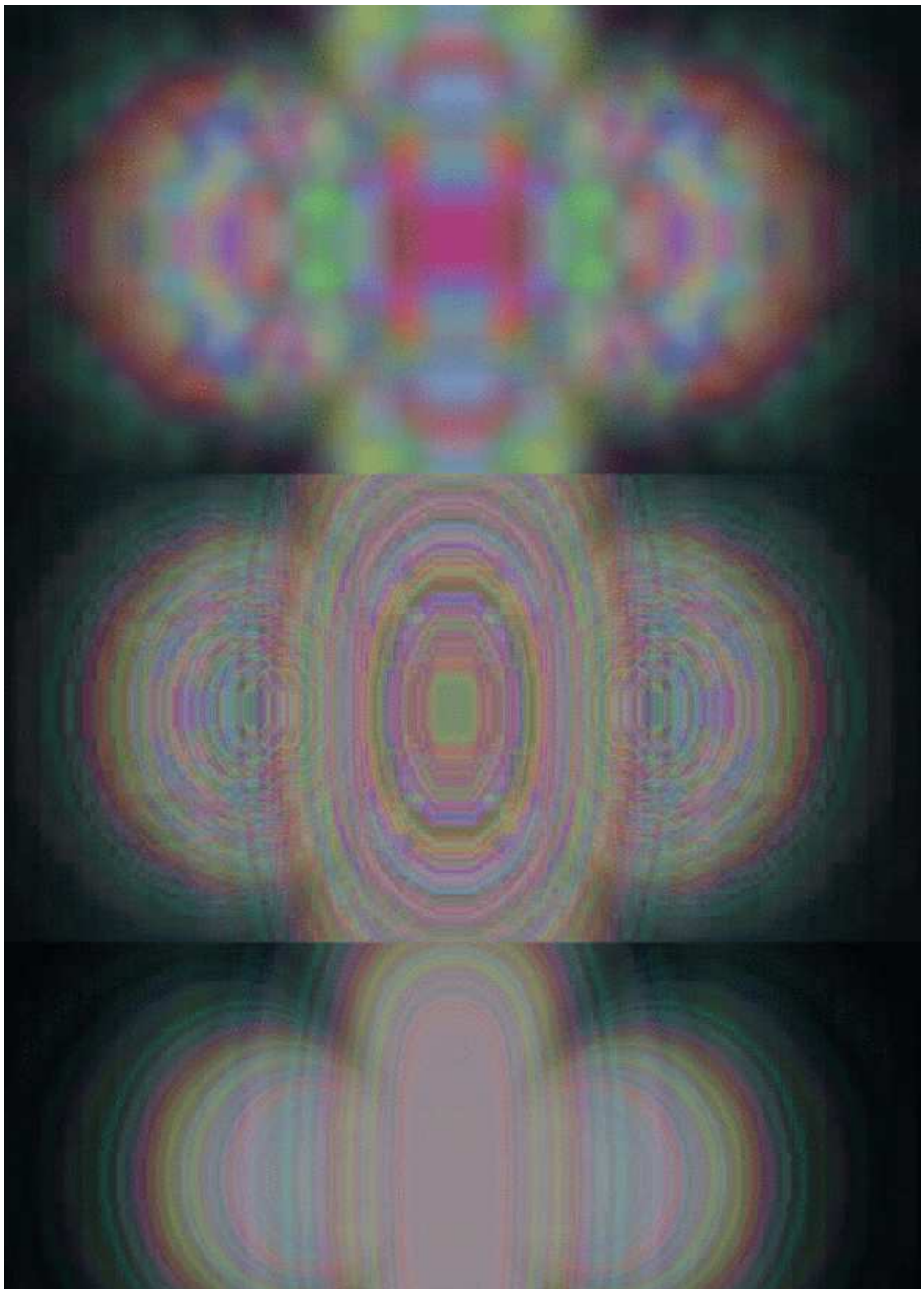


Figure 16.4: Comparison of the results of pre-, post- and pre-integrated classification for a random transfer function. Pre-classification (top) does not reproduce high frequencies of the transfer function. Post-classification reproduces the high frequencies on the slice polygons (middle). Pre-integrated classification (bottom) produces the best visual result due to the reconstruction of high frequencies from the transfer function in the volume.

Shading Artifacts

It is common to interpret a volume as a self-illuminated gas that absorbs light emitted by itself. If external light sources have to be taken into account, a shading stage is inserted into the volume rendering pipeline. Shading can greatly enhance depth perception and manifest small features in the data; however, it is another common source of artifacts (see figure 17.1, left). Shading requires a per-voxel gradient to be computed that is determined directly from the volume data by investigating the neighborhood of the voxel. Although the newest generation of graphics hardware permits calculating of the gradient at each voxel on-the-fly, in the majority of the cases the voxel gradient is pre-computed in a pre-processing step. This is due to the limited number of texture fetches and arithmetic instructions of older graphics hardware in the fragment processing phase of the OpenGL graphics pipeline and as well as of performance considerations. For scalar volume data the gradient vector is defined by the first order derivative of the scalar field $I(x, y, z)$, which is defined as by the partial derivatives of I in the x-, y- and z-direction:

$$\vec{\nabla} I \;=\; (I_x,\, I_y,\, I_z) \;=\; \left(\frac{\partial}{\partial x} I,\, \frac{\partial}{\partial y} I,\, \frac{\partial}{\partial z} I\right). \tag{17.1}$$

The length of this vector defines the local variation of the scalar field and is computed using the following equation:

$$\left\| \vec{\nabla} I \right\| \;=\; \sqrt{{I_x}^2 + {I_y}^2 + {I_z}^2}. \tag{17.2}$$

Gradients are often computed in a pre-processing step. To access those pre-computed gradient during rendering, gradients are usually normalized, quantized to 8-bits and stored in the RGB channels of a separate volume texture. For performance reasons, often the volume data is stored together with the gradients in the alpha channel of that same textures, so that a single texture lookup provides the volume data and gradients at the same time.

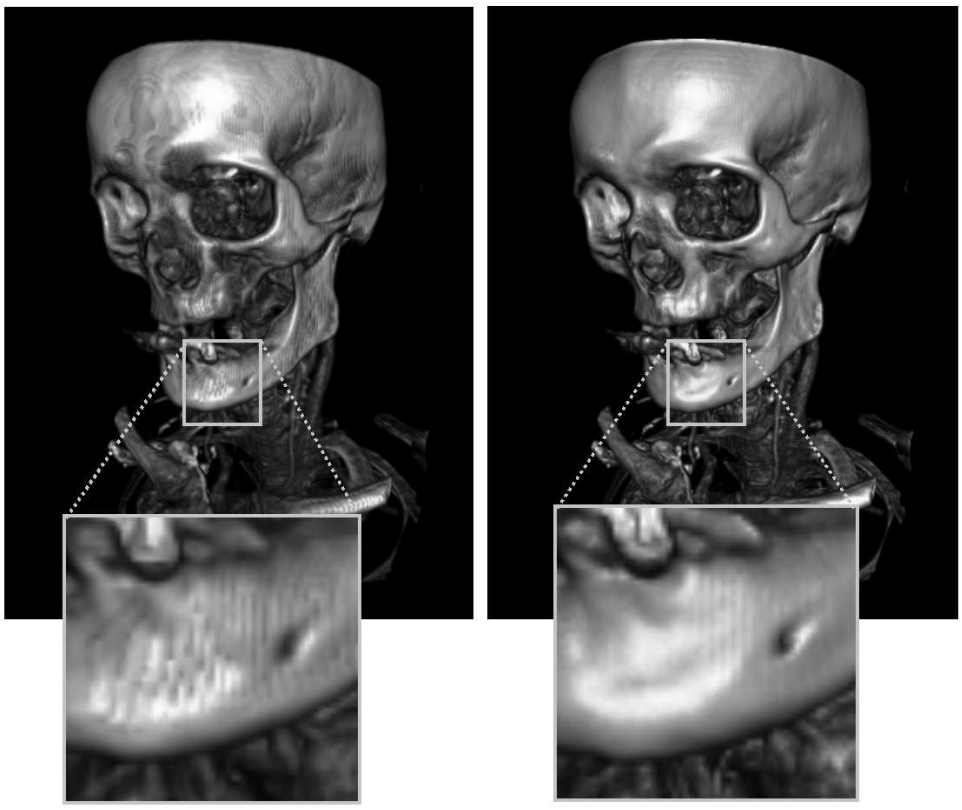


Figure 17.1: Comparison between pre-computed and quantized gradients (left) with on-the-fly gradient computation (right).

Aside from the higher memory requirements for storing pre-computed gradients and the pre-processing time, quantizing gradients to 8 bit precision can cause artifacts in the resulting images, especially if the original volume data is available at a higher precision. Even worse, gradients are interpolated in the filtering step of the volume rendering pipeline. Note, that when interpolating two normalized gradients an unnormalized normal may be generated. Previous graphics hardware did not allow gradients renormalized gradients in the fragment stage. Such unnormalized and quantized gradients cause dark striped artifacts which are visible the left image of figure 17.1.

One possible solution to this problem is to store the pre-computed gradients at higher precision in a 16 bit fixed point or 32 bit floating point 3D texture and apply another normalization in the fragment processing stage on interpolated gradients. Those high-precision texture formats are

available on newer graphics hardware; however, the increased amount of texture memory required to store such high-precision gradients does not permit this solution for high-resolution volumetric data.

A significantly better solution is to compute high-precision gradients on-the-fly. For a central differences gradient we need to fetch the six neighboring voxel values at the sampling position. For this purpose we provide six additional texture coordinates to the fragment program, each shifted by one voxel distance to the right, left, top, bottom, back or front. Using this information, a central differences gradient can be computed per fragment. The resulting gradient is normalized and used for shading computations. The following Cg fragment program looks up a sample along the rays, performs a classification, computes a gradient from additional neighboring samples and finally computes the shading:

```
struct fragIn {
    float4 Hposition : POSITION;
    float3 TexCoord0 : TEXCOORD0;
    float3 TexCoord1 : TEXCOORD1;
    float3 TexCoord2 : TEXCOORD2;
    float3 TexCoord3 : TEXCOORD3;
    float3 TexCoord4 : TEXCOORD4;
    float3 TexCoord5 : TEXCOORD5;
    float3 TexCoord6 : TEXCOORD6;
    float3 TexCoord7 : TEXCOORD7;
    float3 VDir      : COLOR0;
};

float4 main(fragIn IN, uniform sampler3D Volume,
                       uniform sampler2D TransferFunction,
                       uniform half3 lightdir,
                       uniform half3 halfway,
                       uniform fixed ambientParam,
                       uniform fixed diffuseParam,
                       uniform fixed shininessParam,
                       uniform fixed specularParam) : COLOR
{
   fixed4 center;
   // fetch scalar value at center
   center.ar = (fixed)tex3D(Volume, IN.TexCoord0.xyz).x;
   // classification
   fixed4 classification = (fixed4)tex2D(TransferFunction, center.ar);
```

```
    // samples for forward differences
    half3 normal;
    half3 sample1;
    sample1.x = (half)tex3D(Volume, IN.TexCoord2).x;
    sample1.y = (half)tex3D(Volume, IN.TexCoord4).x;
    sample1.z = (half)tex3D(Volume, IN.TexCoord6).x;

    // additional samples for central differences
    half3 sample2;
    sample2.x = (half)tex3D(Volume, IN.TexCoord3).x;
    sample2.y = (half)tex3D(Volume, IN.TexCoord5).x;
    sample2.z = (half)tex3D(Volume, IN.TexCoord7).x;

    // compute central differences gradient
    normal = normalize(sample2.xyz  sample1.xyz);
    // compute diffuse lighting component
    fixed diffuse = abs(dot(lightdir, normal.xyz));

    // compute specular lighting component
    fixed specular = pow(dot(halfway, normal.xyz),
                         shininessParam);

    // compute output color
    OUT.rgb =
        ambientParam * classification.rgb
      + diffuseParam * diffuse * classification.rgb
      + specularParam * specular;

    // use alpha from classification as output alpha
    OUT.a = classification.a;

    return OUT;
}
```

The resulting quality of on-the-fly gradient computation computation is shown in the image on the right of figure 17.1. The enhanced better quality compared to pre-computed gradients is due to the fact that we used filtered scalar values to compute the gradients compared to filtered gradients. This provide much nicer and smoother surface shading, which even allows reflective surfaces to look smooth (see figure 17.2). Besides

this advantage, no additional memory is wasted to store pre-computed gradients. This is especially important for high-resolution volume data that already consumes a huge amount of texture memory or must be bricked to be rendered (see chapter 17.2). This approach allows even higher quality gradients at the cost of additional texture fetches, e.g. sobel gradients.

However, the improved quality comes at the cost of additional memory reads which considerably decrease performance due to memory latency. It is important that those expensive gradient computations are only performed when necessary. Several techniques, like space-leaping, early-ray termination and deferred shading (which are discussed in chapter **??**) will allow real-time performance, even when computing gradients on-the-fly.

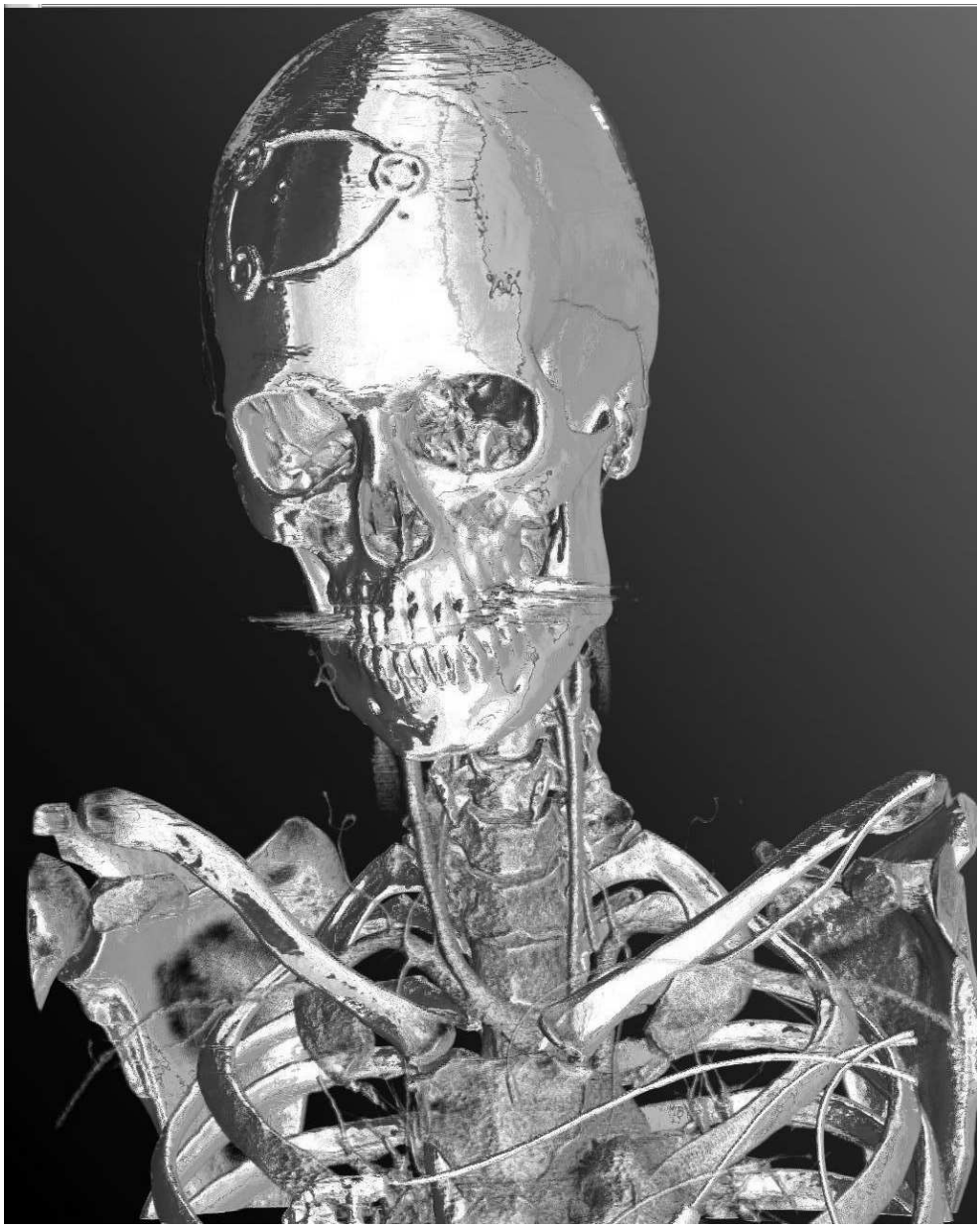


Figure 17.2: Reflective environment mapping computed with on-the-fly gradient computation. Note the smoothness of the surface.

Blending Artifacts

The final step of the rendering pipeline involves combining color values generated by previous stages of the pipeline with colors written into the frame buffer during integration. As discussed in previous chapters, this is achieved by blending RGB colors with their alpha values into the frame buffer. A large number of samples along the rays through the volume are blended into the frame buffer. Usually, color values in this stage are quantized to 8-bit precision. Therefore, quantization errors are accumulated very quickly when blending a large number of quantized colors into the frame buffer, especially when low alpha values are used. This is due to the fact, that the relative error for small 8 bit fixed point quantization is much greater than for large numbers. Figure 18.1 demonstrates blending artifacts for a radial distance volume renderer with low alpha values. In contrast to fixed point formats, floating point number allow higher precision for small numbers than for large numbers.

Floating point precision was introduced recently into the pixel pipeline of graphics hardware. The first generation of graphics hardware with floating-point support throughout the pipeline does not support blending with floating point precision. Therefore, blending must

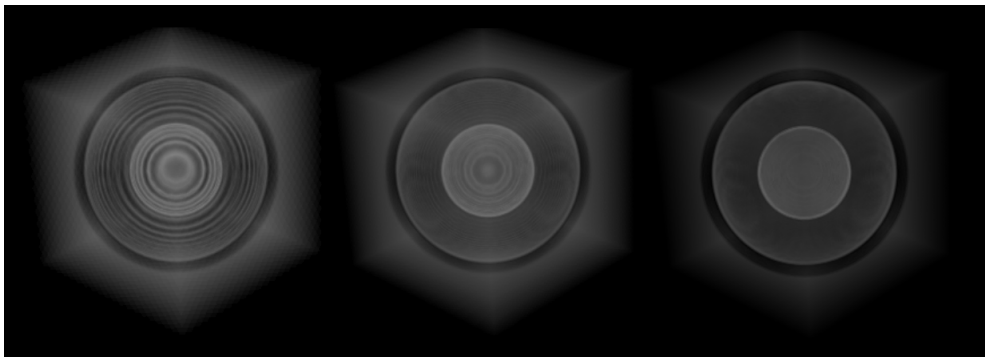


Figure 18.1: Comparison between 8-bit (left), 16-bit (middle) and 32-bit blending (right).

be implemented in a fragment shader. As the on-screen frame buffer still only supports 8-bit precision, off-screen pbuffers are required for high-precision blending. The fragment shader has to read the current contents of the floating-point pbuffer, blend the incoming color with the frame buffer and write the result back into the pbuffer. To bind a pbuffer as an input image to a fragment program, the pbuffer is defined as a so-called rendertexture; i.e., a texture that can be rendered to. To read the current contents of the pbuffer at the rasterization position, the window position (WPOS) that is available in fragment programs can directly be used as a texture coordinate for a rectangle texture fetch. Figure 18.2 illustrates the approach while the following Cg source code demonstrates the approach with a simple post-classification fragment program with over-operator compositing:

```
struct v2f_simple {
    float3 TexCoord0 : TEXCOORD0;
    float2 Position : WPOS;
};

float4 main(v2f_simple IN,
            uniform sampler3D Volume,
            uniform sampler1D TransferFunction,
            uniform samplerRECT RenderTex,
            ) : COLOR
{
    // get volume sample
    half4 sample = x4tex3D(Volume, IN.TexCoord0);
    // perform classification to get source color
    float4 src = tex1D(TransferFunction, sample.r);
    // get destination color
    float4 dest = texRECT(RenderTex, IN.Position);
    // blend
    return (src.rgba * src.aaaa) +
           (float4(1.0, 1.0, 1.0, 1.0)-src.aaaa) * dest.rgba;
}
```

It should be noted, that the specification of the render texture extension explicitly states that the result is undefined when rendering to a texture and reading from the texture at the same time. However, current graphics hardware allows this operation and produces correct results when reading from the same position that the new color value is written to.

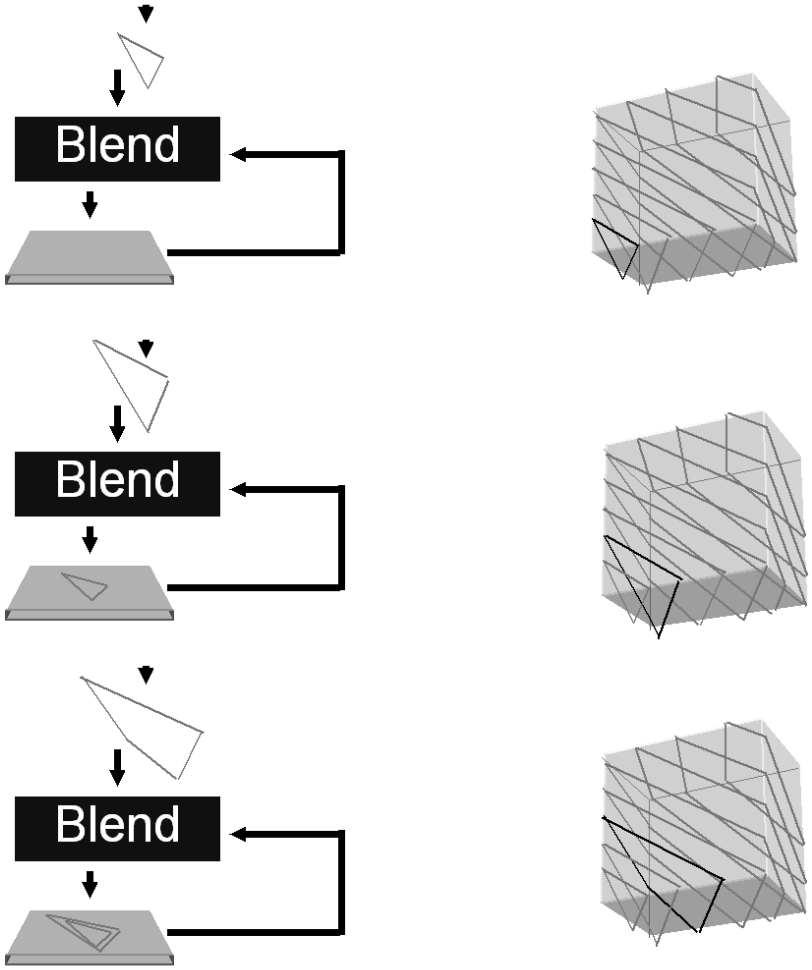


Figure 18.2: Programmable blending with a pbuffer as input texture and render target at the same time.

If you feel uncomfortable with this solution, you can use ping-pong blending as an alternative (see figure 18.3). Ping-pong blending alternates the rendering target to prevent read-write race conditions. To avoid context switching overhead when changing rendering targets a double-buffered pbuffer can be employed, whose back and front buffer then are used for ping-pong blending.

As demonstrated in the middle image of figure 18.1 even 16-bit floating point precision might not be sufficient to accurately integrate colors with low-alpha values into the frame buffer. However, as memory access does not come for free, performance decreases as a function of precision. Therefore, it is necessary to find a good balance between quality and performance. For most applications and transfer functions 16-bit floating point blending should produce acceptable results.

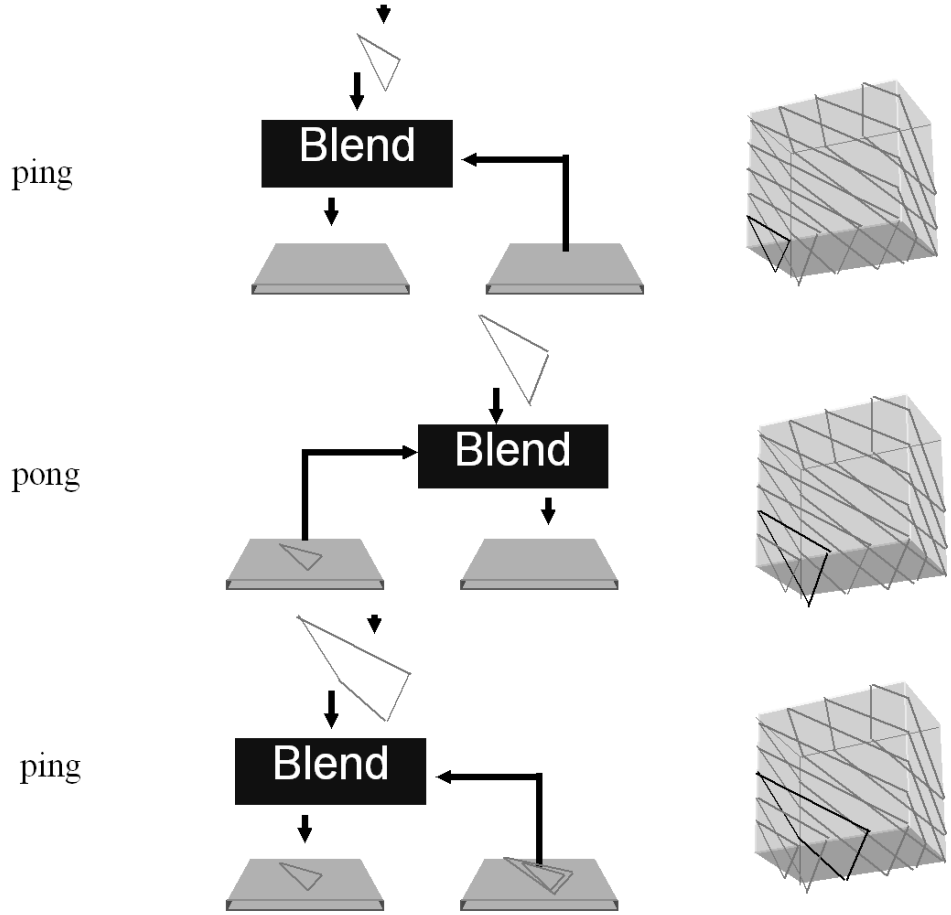


Figure 18.3: Programmable blending with a pbuffer as input texture and render target at the same time.

Summary

Artifacts are introduced in various stages of the volume rendering process. However, the high precision texture formats and computations in combination with the advanced programmability of today's GPUs allow artifacrs to be suppressed or even allow to remove them almost completely. All of the techniques presented to prevent artifacts can be implemented quite efficiently using programmable graphics hardware to achieve real-time performance. However, those optimization do not come for free - to maximize performance trade-offs between quality and performance often have to be made.

The human visual system is mess sensitive to artifacts in moving pictures that static images. This phenomena is evident by comparing a still image with non-static images from a TV screen. Therefore, for some applications it is acceptable to trade off quality for performance while the volumetric object is moving, and use higher quality when the object becomes stationary.

Course Notes T7

Real-Time Volume Graphics

Literature

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VRVis Research Center, Vienna, Austria

Joe M. Kniss
SCI Institute, University of Utah, USA

Christof Rezk Salama
University of Siegen, Germany

Bibliography

[1] Andreas H. König and Eduard M. Gröller. Mastering transfer function specification by using volumepro technology. Technical Report TR-186-2-00-07, Vienna University of Technology, March 2000.

[2] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. The Contour Spectrum. In *Proceedings IEEE Visualization 1997*, pages 167–173, 1997.

[3] Uwe Behrens and Ralf Ratering. Adding Shadows to a Texture-Based Volume Renderer. In *1998 Volume Visualization Symposium*, pages 39–46, 1998.

[4] J. Blinn. Models of Light Reflection for Computer Synthesized Pictures . *Computer Graphics*, 11(2):192–198, 1977.

[5] J. Blinn and M. Newell. Texture and Reflection in Computer Generated Images. *Communcations of the ACM*, 19(10):362–367, 1976.

[6] J. F. Blinn. Jim blinn's corner: Image compositing–theory. *IEEE Computer Graphics and Applications*, 14(5), 1994.

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

[8] Yoshinori Dobashi, Kazufumi Kanede, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Hishita. A Simple, Efficient Method for Realistic Animation of Clouds. In *Siggraph 2000*, pages 19–28, 2000.

[9] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proc. of SIGGRAPH '88*, pages 65–74, 1988.

[10] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (Second Edition)*. Wiley-Interscience, 2001.

[11] D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, July 1998.

[12] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proc. Graphics Hardware*, 2001.

[13] T. J. Farrell, M. S. Patterson, and B. C. Wilson. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the non-invasive determination of tissue optical properties in vivo. *Medical Physics*, 19:879–888, 1992.

[14] R. Fernando and M. Kilgard. *The Cg Tutorial - The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley, 2003.

[15] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principle And Practice*. Addison-Weseley, 1993.

[16] N. Greene. Environment Mapping and Other Applications of World Projection. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[17] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Proc. of Vision, Modeling, and Visualization 2001*, pages 105–112, 2001.

[18] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Vision, Modeling, and Visualization 2002*, pages 155–162, 2002.

[19] M.J. Harris and A. Lastra. Real-time cloud rendering. In *Proc. of Eurographics 2001*, pages 76–84, 2001.

[20] Taosong He, Lichan Hong, Arie Kaufman, and Hanspeter Pfister. Generation of Transfer Functions with Stochastic Search Techniques. In *Proceedings IEEE Visualization 1996*, pages 227–234, 1996.

[21] James T. Kajiya and Brian P. Von Herzen. Ray Tracing Volume Densities. In *ACM Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 165–173, July 1984.

[22] A. Kaufman. Voxels as a Computational Representation of Geometry. In *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, 1994.

[23] Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Moller. Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications. In *Proceedings Visualization 2003*, pages 513–520. IEEE, October 2003.

[24] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multi-Dimensional Transfer Functions for Interactive Volume Rendering. *TVCG*, pages 270–285, July-September 2002.

[25] Joe Kniss, Simon Premoze, Milan Ikits, Aaron Lefohn, Charles Hansen, and Emil Praun. Gaussian Transfer Functions for Multi-Field Volume Visualization. In *Proceedings IEEE Visualization 2003*, pages 497–504, 2003.

[26] Philip Lacroute and Marc Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. In *ACM Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 451–458, July 1994.

[27] E. LaMar, B. Hamann, and K.I. Joy. *Multiresolution Techniques for Interactive Texture-Based Volume Visualization.* IEEE Visualization '99 Proceedings, pp. 355–361, 1999.

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

[29] J. Marks, B. Andalman, P.A. Beardsley, and H. Pfister et al. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In *ACM Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 389–400, August 1997.

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

[31] M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D-texture Based Volume Rendering Using OpenGL and Extensions. In *Proc. IEEE Visualization*, 1999.

[32] H. Nyquist. Certain topics in telegraph transmission theory. In *Trans. AIEE, vol. 47*, pages 617–644, 1928.

[33] Bui Tuong Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[34] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[35] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proc. of IEEE Visualization 2000*, pages 109–116, 2000.

[36] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. http://www.opengl.org.

[37] C. E. Shannon. Communication in the presence of noise. In *Proc. Institute of Radio Engineers, vol. 37, no.1*, pages 10–21, 1949.

[38] Lihong V. Wang. Rapid modelling of diffuse reflectance of light in turbid slabs. *J. Opt. Soc. Am. A*, 15(4):936–944, 1998.

[39] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH '98*, pages 169–178, 1998.

[40] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-weighted color interpolation for volume sampling. In *Proc. of IEEE Symposium on Volume Visualization*, pages 135–142, 1998.

EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces

Markus Hadwiger Christian Sigg* Henning Scharsach Katja Bühler Markus Gross*

VRVis Research Center * ETH Zürich

Figure 1: *We render high-quality implicit surfaces on regular grids, e.g., distance fields or medical CT scans, in real-time without pre-computing additional per-voxel information. Gradients with C^1 continuity, second-order derivatives, and surface curvature are computed exactly for each output pixel using tri-cubic filtering. Applications include surface interrogation and visualizing levelset computations by color mapping curvature measures (center), and ridge and valley lines (left and right).*

Abstract

This paper presents a real-time rendering pipeline for implicit surfaces defined by a regular volumetric grid of samples. We use a ray-casting approach on current graphics hardware to perform a direct rendering of the isosurface. A two-level hierarchical representation of the regular grid is employed to allow object-order and image-order empty space skipping and circumvent memory limitations of graphics hardware. Adaptive sampling and iterative refinement lead to high-quality ray/surface intersections. All shading operations are deferred to image space, making their computational effort independent of the size of the input data. A continuous third-order reconstruction filter allows on-the-fly evaluation of smooth normals and extrinsic curvatures at any point on the surface without interpolating data computed at grid points. With these local shape descriptors, it is possible to perform advanced shading using high-quality lighting and non-photorealistic effects in real-time.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism Color, shading, shadowing, and texture

1. Introduction

Rendering isosurfaces represented implicitly by a volume of function samples is an important task in visualization, for example in medical applications, where volume data are naturally acquired directly, e.g., through CT or MRI scans, as well as a wide spectrum of other graphics disciplines including modeling and animation [MBWB02], and levelset simulation [LKHW03]. More general, implicit models are often specified and modified on volumetric grids such as regularly sampled distance fields, e.g., in levelset methods. Implicit representations naturally represent shapes of complex and changing topology. However, a major limitation of implicits is that the isosurface has to be extracted from the underlying volumetric representation for display. High-quality rendering at interactive speeds is a major bottleneck, particularly when the isosurface changes over time. When an implicit is represented by a discrete set of samples, rendering involves reconstruction of the data and the reconstruction filter is of crucial importance for image quality, especially for gradient reconstruction [MMK*98].

We present a real-time rendering pipeline for isosurfaces of dense volumetric grids of function samples that achieves

both high rendering quality and performance on current consumer graphics hardware (GPUs). Our algorithms are generally independent of specific hardware but we assume support for volumetric textures, render-to-texture and looping in fragment programs (e.g., ShaderModel 3.0). We address several shortcomings in existing GPU isosurface rendering approaches, particularly the lack and inefficiency of advanced shading, and texture memory usage. Modern GPUs are able to perform standard ray-casting of small regularly sampled data sets [KW03]. However, advanced shading, e.g., curvature-based transfer functions [HKG00, KWTM03], is still the domain of off-line rendering. The amount of texture memory limits data sizes significantly. This problem is aggravated by the demand of high-quality rendering for voxel data of 16-bit precision or more and lossless compression.

As a central part of our rendering pipeline, we support tri-cubic filtering throughout. Cubic filters allow for precise evaluations of differential properties of the isosurface, such as the normal and curvature, which both play a vital role in visualization, modeling, and simulation. These shape descriptors can be used for various advanced shading effects such as accessibility shading [Mil94], visualizing implicit surface curvature [KWTM03], and flow along curvature directions [vW03]. See Figure 1 for examples. In contrast to direct volume rendering, for isosurfaces only one sample position contributes to the color of a single pixel. Therefore, our method employs a ray-casting pass only for determining ray/surface intersections, and defers the computation of surface shape descriptors and shading to image space, where they are evaluated once per visible surface sample only. On-demand caching techniques are employed to dynamically download bricks of data only when they contain parts of the isosurface. Because only a small fraction of the grid samples contributes to the definition of an isosurface, this leads to significant reduction of texture memory usage without the need for lossy compression. See Figure 2 for an example. In summary, the combination of real-time performance and high quality yields a general-purpose rendering front-end for many powerful applications of implicit surfaces. The major contribution is a system that integrates the following:

- Tri-cubic filtering and high-quality shading with non-photorealistic effects using on-the-fly computation of smooth second-order geometric surface properties.
- Object space culling and empty space skipping without any per-sample cost during ray-casting.
- Precise ray/surface intersections without global oversampling, by combining adaptive resampling and iterative refinement of intersections with image order complexity.
- A very simple 3D brick cache alleviates GPU memory limitations significantly.
- Principal surface curvatures are computed in a simpler way than in previous approaches [KWTM03].

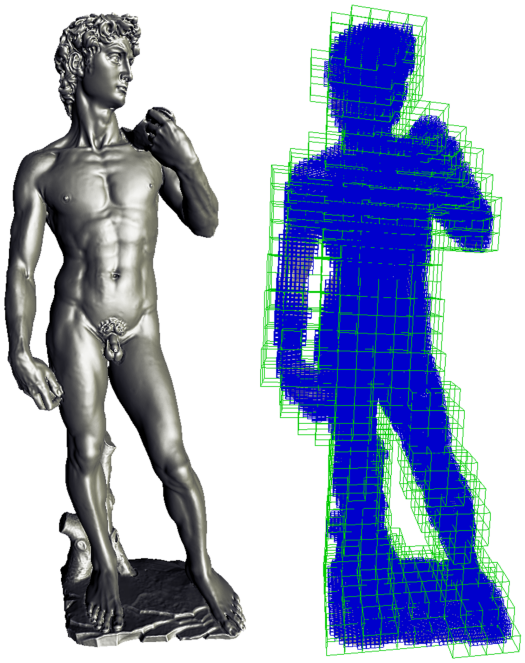


Figure 2: *Michelangelo's David extracted and shaded with tri-cubic filtering as isosurface of a 576x352x1536 16-bit distance field at 10 fps. The distance field is subdivided into two levels: a fine level for empty space skipping during ray-casting (blue) and a coarse level for texture caching (green).*

Previous Work

Our work is related to a large amount of previous research on volume rendering and rendering isosurfaces of volumetric data such as CT or MRI scans, as well as the area of implicit surfaces in general, especially when an implicit is represented by a grid of function samples, e.g., in levelset methods [MBWB02]. Although isosurfaces are often converted to triangle meshes for rendering [LC87], this produces very complex models and interactive changes of the isovalue or the volume itself are difficult to deal with. The two major approaches for rendering isosurfaces directly are ray-casting [Bar86, Lev88], and sampling ray-surface intersections on graphics hardware via slicing [WE98]. Although implicits are well-suited for finding guaranteed ray-surface intersections [KB89], precise computations and high-quality reconstruction are expensive. Hence interactive rates with high-quality or analytic ray-surface intersections and gradients have only been achieved by implementations using multiple CPUs [PSL*98, PPL*99] or clusters [DPH*03]. Different trade-offs have been presented [NMHW02, MKW*04]. The parallel architecture of GPUs has also been used extensively for interactive volume rendering, usually via slicing [WE98, EKE01]. In addition to hybrid CPU/GPU ray-casting [WS01], ray-casting on GPUs has been shown for small data sets [KW03, Gre04]. Adaptive sampling rates can be achieved by using pre-computed importance volumes [RGW*03]. Aliasing artifacts due to undersampling during slicing can be reduced by pre-integration [EKE01],

which also yields sharp isosurface boundaries, but assumes piecewise linear data variation along all viewing rays instead of tri-linear or higher-order reconstruction. All other previous interactive approaches for rendering isosurfaces of volume data are restricted to tri-linear data interpolation, and usually interpolate gradients pre-computed at grid points. If the volume data have not been scanned directly, signed distance fields are a natural choice as input for our rendering pipeline [WSE99]. Levelset methods change the distance fields dynamically and have many powerful applications such as surface editing and processing operators [MBWB02], and surface deformations [TO99]. Implicits are also well-suited for CSG modeling. In addition to reconstructing an isosurface, we are computing implicit surface curvature [KWTM03]. The space of principal curvature magnitudes is intuitive for shape depiction [HKG00], and can be used for non-photorealistic volume rendering [RE01] such as ridge and valley lines [IFP95]. Curvature directions can be visualized effectively by advecting dense noise textures [vW03], which we do entirely in image space [LJH03] on a per-pixel basis. The texture memory limitation for large volumes has been tackled by various means of lossy compression [GWGS02, SW03], which are not well suited for high-quality rendering. Texture packing has been used for static lossless compression [KE02], improved rendering performance [LMK03], and sparse levelset computations [LKHW03]. Octrees have also been used [LHJ99, WWH*00]. Our texture caching approach combines adaptive texture look-ups during rendering [KE02] with dynamically updated packed data [LKHW03].

2. Pipeline Overview

This section gives a high-level overview of our rendering pipeline, which is illustrated in Figure 3. The basic input is a regularly sampled scalar volume. The first stage (top row of Figure 3) performs ray-casting through the volume in order to obtain a floating point image of ray/isosurface intersection positions in volume coordinates, which drives the following stages in image space (lower two rows of Figure 3). The ray-casting stage (Section 3) is the only part of the pipeline that has object space complexity. All other computations (such as computing derivatives; Section 4.1) and shading operations (such as color-coding curvature; Section 4.2) are deferred to image space and thus have image space complexity [ST90].

The volume is subdivided into two regular grid levels: a fine level to facilitate empty space skipping (Section 3.1), and a coarse level to circumvent memory limitations of graphics hardware (Section 3.4). We call the elements of the fine subdivision level *blocks*, and those of the coarse level *bricks*. For each block we track min-max values of a set of voxels. Rays are started on block bounding faces and cast into the volume using adaptive sampling (Section 3.2). The last operation of the ray-casting stage iteratively refines isosurface hit-points (Section 3.3). This is done with a constant number of steps of image space complexity and is thus

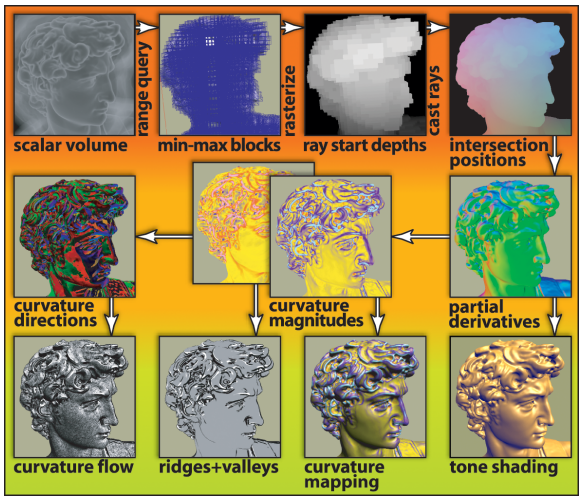


Figure 3: *Overview of our rendering pipeline. The top row operates with object space complexity until the refinement of ray/isosurface intersection positions. The middle row stages compute differential surface properties with image space complexity, and the bottom row stages perform deferred shading in image space.*

the transition from object to image space. The result of hit-point refinement is an image of high-quality ray/isosurface intersection positions. If the whole volume does not fit into graphics memory, rays are cast through a dynamic cache texture storing active bricks (Section 3.4). The cache is updated on-the-fly according to the current isovalue. An additional low-resolution texture references the positions of bricks of the volume in the cache.

The image space pipeline stages generate a series of images of differential isosurface properties, which are then used in a final shading pass to generate an output image using a variety of shading styles. Surface properties are computed at the exact positions of ray/isosurface intersections specified by the intersection image. Computing the first and second partial derivatives of the scalar volume yields floating point images for the components of the gradient and the Hessian matrix (Section 4.1). These derivatives are then used to compute curvature measures, which are likewise written into floating point images. The output image is generated in a final image space shading pass with a variety of effects that build on the shape descriptors computed before. The gradient image can be used for all shading models that require a surface normal, such as standard Blinn-Phong or tone shading. Curvature measures can be mapped to colors via 1D or 2D transfer functions, which is well-suited for shape depiction. For example drawing ridge and valley lines without generating actual line primitives. Pixels that correspond to ridge or valley areas are identified on a per-pixel basis via a curvature transfer function. Curvature directions are also effective shape cues, and we illustrate the curvature field on the isosurface with image space flow advection.

3. Ray-Casting

The basic idea of GPU-based ray-casting is to store the entire volume in a single 3D texture, and drive a fragment program that casts rays into the volume. Each pixel corresponds to a single ray $\mathbf{p}(t,x,y) = \mathbf{c} + t\ \mathbf{d}(x,y)$ in volume coordinates. Here, the normalized direction vector $\mathbf{d}(x,y)$ can be computed from the camera position $\mathbf{c}$ and the screen space coordinates (x,y) of the pixel. The range of depths $[t_{start}(x,y), t_{exit}(x,y)]$ which has to be searched for an isosurface intersection is computed per frame during initialization. In the simplest case, t_{start} is obtained by rasterizing the front faces of the volume bounding box with the corresponding distance to the camera. Rendering the back faces of the bounding box yields the depths t_{exit} of each ray exiting the volume.

In contrast to earlier approaches, we are using a single rendering pass and looping in the fragment shader for casting through the volume in front-to-back order instead of multiple passes [KW03], and employ object-order in addition to image-order empty space skipping. Most importantly, we overcome the following limitations:

- Empty space skipping overhead is reduced by using a two-level approach. Most empty space is skipped with no cost using modified ray segments $[t_{start}(x,y), t_{exit}(x,y)]$. Only for a small number of samples empty space has to be skipped on a sample-by-sample basis, which is accelerated via an adaptive sampling strategy.
- The quality of ray/isosurface intersection positions is refined by an iterative bisection procedure, which yields quality identical to much higher constant sampling rates [KW03] except at silhouette edges. A simple adaptive approach improves the quality of silhouette edges, without significant book-keeping overhead [RGW*03].
- The entire volume is not required to fit in GPU memory. Instead of casting through the original volume, we sample a brick cache texture storing only bricks intersected by the isosurface. Fast culling and LRU cache brick replacement allow changing the isovalue in real-time.

3.1. Empty Space Skipping

In order to facilitate object-order empty space skipping without per-sample overhead, we maintain min-max values of a regular subdivision of the volume into small blocks, e.g., with 4^3 or 8^3 voxels per block. These blocks do not actually re-arrange the volume. For each block, a min-max value is simply stored in an additional structure for culling. If the whole volume does not fit in GPU memory, however, a second level of coarser bricks is maintained, which is described in Section 3.4. Whenever the isovalue changes, blocks are culled against it using their min-max information and a range query [CSS98], which determines their active status. See Figure 4. The view-independent geometry of active block bounding faces that are adjacent to inactive blocks is kept in GPU memory for fast rendering.

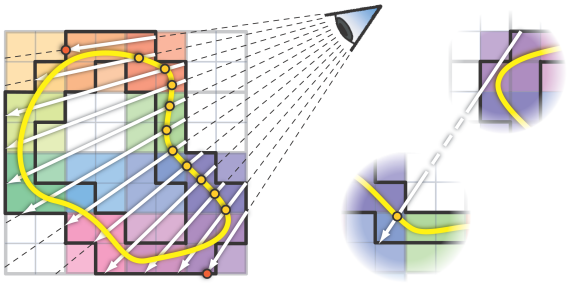


Figure 4: *Ray-casting with object-order empty space skipping. The bounding geometry (black) between active and inactive blocks that determines start and exit depths for the intersection search along rays (white) encloses the isosurface (yellow). Colored bricks of 2x2 blocks reference bricks in the cache texture (Figure 6). White bricks are not in the cache. Actual ray termination points are shown in yellow and red, respectively.*

In order to obtain ray start depths $t_{start}(x,y)$, the front faces of the block bounding geometry are rendered with their corresponding distance to the camera. The front-most points of ray intersections are retained by enabling a corresponding depth test (e.g., `GL_LESS`). For obtaining ray exit depths $t_{exit}(x,y)$ we rasterize the back faces with an inverted depth test that keeps only the farthest points (e.g., `GL_GREATER`). Figure 4 shows that this approach does not exclude inactive blocks from the search range if they are enclosed by active blocks with respect to the current viewing direction. The corresponding samples are skipped on a per-sample basis early in the ray-casting loop. However, most rays hit the isosurface soon after being started and are terminated quickly (yellow points in Figure 4, left). Only a small number of rays on the outer side of the isosurface silhouette are traced for a larger distance until they hit the exit position of the block bounding geometry (red points in Figure 4, left). The right side of Figure 4 illustrates the worst case scenario, where rays are started close to the view point, miss the corresponding part of the isosurface, and sample inactive blocks with image-order empty space skipping until they enter another part of the isosurface bounding geometry and are terminated or exit without any intersection. In order to minimize the performance impact when the distance from ray start to exit or termination is large, we use an adaptive strategy for adjusting the distance between successive samples along a ray.

3.2. Adaptive Sampling

In order to find the position of intersection for each ray, the scalar function is reconstructed at discrete sampling positions $\mathbf{p}_i(x,y) = \mathbf{c} + t_i\mathbf{d}(x,y)$ for increasing values of t_i in $[t_{start}, t_{exit}]$. The intersection is detected when the first sample lies behind the isosurface, e.g., when the sample value is smaller than the isovalue. Note that in general the exact intersection occurs somewhere between two successive samples. Due to this discrete sampling, it is possible that an intersec-

tion is missed entirely when the segment between two successive samples crosses the isosurface twice. This is mainly a problem for rays near the silhouette. Guaranteed intersections even for thin sheets are possible if the gradient length is bounded by some value L [KB89]. Note that for distance fields, L is equal to 1. For some sample value f, it is known that the intersection at isovalue ρ cannot occur for any point closer than $h = |f - \rho|/L$. Yet, h can become arbitrarily small near the isosurface, which would lead to an infinite number of samples for guaranteed intersections.

We use adaptive sampling to improve intersection detection. The actual intersection position of an intersection that has been detected is then further refined using the approach described in Section 3.3. We have found that completely adaptive sampling rates are not well suited for implementations on graphics hardware. These architectures use multiple pipelines where small tiles of neighboring pixels are scan-converted in parallel using the same texture cache. With completely adaptive sampling rate, the sampling positions of neighboring pixels diverge during parallel execution, leading to under-utilization of the cache. Therefore, we use only two different discrete sampling rates. The *base sampling rate* r_0 is specified directly by the user where 1.0 corresponds to a single voxel. It is the main tradeoff between speed and minimal sheet thickness with guaranteed intersections. In order to improve the quality of silhouettes (see Figure 5), we use a second *maximum sampling rate* r_1 as a constant multiple of r_0: $r_1 = nr_0$. We are currently using $n = 8$ in our system. However, we are not detecting silhouettes explicitly at this stage, because it would be too costly. Instead, we automatically increase the sampling rate from r_0 to r_1 when the current sample's value is closer to the isovalue ρ by a small threshold δ. In our current implementation, δ is set by the user as a quality parameter, which is especially easy for distance fields where the gradient magnitude is 1.0 everywhere. In this case, a constant δ can be used for all data sets, whereas for CT scans it has to be set according to the data.

3.3. Intersection Refinement

Once a ray segment containing an intersection has been detected, the next stage determines an accurate intersection position using an iterative bisection procedure. In one iteration, we first compute an approximate intersection position assuming a linear field within the segment. Given the sample values f at positions $\mathbf{x}$ for the near and far ends of the segment, the new sample position is

$$\mathbf{x}_{new} = (\mathbf{x}_{far} - \mathbf{x}_{near}) \frac{\rho - f_{near}}{f_{far} - f_{near}} + \mathbf{x}_{near} \qquad (1)$$

Then the value f_{new} is fetched at this point and compared to the isovalue ρ. Depending on the result, we update the ray segment with either the front or the back sub-segment. If the new point lies in front of the isosurface (e.g. $f_{new} > \rho$), we set $\mathbf{x}_{near}$ to $\mathbf{x}_{new}$, otherwise we set $\mathbf{x}_{far}$ to $\mathbf{x}_{new}$ and repeat. We have found empirically that a fixed number of four iteration steps is enough for high-quality intersection positions.

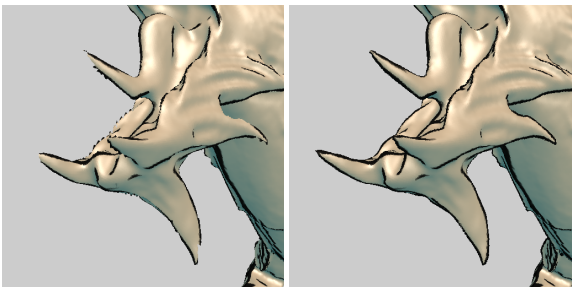


Figure 5: *The left image illustrates a small detail of the asian dragon model with a sampling rate of 0.5. On the right, adaptive sampling increases the sampling rate to 4.0 close to the isosurface. Note that except at the silhouettes there is no visible difference due to iterative refinement of intersections.*

3.4. Brick Caching

For any possible isovalue, many of the blocks described in Section 3.1 do not contain any part of the isosurface. In addition to improving rendering performance by skipping empty blocks, this fact can also be used for reducing the effective memory footprint of relevant parts of the volume significantly. Whenever the isovalue changes, the corresponding range query also determines the active status of bricks of coarser resolution, e.g., 32^3 voxels. The colored squares in Figure 4 depict these bricks with a size of 2*x*2 blocks per brick for illustration purposes. In contrast to blocks, bricks re-arrange the volume and include neighbor samples to allow filtering without complicated look-ups at the boundaries, i.e., a brick of resolution n^3 is stored with size $(n+1)^3$ [KE02]. This overhead is inversely proportional to the brick size, which is the reason for using two levels of subdivision. Small blocks fit the isosurface tightly for empty space skipping and larger bricks avoid excessive storage overhead for memory management.

In order to decouple the volume size from restrictions imposed by GPUs on volume resolution (e.g., 512^3 on NVIDIA GeForce 6) and available video memory (e.g., 256MB), we can perform ray-casting directly on a re-

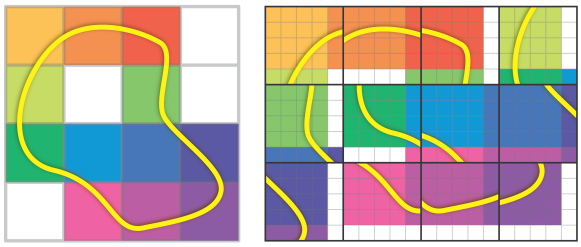


Figure 6: *A low-resolution brick reference texture (left) stores references from volume coordinates to texture cache bricks (right). The reference texture is sampled in the fragment shader to transform volume coordinates into brick cache texture coordinates. White bricks denote* null *references for bricks that are not resident in the cache.*

arranged brick structure. Similar to the idea of adaptive texture maps [KE02], we maintain an additional low-resolution floating point reference texture (e.g., 16^3 for a 512^3 volume with 32^3 bricks) storing texture coordinate offsets of bricks in a single brick cache texture that is always resident in GPU memory (e.g., a 512x512x256 texture). However, both the reference and the brick cache texture are maintained dynamically and not generated in a pre-process [KE02]. Figure 6 illustrates the use of the reference and brick cache textures. Note that since no gradient reconstruction or shading is performed during ray-casting, no complicated neighbor lookups are required at this stage. When the isovalue changes, bricks that potentially contain a part of the isosurface are downloaded into the brick cache texture. Inactive bricks are removed with a simple LRU (least recently used) strategy when their storage space is required for active bricks. Bricks that are currently not resident in the cache texture are specially marked at the corresponding position in the reference texture (shown as white squares in Figure 6). During ray-casting, samples in such bricks are simply skipped.

4. Deferred Shading

While the last section showed how to compute accurate ray/surface intersections for each pixel, this section describes how to turn the position image into a high quality rendering using deferred shading. All algorithms described here have image space complexity, meaning that they are independent of the size of the grid data. Each pass of the deferred shading stage writes a different property of the intersection position to an off-screen pixel buffer [ST90]. The result of one pass can serve as input for successive passes by mapping the pixel buffer as a texture. The final shading pass uses the property images to render the shaded isosurface to the viewport.

Figure 7: *Color mapping of maximum principal curvature magnitude using a 1D color look-up table (dragon data set with 512x512x256 samples).*

operation	#passes	inputs	outputs
Ray-Casting	3 [3]	*volume*	pos
Gradient	3 [6]	pos, *volume*	**g**
Hessian	6 [12]	pos, *volume*	**H**
Curvature	1 [13]	**g**, **H**	$\kappa_{1,2}$, $\mathbf{e}_{1,2}$
Shading	1 [14]	pos, **g**, $\kappa_{1,2}$, $\mathbf{e}_{1,2}$	image

Table 1: *Number of image space rendering passes and required input images for differential properties and deferred shading. Pass counts in brackets denote total number of passes after the intersection position computation.*

4.1. Differential Surface Properties

The appendix describes briefly how we quickly evaluate cubic reconstruction filters and their partial derivatives. See [SH05] for more details. This section shows that these basic capabilities can be exploited to calculate differential properties of isosurfaces from the scalar volume. In our implementation on a NVIDIA GeForce 6800, each property is calculated in one to six rendering passes, where each of these passes renders only a single screen-aligned quad in order to invoke the fragment shader for every output pixel. An overview of the number and types of rendering passes is given in Table 1.

Partial derivatives. The first differential property of the scalar volume that we need to reconstruct is its gradient $\mathbf{g} = \nabla f$, which we use as implicit surface normal and for curvature computations. The surface normal is the normalized gradient of the volume, or its negative, depending on the notion of being inside/outside the object: $\mathbf{n} = \pm\mathbf{g}/|\mathbf{g}|$. We compute **g** in three rendering passes, each of which evaluates a tri-cubic B-spline convolution sum in order to compute one of the three first-order partial derivatives via eight texture fetches from the 3D volume texture, plus three fetches from 1D filter weight textures [SH05]. The calculated gradient is stored in a single RGB floating point image, see Figure 3(derivatives). The Hessian $\mathbf{H} = \nabla\mathbf{g}$, comprised of all second partial derivatives of the volume, is calculated analogously. Due to symmetry, only six unique components need to be calculated, which is done in six rendering passes using either eleven or fourteen texture fetches each. The six calculated coefficients of **H** are stored in two RGB floating point images.

Extrinsic curvature. The first and second principal curvature magnitudes (κ_1, κ_2) of the isosurface can be estimated directly from the gradient **g** and the Hessian **H** [KWTM03], whereby tri-cubic filtering in general yields high-quality results. We do this in a single rendering pass, which uses the three partial derivative RGB floating point images generated by previous pipeline stages as input textures. The principal curvature magnitudes amount to two eigenvalues of the shape operator **S**, defined as the tangent space projection of the normalized Hessian:

$$\mathbf{S} = \mathbf{P}^T \frac{\mathbf{H}}{|\mathbf{g}|}\mathbf{P}, \qquad \mathbf{P} = \mathbf{I} - \frac{\mathbf{g}\mathbf{g}^T}{|\mathbf{g}|^2} \tag{2}$$

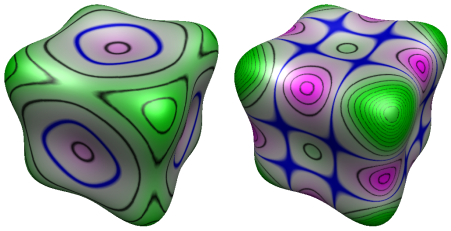


Figure 8: *Curvature Mapping of a* 64^3 *synthetic data set. Mean curvature* $(\kappa_1+\kappa_2)/2$ *(left), and Gaussian curvature* $\kappa_1\kappa_2$ *(right). Our renderer is capable to reproduce images from [KWTM03] at interactive rates. Data set and color mapping function are courtesy of Gordon Kindlmann.*

where **I** denotes the identity matrix. The eigenvalue corresponding to the eigenvector **g** vanishes, and the other two eigenvalues are the principal curvature magnitudes. Because one eigenvector is known, it is possible to solve for the remaining two eigenvectors in the two-dimensional tangent space without ever computing **S** explicitly. This results in reduced amount of operations and improved accuracy compared to the approach given in [KWTM03]. The transformation of the shape operator **S** to some orthogonal basis $(\mathbf{u},\mathbf{v})$ of the tangent space is given by

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = (\mathbf{u},\mathbf{v})^T \frac{\mathbf{H}}{|\mathbf{g}|}(\mathbf{u},\mathbf{v}) \qquad (3)$$

Eigenvalues of **A** can now be computed using the direct formulas for 2x2 matrices. The two eigenvectors of the shape operator **S** corresponding to the principal curvature directions are computed by transforming the eigenvectors of **A** back to three-dimensional object space.

$$\kappa_{1,2} = \frac{1}{2}\left(\mathrm{trace}(\mathbf{A}) \pm \sqrt{\mathrm{trace}(\mathbf{A})^2 - 4\det(\mathbf{A})}\right) \qquad (4)$$

$$\mathbf{e}_i = \kappa_i\mathbf{u} + (\kappa_i + a_{22} - a_{11})\mathbf{v} \qquad (5)$$

This amounts to a moderate number of vector and matrix multiplications, solving a quadratic polynomial, and three texture instructions. The curvature magnitudes and directions are rendered to two floating point targets.

4.2. Shading Effects

After the computation of differential surface properties, the resulting floating point images can be used for deferred shading in image space. Hence, all shading is decoupled from the volume and only calculated for actually visible pixels. This section outlines some of the example shading modes that we have implemented. This is only a small selection of possible rendering modes that can be used in our pipeline.

Shading from gradient image. The simplest shading equations depend on the normal vector of the isosurface. We have implemented standard Blinn-Phong shading and tone shading.

Figure 9: *Asian dragon data set (512x256x256). Left: tone shading. Right: tone shading blended with accessibility shading, allowing better depiction of local surface details.*

Curvature color mapping. The extrinsic curvature can be visualized on the isosurface by mapping curvature measures to colors via lookup textures. First and second principal curvatures, mean curvature $(\kappa_1+\kappa_2)/2$ and Gaussian curvature $\kappa_1\kappa_2$ can be visualized using a 1D lookup texture (see Figures 7 and 8) and give a good understanding of the local shape of the isosurface. Using a two-dimensional lookup texture for the (κ_1,κ_2) domain allows to highlight different structures on the surface. Figure 9 shows approximated accessibility shading [Mil94]. In this case, we have used a simple 1D curvature transfer function to darken areas with large negative maximum curvature. A 2D curvature function could also be employed for this purpose, giving finer control over the appearance.

Curvature-aligned flow advection. Direct mappings of principal curvature direction vectors to RGB colors are hard to interpret, see Figure 3(curvature directions). Instead of showing curvature directions directly, we visualize them with an approach based on image-based flow visualiza-

Figure 10: *Dense flow advected in the direction of maximum principal curvature (head of the David data set with* 512^3 *samples).*

tion [vW03]. In particular, we are advecting flow on the surface entirely in image space [LJH03] by computing advection on a per-pixel basis according to the underlying vector field of principal curvature directions. Image-based flow advection methods can be used on surfaces without parametrization by projecting a 3D flow field to the 2D image plane and advecting entirely in the image [vW03]. We do this by simply projecting each 3D curvature direction vector stored in the corresponding floating point image to the image plane immediately before performing advection for a given pixel. Image-based flow advection easily attains real-time rates, which complements the capability of our pipeline to generate the underlying, potentially unsteady, flow field in real-time. See Figure 10 for an example. A problem with advecting flow along curvature directions is that their orientation is not uniquely defined and thus seams in the flow cannot be entirely avoided [vW03]. Although these seams are visible when looking closely, we have found them to be not very disturbing in practice. Even though the flow field we are computing from curvature directions contains clearly visible patches (Figure 3: curvature directions), the resulting flow has much higher quality (Figure 10).

Non-photorealistic effects. Curvature information can be used for a variety of non-photorealistic rendering modes. We have implemented silhouette outlining taking curvature into account in order to control thickness, and depicting ridge and valley lines specified via colors in the (κ_1, κ_2) domain [KWTM03]. See Figures 11 and 8. In our pipeline, rendering modes such as these are simple operations that can be carried out in a single final shading pass, usually in combination with other parts of a larger shading equation, e.g., tone shading or solid texturing. We find the combination of curvature magnitude color maps and curvature-directed flow especially powerful for visualizing surface shape, e.g., as guidance during modeling.

5. Results

Volume rendering. Since the input to our rendering pipeline is an arbitrary scalar volume, it is naturally applicable to the rendering of isosurfaces such as the CT scan shown in Figure 11. We have integrated our renderer into an existing volume rendering framework as high-quality isosurface rendering front-end. In particular, real-time curvature estimation can be used to guide volume exploration, e.g., visualizing isosurface uncertainty, as has been proposed previously for off-line volume rendering [KWTM03].

Rendering from distance fields. For surface editing using a levelset approach, an initial implicit representation of the surface is usually generated by computing the signed distance to a triangle mesh. We used a variation of radially weighted linear fields [Nie04] to compute high resolution distance fields from triangle meshes, see Figures 7 and 9 for examples. Our rendering pipeline could easily be extended to include on the fly evaluation of CSG operations between multiple distance fields using min/max operations.

5.1. Rendering Performance

Table 2 gives performance numbers of our rendering pipeline corresponding to the figures shown in this paper. Except for very small volumes, the overall performance is dominated by the initial volume sampling step that computes approximate intersection positions. This fact is illustrated in Table 3. Although differential surface properties are expensive to compute in general, the fact that all of these computations have image space complexity combined with fast filtering decrease their impact on overall frame rate significantly. Even more important, the time spent in these computations is constant with respect to sampling rate and volume resolution. The same is true for intersection optimization via bisection. Table 4 illustrates the performance impact of different sampling rates. With respect to adaptive sampling, we compare constant sampling rates with the same rates for the maximum sampling rate r_1 that is used close to the isosurface (Section 3.2). We observe that although the overhead introduced by bricking is significant, it can be reduced via adaptive sampling so that overall performance is about 80-85% of rendering without bricking and without adaptive sampling.

data set	grid size	figure	fps
asian dragon	512x256x256	1	20.3
asian dragon	512x256x256	9	24.0
david head	512x512x512	1	15.3
david head	512x512x512	10	14.9
david	576x352x1536	2	10.3
cube	64x64x64	8	29.6
dragon	512x512x256	7	11.7

Table 2: *Performance of the renderings shown in the figures. Frame rates are given in frames per second for a 512x512 viewport. Four bisection steps have always been used, since they do not influence overall performance significantly.*

bounding geometry	ray-cast	differential properties	shading
1.7%	66.1%	31.0%	1.1%

Table 3: *Relative performance of the different stages of the pipeline for asian dragon rendering of Figure 1. Rendering performance is dominated by the surface intersection time.*

adaptive sampling	brick size	sampling rate (adaptive: r_1) 0.25	0.5	1	2	4	8
no	**none**	33.2	29.0	22.7	16.9	12.4	
no	**32**	23.8	19.5	16.1	11.7	7.2	
$r_1 = 8r_0$	**none**			34.6	27.4	20.3	15.2
$r_1 = 8r_0$	**32**			19.2	13.8	10.2	6.9

Table 4: *Rendering performance in frames per second corresponding to different sampling rates for asian dragon rendering of Figure 1. Brick caching introduces an additional texture indirection per sample (Section 3.4). Adaptive sampling (Section 3.2; $n = 8$) with bricking reduces this overhead compared to constant sampling.*

5.2. Discussion and Limitations

This section discusses some limitations of our system. A problem that can be seen in Figure 11, is that even when cubic filters are used, the curvature computed on actual scanned data contains visible noise. However, the quality of cubic filters is almost indistinguishable from filters up to order seven [KWTM03]. In any case it is important to use full 32-bit floating point precision for all GPU computations.

A limitation of our bisection approach for intersection is that in comparison to an analytic root search [DPH*03] or isolation of exactly one intersection [MKW*04], our discrete sampling with fixed step size does not guarantee correct detection of segments with multiple intersections. Furthermore, our bisection search might not find the intersection closest to the camera in such configurations.

A disadvantage of all deferred shading pipelines in general is the memory consumption of the image buffers. We are maintaining up to six window-sized images consisting of four 32-bit floating point channels each, which consumes a significant amount of GPU memory for high rendering resolutions and thus decreases the maximum volume or brick cache size.

Another consideration is whether to use an interpolating filter, such as tri-linear interpolation or Catmull-Rom cubic splines, or a smoothing filter such as the cubic B-spline for reconstruction purposes. A very good combination seems to be using an interpolating filter for value reconstruction, and a smoothing filter for reconstructing derivatives.

6. Conclusions

We have presented a rendering pipeline for real-time rendering of isosurfaces defined implicitly by regularly sampled scalar volumes. Using empty space skipping techniques and brick caching, we are able to render volumes of large sizes that would not fit into GPU texture memory at once at interactive rates. In comparison to volume rendering algorithms which perform color integration along the viewing ray, our method is optimized for rendering of isosurfaces. Because only one sample position contributes to the color of each pixel, differential surface properties can be computed on-the-fly in image space as part of the deferred shading stage. Due to its general nature, our pipeline is applicable to many practical problems involving implicit surfaces, such as volume rendering of scientific or medical data, modeling, morphing, and surface investigation using non-photorealistic techniques.

We would like to thank Gordon Kindlmann, Bob Laramee, Jiří Hladuvka, and Christof Rezk-Salama for their help and valuable contributions. The VRVis research center is funded in part by the Austrian Kplus project. The second author has been supported by Schlumberger Cambridge Research. The medical data sets are courtesy of Tiani MedGraph. The David model is courtesy of the Digital Michelangelo Project.

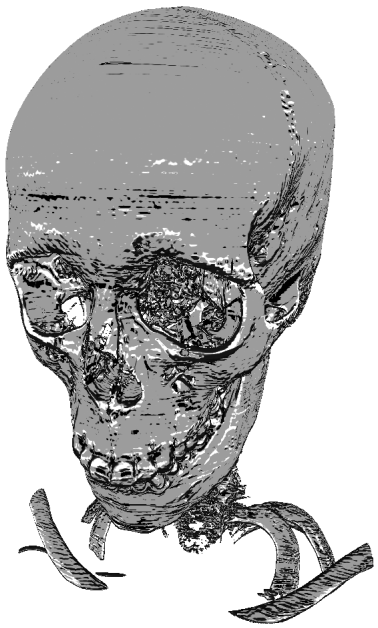


Figure 11: *Contours modulated with curvature in view direction, and ridges and valleys on an isosurface of a 512x512x333 CT scan of a human head.*

References

[Bar86] BARR A. H.: Ray tracing deformed surfaces. In *Proc. of SIGGRAPH '86* (1986), pp. 287 – 296.

[CSS98] CHIANG Y.-J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *Proc. of IEEE Visualization '98* (1998), pp. 167–174.

[DPH*03] DEMARLE D., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed interactive ray tracing for large volume visualization. In *Proc. of IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 87–94.

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. of Graphics Hardware 2001* (2001), pp. 9–16.

[Gre04] GREEN S.: Procedural volumetric fireball effect. In *NVIDIA SDK samples* (2004).

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *Proc. of IEEE Visualization 2002* (2002), pp. 53–60.

[HKG00] HLADUVKA J., KÖNIG A., GRÖLLER E.: Curvature-based transfer functions for direct volume rendering. In *Proc. of SCCG 2000* (2000), pp. 58–65.

[IFP95] INTERRANTE V., FUCHS H., PIZER S.: Enhancing transparent skin surfaces with ridge and valley lines. In *Proc. of IEEE Visualization '95* (1995), pp. 52–59.

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. In *Proc. of SIGGRAPH '89* (1989), pp. 297 – 306.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. of Graphics Hardware 2002* (2002), pp. 7–15.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 287–292.

[KWTM03] KINDLMANN G., WHITAKER R., TASDIZEN T., MÖLLER T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. of IEEE Visualization 2003* (2003), pp. 513–520.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. of SIGGRAPH '87* (1987), pp. 163–169.

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37.

[LHJ99] LAMAR E., HAMANN B., JOY K.: Multiresolution techniques for interactive texture-based volume visualization. In *Proc. of IEEE Visualization '99* (1999), pp. 355–361.

[LJH03] LARAMEE B., JOBARD B., HAUSER H.: Image space based visualization of unsteady flow on surfaces. In *Proc. of IEEE Visualization 2003* (2003), pp. 131–138.

[LKHW03] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of IEEE Visualization 2003* (2003), pp. 75–82.

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 317–324.

[MBWB02] MUSETH K., BREEN D. E., WHITAKER R. T., BARR A. H.: Level set surface editing operators. In *Proc. of SIGGRAPH 2002* (2002), pp. 330–338.

[Mil94] MILLER G.: Efficient algorithms for local and global accessibility shading. In *Proc. of SIGGRAPH '94* (1994), pp. 319–326.

[MKW*04] MARMITT G., KLEER A., WALD I., FRIEDRICH H., SLUSALLEK P.: Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proc. of Vision, Modeling, and Visualization* (2004), pp. 429–435.

[MMK*98] MÖLLER T., MÜLLER K., KURZION Y., MACHIRAJU R., YAGEL R.: Design of accurate and smooth filters for function and derivative reconstruction. In *Proc. of IEEE VolVis '98* (1998), pp. 143–151.

[Nie04] NIELSON G.: Radial hermite operators for scattered point cloud data with normal vectors and applications to implicitizing polygon mesh surfaces for generalized CSG operations and smoothing. In *Proc. of IEEE Vis. 2004* (2004), pp. 203–210.

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. In *Proc. of VisSym 2002* (2002), pp. 77–86.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 238–250.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Proc. of IEEE Visualization '98* (1998), pp. 233–238.

[RE01] RHEINGANS P., EBERT D.: Volume illustration: Nonphotorealistic rendering of volume models. In *Proc. of IEEE Visualization 2001* (2001), pp. 253–264.

[RGW*03] RÖTTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proc. of VisSym 2003* (2003), pp. 231–238.

[SH05] SIGG C., HADWIGER M.: Fast third-order texture filtering. In *GPU Gems 2, Matt Pharr (ed.)* (2005), Addison-Wesley, pp. 313–329.

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3D shapes. In *Proc. of SIGGRAPH '90* (1990), pp. 197–206.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 293–300.

[TO99] TURK G., O'BRIEN J. F.: Shape transformation using variational implicit functions. In *Proc. of SIGGRAPH '99* (1999), pp. 335–342.

[vW03] VAN WIJK J.: Image based flow visualization for curved surfaces. In *Proc. of IEEE Visualization 2003* (2003), pp. 745 – 754.

[WE98] WESTERMANN R., ERTL T.: Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH '98* (1998), pp. 169–177.

[WS01] WESTERMANN R., SEVENICH B.: Accelerated volume ray-casting using texture mapping. In *Proc. of IEEE Visualization 2001* (2001), pp. 271–278.

[WSE99] WESTERMANN R., SOMMER O., ERTL T.: Decoupling polygon rendering from geometry using rasterization hardware. In *Proc. of Eurographics Workshop on Rendering* (1999), pp. 45–56.

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMAN K., ERTL T.: Level-of-detail volume rendering via 3D textures. In *Proc. of IEEE VolVis 2000* (2000), pp. 7–13.

Appendix: Fast tri-cubic interpolation

To reconstruct a texture with a cubic B-spline filter at texture coordinate x, the convolution sum

$$f(x) = w_0(x)\, f_{i-1} + w_1(x)\, f_i + w_2(x)\, f_{i+1} + w_3(x)\, f_{i+2} \qquad (6)$$

of four weighted neighboring texels f_i has to be evaluated. Note that the weights are periodic in the sample positions of the input texture. The number of texture fetches is reduced by employing the linear filtering capability of GPU texture units. Instead of fetching all four neighbors independently, we fetch two consecutive samples at the same time using linear interpolation and perform a single weighted sum.

$$f(x) = g_0(x)\, f_{\lfloor x \rfloor - h_0(x)} + g_1(x)\, f_{\lfloor x \rfloor + h_1(x)} \qquad (7)$$

The weight functions g_i and offset functions h_i are pre-computed and stored in a lookup texture.

$$g_0(x) = w_0(x) + w_1(x), \qquad h_0(x) = 1 - \frac{w_1(x)}{w_0(x) + w_1(x)} \qquad (8)$$

$$g_1(x) = w_2(x) + w_3(x), \qquad h_1(x) = 1 + \frac{w_3(x)}{w_2(x) + w_3(x)} \qquad (9)$$

The extension to three dimensional textures is straight-forward due to separability of tensor-product B-splines, and it is possible to evaluate a tri-cubic filter with 64 summands using just eight tri-linear texture fetches. In order to compute partial derivatives, the functions g_i and h_i are computed using the appropriate derivatives of w_i. More details can be found in [SH05].

Eurographics/ IEEE-VGTC Symposium on Visualization (2006)
Thomas Ertl, Ken Joy, and Beatriz Santos (Editors)

Perspective Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications

Henning Scharsach[1] Markus Hadwiger[1] André Neubauer[2] Stefan Wolfsberger[3] Katja Bühler[1]

[1]VRVis Research Center [2]Tiani Medgraph [3]Department of Neurosurgery, Medical University Vienna

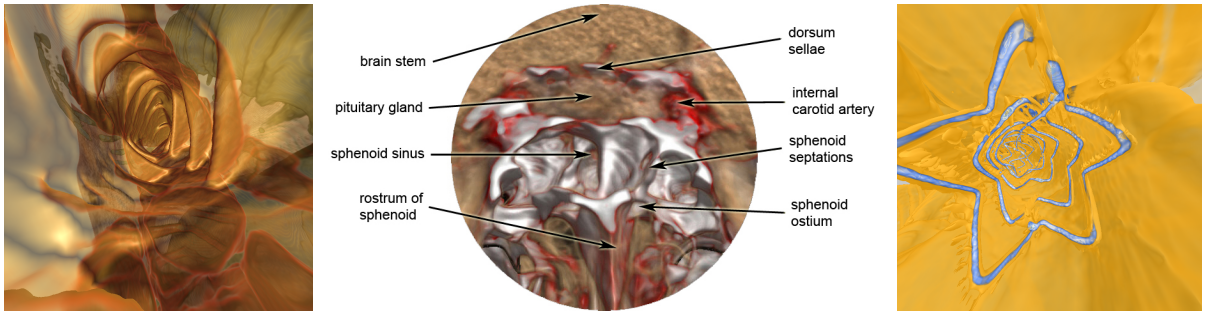


Figure 1: *The three main application areas of our system. Left: Virtual colonoscopy. The semi-transparent isosurface is blended with a direct volume ray-casting of the area behind. Center: Planning of pituitary surgery for tumor removal. Using direct volume rendering in combination with semi-transparent isosurfaces allows for visualization of otherwise occluded objects. Right: Virtual angioscopy of the aorta and a stent that supports it. Two different isovalues have to be used in this case.*

Abstract

Virtual endoscopy has proven to be a very powerful tool in endoscopic surgery. However, most virtual endoscopy systems are restricted to rendering isosurfaces or require segmentation in order to visualize additional objects behind occluding tissue. This paper presents a system for real-time perspective direct volume and isosurface rendering, which allows to simultaneously visualize both the interesting tissue and everything that is behind. Large volume data can be viewed seamlessly from inside or outside the volume without any pre-computation or segmentation. Our system uses a novel ray-casting pipeline for GPUs that has been optimized for the requirements of virtual endoscopy and also allows easy incorporation of auxiliary geometry, e.g., for displaying parts of the endoscopic device, pointers, or grid lines for orientation purposes. We present three main applications of this system and the underlying ray-casting algorithm. Although our ray-casting approach is of general applicability, we have specifically applied it to virtual colonoscopy, virtual angioscopy, and virtual pituitary surgery.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism Color, shading, shadowing, and texture

1. Introduction

Virtual endoscopy has become a very powerful tool for aiding endoscopic surgery procedures, from initial surgeon training to preoperative planning and intraoperative support [BHH*04, Bar05]. For rendering, most systems for virtual endoscopy are focusing on rendering isosurfaces, e.g., to depict tissue walls surrounding the current position of the endoscope. The two main approaches for rendering isosurfaces are to extract explicit geometry [BHH*04, Bar05], e.g., with a variant of the marching cubes algorithm [LC87], or to use first-hit ray-casting in order to determine the intersection of viewing rays with the isosurface [NMHW02, NWF*]. While the first category is able to achieve very high performance, especially when graphics hardware is used for rendering the surface geometry [HMK*97, BS99], the fact that explicit geometry can usually not be extracted interactively hampers isovalue changes. Furthermore, high-resolution isosurface geometry is very memory intensive. On the other hand, ray-casting approaches for virtual endoscopy have the flexibility to change the isovalue interactively, but cannot make use of hardware-accelerated polygon rasterization and are usually implemented with custom CPU algorithms [NWF*].

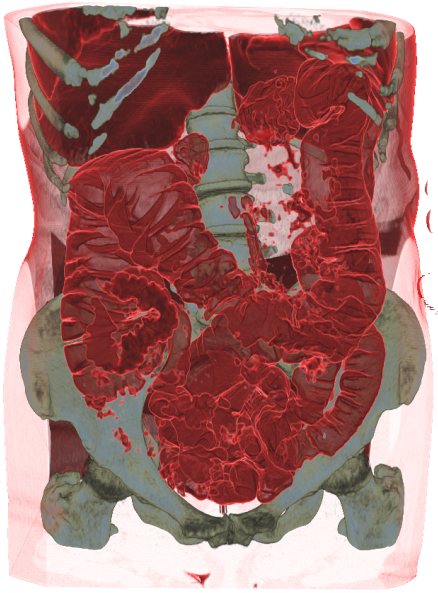


Figure 2: *Overview of a 512x512x478 colonoscopy data set rendered with our system. Figure 1 (left) and Figure 3 show inside views using the same data set and transfer function.*

Though isosurface-renderings can be enhanced to include more information about surface properties [NWF*], a large part of the information contained in a medical scan is still discarded during rendering because everything in front and especially most of the information behind the isosurface is not rendered. A crucial example is pituitary surgery for removing a tumor at the pituitary gland near the internal carotid artery, which is hidden behind a bone structure that must be punctured by the surgeon without damaging the artery behind it [NWF*] (see Figure 1 (center)). Most previous approaches have required segmentation in order to visualize these crucial occluded structures, which is a very time-consuming task and only allows to depict objects as isosurfaces of smoothed binary masks [NFW*04].

In contrast, direct volume rendering is only common for viewing volume data from the outside, because most approaches either use orthogonal projection or incur artifacts with perspective projection due to incorrect opacity correction [EHK*04]. Endoscopic views, however, require perspective projection with large fields of view and correct computation of the volume rendering integral, which necessitated special hardware architectures so far to achieve interactivity [MB01]. GPU-based ray-casting on the other hand can deliver this interactivity for either perspective or orthogonal projections [KW03], but requires far more effort to overcome the inherent problems. Most recent approaches build on single-pass ray-casting, where the entire volume is traversed in a single rendering pass using data-dependent looping in the hardware fragment shader [SSK*05, HSS*05, KSS*05]. In order to support large data sets, a bricked volume can be rendered in correct visibility order by performing ray-casting for each brick individually [HQK05]. This, however, incurs per-brick setup overhead in contrast to single-pass approaches.

Our system uses a novel ray-casting pipeline for GPUs supporting Shader Model 3.0 (e.g., NVIDIA GeForce 6800 or ATI Radeon X1800) that uses perspective projection and also allows easy incorporation of auxiliary geometry, e.g., for displaying parts of the endoscopic device, pointers, or grids for orientation. We seamlessly integrate direct volume rendering with isosurface rendering and achieve real-time performance for both fly-through and outside views without the need for pre-computation or segmentation, which greatly facilitates use by physicians that are working under enormous time pressure.

Although our system can essentially be used for virtual endoscopy in general, we have specifically applied it to three different applications: Virtual colonoscopy [HMK*97], virtual angioscopy [NMHW02], and virtual endonasal transsphenoidal pituitary surgery [NWF*]. We have integrated our system into a commercial medical workstation software, and the application to neurosurgery is already in regular clinical use for operation planning at the department of neurosurgery at the Medical University Vienna.

2. Applications

Virtual endoscopy has proven to be a useful tool for a number of different applications, including pre-operative planning, diagnostic purposes, teaching and practicing with endoscopic tools and even the emerging field of aided intra-operative navigation. The aim of all of these systems - especially the latter one - is to aid medical doctors with a computer generated view of a certain position and orientation of the endoscope that resembles the real endoscopic view as closely as possible while at the same time supplying additional information that would not be visible otherwise. This additional information may include waypoints that prevent deviation from the optimal path, tissue right behind visible structures, the endoscope and attached endoscopic tools themselves, means to measure certain structures to get a better impression of the surrounding tissue and emphasizing important parts like nerves or bigger blood vessels that must not be damaged by any means.

To fulfill all these requirements, a suitable renderer has to be able to visualize many semi-transparent structures at the same time while always keeping the focus on the main object of interest: The surrounding walls of the blood vessel, colon or other structure that the endoscope is travelling through. What makes this task even more difficult is the fact that we are facing three different types of objects here that all have to be visualized accordingly: Isosurfaces like the walls have to be extracted and lit in a way that gives a good overview about surface properties and makes small deformations easily detectable. Regions of interest like tissue behind

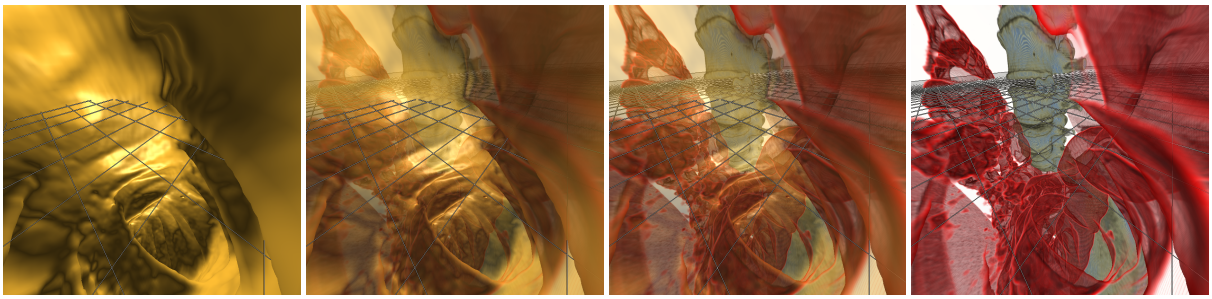


Figure 3: *Changing the isosurface opacity allows to focus more on the foreground rendered with isosurfacing (e.g., the colon) or the background rendered with direct volume rendering. Correctly integrating polygonal geometry such as the grid shown here facilitates spatial orientation. Isosurface opacity also influences rendering performance, as illustrated in Tables 1 and 2.*

walls, important nerves or organs should be visualized using direct volume rendering, giving the user the possibility to distinguish different parts of tissue by assigning a suitable transfer function. Finally, endoscopic tools, grids and pointers, which will aid the surgeon in orientation and estimation of magnitude, are made up of lines and polygons and should be visualized using the normal OpenGL pipeline.

Bringing these visualization techniques together while still preserving interactive framerates was the primary goal of our system, and in the following we will present three fields of application where this leads to significant improvements over previous systems in both expressiveness and applicability.

2.1. Virtual Colonoscopy

A regular colonoscopy enables doctors to examine the last portion of the gastrointestinal tract and look for problem areas such as inflamed tissue, abnormal growths and ulcers. The main purpose is to detect early signs of cancer in the colon and rectum and further analyze or remove them, if possible. Virtual colonoscopy can greatly speed up this process by enabling doctors to check for any kind of abnormality beforehand, and only perform a real colonoscopy when polyps have to be removed or samples have to be taken.

Unfortunately, most virtual colonoscopy systems suffer from the limited expressiveness of the common isosurface extraction, which makes it very difficult to identify all abnormalities and may even lead to false diagnoses because important features where simply overlooked. With doctors hesitating to adopt a technology that has not really proven reliability yet, not many of these systems have found their way into clinical practice. Another shortcoming is that the visualization often lacks resemblance to the real images, making it very difficult for doctors to estimate tissue properties and recognize certain parts later on.

The solution we propose is to combine isosurface extraction for visualization of the colon with a DVR of the regions of interest behind the intestinal wall, thus gaining additional information about the respective tissue. This not only leads to a much more expressive image, but also provides important clues about the position and orientation within the colon, because the whole gastrointestinal tract can be seen at any time. In the case of an intra-operative system, this makes it extremely simple to quickly find areas of interest again, thus greatly speeding up the surgery.

For diagnostic purposes, an automatic path can be calculated which provides a convenient and quick fly-through from the beginning of the colon to the rectum, giving an overview over the large intestine in a matter of minutes.

2.2. Virtual Angioscopy

Virtual angioscopy (virtual endoscopy inside blood vessels) is primarily used for detecting stenoses and calcifications in blood vessels. With many blood vessels being too narrow for a normal endoscope, virtual angioscopy is in many cases the only alternative to the tedious process of examining 2D-slices of a CT scan. Besides the small size of the structures, the specific nature of an angiography requires a slight modification of the rendering pipeline: Because some kind of contrast medium is injected to identify important vessels more easily, density values inside the vessels are higher than those outside. On the other hand, calcifications inside the vessel have an even higher density value, which means that we face two different isosurfaces: One at a certain minimum threshold (i.e., the outer walls) and one at a maximum threshold above that of the contrast medium, which in most cases are calcifications but can also be structures such as stents [BDV*97]. An example that has been generated with our system is illustrated in Figure 1 (right). In this case, the first isosurface corresponds to the stent, and the second isosurface corresponds to the aortic wall. Everything outside one of the isosurfaces (i.e. below the minimum or above the maximum threshold) will be rendered with DVR again, which supplies additional information about the tissue density thus aiding doctors in detecting calcifications and again

facilitates estimating the absolute position and orientation through the visualization of certain body landmarks.

A further interesting application is the implantation of stents, which are small tubular prostheses that are inserted into an artery via an endovascular procedure and are mainly used to enlarge a stenosis (i.e. a local narrowing of the arterial lumen). Like calcifications, stents have a density threshold well above that of the contrast medium, which requires the rendering of two different isosurfaces again. When implanting a stent, the position in respect to other landmarks of the body (e.g. heart, lung or bones) is especially important, which is achieved through a whole DVR behind the semi-transparent isosurfaces again.

2.3. Endonasal Transsphenoidal Pituitary Surgery

Endonasal transsphenoidal pituitary surgery is a minimally invasive endoscopic procedure, mainly applied to remove various kinds of pituitary tumors. Virtual endoscopy can aid medical doctors by simulating this challenging surgery beforehand and planning the approach and ideal target position of the endoscopic intervention. Especially in the case of an intra-operative environment, the system should provide visual feedback about important nerves, blood vessels and other significant landmarks that should not be damaged, thus assisting the doctor in finding the optimal path.

In order to do this, those important landmarks have to be identified and should be visualized throughout the whole process. To avoid the necessity of pre-segmenting every single one of those objects, the whole head should be rendered with a semi-transparent DVR. If a suitable transfer function is selected, this not only warns the doctor whenever an important object is nearby that requires special attention, but also avoids deviation of the optimal path by always visualizing the main object of interest (e.g. the tumor).

Visualizing the endoscopic tools is another important aspect: Especially when planning the optimal path and ideal target position, the size and proportions of these tools with respect to the surrounding tissue is crucial. Because a volumetric approach would raise some serious issues (rotating the tools, limited resolution etc.) and probably not resemble the real appearance closely enough, a polygonal representation of the tools should be merged with the volumetric scene. Of course, intersections have to be calculated and displayed correctly, otherwise the perceived information about available space and proportions may be misleading.

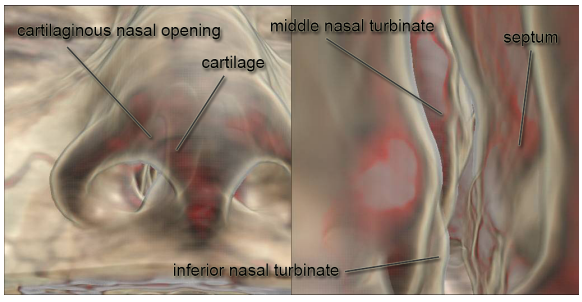


Figure 4: *In endonasal pituitary surgery the endoscope enters through the nose and is advanced to the sphenoid sinus and the pituitary gland behind it (see Figure 1 (center)).*

Furthermore, when encountering objects of interest like small passages or even the tumor itself, there should be a way to measure this object or at least get a first estimation of its size. Other auxiliary graphical elements like grids and pointers can assist the user here, again necessitating a rendering pipeline that can deal with both polygonal and volumetric objects.

3. Hybrid Ray-Casting for Virtual Endoscopy

Choosing the right volume rendering mode for a particular application is crucial for extracting the useful and important information that the user wants to see. In the case of virtual endoscopy, using only isosurface rendering discards a lot of information behind the surface that can provide additional insight into the properties of the underlying tissue, as well as crucial information about occluded structures. In order to alleviate this problem, our system combines isosurface and direct volume ray-casting in a single rendering. We render a shaded semi-transparent isosurface in front, and perform unshaded direct volume rendering behind it. This rendering mode achieves our goals and also avoids visual confusion of isosurface shading and volume shading. The focus is still on the isosurface, but important background information is available at any time. Shading the isosurface is important for shape perception. Direct volume rendering then provides additional information, such as the tissue structure close to the isosurface, as well as depicting background objects farther behind. This leads to very expressive images and allows for considerable flexibility. The basic algorithm is very simple and leverages standard polygon rasterization for ray setup, and a very short fragment shader loop for actual ray-casting. In addition to the volume, intersecting polygonal geometry is integrated seamlessly.

3.1. Algorithm Overview

The ray-casting pipeline of our system combines object-order and image-order stages in order to find a balance between the two and leverage the parallel processing of modern GPUs. For culling of irrelevant subvolumes, a regular grid of min-max values for bricks of size 8^3 is stored along with the volume. Ray-casting itself is performed in a single rendering pass in order to avoid the setup overhead of casting each brick separately [HQK05]. The first step of the algorithm culls bricks on the CPU and generates two separate bit arrays that determine whether a brick is *active* or *inactive*.

The first bit array contains the state of bricks with respect to the isosurface. A brick is active when it intersects the isosurface. The second bit array contains the state of bricks with respect to the transfer function. A brick is active when it is not completely transparent.

In the object-order stage on the GPU, these two bit arrays are used to rasterize brick boundary faces in several rendering passes. The result of these rendering passes are two images that drive the subsequent ray-casting stage. The first image, the *ray start position image*, contains the volume coordinate positions where ray-casting should start for each pixel. Coordinates are stored in the RGB components, and the alpha component is one when a ray should be started, and zero when no ray should be started. The second image, the *ray length image* contains the direction vectors for ray-casting in the RGB components and the length of each ray in the alpha component. Note that the direction vectors could easily be computed in the fragment shader from the camera position and the ray start positions as well. However, the ray length must be rendered into an image that is separate from the ray start positions due to read-write dependencies, which can then also be used for storing the direction vectors that are needed for ray length computation anyway.

The main steps of our ray-casting approach for each pixel are:

1. Compute the initial ray start position on the near clipping plane of the current viewport. When the start position is in an inactive brick with respect to the isosurface, calculate the nearest intersection point with the boundary faces of active isosurface bricks, in order to skip empty space. The result is stored in the *ray start position image*.
2. Compute the ray length until the last intersection point with boundary faces of bricks that are active either due to the isosurface or the transfer function or both. The result is stored in the *ray length image*.
3. Optionally render opaque polygonal geometry and overwrite the ray length image where the distance between the ray start position and the geometry position is less than the stored ray length.
4. Cast from the start position stored in the *ray start position image* along the direction vector until the accumulated opacity reaches a specified threshold (*early ray termination*) or the ray length given by the *ray length image* is exceeded. The result of ray-casting is stored in a separate compositing buffer.
5. Blend the ray-casting compositing buffer on top of the polygonal geometry.

The two main acceleration schemes exploited here are *empty space skipping* and *early ray termination*. For the former, view-independent culling of bricks and rasterization of their boundary faces are employed (Section 3.2), whereas the latter is handled during ray-casting (Section 3.3).

3.2. Culling and Brick Boundary Rasterization

Because we are using hybrid isosurface and direct volume rendering, culling has to determine two different sets of active/inactive states for all bricks, which are stored in separate bit arrays. Each brick is either inactive, active with respect to the isosurface, active with respect to the transfer function, or active with respect to both. In order to determine ray start positions and ray lengths, we employ rasterization of the boundary faces between active and inactive bricks, which is illustrated in Figure 5. To handle brick culling efficiently, the minimum and maximum voxel values of each brick are stored along with the volume, which are compared at run-time with the isovalue and the transfer function, respectively. A brick can be safely discarded when the opacity is always zero between those two values, which can be determined very quickly using summed area tables [GBKG04].

Rasterizing the boundary faces between active and inactive bricks results in object-order empty space skipping. It prunes the rays used in the ray-casting pass and implicitly excludes most inactive bricks. Note, however, that our approach does not exclude all empty space from ray-casting, which can be seen for ray r_3 in Figure 5 (left). This is a trade-off that enables ray-casting without any per-brick setup overhead and works extremely well in practice, which is also illustrated by the performance figures in Section 4. The border between active and inactive bricks defines a surface that can be rendered as standard OpenGL geometry with the corresponding position in volume coordinates encoded in the RGB colors. Corresponding to the two bit arrays of active bricks that results from culling, two sets of boundary faces must be considered. All vertices of brick bounding geometry are constantly kept in video memory. Only two additional index arrays referencing the vertices of active boundary faces have to be updated every time the isovalue or the transfer function changes. As long as the near clipping plane does not intersect the bounding geometry, rays can always be started at the brick boundary front faces. However, if such an in-

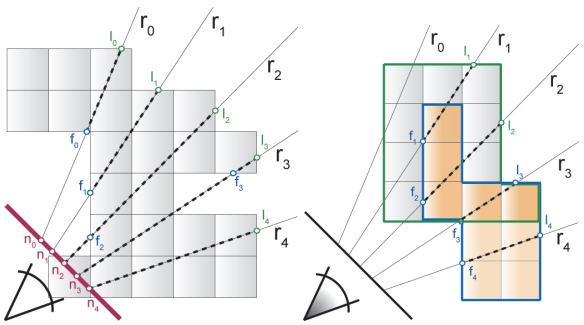


Figure 5: *Determining ray start positions and ray lengths using rasterization of brick boundary faces. Left: The basic case described in Section 3.2.1. Right: Extended cases for endoscopy rendering, which are described in Section 3.2.2.*

tersection occurs, it will produce holes in the front-facing geometry, which results in some rays not being started at all, and others started at incorrect positions. Figure 6 illustrates this problem. In an endoscopic view, we constantly face this situation, so rays typically need to be started at the near clipping plane, which is shown in Figure 5 in the case of points n_2-n_4. To avoid casting through empty space, rays should not be started at the near clipping plane if the starting position is in an inactive brick but at the next intersection with active boundary faces, such as rays r_0 and r_1 in Figure 5. These rays are started at f_0 and f_1, instead of being starting at n_0 and n_1. We achieve this by drawing the near clipping plane first and the front faces afterwards, which ensures that whenever there are no front faces to start from, the position of the near clipping plane will be taken. However, since the non-convex bounding geometry often leads to multiple front faces for a single pixel, the next front face is used when the first front face is clipped, which results in incorrect ray start positions. The solution is to detect when a ray intersects a back face before the first front face that is not clipped.

3.2.1. The Basic Case

When only one bit array of active bricks is used, e.g., direct volume rendering is used without isosurfacing, the basic steps to obtain the *ray start position image* are as follows:

1. Disable depth buffering. Rasterize the entire near clipping plane into the color buffer. Set the alpha channel to zero everywhere.
2. Enable depth buffering. Disable writing to the RGB components of the color buffer. Rasterize the *nearest back faces* of all active bricks into the depth buffer, e.g., by using a depth test of `GL_LESS`. Set the alpha channel to one where fragments are generated.
3. Enable writing to the RGB components of the color buffer. Rasterize the *nearest front faces* of all active bricks, e.g., by once again using a depth test of `GL_LESS`. Set the alpha channel to one where fragments are generated.

This ensures that all possible combinations shown in Figure 5 (left) are handled correctly. Rasterizing the nearest front faces makes sure that all near plane positions in inactive bricks will be overwritten by start positions on active bricks that are farther away (rays r_0 and r_1). Rasterizing the nearest back faces before the front faces ensures that near plane positions inside active blocks will not be overwritten by front faces that are farther away (rays r_2 and r_3). Brick geometry that is nearer than the near clipping plane is automatically clipped by the graphics subsystem. After that, the *ray length image* can be computed, which first of all means finding the last intersection points of rays with the bounding geometry. The basic steps are:

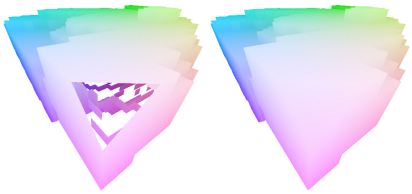


Figure 6: *Holes resulting from near clipping plane intersection (left) must be filled with valid starting positions (right).*

1. Rasterize the *farthest back faces*, e.g., by using a depth test of `GL_GREATER`.
2. During this rasterization, sample the ray start position image and subtract it from the back positions obtained via rasterization of the back faces. This yields the ray vectors and the ray lengths from start to end position.
3. Multiply all ray lengths with the alpha channel of the ray start position image (which is either 1 or 0).

These steps can all be performed in the same fragment shader. Drawing the back faces of the bounding geometry results in the last intersection points of rays and active brick geometry, which are denoted as l_i in Figure 5. Subtracting end positions from start positions yields the ray vectors, which can then be normalized and stored in the RGB components of the *ray length image* together with the ray lengths in the alpha channel. Note that the alpha channel of the ray length image has consistently be set to zero where a ray should not be started at all, which is exploited in the ray-casting pass (Section 3.3).

3.2.2. Combining Isosurfacing and DVR

The basic case described in the previous section must be extended when isosurface and direct volume rendering are combined:

1. Rasterization passes for the ray start position image must use the bit array containing the state of bricks with respect to the isosurface. The transfer function is disregarded.
2. Rasterization for generation of the ray length image must treat all bricks as active that are active with respect to either the isosurface or the transfer function or with respect to both.

Figure 5 (right) illustrates all possible cases. Ray r_0 is never cast because it never intersects isosurface bricks. Both ray r_1 and ray r_2 start at isosurface bricks and terminate at transfer function bricks that are inactive with respect to the isosurface. Ray r_3 starts at an isosurface brick but terminates at a transfer function brick that is also active with respect to the isosurface. Ray r_4 starts and ends at isosurface bricks that are inactive with respect to the transfer function.

3.3. Ray-Casting

Our system employs a ray-casting fragment shader that performs the entire casting step for both the isosurface and

the DVR part behind it in a single rendering pass. Therefore, the GPU is required to support data-dependent looping and branching in the fragment shader, e.g., an NVIDIA GeForce 6 or an ATI Radeon X1800. The shader is essentially comprised of two successive ray-casting loops, which perform first-hit ray-casting followed by DVR ray-casting. The first loop in the fragment shader starts at the position given by the *ray start position image* and traverses the ray until it finds an intersection with the isosurface. After an intersection has been detected, the actual intersection position is refined using iterative bisection with a fixed number of four iterations [HSS*05]. Then, the gradient at the intersection position is computed using central differences and the isosurface is shaded using the standard Blinn-Phong model. The shaded result is weighted according to the opacity of the isosurface, which can be set arbitrarily via a fragment shader parameter. Figure 3 illustrates changing isosurface opacity. After the isosurfacing part of the shader, DVR ray-casting continues with the initial alpha set to the opacity of the isosurface and performs sampling and compositing until a specified alpha threshold is exceeded. Checking against this threshold results in *early ray termination*. That is, the DVR ray-casting loop is terminated as soon as all subsequent samples cannot contribute to the final pixel color anymore. Note that early ray termination naturally depends significantly on the constant opacity of the isosurface. Tables 1 and 2 clearly show that the frame rate increases considerably with increasing isosurface opacity. In order to avoid visual interference with the shaded isosurface that is in front, no further shading is performed in the DVR compositing loop. Naturally, this also increases performance accordingly. Note that the ray-casting shader only performs ray-casting for pixels with a ray length greater than zero, which also excludes rays from processing that do not intersect active bricks at all as described in Section 3.2.1.

3.4. Geometry Intersection

Many applications for virtual endoscopy require both volumetric and polygonal data to be present in the same scene.

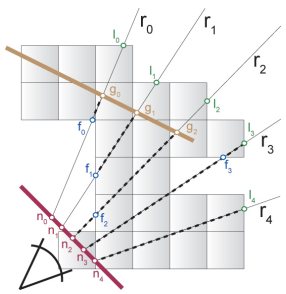


Figure 7: *When rays intersect opaque polygonal geometry, they are terminated immediately. This is achieved by modifying the ray length image accordingly.*

Naturally, intersections of the volume and geometry have to achieve a correct visibility order, and in many cases looking at the intersections of the geometry and the isosurface is the reason for rendering geometry in the first place. A very powerful use of combining geometry with volume rendering is to display grid lines for orientation purposes, which is illustrated in Figure 3. We use a planar grid consisting of lines, which is a very simple but powerful means for assessing spatial location. This grid plane can be translated in the orthogonal direction and can also be rotated arbitrarily. Also, parts that do not contribute to the final image because they are occluded by geometry should not perform ray-casting at all. An easy way to achieve this is to terminate rays once they hit a polygonal object by modifying the ray length image accordingly. This is illustrated in Figure 7. Of course, ray lengths should only be modified if a polygonal object is closer to the view point than the initial ray length. This problem can again be solved by using the depth test and extending the algorithm described in Section 3.2.1, leading to the complete algorithm outlined in Section 3.1.

After rendering the back faces of active/inactive brick boundaries with their respective depth values (and depth test set to `GL_GREATER`), the intersecting geometry is rendered to the same buffer, with the corresponding volume coordinates encoded in the color channel. With the depth test reversed to `GL_LESS`, only those parts will be drawn that are closer to the view point than the initial ray lengths. This approach modifies ray-casting such that it results in an image that looks as if it was intersected with an invisible object. Blending this image on top of the actual geometry in the last pass of the algorithm results in a rendering with correct intersections and visibility order.

4. Rendering Performance

Tables 1 and 2 give rendering performance results of our system. Setting the transfer function to a simple alpha ramp illustrates the effectiveness of early ray termination (Table 1). Setting the isosurface to full opacity will result in immediate ray termination when the isosurface is hit, which yields performance figures similar to rendering isosurfaces only. The less opacity the isosurface adds to the image, the longer the ray has to travel to accumulate full opacity in the direct volume rendering stage. Choosing a more complex transfer

alpha ramp TF	isosurface opacity 100%	80%	50%	0%
Minimum	40.3 fps	32.4 fps	29.5 fps	28.3 fps
Maximum	64.7 fps	54.8 fps	48.6 fps	44.4 fps
Average	58.2 fps	46.3 fps	41.2 fps	37.5 fps

Table 1: *Performance comparison of different isosurface opacities for the colonoscopy dataset with a simple ramp as transfer function. Measured for a 512x512 viewport on a GeForce 7800.*

complex TF	100%	isosurface 80%	opacity 50%	0%
Minimum	40.3 fps	11.1 fps	10.0 fps	9.4 fps
Maximum	64.7 fps	18.6 fps	16.7 fps	15.9 fps
Average	58.2 fps	16.8 fps	14.6 fps	13.2 fps

Table 2: *Performance comparison of different isosurface opacities for the colonoscopy dataset with a complex transfer function that prevents early ray termination most of the time. Measured for a 512x512 viewport on a GeForce 7800.*

function with low alpha values results in performance reduction because in this case early ray termination is ineffective for many rays (Table 2).

5. Conclusions

We have described an effective system for virtual endoscopy that uses GPU-based ray-casting in order to achieve real-time performance. The combination of isosurface and direct volume ray-casting has proven to be very useful during endoscopic planning in order to inspect structures that would otherwise be hidden behind the isosurface.

Using the computational power of today's GPUs in a hardware-based approach as described here, simultaneous isosurface and direct volume ray-casting is feasible at interactive frame rates, which has traditionally been substituted by pure isosurfacing or requiring segmentation. Merging this capability with a flexible rendering pipeline that can handle both volumetric and polygonal data, we have presented a system that is capable of meeting the visualization demands of medical doctors in diagnostic as well as intra-operative environments. The effectiveness and applicability of this virtual endoscopy system has been shown in three different fields of endoscopic procedures: virtual colonoscopy, virtual angioscopy and pituitary surgery. For neurosurgery, our system is already in clinical use, and we will investigate the clinical applicability of the other applications we have presented in the future.

Acknowledgments. The authors would like to thank Christian Sigg for important input. The VRVis research center is funded in part by the Austrian Kplus project. The medical data sets are courtesy of Tiani MedGraph.

References

[Bar05] BARTZ D.: Virtual endoscopy in research and clinical practice. In *Computer Graphics Forum* (2005), p. 24(1).

[BDV*97] BEIER J., DIEBOLD T., VEHSE H., BIAMINO G., FLECK E., FELIX R.: Virtual endoscopy in the assessment of implanted aortic stents. In *Proc. of Computer Assisted Radiology* (1997), pp. 183–188.

[BHH*04] BARTZ D., HARDENBERGH J., HAUTH M., MUELLER K., WACKER M., WU Y.: Advanced virtual medicine: Techniques and applications for virtual endoscopy and soft-tissue-simulation. In *IEEE Visualization 2004 Tutorial Notes* (2004).

[BS99] BARTZ D., SKALEJ M.: VIVENDI - a virtual ventricle endoscopy system for virtual medicine. In *Data Visualization (Proc. of Symposium on Visualization)* (1999), pp. 155–166.

[EHK*04] ENGEL K., HADWIGER M., KNISS J., LEFOHN A., REZK-SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. Course Notes SIGGRAPH 2004, 2004.

[GBKG04] GRIMM S., BRUCKNER S., KANITSAR A., GRÖLLER E.: Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics* (2004), pp. 1–8.

[HMK*97] HONG L., MURAKI S., KAUFMAN A., BARTZ D., HE T.: Virtual voyage: Interactive navigation in the human colon. In *Proceedings of SIGGRAPH'97* (1997), pp. 27–34.

[HQK05] HONG W., QIU F., KAUFMAN A.: Gpu-based object-order ray-casting for large datasets. In *Proceedings of Volume Graphics 2005* (2005).

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proceedings of Eurographics 2005* (2005), pp. 303–312.

[KSS*05] KLEIN T., STRENGERT M., STEGMAIER S., , ERTL T.: Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Proceedings of IEEE Visualization 2005* (2005), pp. 123–230.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proc. of IEEE Visualization 2003* (2003), pp. 287–292.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. of SIGGRAPH '87* (1987), pp. 163–169.

[MB01] MEISSNER M., BARTZ D.: Translucent and opaque direct volume rendering for virtual endoscopy applications. In *Proceedings of Volume Graphics 2001* (2001).

[NFW*04] NEUBAUER A., FORSTER M., WEGENKITTL R., MROZ L., BÜHLER K.: Efficient display of background objects for virtual endoscopy using flexible first-hit ray casting. In *Proceedings of VisSym 2004* (2004), pp. 301–310.

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. In *Proceedings of VisSym 2002* (2002), pp. 77–ff.

[NWF*] NEUBAUER A., WOLFSBERGER S., FORSTER M., MROZ L., WEGENKITTL R., BÜHLER K.: Advanced virtual endoscopic pituitary surgery. *IEEE Transactions on Visualization and Computer Graphics 11*, 5, 497–507.

[NWF*04] NEUBAUER A., WOLFSBERGER S., FORSTER M., MROZ L., WEGENKITTL R., BÜHLER K.: STEPS - an application for simulation of transsphenoidal endonasal pituitary surgery. In *Proc. of IEEE Visualization* (2004).

[SSK*05] STEGMAIER S., STRENGERT M., KLEIN T., , ERTL T.: A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of Volume Graphics 2005* (2005), pp. 187–195.