Course Notes 28

Real-Time Volume Graphics

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Real-Time Volume Graphics

Abstract The tremendous evolution of programmable graphics hardware has made high-quality real-time volume graphics a reality. In addition to the traditional application of rendering volume data in scientific visualization, the interest in applying these techniques for real-time rendering of atmospheric phenomena and participating media such as fire, smoke, and clouds is growing rapidly. This course covers both applications in scientific visualization, e.g., medical volume data, and real-time rendering, such as advanced effects and illumination in computer games, in detail. Course 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 basic texture-based approaches including hardware ray casting, the algorithms are improved and expanded incrementally, covering local and global illumination, scattering, pre-integration, implicit surfaces and non-polygonal isosurfaces, transfer function design, volume animation and deformation, dealing with large volumes, high-quality volume clipping, rendering segmented volumes, higher-order filtering, and non-photorealistic volume rendering. Course participants are provided with documented source code covering 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 basic visualization techniques is helpful, but not required.

Level of Difficulty **Intermediate-Advanced.** Participants should have a working knowledge of computer graphics and some background in graphics programming APIs such as OpenGL or DirectX. Familiarity with basic visualization techniques is helpful, but not required.

Contact

Markus Hadwiger
(course organizer)
VRVis Research Center for
Virtual Reality and Visualization
Donau-City-Straße 1
A-1220 Vienna, Austria
phone: +43 1 20501 30603
fax: +43 1 20501 30900
email: msh@vrvis.at

Christof Rezk Salama
(course organizer)
Computer Graphics Group
University of Siegen
Hölderlinstr. 3
57068 Siegen, Germany
phone: +49 271 740-2826
fax: +49 271 740-3337
email: rezk@fb12.uni-siegen.de

Klaus Engel
Siemens Corporate Research
Imaging & Visualization Department
755 College Road East
Princeton, NJ 08540
phone: +1 (609) 734-3529
fax: +1 (609) 734-6565
email: klaus.engel@scr.siemens.com

Joe Michael Kniss
Scientific Computing and Imaging
School of Computing
University of Utah, USA
50 S. Central Campus Dr. #3490
Salt Lake City, UT 84112
email: jmk@cs.utah.edu
phone: +1 801-581-7977

Aaron E. Lefohn
Center for Image Processing
and Integrated Computing
Computer Science Department
University of California, USA
Davis, CA 95616-8562
email: lefohn@cs.ucdavis.edu

Daniel Weiskopf
Institute of Visualization
and Interactive Systems
University of Stuttgart
Universitätsstr. 38
70569 Stuttgart, Germany
email: weiskopf@vis.uni-stuttgart.de

Lecturers

Klaus Engel
Siemens Corporate Research (SCR) in Princeton/NJ

Since 2003, Klaus Engel has been a researcher at Siemens Corporate Research (SCR) in Princeton/NJ. He received a PhD in computer science from the University of Stuttgart in 2002 and a Diplom (Masters) of computer science from the University of Erlangen in 1997. From January 1998 to December 2000, he was a research assistant at the Computer Graphics Group at the University of Erlangen-Nuremberg. From 2000 to 2002, he was a research assistant at the Visualization and Interactive Systems Group of Prof. Thomas Ertl at the University of Stuttgart. He has presented the results of his research at international conferences, including IEEE Visualization, Visualization Symposium and Graphics Hardware. In 2001, his paper *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading* has won the best paper award at the SIGGRAPH/Eurographics Workshop on Graphics Hardware. He 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.*
Detailed information about this research projects are available online at:
http://wwwvis.informatik.uni-stuttgart.de/~engel/

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

Markus Hadwiger is a researcher in the *Basic Research in Visualization* group at the VRVis Research Center in Vienna, Austria, and a PhD student at the Vienna University of Technology. The focus of his current research is exploiting consumer graphics hardware for high quality visualization at interactive rates, especially volume rendering for scientific visualization. First results on high quality filtering and reconstruction of volumetric data have been presented as technical sketch at SIGGRAPH 2001, and as a paper at Vision, Modeling, and Visualization 2001. He is regularly teaching courses and seminars on computer graphics, visualization, and game programming.

Before concentrating on scientific visualization, he was working in the area of computer games and interactive entertainment. His master's thesis "Design and Architecture of a Portable and Extensible Multiplayer 3D Game Engine" describes the game engine of Parsec (http://www.parsec.org/), a still active cross-platform game project, whose early test builds have been downloaded by over 100.000 people, and were also included on several Red Hat and SuSE Linux distributions.

Information about current research projects can be found at:
http://www.VRVis.at/vis/
http://www.VRVis.at/vis/research/hq-hw-reco/

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 Kniss recently receeved his Masters degree in computer science from the University of Utah, where he is currently pursuing his PhD and a member of the Scientific Computing and Imaging Institute. His current research focuses on interactive volume rendering techniques for scientific discovery and visualization. His current interests are volume light transport (global illumination), user interfaces for scientific visualization, and hardware rendering techniques. He has published numerous papers on these topics and organized two courses that have provided an extensive and in depth overview of volume rendering techniques.

Current research activities can be found on line at:
http://www.cs.utah.edu/~jmk/research.html
http://www.cs.utah.edu/~jmk/simian/index.html

Aaron E. Lefohn
Center for Image Processing and Integrated Computing
Computer Science Department
University of California, Davis
Davis, CA 95616-8562
email: lefohn@cs.ucdavis.edu
Phone:(530) 754-9470
Fax: (530) 752-8894

Aaron Lefohn is a Ph.D. student in the computer science department at the University of California at Davis and a graphics software engineer at Pixar Animation Studios. His current research includes general computation with graphics hardware, interactive high-quality rendering, and data-parallel programming models. His IEEE Visualization 2003 paper, "Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware" was one of six papers nominated for the Best Paper award and an image from the paper was used on the conference proceedings. Aaron has published papers in a number of international conferences and journals; including IEEE Visualization, Medical Image Computing and Computer Assisted Intervention (MICCAI), IEEE Transactions on Visualization and Computer Graphics, IEEE Computer Graphics and Applications, and Eurographics Computer Graphics Forum. In addition to his publications, Aaron was a tutorial presenter at IEEE Visualization 2003 and has given invited talks on his work at ATI, Pixar, and IBM Watson Research. Aaron completed an M.S. in computer science at the University of Utah in 2003, an M.S. in theoretical chemistry from the University of Utah in 2001, and a B.A. in chemistry from Whitman College in 1997. Aaron is currently studying under John Owens at the University of Califonia, Davis and is an NSF graduate fellow in computer science.

Christof Rezk Salama
Computergraphik und Multimediasysteme,
University of Siegen,
Hölderlinstr. 3,
57068 Siegen, Germany
phone: +49 271-740-3315
fax: +49 271-740-3337
email: rezk@fb12.uni-siegen.de

Christof Rezk Salama has studied computer science at the University of Erlangen-Nuremberg. In 2002 he received a PhD at the Computer Graphics Group in Erlangen as a scholarship holder at the graduate college "3D Image Analysis and Synthesis". Since October 2003 he is working as a postdoc at the Computer Graphics and Multimedia Group at the University of Siegen, Germany. The results of his research have been presented at international conferences, including SIGGRAPH, IEEE Visualization, Eurographics, MICCAI and Graphics Hardware. His PhD thesis is titled "volume rendering techniques for general purpose graphics hardware" and has won the dissertation award of the German Staedtler

Foundation. The volume rendering software OpenQVis that he started as a research project is now open source and has recently won a software innovation award from the German government. He has regularly taught courses on graphics programming, tutorials and seminars on computer graphics, geometric modeling and scientific visualization. He has gained practical experience in several scientific projects in medicine, geology and archaeology.

Detailed information about this research projects are available online at:
http://www9.informatik.uni-erlangen.de/Persons/Rezk
http://openqvis.sourceforge.net

Daniel Weiskopf
Institute of Visualization and Interactive Systems
University of Stuttgart
Universitätsstr. 38
70569 Stuttgart, Germany
email: weiskopf@vis.uni-stuttgart.de

Daniel Weiskopf is senior researcher and teacher of computer science at the Institute of Visualization and Interactive Systems at the University of Stuttgart (Germany). He studied physics at the University of Tübingen (Germany), San Francisco State University, and the University of California at Berkeley. He received a Diplom (Masters) of physics in 1997 and a PhD in theoretical astrophysics in 2001, both from the University of Tübingen. Daniel Weiskopf authored several articles on scientific visualization and computer graphics; he won the Best Case Study award at IEEE Visualization 2000 for his paper on general relativistic ray tracing. At the University of Stuttgart, he is regularly teaching courses and seminars on computer graphics, visualization, image synthesis, geometric modeling, computer animation, and human-computer interaction. He is organizing and/or presenting in tutorials at distinguished computer graphics conferences; e.g., he was involved in: SIGGRAPH 2001 Course 16 ("Visualizing Relativity"), Eurographics 2002 Tutorial 4 ("Programmable Graphics Hardware for Interactive Visualization"), Eurographics 2003 Tutorial 7 ("Programming Graphics Hardware"), and the IEEE Visualization 2003 Tutorial on "Interactive Visualization of Volumetric Data on Consumer PC Hardware". In addition to his research on computer graphics, he is interested in using visualization for communicating complex physical concepts to the public: several of his films were featured at major European festivals of scientific animations

and TV broadcasts; his visualizations have been included in a number of popular-science publications. (For a list of references, see
http://www.vis.uni-stuttgart.de/~weiskopf/publications).
Major research interests include scientific visualization, virtual reality, interaction techniques, non-photorealistic rendering, computer animation, special and general relativity.

Course Syllabus

MORNING

Welcome and Speaker Introduction [10 min] (Ch. Rezk Salama) **8:30 – 8:40**

- Introductory words, speaker introduction
- Motivation
- Course overview

Introduction to GPU-Based Volume Rendering [60 min] (Ch. Rezk Salama) **8:40 – 9:40**

- Structure of modern GPUs (graphics processing units), rendering pipeline
- Low-level programming, shading languages
- Physical background of volume rendering
- Volume rendering integral
- Traditional ray casting
- 2D texture-based volume rendering
- 3D texture-based volume rendering
- 2D multi-texture volume rendering
- Compositing

GPU-Based Ray Casting [35 min] (D. Weiskopf) **9:40–10:15**

- Ray casting in regular grids (ray setup, ray integration)
- Ray casting in tetrahedral grids
- Acceleration techniques (early ray termination, empty space skipping, adaptive sampling)

BREAK [15 min] **10:15-10:30**

Local Illumination for Volumes [25 min] (M. Hadwiger) **10:30-10:55**

- Gradient estimation

- Per-pixel Illumination
- Reflection Mapping
- Non-Polygonal Shaded Isosurfaces

10:55-11:20 **Transfer Function Design: Classification** [25 min] (J. Kniss)

- Pre- versus post-classification
- Transfer functions
- Implementations
- Multi-dimensional transfer functions
- image-driven and data-driven methods, user interaction and feedback

11:20-11:45 **Transfer Function Design: Optical Properties** [25 min] (J. Kniss)

- Alternative lighting strategies
- Data vs. computation tradeoff
- Shadowing
- Translucency
- General light transport & global illumination solutions
- Balancing quality and speed

11:45-12:15 **Pre-Integration and High-Quality Filtering** [30 min] (K. Engel)

- Sampling and reconstruction
- Pre-integrated classification
- Texture-based pre-integrated volume rendering
- Higher-order filtering of volumetric data
- Rasterization isosurfaces using dependent textures

12:15-1:45 **LUNCH BREAK** [75 min]

AFTERNOON

1:45-2:30 **Atmospheric Effects, Participating Media, and Scattering** [45 min] (J. Kniss)

- Scattering
- Perlin noise and volumetric effects
- Clouds, smoke, and fire

2:30-3:00 **High-Quality Volume Clipping** [30 min] (D. Weiskopf)

- Object-space approach by tagged volumes
- Image-space approach by depth textures
- Pre-integrated volume rendering and clipping

- Consistent illumination for clipped volumes

Non-Photorealistic Volume Rendering and Segmented Volumes [30 min] (M. Hadwiger) 3:00-3:30

- Adapting NPR techniques to volumes
- Tone shading
- Curvature-based NPR techniques
- Rendering segmented data
- Per-object rendering modes and attributes
- Per-object compositing modes (two-level volume rendering)

BREAK [15 min] 3:30-3:45

Volume Deformation and Animation [30 min] (Ch. Rezk Salama) 3:45-4:15

- Theoretical background
- Piecewise linear patches
- Deformation using dependent textures
- Illumination

Dealing With Large Volumes [30 min] (K. Engel) 4:15-4:45

- Bricking
- Caching strategies
- Compression techniques
- Shading of large volumes
- Acceleration techniques

Rendering from Difficult Data Formats [30 min] (A. Lefohn) 4:45-5:15

- Motivation: Compression, volume computation, dynamic volumes
- Example of *difficult* data formats: stack of 2D slices, sparse, compressed
- Rendering techniques
- On-the-fly reconstruction
- Examples from compression, computation, etc.
- Deferred filtering for compressed data

Summary, Questions and Answers [15min] (all speakers) 5:15-5:30

Contents

I Introduction 1

1 Volume Rendering 2
- 1.1 Volume Data . . . 3
- 1.2 Direct Volume Rendering . . . 4
  - 1.2.1 Optical Models . . . 5
  - 1.2.2 The Volume Rendering Integral . . . 6
  - 1.2.3 Ray-Casting . . . 8
  - 1.2.4 Alpha Blending . . . 9
  - 1.2.5 The Shear-Warp Algorithm . . . 10
- 1.3 Maximum Intensity Projection . . . 11

2 Graphics Hardware 13
- 2.1 The Graphics Pipeline . . . 13
  - 2.1.1 Geometry Processing . . . 14
  - 2.1.2 Rasterization . . . 15
  - 2.1.3 Fragment Operations . . . 16
- 2.2 Programmable GPUs . . . 18
  - 2.2.1 Vertex Shaders . . . 18
  - 2.2.2 Fragment Shaders . . . 20

II GPU-Based Volume Rendering 23

3 Sampling a Volume Via Texture Mapping 24
- 3.1 Proxy Geometry . . . 26
- 3.2 2D-Textured Object-Aligned Slices . . . 27
- 3.3 2D Slice Interpolation . . . 32
- 3.4 3D-Textured View-Aligned Slices . . . 34
- 3.5 3D-Textured Spherical Shells . . . 35
- 3.6 Slices vs. Slabs . . . 36

4 Components of a Hardware Volume Renderer **37**
4.1 Volume Data Representation 37
4.2 Volume Textures 38
4.3 Transfer Function Tables 39
4.4 Fragment Shader Configuration 40
4.5 Blending Mode Configuration 41
4.6 Texture Unit Configuration 42
4.7 Proxy Geometry Rendering 43

III GPU-Based Ray Casting **45**

5 Introduction to Ray Casting **46**

6 Ray Casting in Uniform Grids **51**

7 Ray Casting in Tetrahedral Grids **59**

IV Local Illumination **66**

8 Basic Local Illumination **67**

9 Non-Polygonal Isosurfaces **72**

10 Reflection Maps **74**

11 Deferred Shading **76**

12 Deferred Gradient Reconstruction **80**

13 Other Differential Properties **82**

V Classification **84**

14 Introduction **85**

15 Classification and Feature Extraction **86**
15.1 The Transfer Function as a Feature Classifier 87
15.2 Guidance 87
15.3 Summary 91

16 Implementation **95**

VI Optical Properties **97**

17 Introduction **98**

18 Light Transport **99**
18.1 Traditional volume rendering 99
18.2 The Surface Scalar 101
18.3 Shadows 102
18.4 Translucency 105
18.5 Summary 110

19 User Interface Tips **112**

VII High-Quality Volume Rendering **114**

20 Sampling Artifacts **117**

21 Filtering Artifacts **121**

22 Classification Artifacts **125**

23 Shading Artifacts **133**

24 Blending Artifacts **139**

25 Summary **144**

VIII Volume Clipping **145**

26 Introduction to Volume Clipping **146**

27 Depth-Based Clipping **147**

28 Clipping via Tagged Volumes **153**

29 Clipping and Consistent Shading **155**

30 Clipping and Pre-Integration **162**

IX Non-Photorealistic Volume Rendering **166**

31 Introduction **167**

32 Basic Non-Photorealistic Rendering Modes **168**

33 Rendering from Implicit Curvature **172**

X Segmented Volumes 177

34 Introduction 178

35 Segmented Data Representation 182

36 Rendering Segmented Data 183

37 Boundary Filtering 190

38 Two-Level Volume Rendering 196

39 Performance 200

40 Acknowledgments 203

XI Volume Deformation and Animation 205

41 Introduction 206
41.1 Modeling Paradigms 206
41.2 Volumetric Deformation 207

42 Deformation in Model Space 208
42.1 Depth Sorting 209

43 Deformation in Texture Space 210
43.1 Practical Aspects 211
43.2 Non-uniform Subdivision 213
43.2.1 Edge Constraints 213
43.2.2 Face Constraints 214
43.3 Deformation via Fragment Shaders 214

44 Local Illumination 216

45 Volume Animation 220

XII Dealing with Large Volumes 221

46 Bricking 226

47 Multi-Resolution Volume Rendering 228

48 Compression Techniques **231**
48.1 Wavelet Compression 232
48.2 Packing Techniques 236
48.3 Vector Quantization 239

49 Procedural Techniques **241**

50 Optimizations **244**
50.1 Shading 245
50.2 Empty Space Leaping 245
50.3 Ray Termination 246

51 Summary **248**

XIII Rendering From Difficult Data Formats **249**

52 Rendering From Difficult Data Formats **250**
52.1 Introduction 250
52.2 Volume Domain Reconstruction 251
52.3 Filtering 253
52.4 Conclusions 254

XIV Literature **255**

Course Notes 28
Real-Time Volume Graphics

Introduction

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

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 [68, 17] 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 [19], 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.

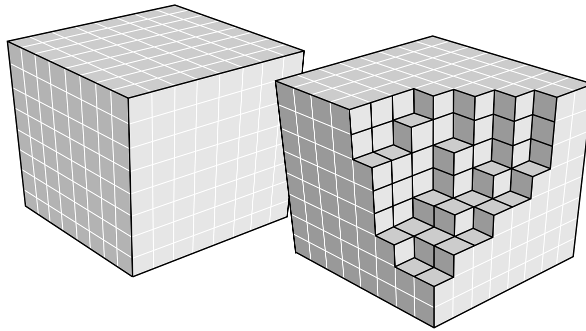


Figure 1.1: Voxels constituting a volumetric object after it has been discretized.

1.1 Volume Data

Discrete volume data set can be thought of as a simple three-dimensional array of cubic elements (voxels[1]) [46], 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 [34, 35].

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 [76]. The scalar value is virtually mapped to physical quantities which describe light interaction at the respective point in 3D-space. This mapping is

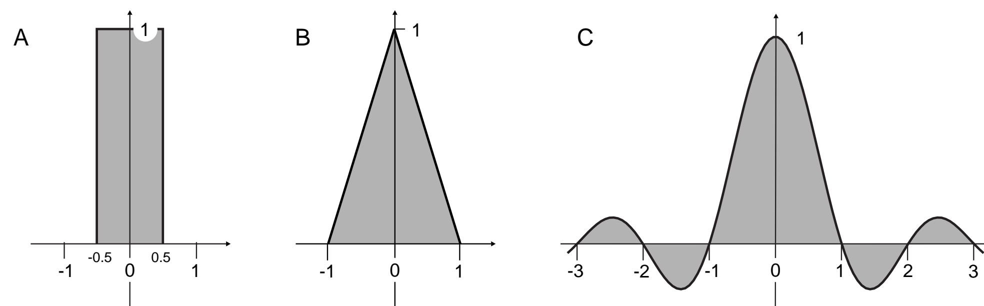


Figure 1.2: Reconstruction filters for one-dimensional signals. In practise, box filter *(A)* and tent filter *(B)* are used instead of the ideal *sinc*-filter *(C)*.

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 [76], 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

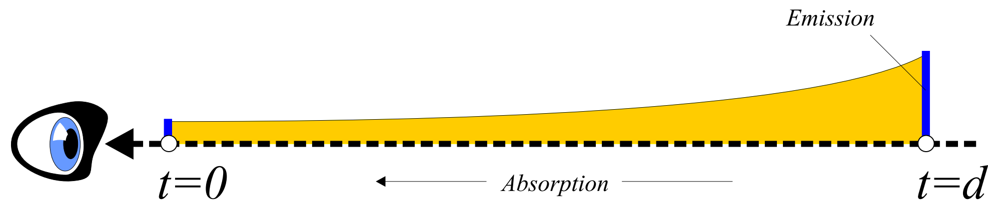


Figure 1.3: An amount of radiant energy emitted at $t = d$ is partially absorbed along the distance d.

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 [68] 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 [68] 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* [112], which are also known as *associated colors* [9]. 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 [112].

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 [64] 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

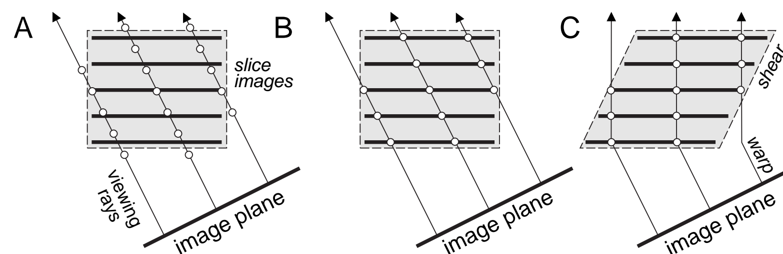


Figure 1.4: The shear-warp algorithm for orthogonal projection.

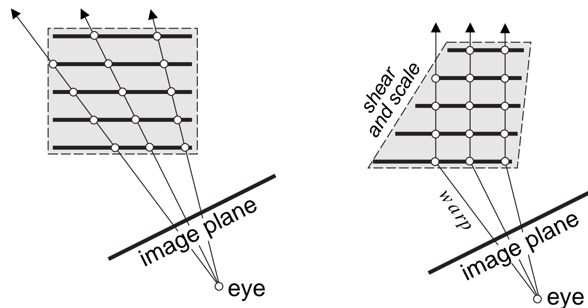


Figure 1.5: The shear-warp algorithm for perspective projection.

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.

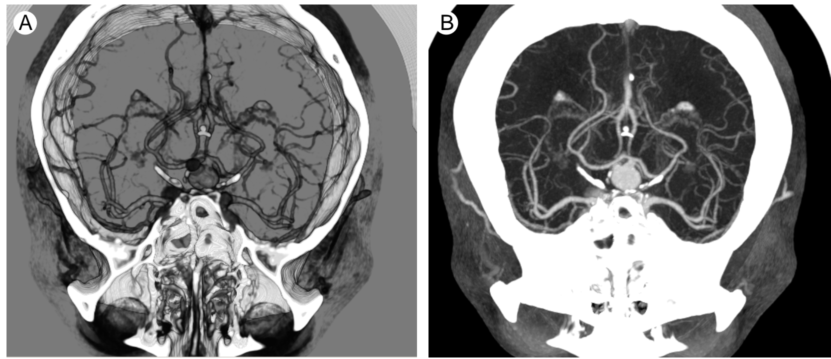


Figure 1.6: A comparison of direct volume rendering (A), and maximum intensity projection (B).

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 [26]. 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*.

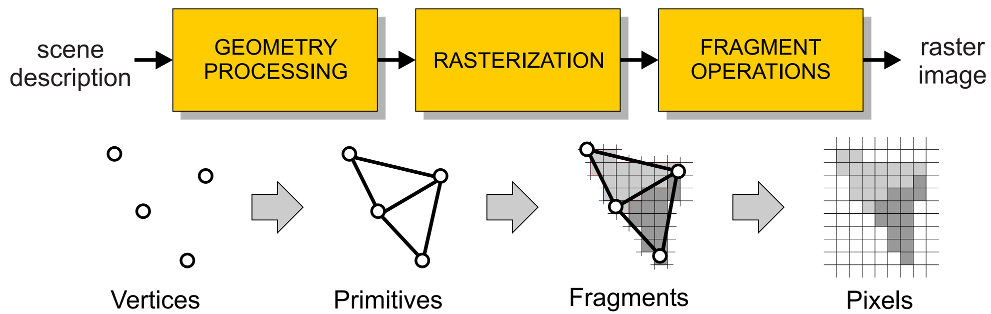


Figure 2.1: The standard graphics pipeline for display traversal.

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.

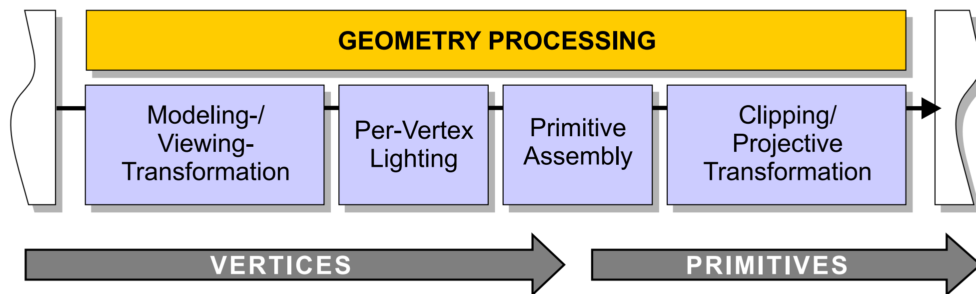


Figure 2.2: Geometry processing as part of the standard graphics pipeline.

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 [82] 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 pro-

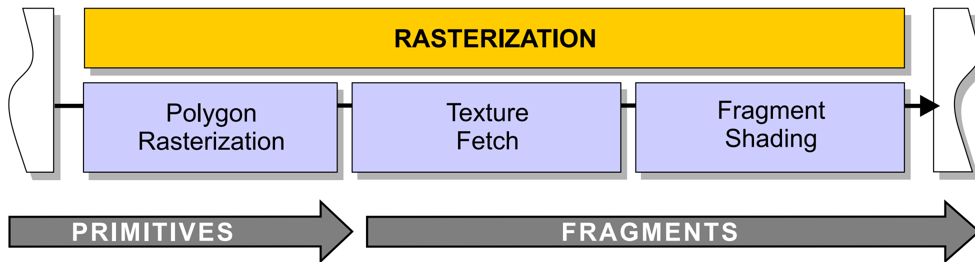


Figure 2.3: Rasterization as part of the standard graphics pipeline.

cess 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

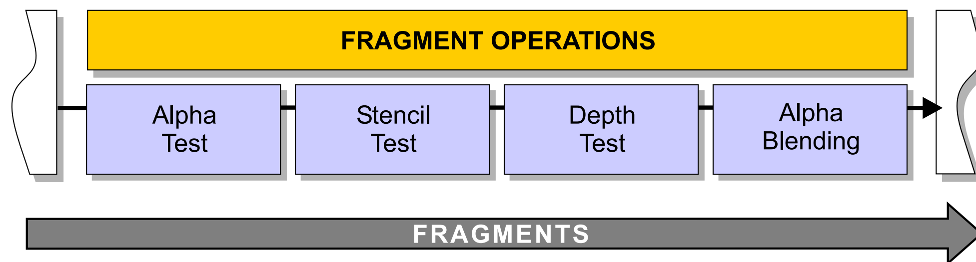


Figure 2.4: Fragment operations as part of the standard graphics pipeline.

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 [91, 26]. 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 [91] 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;

}
```

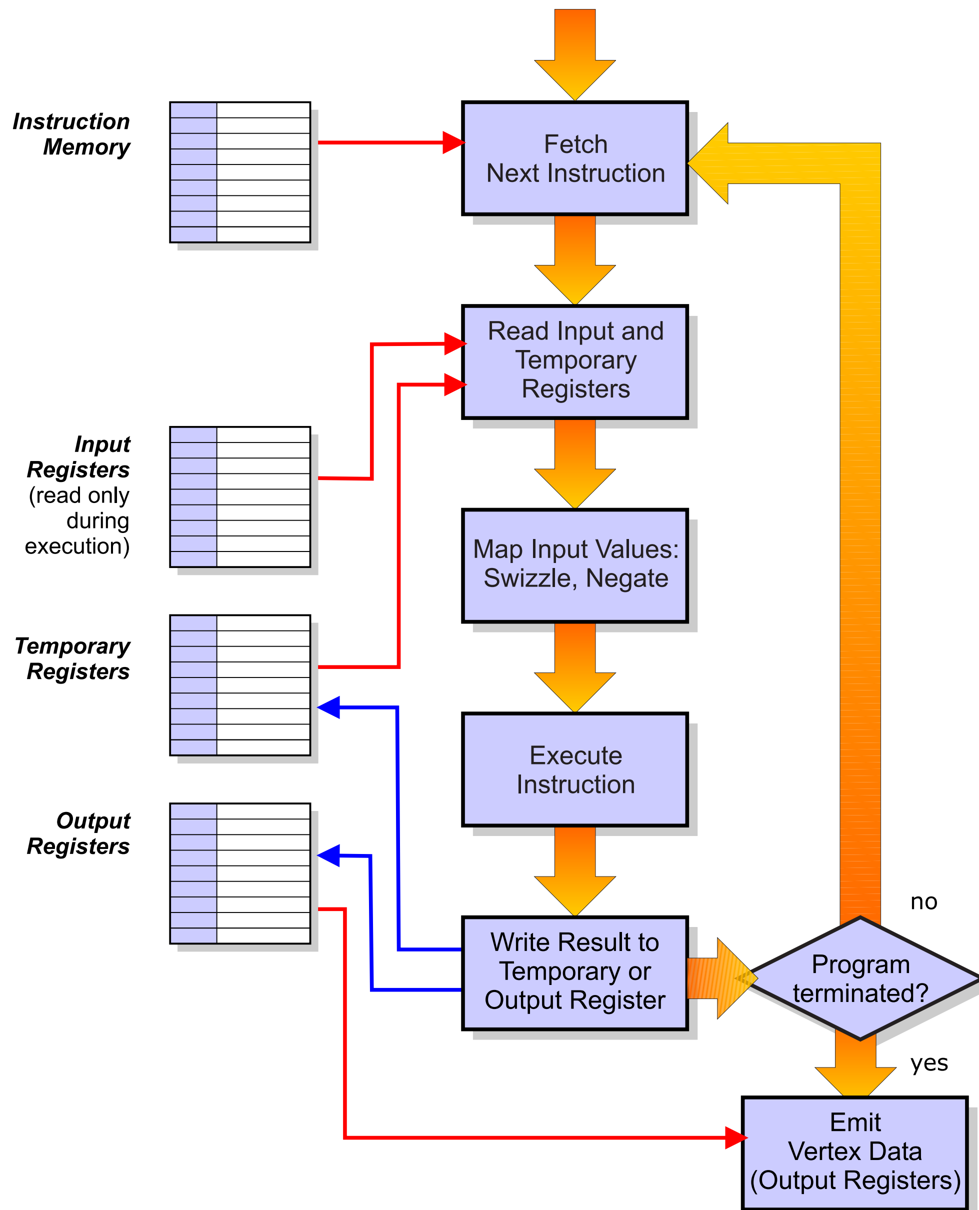


Figure 2.5: The programmable vertex processing unit executes a vertex program stored in local video memory. During the execution a limited set of input-, output- and temporary registers is accessed.

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 [25]

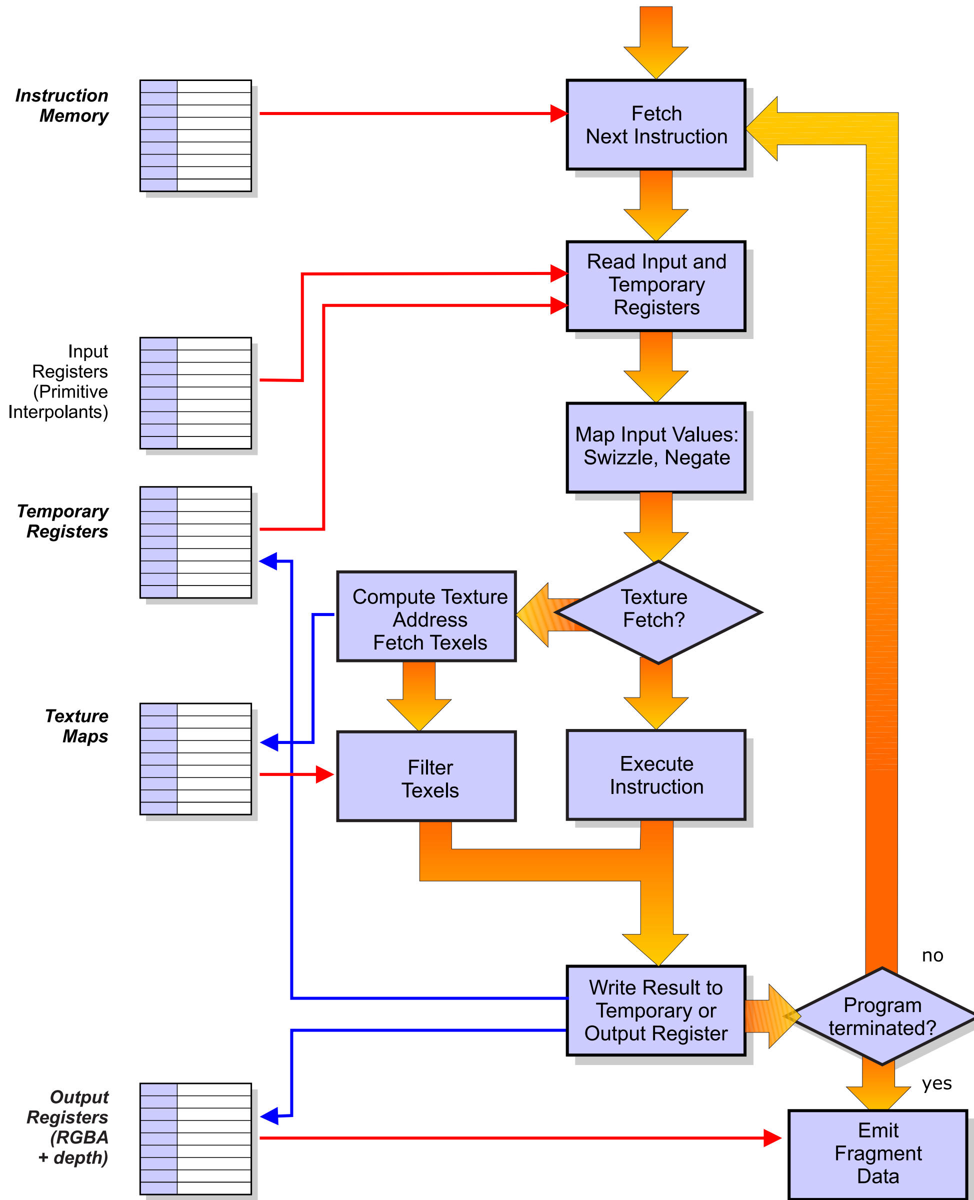


Figure 2.6: For each fragment, the programmable fragment processor executes a micro-program. In addition to reading the input and temporary registers, the fragment processor is able to generate filtered texture samples from the texture images stored in vide memory.

Course Notes 28

Real-Time Volume Graphics

GPU-Based Volume Rendering

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

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

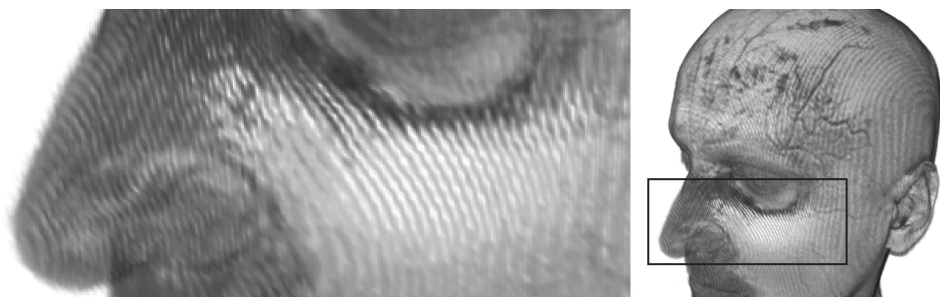


Figure 3.1: Rendering a volume by compositing a stack of 2D texture-mapped slices in back-to-front order. If the number of slices is too low, they become visible as artifacts.

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.

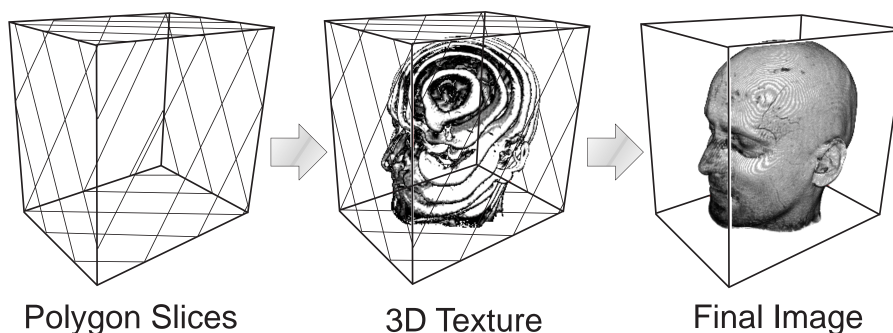


Figure 3.2: View-aligned slices used as proxy geometry with 3D texture mapping.

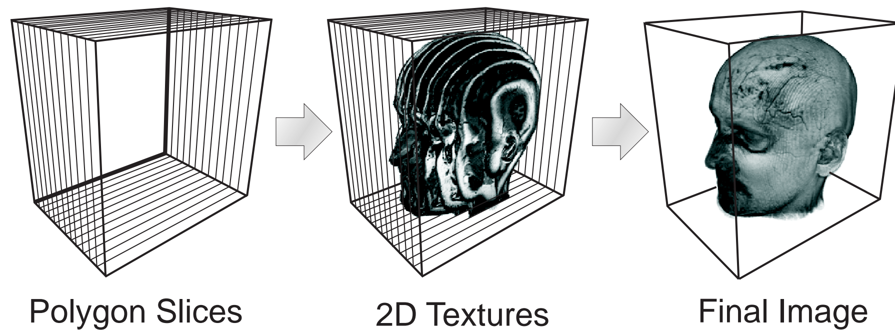


Figure 3.3: Object-aligned slices used as proxy geometry with 2D texture mapping.

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 [11]. 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

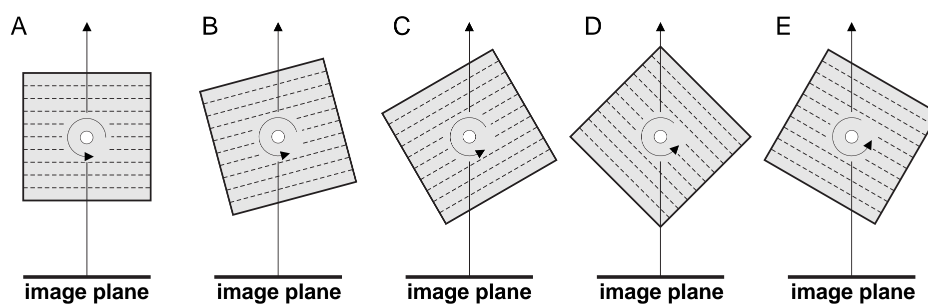


Figure 3.4: Switching the slice stack of object-aligned slices according to the viewing direction. Between image (C) and (D) the slice stack used for rendering has been switched.

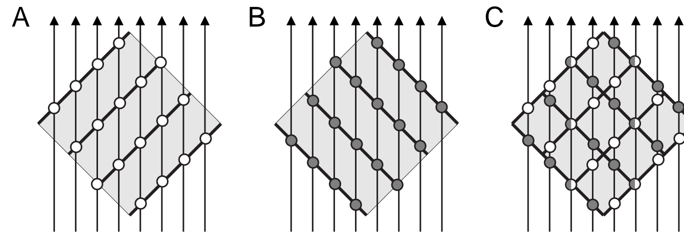


Figure 3.5: The location of sampling points changes abruptly (C), when switching from one slice stack (A), to the next (B).

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

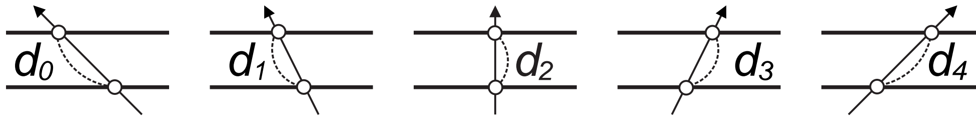


Figure 3.6: The distance between adjacent sampling points depends on the viewing angle.

values, it must also be applied to the respective RGB colors, if these are stored as opacity-weighted colors, which usually is the case [112].

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 [83].

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.

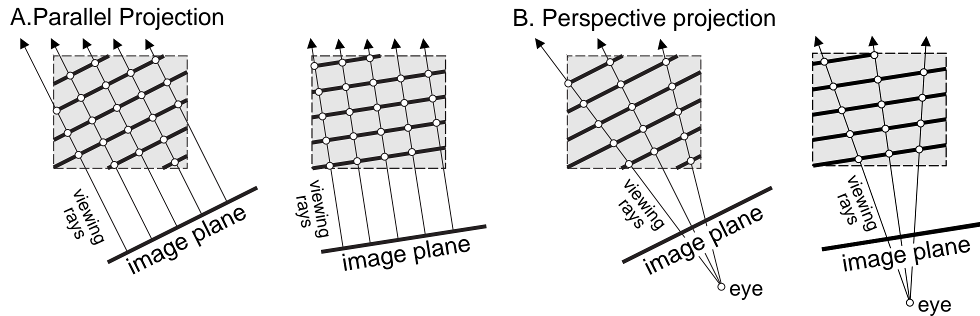


Figure 3.7: Sampling locations on view-aligned slices for parallel (A), and perspective projection (B), respectively.

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 [65]. 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 [21], 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* [112], or *associated* [9] 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 28

Real-Time Volume Graphics

GPU-Based Ray Casting

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Introduction to Ray Casting

The previous Part on "GPU-Based Volume Rendering" describes a traditional and widely used approach to volume rendering on graphics hardware. This approach uses a 2D proxy geometry to sample the underlying 3D data set. The predominant proxy geometries are either view-aligned slices (through a 3D texture) or axis-aligned slices (oriented along a stack of 2D textures). Slice-based volume rendering owes its success to a number of important reasons: (1) a high bandwidth between texture memory and rasterization unit, (2) built-in interpolation methods for a fast resampling (bilinear for stacks of 2D textures or trilinear in 3D textures), and (3) a high rasterization performance. Moreover, the core rendering routines are quite simple to implement.

Despite of these benefits, slice-based volume rendering has a number of significant disadvantages—especially for large data sets. As the number and the position of the slices are directly determined by the volume data set, this object-space approach is strongly influenced by the complexity of the data set. Output sensitivity, however, should be the ultimate goal of any computer graphics algorithm. An image-space approach that takes into account the complexity of the generated image may come closer to this goal. In volume rendering the overhead for a naive object-space technique can be quite large because a significant number of fragments does not contribute to the final image. Typically, only 0.2% to 4% of all fragments are visible [61]. Most volume rendering applications focus on visualizing boundaries of objects or selected material regions. Therefore, large parts of a volume data set are set completely transparent and are not visible. In addition, many of the remaining fragments are invisible due to occlusion effects. Figures 5.1 and 5.2 show two typical examples for volume visualization that contain only a small fraction of visible fragments.

Ray casting [68] is a well-known approach to address these issues. As outlined in Part II ("GPU-Based Volume Rendering"), ray casting

represents a direct numerical evaluation of the volume rendering integral. For each pixel in the image, a single ray is cast into the volume (neglecting possible super sampling on the image plane). Then the volume data is resampled at discrete positions along the ray. Figure 5.3 illustrates ray casting. By means of the transfer function the scalar data values are mapped to optical properties that are the basis for accumulating light information along the ray. The back-to-front compositing scheme,

$$C_{\text{dst}} = (1 - \alpha_{\text{src}})C_{\text{dst}} + C_{\text{src}} \quad ,$$

is changed into the front-to-back compositing equation,

$$\begin{aligned} C_{\text{dst}} &= C_{\text{dst}} + (1 - \alpha_{\text{dst}})C_{\text{src}} \\ \alpha_{\text{dst}} &= \alpha_{\text{dst}} + (1 - \alpha_{\text{dst}})\alpha_{\text{src}} \quad . \end{aligned} \tag{5.1}$$

In this way, compositing can be performed in the same order as the ray traversal.

The ray casting algorithm can be split into the following major parts. First, a light ray needs to be set up according to given camera parameters and the respective pixel position. Then, the ray has to be traversed by stepping along the ray, i.e., a loop needs to be implemented. Optical properties are accumulated within this loop. Finally, this process has to stop when the volume is traversed.

Ray casting can easily incorporate a number of acceleration techniques to overcome the aforementioned problems of slice-based volume rendering. Early ray termination allows us to truncate light rays as soon as we know that volume elements further away from the camera are occluded. Ray traversal can be stopped when the accumulated opacity α_{dst} reaches a certain user-specified limit (which is typically very close to 1). Another advantage is that the step sizes for one ray can be chosen independently from other rays, e.g., empty regions can be completely skipped or uniform regions can be quickly traversed by using large step sizes.

All these aspects are well-known and have been frequently used in CPU-based implementations of ray casting. The main goal of this part of the course is to describe how ray casting and its acceleration techniques can be realized on a GPU architecture. Ray casting exhibits an intrinsic parallelism in the form of completely independent light rays. This parallelism can be easily mapped to GPU processing, for example, by associating the operations for a single ray with a single pixel. In this way, the built-in parallelism for a GPU's fragment processing (multiple pixel pipelines) is used to achieve efficient ray casting. In addition,

volume data and other information can be stored in textures and thus accessed with the high internal bandwidth of a GPU.

A GPU implementation of ray casting, however, has to address some issues that are not present on CPUs. First, loops are not supported in fragment programs. Therefore, the traversal of a ray needs to be initiated by a CPU-based program. Second, it is hard to implement conditional breaks, which are required to stop ray traversal. This problem is particularly important when acceleration techniques are implemented.

We discuss GPU-based ray casting in the following two chapters. Chapter 6 focuses on ray casting for uniform grids, while Chapter 7 describes ray casting for tetrahedral grids.

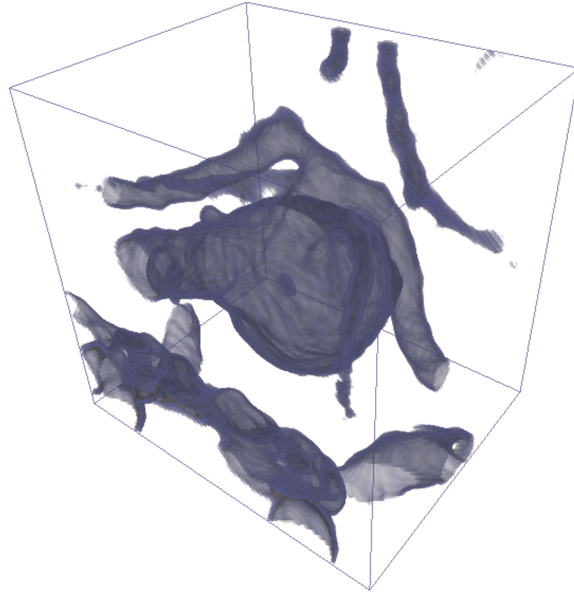


Figure 5.1: Volume rendering of an aneurysm data set.

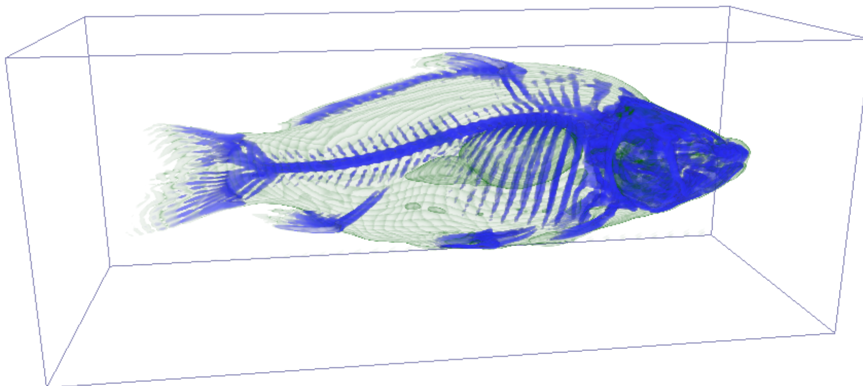


Figure 5.2: Volume rendering of the CT scan of a carp.

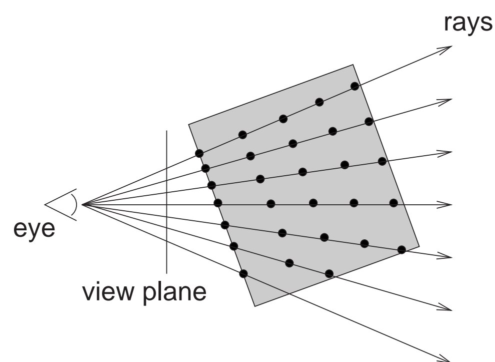


Figure 5.3: Ray casting. For each pixel, one viewing ray is traced. The ray is sampled at discrete positions to evaluate the volume rendering integral.

Ray Casting in Uniform Grids

GPU Ray Casting with Early Ray Termination

Uniform grids have a simple geometric and topological structure. An early implementation of GPU-based ray casting for uniform grids was published by Röttger et al. [84]; a similar approach is taken by Krüger and Westermann [61]. These two papers are the basis for the description in this chapter. Both approaches need the functionality of Pixel Shader 2.0 (DirectX) or comparable functionality via OpenGL fragment programs; both implementations were tested on an ATI Radeon 9700.

We discuss the two main ingredients of GPU-based ray casting: (a) data storage and (b) fragment processing. In graphics hardware data can be stored in, and efficiently accessed from, textures. The data set (its scalar values and, possibly, its gradients) are held in a 3D texture in the same way as for rendering with view-aligned slices (see Part II). In addition, ray casting needs intermediate data that is "attached" to light rays. Due to the direct correspondence between ray and pixel, intermediate data is organized in 2D textures with a one-to-one mapping between texels and pixels on the image plane.

Accumulated colors and opacities are represented in such 2D textures to implement the front-to-back compositing equation (5.1). In addition, the current sampling position along a ray can be stored in an intermediate texture. Typically, just the 1D ray parameter is held, i.e., the length between entry point into the volume and current position [84]. The mentioned intermediate values are continuously updated during ray traversal. Since OpenGL and DirectX have no specification for a simultaneous read and write access to textures, a ping-pong scheme makes such an update possible. Two copies of a texture are used; one texture holds the data from the previous sample position and allows for a read access while the other texture is updated by a write access. The roles of the two textures are exchanged after each iteration. Textures can be efficiently modified via the render-to-texture functionality of DirectX or

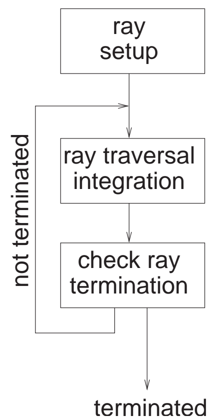


Figure 6.1: Principal structure of multi-pass rendering for GPU ray casting.

the `WGL_ARB_render_texture` support for OpenGL under Windows.

Some intermediate parameters can either be stored in 2D textures or, alternatively, computed on-the-fly within a fragment program. For example, the ray direction can computed once and then stored in a texture (as in [61]) or computed on-the-fly (as in [84]). The ray direction can determined by taking the normalized difference vector between exit point and entry point, or the normalized difference between entry point and camera position.

Other important aspects deal with the implementation of the fragment programs. Different programs are used in different parts of a multi-pass rendering approach. Figure 6.1 shows the principal structure of multi-pass rendering for GPU ray casting.

Ray setup is performed once per frame to compute the initial parameters for the rays, i.e., the entry points into the cube-shaped volume and the direction of the rays (in [61]) or the initial ray parameter (in [84]). The entry points are determined by rendering the frontfaces of the volume and attaching the positions to the vertices. Scanline conversion fills in-between values via interpolation.

Ray traversal and integration are performed via multiple render passes. Fragment are once again generated by rendering the boundary polygons of the volume (here, the frontfaces). The new position is computed by shifting the previous position along the ray direction according to the integration step size. The contribution of this ray segment (between previous and subsequent position) is accumulated according to the compositing equation (5.1). The source color C_{src} and source opacity α_{src} can be obtained by evaluating the transfer function at the sample

points [61] (in the same way as for slice-based rendering from Part II). Alternatively, the integrated contribution of the ray segment between the two points can be taken into account [84]. Here, a linear interpolation of scalar values is assumed between the two points, and the contribution to the volume rendering integral can be obtained from a pre-computed lookup-table. We refer to Part VII for a detailed description of this *pre-integration* approach. For the following discussion it is sufficient to understand that we have a 2D lookup-table that provides the color C_{src} and opacity α_{src} of a ray segment. The two parameters for the lookup-table are the scalar values at the start point and end point of the segment. Regardless whether pre-integration or point sampling are used, a ping-pong scheme is employed to iteratively update the colors and opacities along the rays.

Another important aspect is the implementation of the stopping criterion. Early ray termination leads to cancellation of ray traversal as soon as a certain opacity level is reached. In addition, ray traversal has to end when the ray leaves the volume. This ray termination cannot be included in the above fragment program for ray traversal and integration because GPUs do not support a conditional break for this iteration loop. Therefore, ray termination is implemented in another fragment program that is executed after the shader for traversal and integration (see Figure 6.1). Actual ray termination is implemented by using a depth test and setting the z buffer accordingly, i.e., the z value is specified in a way to reject fragments that correspond to terminated rays. In addition to the termination of single rays on a pixel-by-pixel basis, the whole iteration process (the outer loop in multi-pass rendering) has to stop when all rays are terminated. An asynchronous occlusion query allows us to check how many fragments are actually drawn in a render process; multi-pass rendering is stopped when the occlusion query reports no drawn fragments [84]. An alternative solution is based on the early z-test (as used in [61]). The maximum number of render passes has to be known beforehand, e.g., by computing the worst-case number of sampling steps. Here, all sampling steps are always initiated by multi-pass rendering. Fragments that correspond to terminated rays, however, are rejected by the efficient early z-test. In this way, the time-consuming fragment programs are skipped for terminated rays.

In the remainder of this section, the two implementations [61, 84] are described separately. The approach by Röttger et al. [84] is realized as follows. Three floating-point textures, with two components each, are used for intermediate values: the first texture for accumulated RG values, the second texture for accumulated BA values, and the third texture for

the single-component ray parameter (i.e., the other component is not used). In all rendering passes, fragments are generated by rendering the frontfaces of the volume box. Ray setup and the first integration step are combined in a single render pass (and therefore in a combined fragment program). The implementation uses multiple render targets to update the three aforementioned 2D textures.

A second fragment program realizes a single ray traversal and integration step. Non-normalized ray directions are attached as texture coordinates to the vertices of the volume's bounding box and interpolated during scanline conversion. Actual direction vectors are determined within the fragment program by normalizing to unit length. The current sampling position is computed from the direction vector and the ray parameter (as fetched from the intermediate texture). The subsequent position is obtained by adding the fixed traversal step size. A 3D texture lookup in the data set provides the scalar values at the two endpoints of the current ray segment. These two scalar values serve as parameters for a dependent lookup in the pre-integration table. If (optional) volume illumination is switched on, the gradients included in the 3D data texture are used to evaluate the local illumination model (typically Phong or Blinn-Phong). Finally, all contributions are added and combined with the previously accumulated RGBA values according to the compositing equation (5.1).

A third fragment program is responsible for ray termination. It is executed after each traversal step. The ray termination program checks whether the ray has left the volume or the accumulated opacity has reached its limit. When a ray is terminated, the z buffer is set so that the corresponding pixel is no longer processed (in any future rendering pass for traversal or ray termination).

An asynchronous occlusion query is applied to find out when all rays are terminated. A total number of $2n + 1$ render passes is required if n describes the maximum number of samples along a single ray.

Krüger and Westermann [61] use a similar algorithmic structure, but include some differences in their implementation. Their ray setup is distributed over two different rendering passes. The first pass determines the entry points into the volume. The frontfaces of the volume box are rendered, with the 3D positions attached to the vertices. In-between positions are obtained via interpolation during rasterization. The 3D positions are written to a 2D render texture that we denote "POS". The second pass computes the ray direction by taking the normalized difference between exit and entry points (into or out of the volume). Only in this second pass, fragments are generated by rendering the backfaces

of the volume box. The 3D positions of the exit points are attached to the vertices and interpolated during scanline conversion. The entry points are read from the texture "POS". In addition to the ray direction, the length of each ray is computed in stored in a 2D render texture denoted "DIR".

The main loop iterates over two different fragment programs. In these rendering passes, fragments are generated by drawing the frontfaces of the volume. The first program implements ray traversal and integration. Each render pass samples m steps along the ray by partially rolling out the traversal loop within the fragment program. This avoids some data transfer between fragment program and textures (for the accumulated RGBA values) and therefore increases the rendering performance. In addition, the traversal shader checks whether the ray has left the volume, based on the length of a ray that is read from the texture "DIR". If a ray has left the volume, opacity is set to a value of one. The second shader program implements ray termination. Here, opacity is checked against a given threshold. If the ray is ended (due to early ray termination or because it has left the volume), the z buffer is set so that the corresponding pixel is no longer processed.

Krüger and Westermann [61] use a fixed number of rendering passes and do not employ an occlusion query. The number of passes depends on the maximum length of a ray, the traversal step size, and the number of intermediate steps m. Due to the efficient early z-test, the overhead for possibly unnecessary render passes is small.

Adaptive Sampling

A typical volume data set has different regions with different characteristics. On the one hand, there can be largely uniform, or even completely empty, regions in which a fine sampling of the ray integral is not necessary. On the other hand, details at boundaries between different regions should be represented by an accurate sampling of the ray. To address this issues, we describe the adaptive sampling approach taken by Röttger et al. [84].

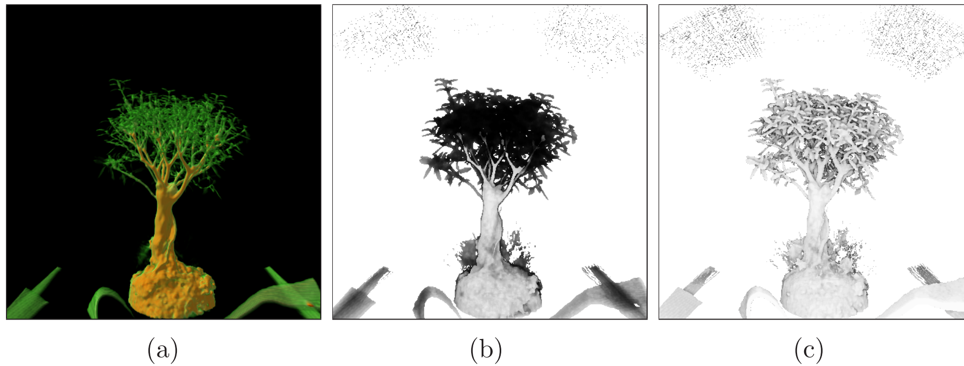
Adaptive sampling relies on an additional data structure that controls the space-variant sampling rate. This *importance volume* describes the maximum isotropic sampling distance, and is computed from a user-specified error tolerance and the local variations of the scalar data set. The importance volume is stored in a 3D texture whose resolution can be chosen independently from the resolution of the scalar data set.

The ray casting algorithm from the previous section needs only a

slight modification to incorporate adaptive sampling. During ray traversal, an additional 3D texture lookup in the importance volume (at the current position) yields the step size for the following iteration. This space-variant step size is now used to compute the next sampling point—instead of a fixed step distance.

This approach relies on pre-integration to determine a segment's contribution to the volume rendering integral. A key feature of pre-integration is its separation of the sampling criteria for the data set and the transfer function [55]. Without pre-integration, the rate for an accurate sampling of the volume rendering integral has to be based on the variations of the RGBA values that result from a mapping of scalar values via the transfer function. If, for example, the scalar values vary only very smoothly and slowly, but the transfer function contains very high frequencies, a high overall sampling rate will have to be chosen for the volume rendering integral. In other words, the sampling rate has to take into account the frequencies of the data set and the transfer function. In contrast, pre-integration "absorbs" the effect of the transfer function by means of a pre-computed lookup-table. Accordingly, only the spatial variations of the data set have to taken into account for an accurate sampling. Therefore, the construction of the importance volume makes use of the structure of the data set only, and does not consider the transfer function. A benefit of this approach is that the importance volume is fixed for a stationary data set, i.e., it is computed only once (by the CPU) and downloaded to the GPU. Unfortunately, another potential case for optimization is not taken into account: If the volume rendering integral contains only low frequencies because of the effect of the simple structure of the chosen transfer function, the sampling rate will be unnecessarily high for a data set containing higher frequencies.

Figure 6.2 shows an example image generated by ray casting with adaptive sampling. Image (a) depicts the original volume visualization. Image (b) visualizes the corresponding number of sampling steps. Due to adaptive sampling, only few steps are needed for the uniform, empty space around the bonsai. Early ray termination reduces the number of samples in the region of the opaque trunk. In the leave's region, the importance volume indicates that a fine sampling is required (due to strongly varying scalar data values). Therefore, many sampling steps are used for this part of the image. Figure 6.2 (c) visualizes the number of sampling steps if a more opaque transfer function is applied to the same data set and under the same viewing conditions. The reduced number of samples for the leaves is striking. This effect is caused by early ray termination: After the first hit on an opaque leaf, a ray is ended.



(a) (b) (c)

Figure 6.2: Comparison of the number of rendering steps for different transfer functions. Image (a) shows the original volume visualization, image (b) the corresponding number of sampling steps (black corresponds to 512 samples). Image (c) visualizes the number of sampling steps for a more opaque transfer function; this illustrates the effect of early ray termination.

Empty-Space Skipping

Empty-space skipping is useful when a volume visualization contains large portions of completely transparent space. Transparency is determined after the transfer function was applied to the data set. An additional data structure is used to identify empty regions. For example, an octree hierarchy can be employed to store the minimum and maximum scalar data values within a node. In combination with the transfer function, these min/max values allow us to determine completely transparent nodes.

Krüger and Westermann [61] reduce the octree to just a single level of resolution—in their implementation to $(1/8)^3$ of the size of the scalar data set. This reduced "octree" can be represented by a 3D texture, with the R and G components holding the minimum and maximum values for each node. Empty regions can only be identified by applying the transfer function to the original data and, afterwards, checking whether the data is rendered visible. We need a mapping from the min/max values to a Boolean value that classifies the node as empty or not. This two-parameter function is realized by a 2D texture that indicates for each min/max pair whether there is at least one non-zero component in the transfer function in the range between minimum and maximum scalar value. This 2D empty-space table depends on the transfer function and has to be updated whenever the transfer function is changed. The empty-space table is computed on the CPU and then uploaded to the GPU. Note that the "octree" 3D texture depends on the data only and

has to be generated just once for a stationary data set.

The fragment program for ray termination is extended to take into account empty-space skipping (cf. the discussion at the end of the first section). The frontfaces of the volume are rendered in the same way as before, while the step size is increased according to the size of the "octree" structure. Due to the larger step size, the number of traversal iterations is decreased. The min/max values are sampled from the "octree" and serve as parameters for a dependent texture lookup in the empty-space table. Empty space is skipped by setting the z value of the depth buffer to the maximum; the code for the fine-grained sampling of the scalar data set is skipped by the early z-test. Conversely, a non-empty node leads to a z value of zero and a subsequent integration of the ray segment. The z value is reset to zero as soon as a non-empty node is found (and if the opacity is still below the threshold for early ray termination).

Ray Casting in Tetrahedral Grids

Unstructured grids are widely used in numerical simulations to discretize the computational domain. Their resolution can be locally adapted to achieve a required numerical accuracy, while minimizing the total number of grid cells. For example, unstructured grids are popular in applications like computational fluid dynamics (CFD). Although unstructured grids may contain a variety of different cell types (such as tetrahedra, hexahedra, or prisms), these grids can always be decomposed into a collection of tetrahedra. Therefore, tetrahedral meshes are the most important type of unstructured grids.

Volume rendering of tetrahedral meshes is traditionally implemented on graphics hardware by means of cell projection, e.g., according to Shirley and Tuchman [94]. Unfortunately, cell projection with non-commutative blending requires a view-dependent depth sorting of cells, which still has to be performed on the CPU. Whenever the camera or the volume is moved, new graphical primitives have to be generated by the CPU and transferred to the GPU. Therefore, cell projection benefits only in parts from the performance increase of GPUs. Another problem of cell projection is that cyclic meshes require special treatment [56]. With the R-buffer architecture [113, 50] order-independent cell projection could be achieved; however, the R-buffer has not been realized yet.

Fortunately, ray casting for tetrahedral meshes can overcome these problems. This chapter describes a ray casting approach that can be completely mapped to the GPU. This approach was proposed by Weiler et al. [102]. A re-print of this paper is included in the course notes. We refer to this paper for details of the implementation and some additional background information. Similarly to ray casting in uniform grids (see previous chapter), the algorithm can be readily parallelized because the operations for each ray are independent. A ray is once again identified with its corresponding pixel on the image plane. Therefore, rays can be processed in parallel on the GPU by mapping the ray casting operations

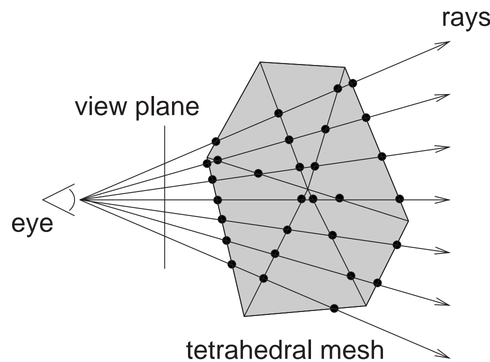


Figure 7.1: Ray traversal in a tetrahedral grid. For each pixel, one viewing ray is traced. The ray is sampled at all intersected cell faces (black dots).

to fragment programs. The rendering performance can additionally be increased by early ray termination.

Basic Structure of the Algorithm

GPU ray casting is based on a ray propagation approach similar to the CPU approach by Garrity [27]. Figure 7.1 illustrates how each viewing ray is propagated in a front-to-back fashion from cell to cell until the whole grid has been traversed. The traversal follows the links between neighboring cells. A ray begins at its first intersection with the mesh, which is determined during an initialization phase.

The traversal is performed in multiple render passes. In each pass the color and opacity contribution of a pixel's current cell is computed analogously to the GPU-based view-independent cell projection by Weiler et al. [104]. Pre-integrated volume rendering is used to determine the contribution of a ray segment within a tetrahedron [86]. Finally, these contributions are accumulated according to the front-to-back compositing equation (5.1). The convexification approach by Williams [110] is used to allow for re-entries of viewing rays in non-convex meshes. Convexification converts non-convex meshes into convex meshes by filling the empty space between the boundary of the mesh and a convex hull of the mesh with imaginary cells.

Although ray casting for uniform grids and tetrahedral grids are strongly related to each other, some important difference have to be taken into account to process tetrahedral meshes. First, the topolog-

ical information about the connectivity of neighboring cells has to be stored, i.e., more complex data structures have to be handled. Second, ray traversal samples the volume at entry/exit points of cells, i.e., intersections between ray and cells need to be computed.

Fragment programs are used to perform all computations for the ray propagation. Fragments are generated by rendering screen-filling rectangles. Each rendering pass executes one propagation step for each viewing ray. The whole mesh is processed by stepping through the volume in multiple passes. The algorithm consists of the following steps:

- Ray setup (initialization)
- While within the mesh:
  - ▷ Compute exit point for current cell
  - ▷ Determine scalar value at exit point
  - ▷ Compute ray integral within current cell via pre-integration
  - ▷ Accumulate colors and opacities by blending
  - ▷ Proceed to adjacent cell through exit point

The algorithm starts by initializing the first intersection of the viewing ray, i.e., an intersection with one of the boundary faces of the mesh. This can be implemented using a rasterization of the visible boundary faces, similarly to ray casting in uniform grids. However, it may also be performed on the CPU as there are usually far less boundary faces than cells in a mesh, and thus, this step is not time critical.

The remaining steps can be divided into the handling of ray integration and ray traversal. These steps have to transfer intermediate information between successive rendering passes. This intermediate information is represented by several 2D RGBA textures that have a one-to-one mapping between texels and pixels on the image plane. The textures contain the current intersection point of the ray with the face of a cell, the index of the cell the ray is about to enter through this face (including the index of the entry face), and the accumulated RGBA values.

The intermediate textures are read and updated during each rendering pass. Since OpenGL and DirectX have no specification for a simultaneous read and write access to textures, a ping-pong scheme makes such an update possible. Two copies of a texture are used; one texture holds the data from the previous sample position and allows for a read access while the other texture is updated by a write access. The roles of the two textures are exchanged after each iteration. Textures can be efficiently modified via the render-to-texture functionality of DirectX or the `WGL_ARB_render_texture` support for OpenGL under Windows.

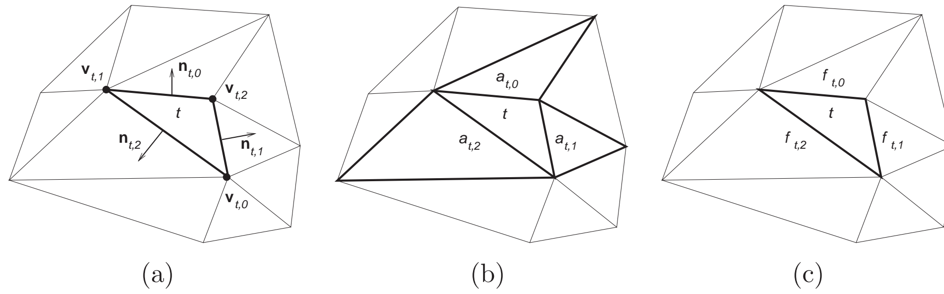


Figure 7.2: Terminology for the mesh representation. (a) The vertex $\mathbf{v}_{t,i}$ is opposite to the i-th face of cell t; the normal $\mathbf{n}_{t,i}$ is perpendicular to the i-th face. (b) The neighboring cell $a_{t,i}$ shares the i-th face. (c) The i-th face of t corresponds to the $f_{t,i}$-th face of t's neighbor $a_{t,i}$.

Mesh Representation

Before we discuss ray integration and ray traversal in more detail, we would like to show how the rather complex mesh data can be represented on the GPU. A tetrahedral mesh contains information on topology (neighboring cells), geometry (position of vertices, normal vectors), and scalar data values. Figure 7.2 illustrates how this information is attached to a mesh. Cells are labelled by an integer index t that ranges from 0 to $n-1$, where n is the number of cells in the mesh. Each tetrahedron t has four faces. The normal vectors on the faces are labelled $\mathbf{n}_{t,i}$, where $i \in \{0, 1, 2, 3\}$ specifies the face. Normal vectors are assumed to point outwards. The four vertices of a tetrahedron t are denoted $\mathbf{v}_{t,i}$; vertex $\mathbf{v}_{t,i}$ is opposite to the i-th face. The neighbor of a tetrahedron t that is adjacent to the i-th face is labelled $a_{t,i}$. The index of the face of $a_{t,i}$ that corresponds to the i-th face of t is called $f_{t,i}$.

In addition to the structure of the mesh, the data values play an important role. The scalar field value $s(\mathbf{x})$ at a point $\mathbf{x}$ can be computed by

$$s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{x}_0) + s(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + (-\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)) \, . \tag{7.1}$$

The gradient of the scalar field, $\mathbf{g}_t$, is constant within a cell because a linear, barycentric interpolation is assumed. The advantage of this representation is that the scalar values inside a cell can be efficiently reconstructed by computing one dot product and one scalar addition, while we still need to store only one vector $\mathbf{g}_t$ and one scalar $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)$ for each cell ($\mathbf{x}_0$ is the position of an arbitrary vertex of the cell).

Table 7.1: Mesh data represented by textures.

data in texture	tex coords			texture data			
	u	v	w	r	g	b	α
vertices	t		i	$\mathbf{v}_{t,i}$			—
face normals	t		i	$\mathbf{n}_{t,i}$			$f_{t,i}$
neighbor data	t		i	$a_{t,i}$		—	—
scalar data	t		—	$\mathbf{g}_t$			$\hat{g}_t$

The mesh data is stored in 2D and 3D RGBA textures at floating-point resolution. Since the mesh data is constant for a stationary data set, these textures are generated in a pre-processing step on the CPU. Table 7.1 gives an overview of this texture representation. Cell indices are encoded in two texture coordinates because their values can exceed the range of a single texture coordinate. 3D textures are used for vertices, face normals, and neighbor data; a 2D texture is used for the scalar data. The textures are accessed via the cell index. For 3D textures the additional w coordinate represents the index of the vertex, face, or neighbor.

Ray Integration and Cell Traversal

Integration along a whole ray is split into a collection of integrations within single cells. The evaluation of the volume rendering integral within a cell can be efficiently and accurately handled by pre-integration [86]. We refer to Part VII for a detailed description of the pre-integration approach. For the following discussion it is sufficient to understand that we have a 3D lookup-table that provides the color C_{src} and opacity α_{src} of a ray segment. The parameters for the lookup-table are the scalar values at the entry point and exit point of the segment, as well as the length of the segment.

The entry point and its scalar value are communicated via the intermediate 2D textures. In addition, the index of the current cell is given in these textures. The exit point is computed by determining the intersection points between the ray and the cell's faces and taking the intersection point that is closest to the eye (but not on a visible face). We denote the index for the entry face by j, the position of the eye by $\mathbf{e}$, and the normalized direction of the viewing ray by $\mathbf{r}$. Then the three intersection points with the faces of cell t are $\mathbf{e} + \lambda_i \mathbf{r}$, where $0 \leq i < 4 \wedge i \neq j$

and

$$\lambda_i = \frac{(\mathbf{v} - \mathbf{e}) \cdot \mathbf{n}_{t,i}}{\mathbf{r} \cdot \mathbf{n}_{t,i}}, \quad \text{with} \quad \mathbf{v} = \mathbf{v}_{t,3-i} \quad .$$

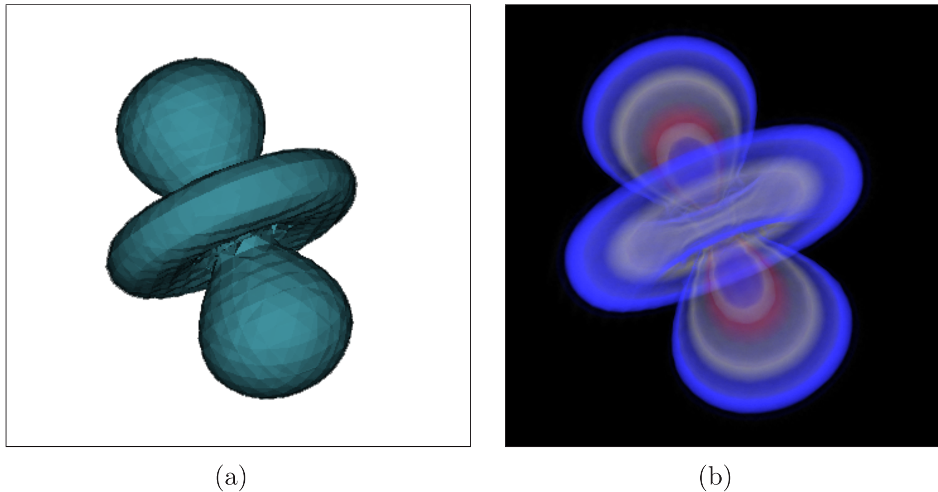
Note that no intersection is checked for the entry face j because this intersection point is already known. A face is visible and its corresponding intersection point should be discarded when the denominator in the above equation is negative. The minimum of the values λ_i is computed in the fragment program to determine the exit point.

The exit point is used to calculate the corresponding scalar value according to Equation (7.1). Also, the distance between exit and entry point is determined. With the length of the ray segment and the two scalar values at the end points of the segment, a lookup in the 3D pre-integration table yields the color C_{src} and opacity α_{src} of this segment. Finally, this RGBA contribution is accumulated according to the compositing equation (5.1).

The traversal of the whole mesh is guided by the current cell index stored in the intermediate textures. The fragment program takes the current index and updates each texel of the texture with the index of the cell adjacent to the face through which the viewing ray leaves the current cell. This index is given by $a_{t,i}$ for the current cell t, where i is the index that corresponds to the exit face. Boundary cells are represented by an index -1, which allows us to determine whether a viewing ray has left the mesh. This approach is only valid for convex meshes, where no "re-entries" into the mesh are possible. Therefore, non-convex meshes are converted into convex meshes by filling the empty space between the boundary of the mesh and a convex hull of the mesh with imaginary cells during a preprocessing step [110]. The current cell index is also used to implement early ray termination: The index is set to -1 when the accumulated opacity has reached the user-specified threshold.

Another important aspect is the implementation of ray termination. Two issues have to be taken into account. First, for each fragment we have to detect whether the corresponding ray is terminated (due to early ray termination or because the ray has left the volume). Second, the whole render loop has to be stopped when all rays are terminated. Note that the same problems are discussed in Chapter 6 on ray casting in uniform grids.

Ray termination is realized by using a depth test and setting the z buffer accordingly, i.e., the z value is specified in a way to reject fragments that correspond to terminated rays. Actual ray termination is implemented in another fragment program that is executed after the shader for traversal and integration. The ray termination shader checks



(a) (b)

Figure 7.3: GPU-based ray casting in a tetrahedral grid. Image (a) shows the isosurface of an orbital data set, image (b) a semi-transparent volume rendering.

whether the current index is -1. In this case, the z value is set to a value that prevents further updates of the corresponding pixel, i.e., subsequent executions of the shader for ray integration and cell traversal are blocked by the efficient early z-test.

An asynchronous occlusion query is employed to determine when all rays have been terminated. The asynchronous delivery of the occlusion query result leads to some additional rendering passes. This effect, however, can be neglected compared to the delay caused by waiting for the result.

Example Images

Figure 7.3 shows example images generated by GPU-based ray casting in a tetrahedral grid. Image (a) depicts the isosurface of an orbital data set, image (b) a semi-transparent volume rendering.

Acknowledgments

Special thanks to Manfred Weiler for proof-reading and fruitful discussions.

Course Notes 28
Real-Time Volume Graphics

Local Illumination

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

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 [82, 7], 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,

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

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

$$\nabla f = (f_x, \, f_y, \, f_z) = (\frac{\delta}{\delta x} f, \, \frac{\delta}{\delta y} f, \, \frac{\delta}{\delta z} f), \tag{8.1}$$

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,

$$||\nabla f|| = \sqrt{{f_x}^2 + {f_y}^2 + {f_z}^2}. \tag{8.2}$$

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

$$f_x(x,y,z) \;=\; f(x+1,y,z) - f(x-1,y,z) \quad \text{with} \quad x,y,z \in \mathbb{N}. \quad (8.3)$$

Derivatives in the other directions are computed analogously. Central differences are usually the method of choice for gradient precomputation. 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.

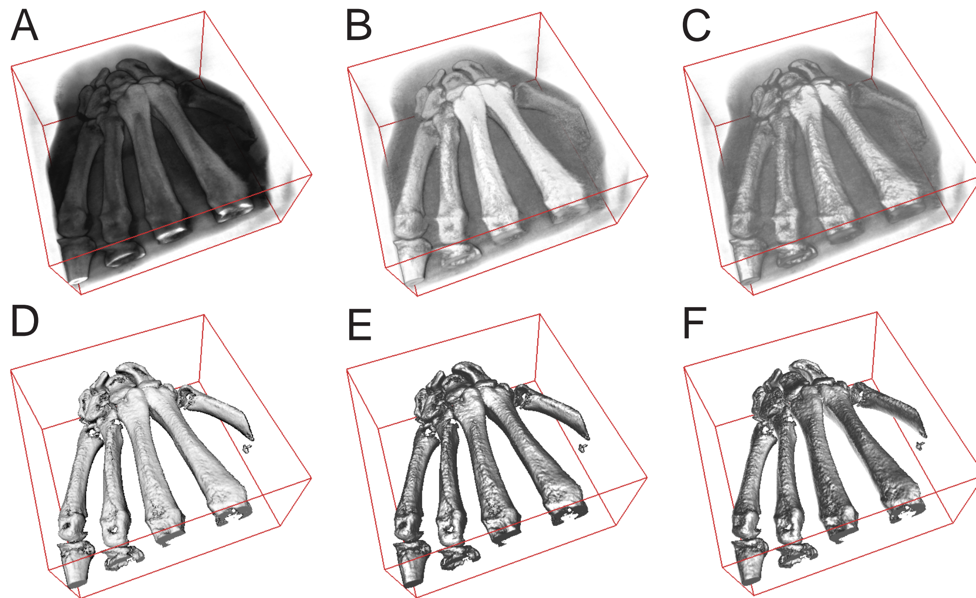


Figure 8.1: CT data of a human hand without illumination (A), with diffuse illumination (B) and with specular illumination (C). Non-polygonal isosurfaces with diffuse (D), specular (C) and diffuse and specular (E) illumination.

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 8.1 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 9.1 (*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:

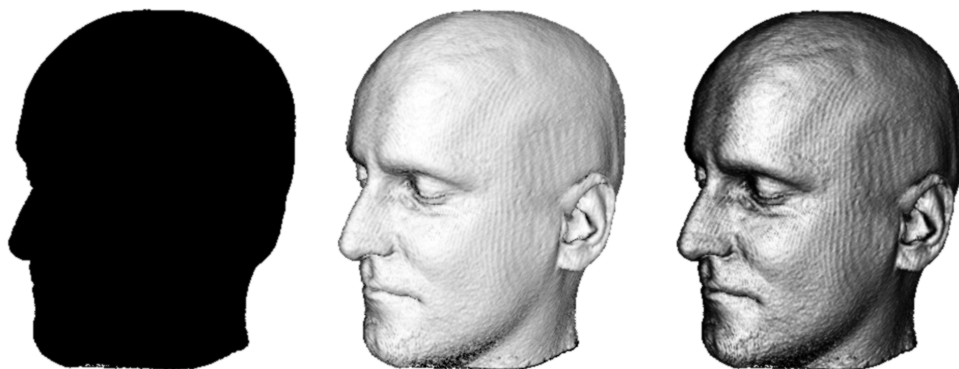


Figure 9.1: Non-polygonal isosurface without illumination (left), with diffuse illumination (middle) and with specular light (right)

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

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

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 [107]. The algorithm was expanded to volume shading my Meissner et al [77]. Efficient implementations on PC hardware are described in [83].

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 [8]. The term *environment mapping* was coined by Greene [32] 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 10.1: 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 10.1. 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 10.2 for example images of isosurface rendering with reflection mapping.

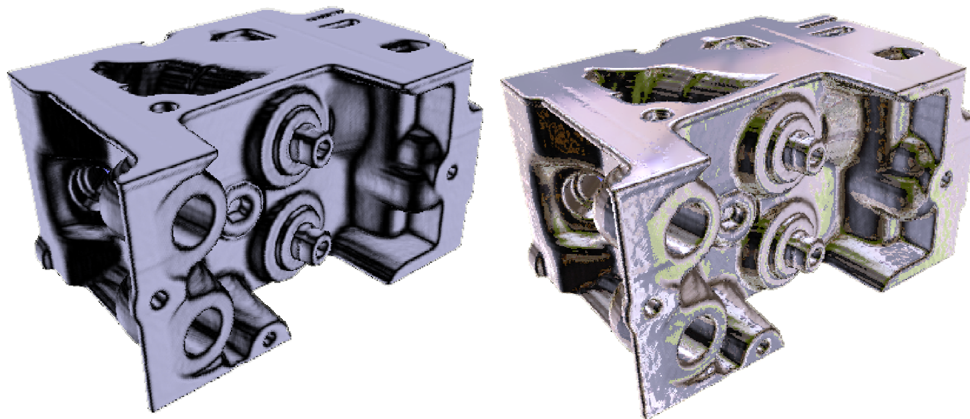


Figure 10.2: Isosurface of the engine block with diffuse reflection map (left) and specular environment map (right).

Deferred Shading

In standard rendering pipelines, shading equations are often evaluated for pixels that are entirely invisible or whose contribution to the final image is negligible. With the shading equations used in real-time rendering becoming more and more complex, avoiding these computations for invisible pixels becomes an important goal.

A very powerful concept that allows to compute shading only for actually visible pixels is the notion of *deferred shading*. Deferred shading computations are usually driven by one or more input images that contain all the information that is necessary for performing the final shading of the corresponding pixels. The major advantage of deferred

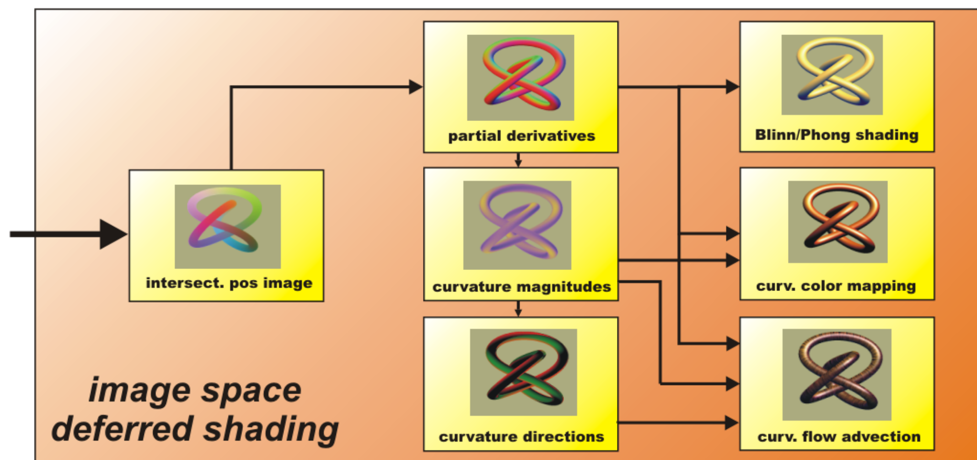


Figure 11.1: Deferred shading computations for an isosurface given as a floating point image of ray-surface intersection positions. First, differential properties such as the gradient and additional partial derivatives can be computed. These derivatives also allow to compute principal curvature information on-the-fly. In the final shading pass, the obtained properties can be used for high-quality shading computations. All of these computations and shading operations have image space instead of object space complexity and are only performed for visible pixels.

computations is that it reduces their complexity from being proportional to object space, e.g., the number of voxels in a volume, to the complexity of image space, i.e., the number of pixels in the final output image. Naturally, these computations are not limited to shading equations per se, but can also include the derivation of additional information that is only needed for visible pixels and may be required as input for shading, such as differential surface properties.

In this section, we describe deferred shading computations for rendering isosurfaces of volumetric data. The computations that are deferred to image space are not limited to actual shading, but also include the derivation of differential implicit surface properties such as the gradient (first partial derivatives), the Hessian (second partial derivatives), and principal curvature information.

Figure 11.1 shows a pipeline for deferred shading of isosurfaces of volume data. The input to the pipeline is a single floating point image storing ray-surface intersection positions of the viewing rays and the

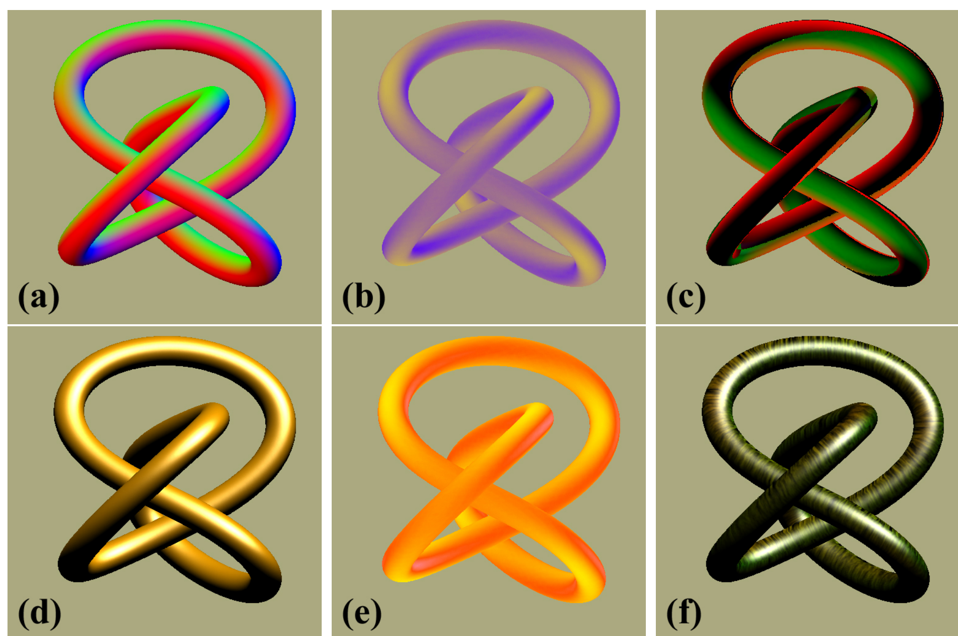


Figure 11.2: Example image space rendering passes of deferred isosurface shading. Surface properties such as (a) the gradient, (b) principal curvature magnitudes (here: κ_1), and (c) principal curvature directions can be reconstructed. These properties can be used in shading passes, e.g., (d) Blinn-Phong shading, (e) color coding of curvature measures (here: $\sqrt{\kappa_1^2 + \kappa_2^2}$), and (f) advection of flow along principal curvature directions.

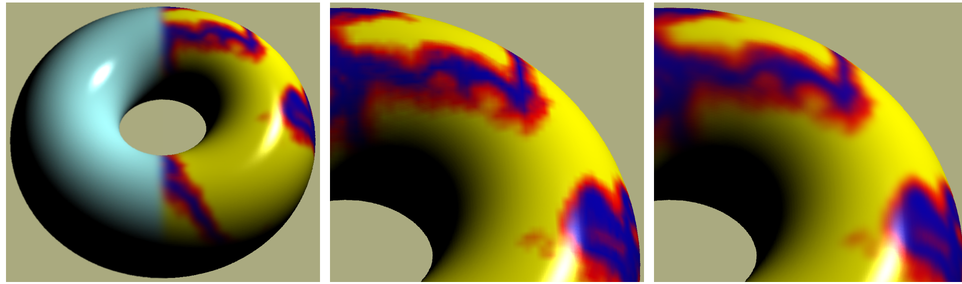


Figure 11.3: Deferred shading of an isosurface from a ray-isosurface intersection and a gradient image (left). Solid texture filtering can be done using tri-linear interpolation (center), or tri-cubic filtering in real-time (right).

isosurface. This image can be obtained via slicing the volume, e.g., using the traditional approach employing the hardware alpha test, or first hit ray casting using one of the recent approaches for hardware-accelerated ray casting.

From this intersection position image, differential isosurface properties such as the gradient and additional partial derivatives such as the Hessian can be computed first. This allows shading with high-quality gradients, as well as computation of high-quality principal curvature magnitude and direction information. Sections 12 and 13 describe high-quality reconstruction of differential isosurface properties.

In the final actual shading pass, differential surface properties can be used for shading computations such as Blinn-Phong shading, color mapping of curvature magnitudes, and flow advection along curvature directions, as well as applying a solid texture to the isosurface. Figure 11.2 shows example image space rendering passes of deferred isosurface rendering.

Shading from gradient image

The simplest shading equations depend on the normal vector of the isosurface, i.e., its normalized gradient. The normal vector can for example be used to compute Blinn-Phong shading, and reflection and refraction mapping that index an environment map with vectors computed from the view vector and the normal vector. See figure 11.3(left) for an example.

Solid texturing

The initial position image that contains ray-isosurface intersection positions can be used for straight-forward application of a solid texture onto an isosurface. Parameterization is simply done by specifying the transformation of object space to texture space coordinates, e.g., via an affine transformation. For solid texturing, real-time tri-cubic filtering can be used instead of tri-linear interpolation in order to achieve high-quality results. See figure 11.3(center and right) for a comparison.

Deferred Gradient Reconstruction

The most important differential property of the isosurface that needs to be reconstructed is the gradient of the underlying scalar field f:

$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^{\mathbf{T}} \tag{12.1}$$

The gradient can then be used as implicit surface normal for shading and 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 bounded by the isosurface: $\mathbf{n} = -\mathbf{g}/|\mathbf{g}|$. The calculated gradient can be stored in a single RGB floating point image, see figure 11.2(a).

Hardware-accelerated high-quality filtering can be used for reconstruction of high-quality gradients by convolving the original scalar volume with the first derivative of a reconstruction kernel, e.g., the derived cubic B-spline kernel shown in figure 12.1(a). The quality difference between cubic filtering and linear interpolation is even more apparent in gradient reconstruction than it is in value reconstruction. Figure 12.2

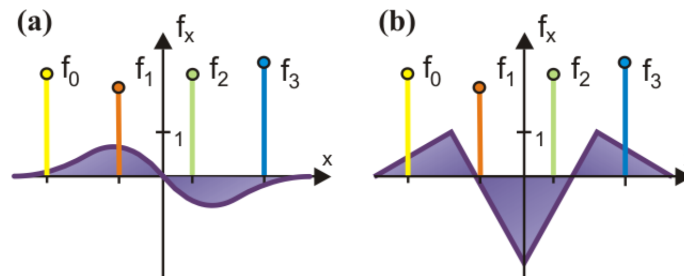


Figure 12.1: The first (a) and second (b) derivatives of the cubic B-spline filter for direct high-quality reconstruction of derivatives via convolution.

shows a comparison of different combinations of filters for value and gradient reconstruction, i.e., linear interpolation and cubic reconstruction with a cubic B-spline kernel. Figure 12.3 compares linear and cubic (B-spline) reconstruction using reflection mapping and a line pattern environment map. Reconstruction with the cubic B-spline achieves results with C^2 continuity.

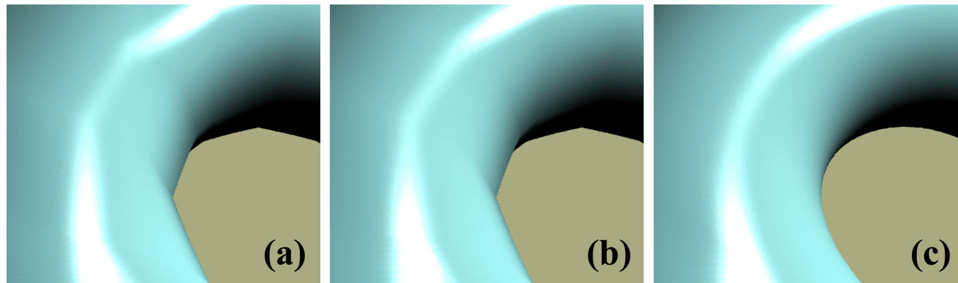


Figure 12.2: Linear and cubic filtering with a cubic B-spline kernel for value and gradient reconstruction on a torus: (a) both value and gradient are linear; (b) value is linear and gradient cubic; (c) both value and gradient are cubic.

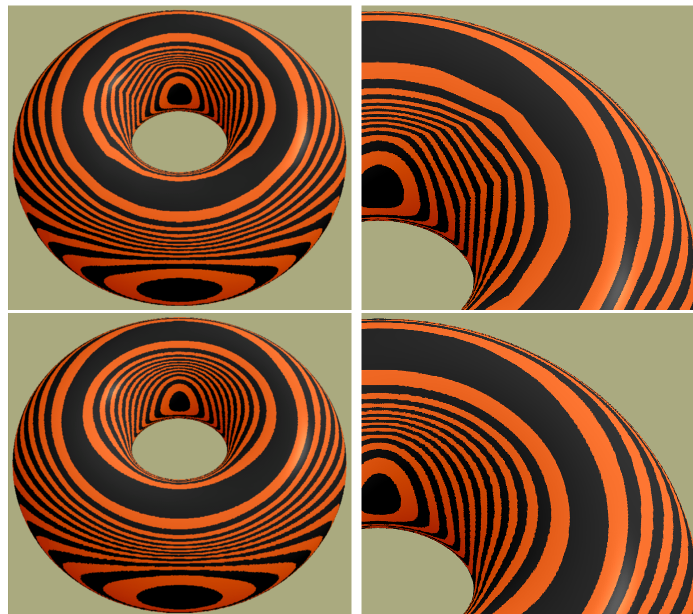


Figure 12.3: Comparing linear and cubic gradient reconstruction with a cubic B-spline using reflection lines. Top row images linear, bottom row cubic filtering.

Other Differential Properties

In addition to the gradient of the scalar volume, i.e., its first partial derivatives, further differential properties can be reconstructed in additional deferred shading passes.

For example, implicit principal curvature information can be computed from the second partial derivatives of the volume. Curvature has many applications in surface investigation and rendering, e.g., non-photorealistic rendering equations incorporating curvature magnitudes in order to detect surface structures such as ridge and valley lines, or rendering silhouettes of constant screen space thickness.

Second partial derivatives: the Hessian

The Hessian $\mathbf{H}$ is comprised of all second partial derivatives of the scalar volume f:

$$\mathbf{H} = \nabla \mathbf{g} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial^2 y} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial^2 z} \end{pmatrix} \tag{13.1}$$

Due to symmetry, only six unique components need to be calculated, which can be stored in three RGB floating point images.

High-quality second partial derivatives can be computed by convolving the scalar volume with a combination of first and second derivatives of the cubic B-spline kernel, for example, which is illustrated in figure 12.1.

Principal curvature magnitudes

The first and second principal curvature magnitudes (κ_1, κ_2) of the iso-surface can be estimated directly from the gradient $\mathbf{g}$ and the Hessian $\mathbf{H}$ [48], whereby tri-cubic filtering in general yields high-quality results. This can be done in a single rendering pass, which uses the three partial derivative RGB floating point images generated by previous pipeline

stages as input textures. It amounts to a moderate number of vector and matrix multiplications and solving a quadratic polynomial.

The result is a floating point image storing (κ_1, κ_2), which can then be used in the following passes for shading and optionally calculating curvature directions. See figure 11.2(b).

Principal curvature directions

The principal curvature magnitudes are the eigenvalues of a 2x2 eigensystem in the tangent plane specified by the normal vector, which can be solved in the next rendering pass for the corresponding eigenvectors, i.e., the 1D subspaces of principal curvature directions. Representative vectors for either the first or second principal directions can be computed in a single rendering pass.

The result is a floating point image storing principal curvature direction vectors. See Figure 11.2(c).

Filter kernel considerations

All curvature reconstructions in this chapter employ a cubic B-spline filter kernel and its derivatives. It has been shown that cubic filters are the lowest order reconstruction kernels for obtaining high-quality curvature estimates. They also perform very well when compared with filters of even higher order [48].

The B-Spline filter is a good choice for curvature reconstruction because it is the only fourth order BC-spline filter which is both accurate and continuous for first and second derivatives [79, 48]. Hence it is the only filter of this class which reconstructs continuous curvature estimates.

However, although B-spline filters produce smooth and visually pleasing results, they might be inappropriate in some applications where data interpolation is required [78]. Using a combination of the first and second derivatives of the cubic B-spline for derivative reconstruction, and a Catmull-Rom spline for value reconstruction is a viable alternative that avoids smoothing the original data [48].

Course Notes 28
Real-Time Volume Graphics

Classification

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

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 [18]. 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 15.1 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 15.1: Pre-classification (left) verses post-classification (right)

15.1 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 [68] 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 15.2(c) and 15.2(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 [49], can help define important landmarks for generating technical illustration-like renderings as seen in Figure 15.3. See [52] 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 15.4.

15.2 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

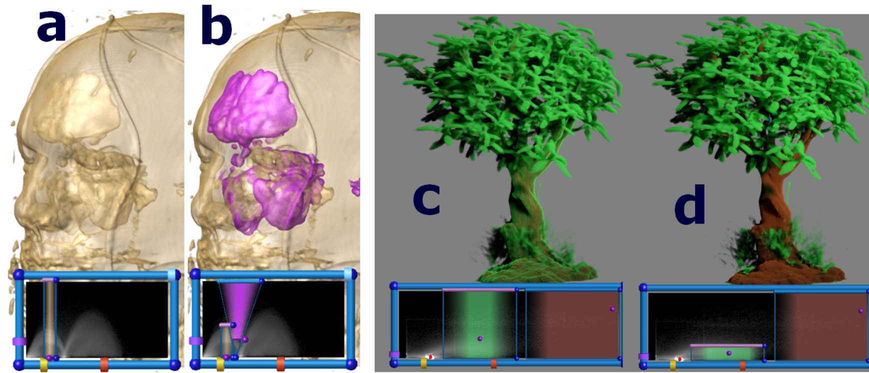


Figure 15.2: 1D (a and c) verses 2D (b and d) transfer functions.

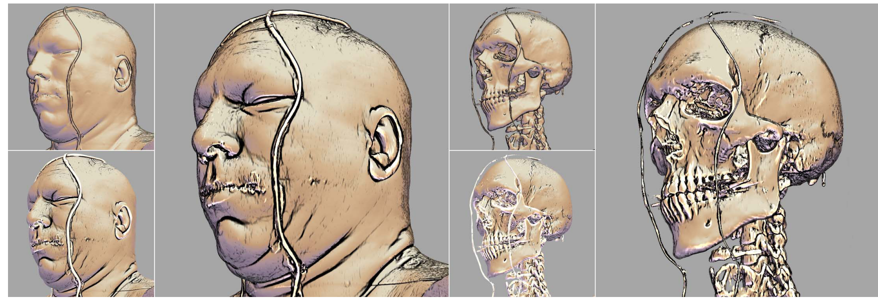


Figure 15.3: Using curvature and shape measures as axes of the transfer function. The upper small images show silhouette edges, the lower small images show ridge valley emphasis, and the large images show these combined into the final illustration. Images courtesy of Gordon Kindlman, use by permission.

(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 piece-wise linear (or higher order) ramps. This can be seen in Figure 15.5. 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.* [41]

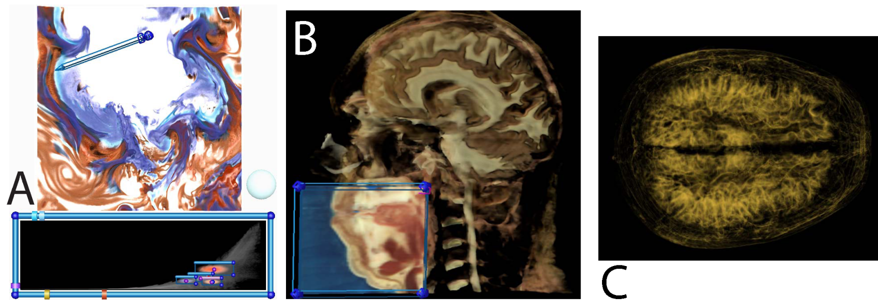


Figure 15.4: General multi-dimensional transfer functions, using multiple primary data values. A is a numerical weather simulation using temperature, humidity, and pressure. B is a color cryosection dataset using red, green, and blue visible light. C is a MRI scan using proton density, T1, and T2 pulse sequences.

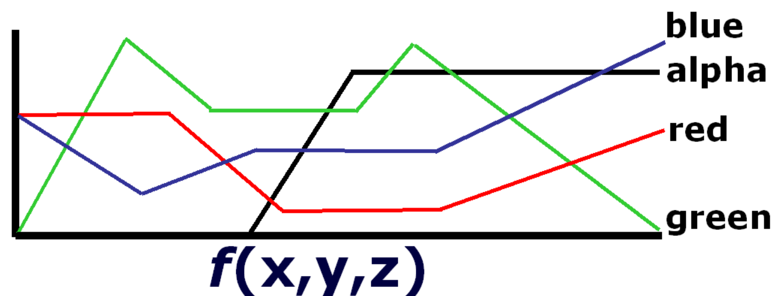


Figure 15.5: An arbitrary transfer function showing how red, green, blue, and alpha vary as a function of data value **f(x,y,z)** .

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 [74] 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 15.6 shows an example of this user interface.

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

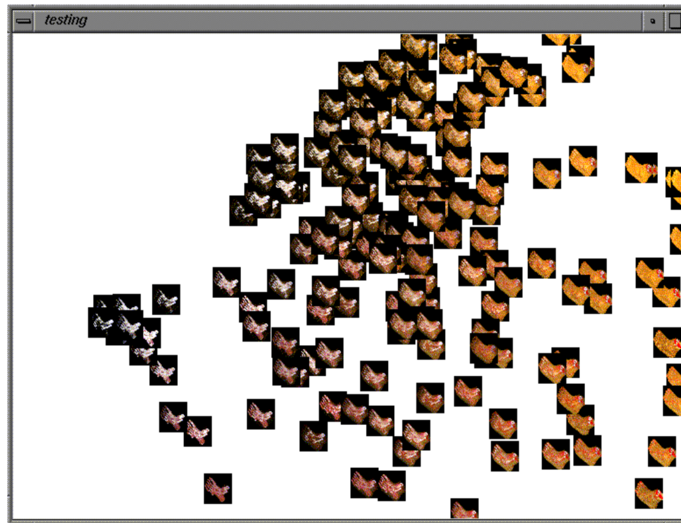


Figure 15.6: The Design Gallery transfer function interface.

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 15.2.

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 15.8(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 [52]. The action of this process can be seen in Figure 15.9.

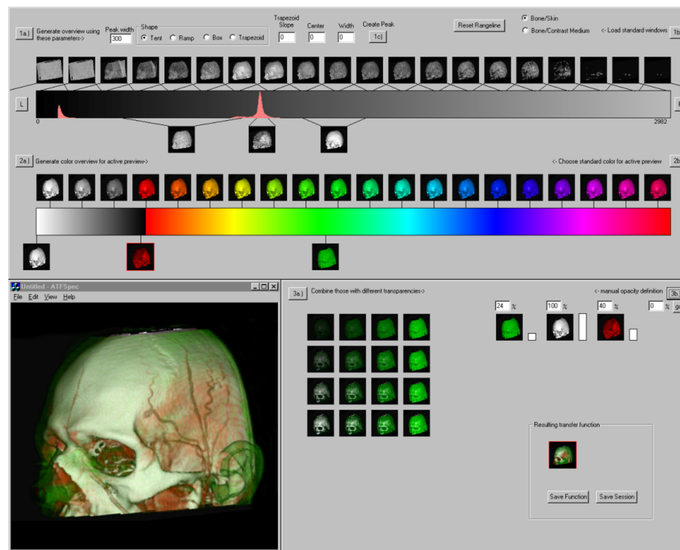
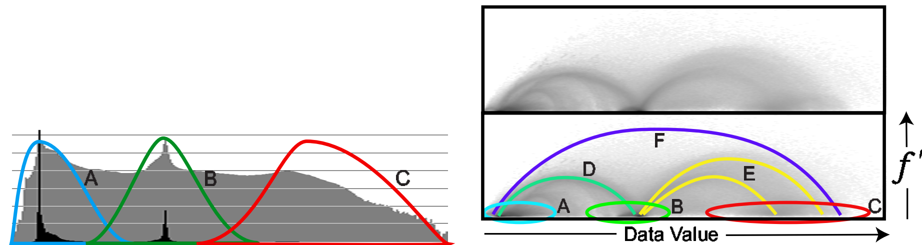


Figure 15.7: A thumbnail transfer function interface.

Often it is helpful to identify discrete regions in the transfer function domain that correspond to individual features. Figure 15.10 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.

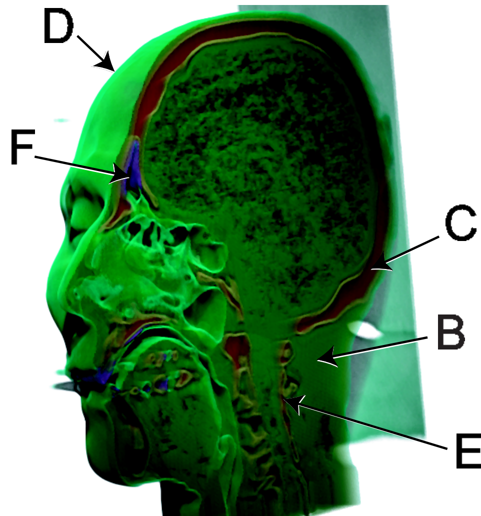
15.3 Summary

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 15.11, 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 15.8: 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).

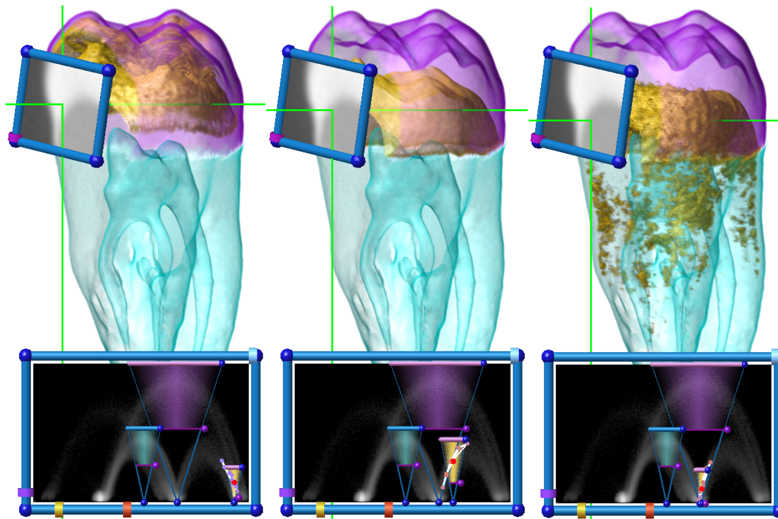


Figure 15.9: Probing and dual-domain interaction.

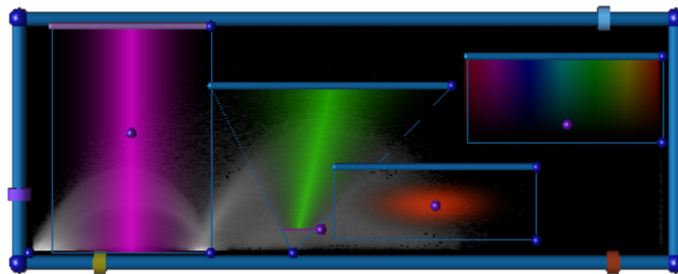


Figure 15.10: Classification widgets

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.

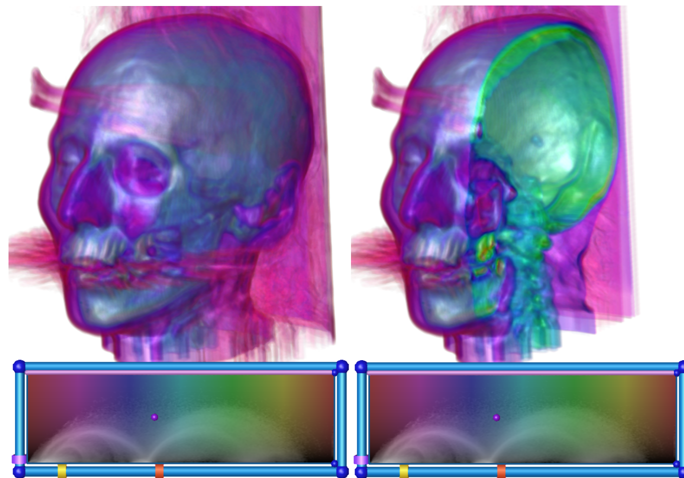


Figure 15.11: The “default” transfer function.

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 [54].

Course Notes 28

Real-Time Volume Graphics

Optical Properties

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Introduction

In the previous chapter 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 [68] is a simplified approximation of a more general volumetric light transport equation first used in computer graphics by Kajiya [44]. 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 [76]. 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 18.1 illustrates the geometric setup common to each of the approximations.

18.1 Traditional volume rendering

The classic volume rendering model originally proposed by Levoy [68] 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:

$$x(s) = x_0 + s\vec{\omega} \tag{18.1}$$

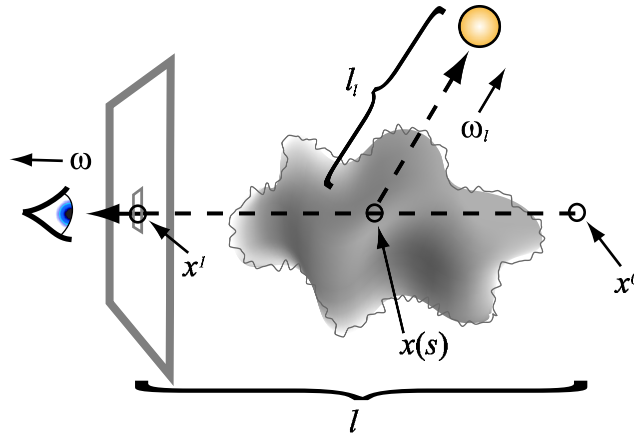


Figure 18.1: The geometric setup for light transport equations.

The classic volume rendering model is then written as:

$$L(x_1, \vec{\omega}) \;=\; T(0, l)L(x_0, \vec{\omega}) + \int_0^l T(s, l)R(x(s))f_s(x(s))L_l ds \quad (18.2)$$

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:

$$T(s, l) = \exp\left(-\int_s^l \tau(s')ds'\right) \quad (18.3)$$

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:

$$L(x_1, \vec{\omega}) = T(0, l)L(x_0, \vec{\omega}) + \int_0^l T(s, l)E(x(s))ds \quad (18.4)$$

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.

18.2 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:

$$R'(s) = R(s)\left((1 - S(s)) + f_s(s)S(s)\right) \tag{18.5}$$

$S(s)$ 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:

$$S(s) = 1 - (1 - \|\nabla f(s)\|)^2 \tag{18.6}$$

Figure 18.2 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.

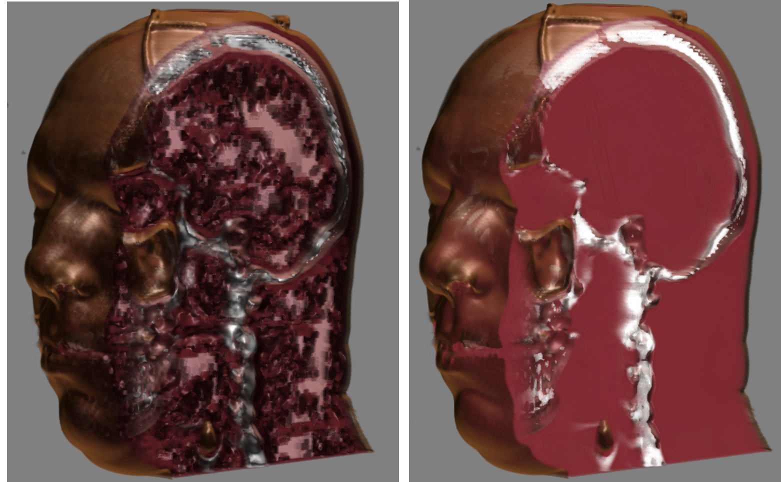


Figure 18.2: Surface shading without (left) and with (right) the surface scalar.

18.3 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:

$$I_{eye} = I_B * T_e(0) + \int_0^{eye} T_e(s) * R(s) * f_s(s) * I_l(s)ds \qquad (18.7)$$

$$I_l(s) = I_l(0) * exp\left(-\int_s^{light} \tau(x)dx\right) \qquad (18.8)$$

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 [4]. 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

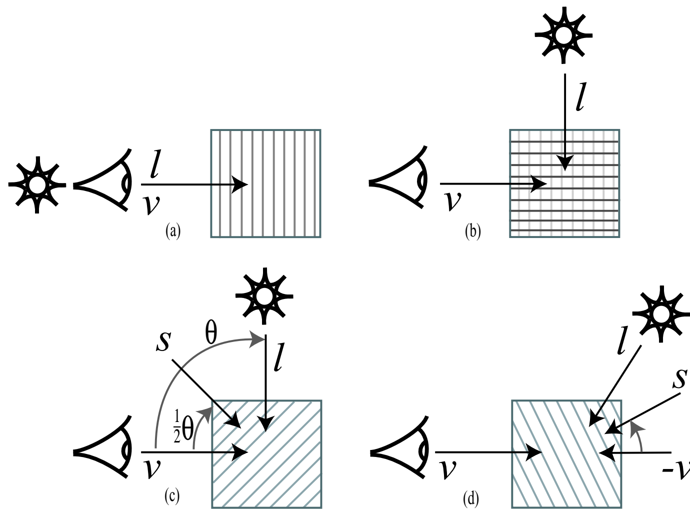


Figure 18.3: Modified slice axis for light transport.

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 [52]. 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 18.3(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 18.3(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

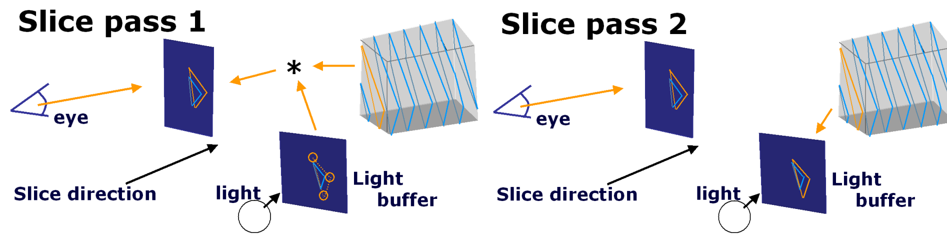


Figure 18.4: Two-pass shadows. Step 1 (left) render a slice for the eye, multiplying it by the attenuation in the light buffer. Step 2 (right) render the slice into the light buffer to update the attenuation for the next pass.

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 18.3(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 18.3(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

$$d_s = \cos(\frac{\theta}{2})d_v. \tag{18.9}$$

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.

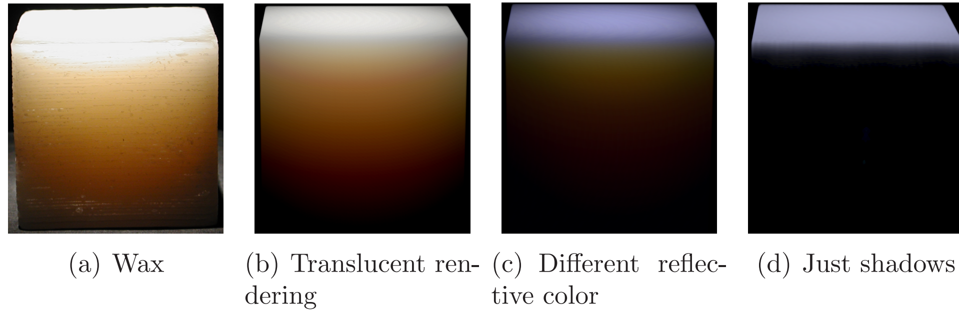
Since we must keep track of the amount of light attenuated at each slice, we utilize an off screen render buffer, known as the *pixel buffer*. This buffer is initialized to $1 - \mathit{light\,intensity}$. It can also be initialized using an arbitrary image to create effects such as spotlights. The projection 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 18.4.

18.4 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 18.5(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 [38, 16]. 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 18.5: 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.

$$I_{eye} = I_0 * T_e(0) + \int_0^{eye} T_e(s) * C(s) * I_l(s) ds \tag{18.10}$$

$$\begin{aligned} I_l(s) \quad = \quad & I_l(0) * exp\left(-\int_s^{light} \tau(x)dx\right) + \\ & I_l(0) * exp\left(-\int_s^{light} \tau_i(x)dx\right) \mathbf{Blur}(\theta) \end{aligned} \tag{18.11}$$

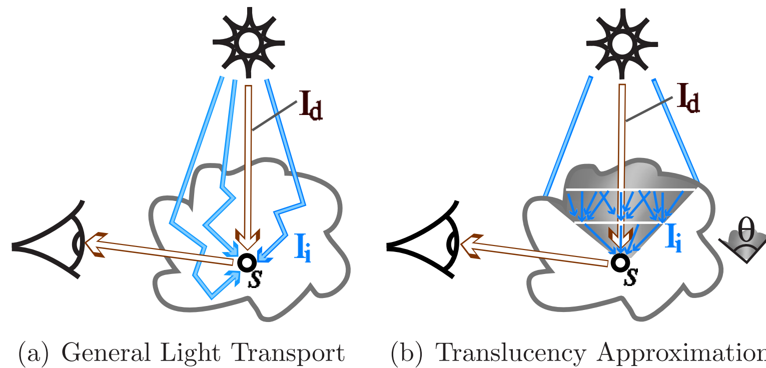
where $\tau_i(s)$ is the indirect light extinction term, $C(s)$ is the reflective color at the sample s, $S(s)$ is a surface shading parameter, and I_l is the sum of the direct and indirect light contributions.

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:

$$\alpha_i = exp(-\tau_i(x)). \tag{18.12}$$

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.

In general, light transport in participating media must take into account the incoming light from all directions, as seen in Figure 18.6(a).



(a) General Light Transport (b) Translucency Approximation

Figure 18.6: 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.

However, the net effect of multiple scattering in volumes is a blurring of light. The diffusion approximation [101, 24] 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 18.11 averages the incoming light within the cone with an apex angle θ in the direction of the light (Figure 18.6(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):

$$offset \leq d\tan(\frac{\theta}{2}) \tag{18.13}$$

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.

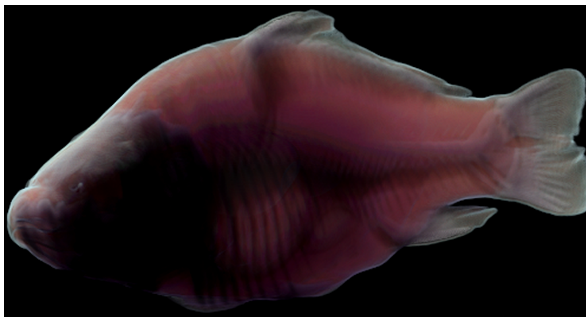
While this process my seem quite complicated, it is straightforward to implement. The *render to texture* extension is part of the **WGL_ARB_render_texture** OpenGL extensions. The key functions are **wglBindTexImageARB()** which binds a *P-Buffer* as a texture, and **wglReleaseTexImageARB()** which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

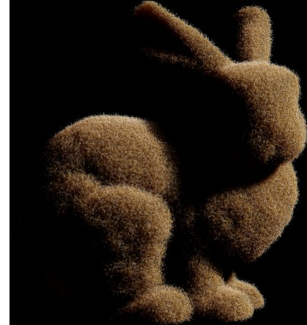
volume rendering can match. This means that the resolution of the light transport is decoupled from that of the data volume's grid, permitting procedural volumetric texturing.

18.5 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 18.7: Example volume renderings using an extended transfer function.

User Interface Tips

Figure 15.5 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 19.1 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 19.1.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 19.2. 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

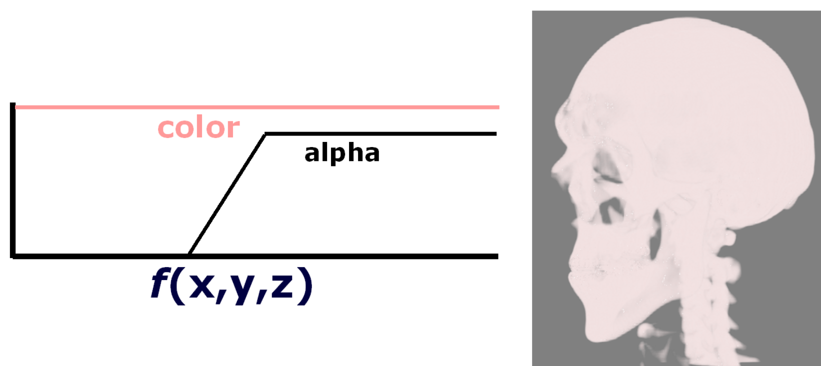


Figure 19.1: An example transfer function for the bone of the Visible Male (left), and the resulting rendering (right).

in this image. This approach 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 19.2. While this approach may not produce images as compelling as surface shaded or shadowed renderings as seen in Figure 19.3, it is advantageous because it doesn't require any extra computation in the rendering phase.

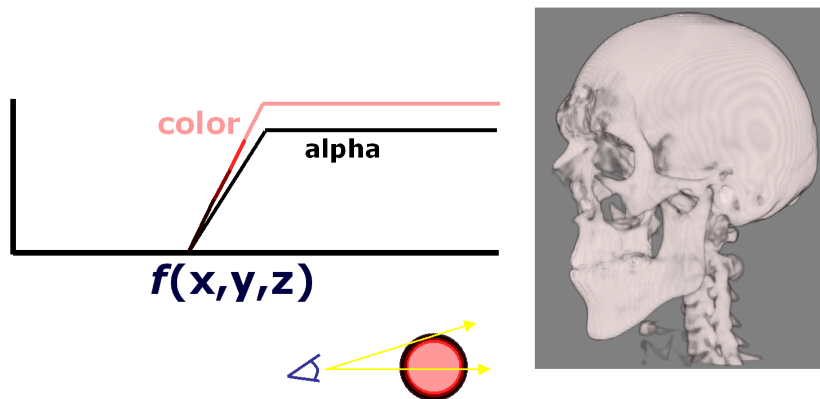


Figure 19.2: Faux shading. Modify the transfer function to create silhouette edges.

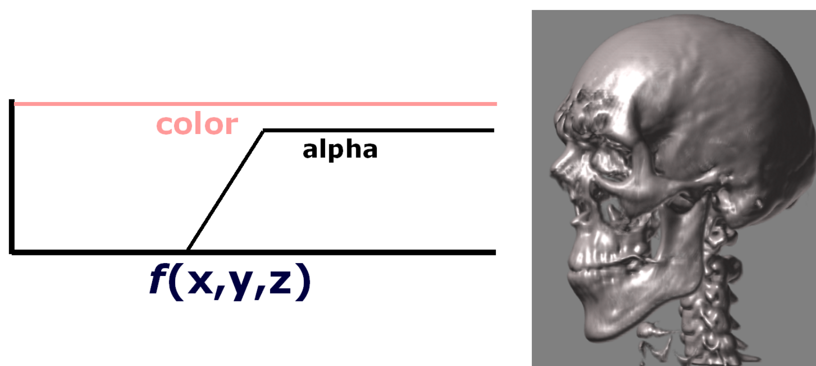


Figure 19.3: Surface shading.

Course Notes 28

Real-Time Volume Graphics

High-Quality Volume Rendering

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

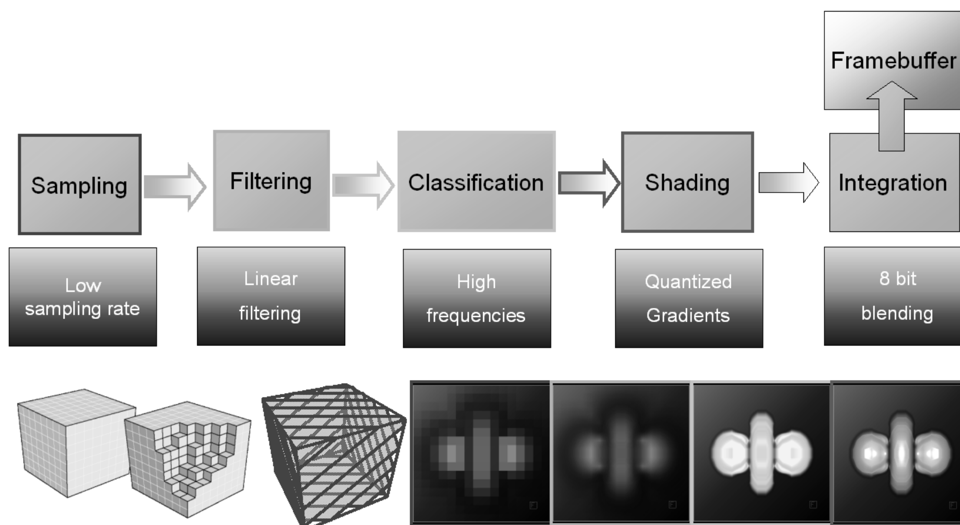


Figure 19.4: Volume rendering pipeline. Each step in this pipeline can introduce artifacts.

Today's GPUs support high-quality volume rendering (see figures 48.2). 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 19.4): 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 20.1). 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 ([81, 92]). 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

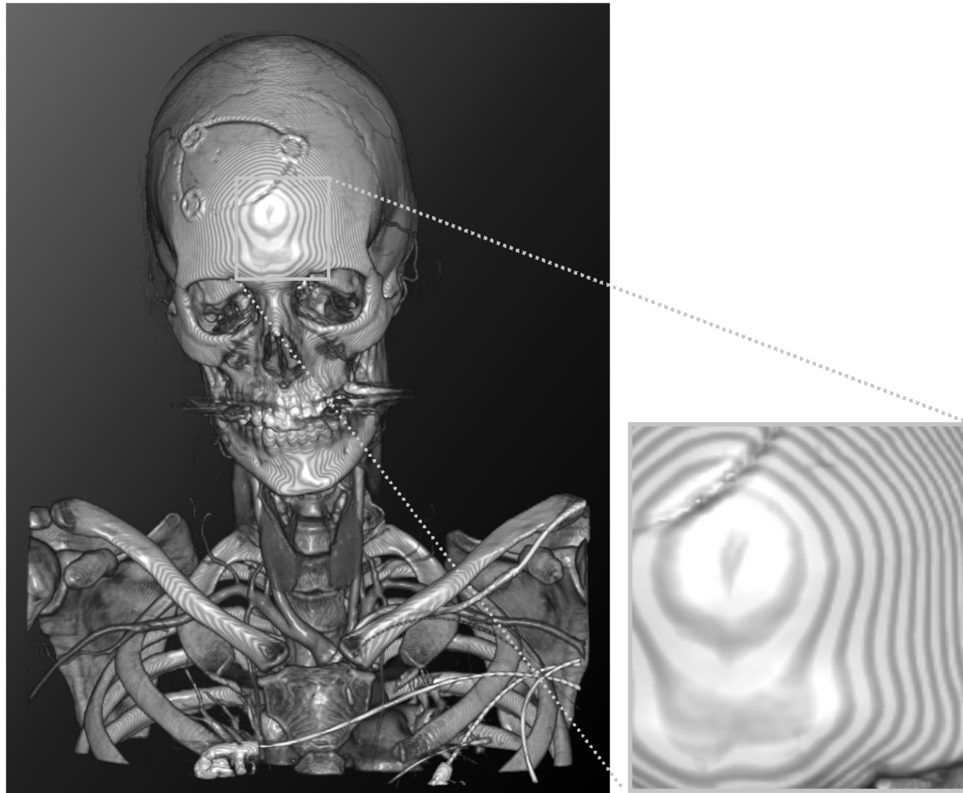


Figure 20.1: Wood grain artifact caused by a low sampling rate.

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 20.2) 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

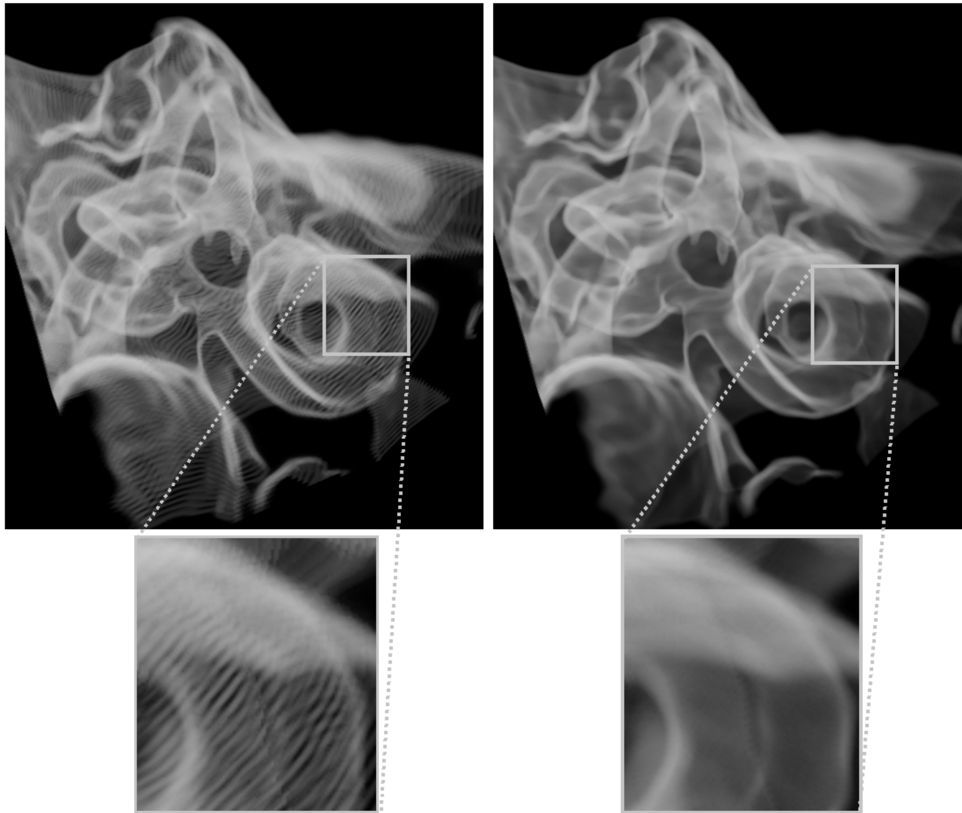


Figure 20.2: Comparison of a visualization of the inner ear with low (left) and high (right) sampling rate.

of the volume data. We will discuss this effect in detail in chapter 22 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.

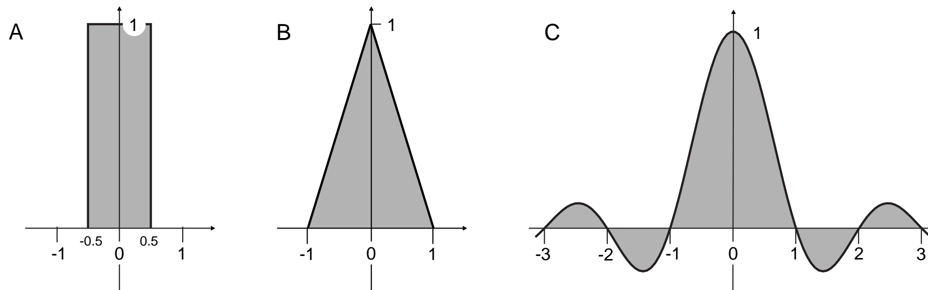


Figure 21.1: Three reconstruction filters: (a) box, (b) tent and (c) sinc filters.

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.

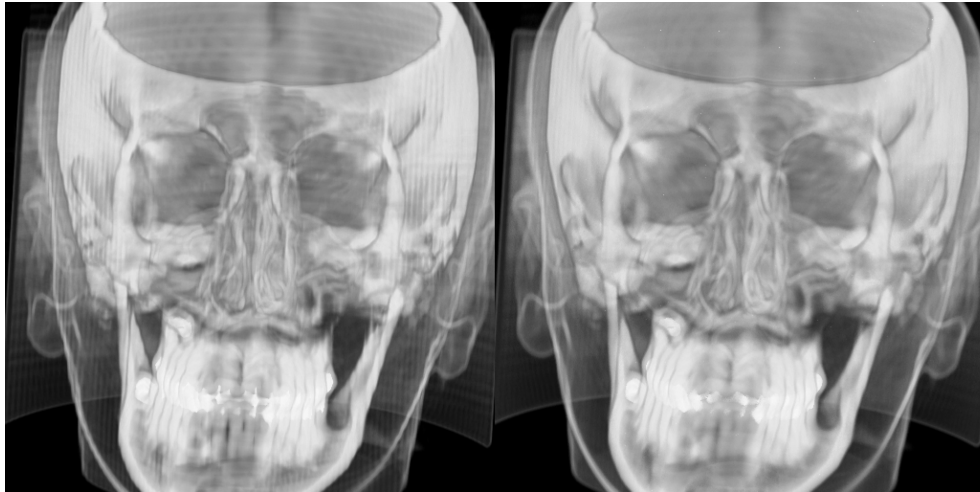


Figure 21.2: Comparison between trilinear filtering and cubic B-spline filtering.

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 21.2). Hadwiger et al.[34] 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:

$$g(t) = (s * h)(t) = \int_{-\infty}^{\infty} s(t - t') \cdot h(t')dt' \qquad (21.1)$$

The discretized form is:

$$g_t = \sum_{i=-I}^{+I} s_{t-i} h_i \qquad (21.2)$$

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.

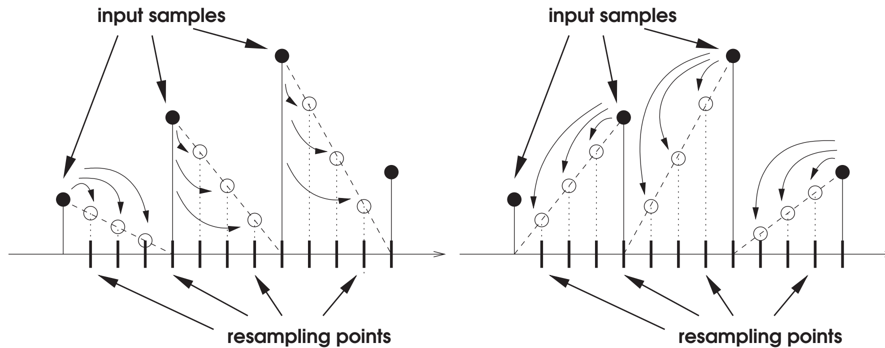


Figure 21.3: Distributing the contributions of all "left-hand" (a), and all "right-hand" (b) neighbors, when using a tent filter.

Figure 21.3 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 21.4). 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-

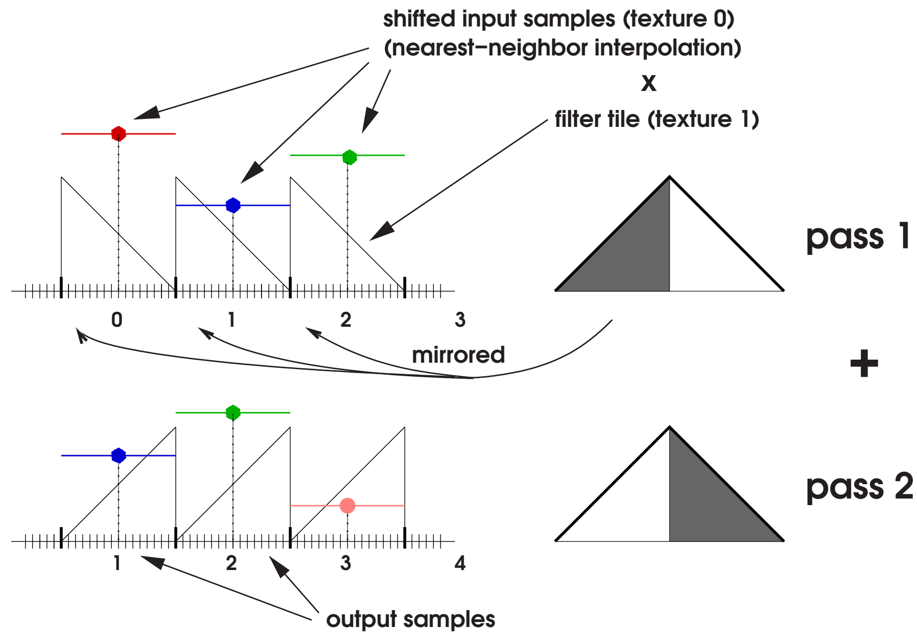


Figure 21.4: Tent filter (width two) used for reconstruction of a one-dimensional function in two passes. Imagine the values of the output samples added together from top to bottom.

fication is desired instead of minification. For minification, pre-filtering approaches like mip-mapping are advantageous. Figure 21.2 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 50 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 22.1. 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.

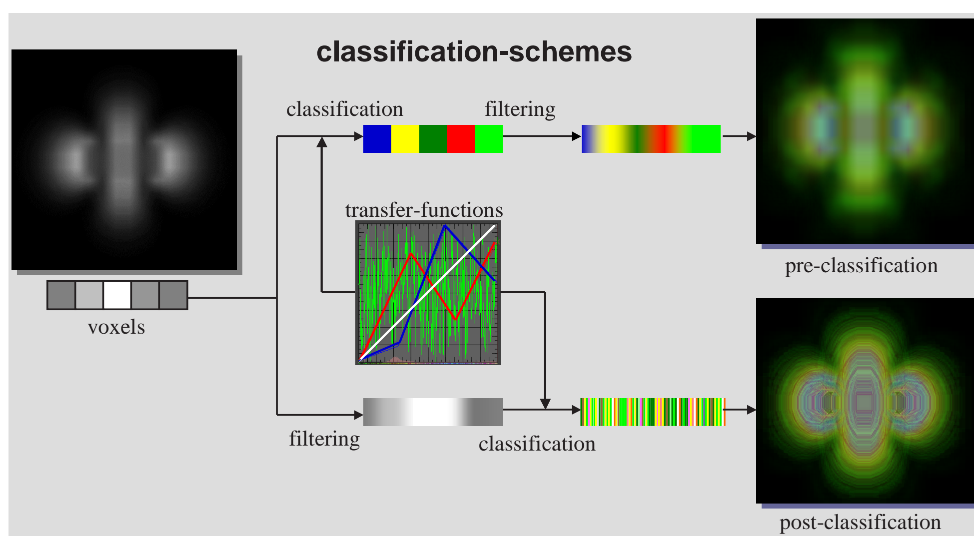


Figure 22.1: Comparison of pre-classification and post-classification. Alternate orders of classification and filtering lead to completely different results. For clarification a random transfer function is used for the green color channel. Piecewise linear transfer functions are employed for the other 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*[86] 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 22.2.) More precisely spoken, the opacity α_i of the i-th segment is approximated by

$$\begin{aligned}\alpha_i &= 1 - \exp\left(-\int_{i\,d}^{(i+1)d} \tau\Big(s\big(\mathbf{x}(\lambda)\big)\Big)\mathrm{d}\lambda\right)\\ &\approx 1 - \exp\left(-\int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big)d\,\mathrm{d}\omega\right). \qquad (22.1)\end{aligned}$$

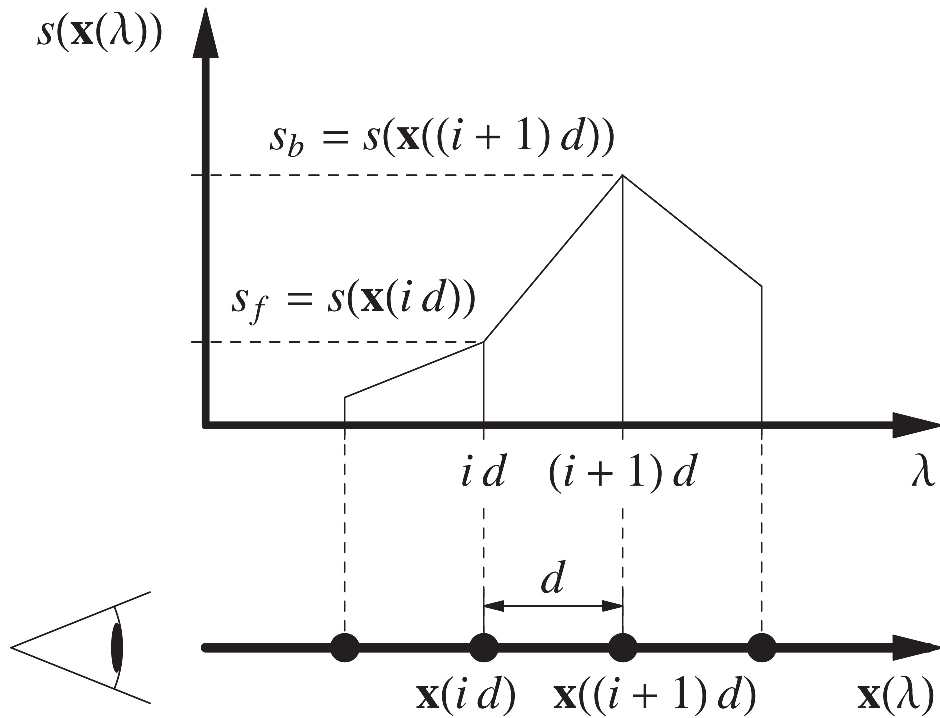


Figure 22.2: Scheme for determining the color and opacity of the i-th ray segment.

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:

$$\begin{aligned} \tilde{C}_i \quad \approx \quad & \int_0^1 \tilde{c}\big((1-\omega)s_f + \omega s_b\big) \\ & \times \exp\Big(-\int_0^{\omega} \tau\big((1-\omega')s_f + \omega' s_b\big)\,d\;\mathrm{d}\omega'\Big)\,d\;\mathrm{d}\omega. \end{aligned} \tag{22.2}$$

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 (22.2) and opacities α_i pre-computed according to Equation (22.1). For non-associated color

transfer function, i.e., when substituting $\tilde{c}(s)$ by $\tau(s)c(s)$, we will also employ Equation (22.1) for the approximation of α_i and the following approximation of the associated color $\tilde{C}_i^\tau$:

$$\tilde{C}_i^\tau \approx \int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) c\big((1-\omega)s_f + \omega s_b\big) \times \exp\Big(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big) d \,\mathrm{d}\omega'\Big) d \,\mathrm{d}\omega. \tag{22.3}$$

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[21], 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,
```

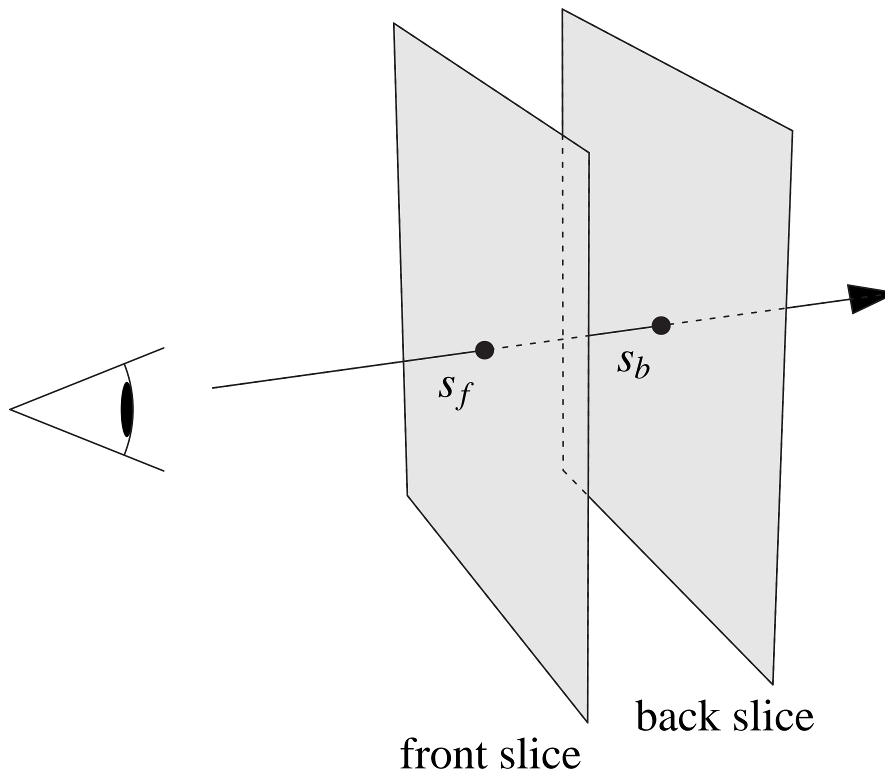


Figure 22.3: A slab of the volume between two slices. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b).

```
                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 22.4. 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 24).

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.

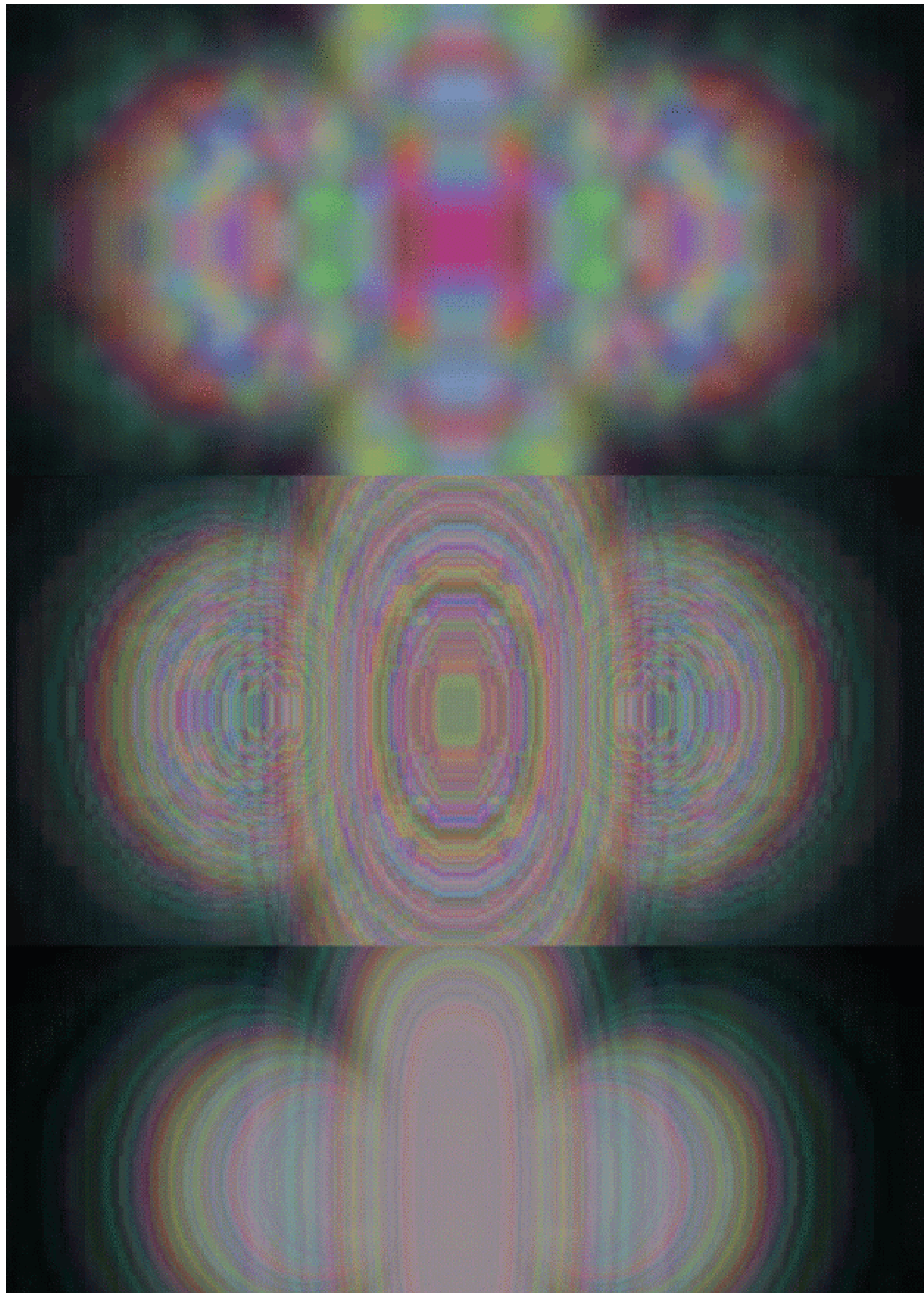


Figure 22.4: Comparison of the results of pre-, post- and pre-integrated classification for a random transfer function. Pre-classification (top) does not reproduce high frequencies of the transfer function. Post-classification reproduces the high frequencies on the slice polygons (middle). Pre-integrated 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 23.1, 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:

$$\vec{\nabla} I \;=\; (I_x,\, I_y,\, I_z) \;=\; \left(\frac{\partial}{\partial x}\, I,\, \frac{\partial}{\partial y}\, I,\, \frac{\partial}{\partial z}\, I\right). \tag{23.1}$$

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

$$\left\| \vec{\nabla} I \right\| \;=\; \sqrt{{I_x}^2 + {I_y}^2 + {I_z}^2}. \tag{23.2}$$

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.

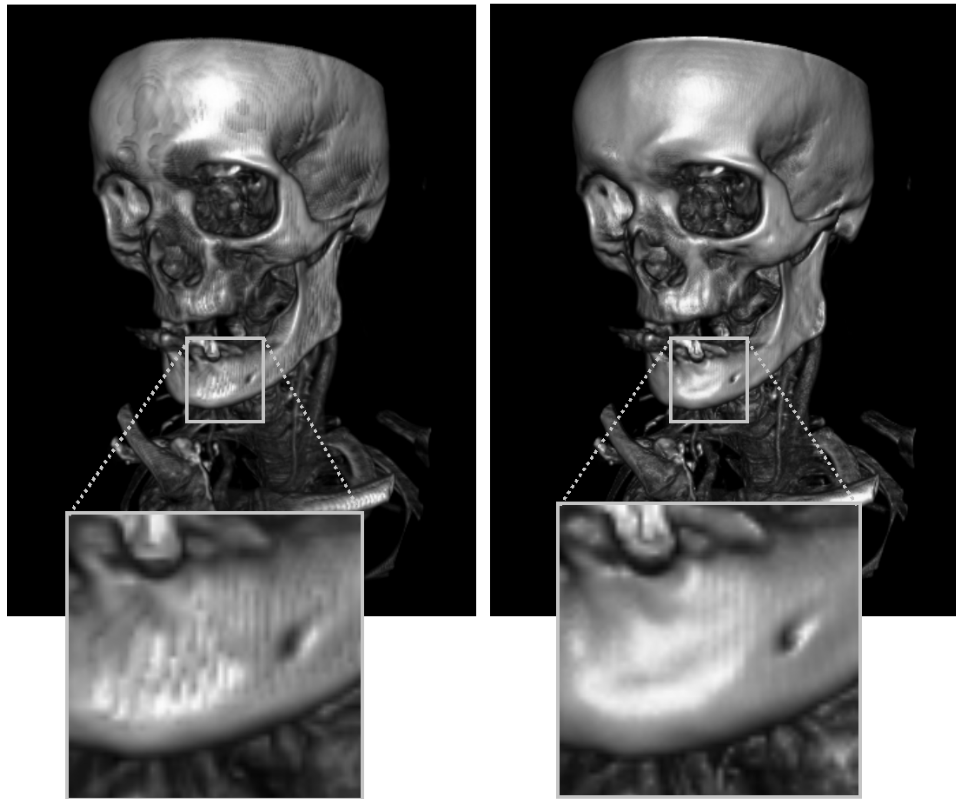


Figure 23.1: Comparison between pre-computed and quantized gradients (left) with on-the-fly gradient computation (right).

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 23.1.

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 23.1. 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 23.2). 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 23.2). 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 50) will allow real-time performance, even when computing gradients on-the-fly.

Figure 23.2: 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 24.1 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

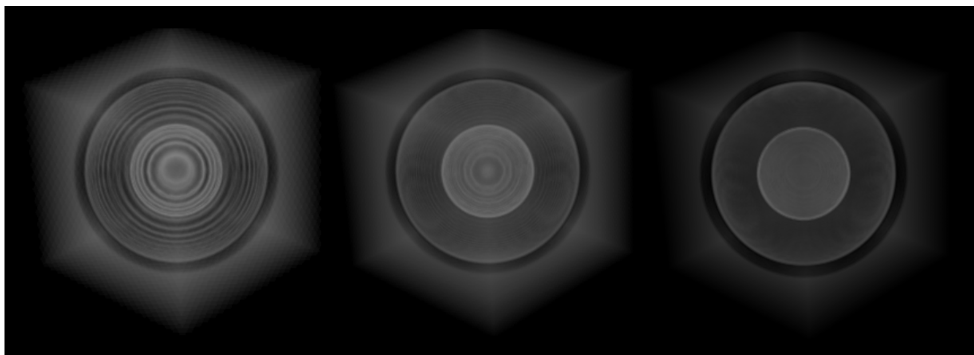


Figure 24.1: Comparison between 8-bit (left), 16-bit (middle) and 32-bit blending (right).

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 24.2 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.

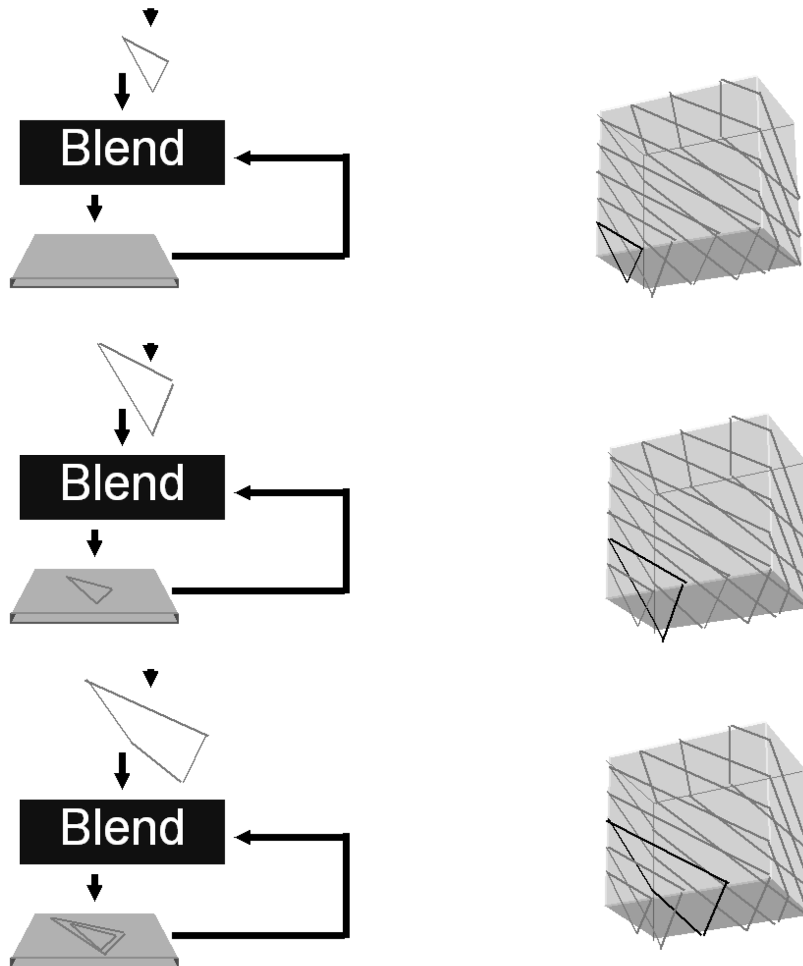


Figure 24.2: Programmable blending with a pbuffer as input texture and render target at the same time.

If you feel uncomfortable with this solution, you can use ping-pong blending as an alternative (see figure 24.3). 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 24.1 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.

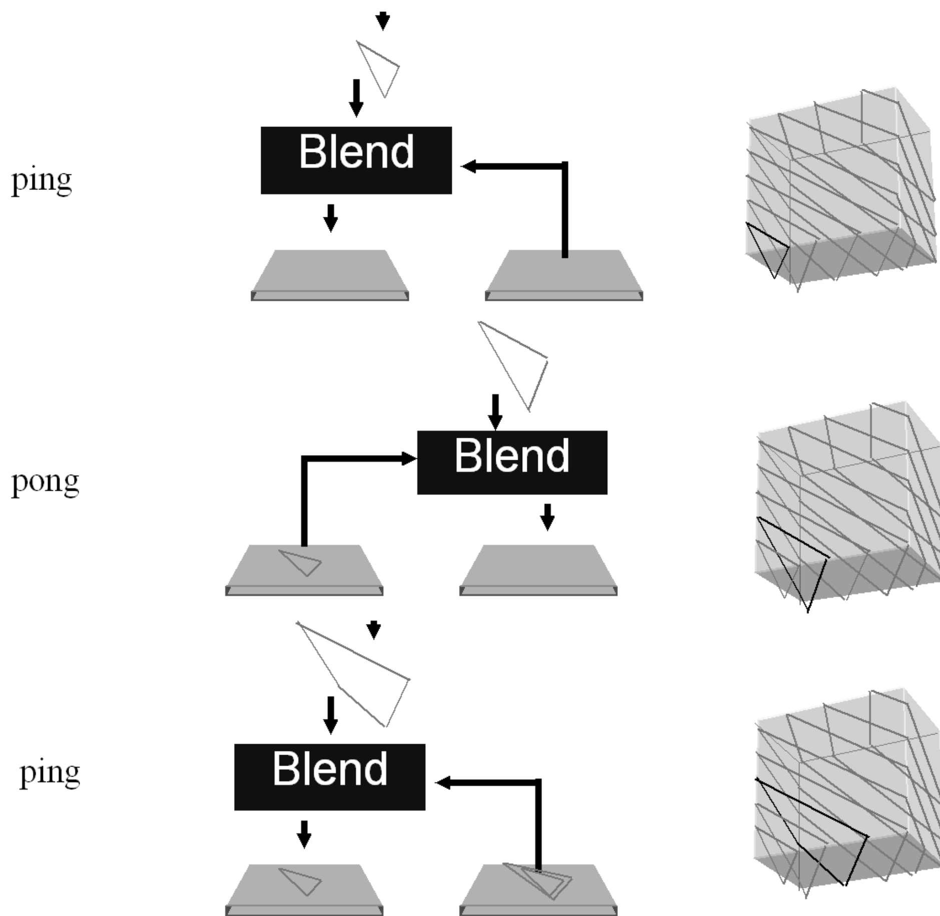


Figure 24.3: Programmable blending with a pbuffer as input texture and render target at the same time.

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 28

Real-Time Volume Graphics

Volume Clipping

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Introduction to Volume Clipping

Volume clipping plays an important role in understanding 3D volumetric data sets because it allows the user to cut away selected parts of the volume based on the position of voxels in the data set. Clipping often is the only way to uncover important, otherwise hidden details of a data set. The goal of this part of the course notes is to present several techniques to efficiently implement complex clip geometries for volume visualization. All methods are tailored to interactive texture-based volume rendering on graphics hardware.

We discuss two basic approaches to volume clipping for slice-based volume rendering. In the first approach (Chapter 27), a clip object is represented by a tessellated boundary surface. The basic idea is to store the depth structure of the clip geometry in 2D textures. This depth structure of the clip object can be used by fragment operations to clip away parts of the volume. In the second approach (Chapter 28), a clip object is voxelized and represented by an additional volume data set. Clip regions are specified by marking corresponding voxels in this volume.

Issues related to volume shading are considered in Chapter 29. The combination of volume shading and volume clipping introduces issues of how illumination is computed in the vicinity of the clipping object. On the one hand, the orientation of the clipping surface should be represented. One the other hand, properties of the scalar field should still influence the appearance of the clipping surface.

Finally, Chapter 30 shows how GPU clipping techniques can be combined with pre-integrated volume rendering (see Part VII on "Pre-Integration" for details on basic pre-integrated volume rendering).

Depth-Based Clipping

This chapter is focused on algorithms for volume clipping that rely on the depth structure of the clip geometry. It is assumed that clip objects are defined by boundary surface representations via triangle meshes. For each fragment of a slice, fragment operations and tests decide whether the fragment should be rendered or not. Detailed background information on this approach can be found in [105, 106].

Generic Description

Figure 27.1 illustrates the depth structure of a typical clip object. The problem is reduced to a 1D geometry along a single light ray that originates from the eye point. There is a one-to-one mapping between light rays and pixels on the image plane. Therefore, the following descriptions can be mapped to operations on the fragments that correspond to the respective position of the pixel in the frame buffer.

We start with an algorithm for clip objects of arbitrary topology and geometry. First, the depth structure of the clip geometry is constructed for the current pixel. This structure holds the depth values for each

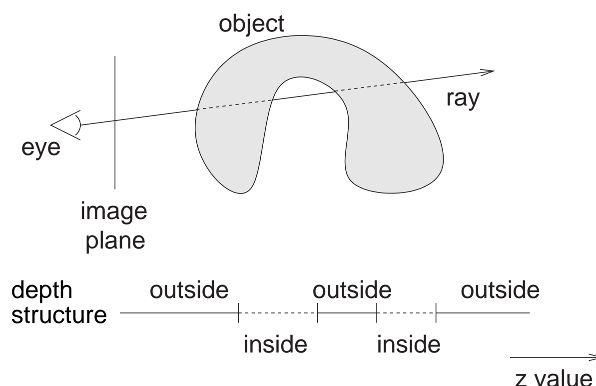


Figure 27.1: Depth structure of a clip geometry.

intersection between eye ray and object. Second, the rendering of each fragment of a volume slice has to check if the fragment is visible or not. Based on the visibility, the fragment is blended into the frame buffer or rejected. We use the term *volume probing* when the volume is clipped away outside the clip geometry. Conversely, a *volume cutting* approach inverts the role of the visibility property—only the volume outside the clip object remains visible.

Clipping against a Single Depth Layer

A straightforward approach uses the depth buffer to store the depth structure. Unfortunately, the depth buffer can hold only one depth value per pixel. Therefore, just a single depth layer can be represented. The implementation of this approach begins with rendering the clip geometry to the depth buffer. Then, writing to the depth buffer is disabled and depth testing is enabled. Finally, the slices through the volume data set are rendered, while the depth test implements the evaluation of the visibility property of a fragment. The user can choose to clip away the volume in front or behind the geometry, depending on the logical operator for the depth test.

Convex Volume Probing

A convex object implies that the number of boundaries along the depth structure is not larger than two because, regardless of the viewing parameter, an eye ray cannot intersect the object more than twice. The special case of a convex clip objects allows us to use a rather simple representation for the depth structure.

For a convex geometry, one algorithm for depth-based volume probing is as follows. First, the depth values z_{front} for the first boundary are determined and written to a texture by rendering the frontfaces of the clip geometry. Then, a fragment program is used to shift the depth values of all fragments in the following rendering passes by $-z_{\text{front}}$ (where z_{front} is read from the previously generated texture). Third, the depth buffer is cleared and the backfaces are rendered into the depth buffer (with depth shift enabled) to build the second boundary. Finally, slices through the volume data set are rendered, without modifying the depth buffer. Depth shift and depth testing, however, are still enabled.

Figure 27.2 illustrates this clipping algorithm. By rendering the backfaces of the clip object with a shift by $-z_{\text{front}}$, entries in the depth buffer are initialized to $z_{\text{back}} - z_{\text{front}}$, where z_{back} is the unmodified depth of

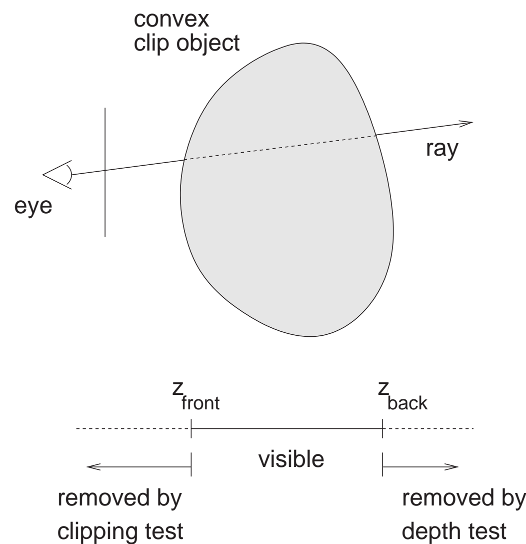


Figure 27.2: Illustration of depth-based volume probing for a convex clip geometry.

the backface's fragment. During the actual volume rendering, fragments that are behind the backface of the clip geometry are rejected by the depth test, while fragments that are in front of the frontface of the clip geometry are removed by clipping against the near plane of the view frustum.

A shift of depth values by $-z_{\text{front}}$ can be realized by a fragment program. The current generation of GPUs (such as ATI Radeon 9600 or higher, or NVidia GeForce FX) allows us to modify a fragment's depth value according to values from a texture and / or numerical operations. The Pixel Shader 2.0 code (DirectX) for such an implementation is shown in Figure 27.3.

The texture that holds the depth structure (here texture sampler stage `s0`) should be a hires texture, such as 16 bit fixed-point or 16/32 bit floating point. This texture is efficiently filled with depth values by directly rendering into this 2D texture. DirectX supports such a render-to-texture functionality; similarly, `WGL_ARB_render_texture` is supported for OpenGL under Windows.

In [105] a comparable implementation on an NVidia GeForce 3 GPU is described. Due to a reduced flexibility of this previous generation of GPUs, a complicated configuration of texture shaders is required. This implementation is recommended when older GPUs have to be supported.

```
ps.2.0       // Pixel Shader 2.0 code

dcl v0       // Vertex coordinates
dcl t0       // Position
dcl t1       // One-to-one mapping on image plane
dcl_2d s0    // Depth layer for front faces
```
... More declarations for standard volume rendering
(data set, transfer function, etc.)
```
// Get z values of front faces
texld r0, t1, s0

// Compute depth of the fragment in device coordinates
// (division by homogeneous w coordinate)
rcp r1,t0.w
mul r2,t0,r1

// Shift depth values
add r3,r2.z,-r0.x
mov oDepth,r3.z
```
... Compute color according to standard volume rendering

Figure 27.3: Pixel Shader 2.0 code for depth shift.

Convex Volume Clipping Based on Texkill

An alternative approach exploits a conditional removal of fragments (i.e., a `texkill` operation) to achieve comparable results. Compared to the algorithm from the previous section, texkill-based clipping introduces the following modifications. First, the depth structures for frontfacing and backfacing parts of the clip geometry are both stored in hires textures, i.e., the depth buffer is not used to clip away volume regions. Second, the fragment program accesses both hires textures and compares the depth of a fragment with the stored depth structure. This comparison leads to a conditional texkill when the fragment lies outside of the visible region. Probing and cutting modes can easily be exchanged by negating the Boolean expression for the comparison.

Multi-Pass Rendering for Concave Clip Objects

If concave clip geometries have to be represented, the management of the depth structure has to be extended to take into account a larger number of possible intersections between the clipping geometry and an eye ray. Concave clipping needs a larger number of textures to store the depth structure, and can be implemented by using multiple render passes.

In the first step of this approach, the depth structure of the clip object is analyzed and neighboring pairs of depth layers are combined. These pairs are organized in a way that ensures that each pair represents one visible segment of the viewing ray, i.e., the volume remains visible within the interval defined by such a pair of z values. Note that single, unpaired depth entries are produced at the front and back parts of the depth structure in the case of volume cutting.

The depth structure can be determined by an algorithm reminiscent of the *depth-peeling* algorithm [22]: The depth structure is constructed by peeling off the geometry in a depth-layer by depth-layer fashion. In a back-to-front scheme, the first depth layer (i.e., the geometry furthest away from the camera) is initialized by rendering with a depth test set to "greater". This depth layer is stored in a hires texture. Subsequent (i.e., closer) layers are extracted by rendering only those objects that are closer than the depth structure of the current hires texture. This behavior is ensured by a fragment program that kills fragments that lie behind the depth value of the current hires texture. The depth values of the rendered fragments are written to the next hires texture—and the same process is continued for the next closer depth layer. The algorithm stops when no fragments pass the fragment program, which can be checked by performing an occlusion query that provides the number of rendered fragments.

The second step comprises the actual multi-pass rendering of the volume. For each pair of depth values from the depth structure, the corresponding region of the volume is rendered. Each depth pair leads to a local volume probing against two boundaries; therefore, rendering can make use of the previously described method for convex volume probing. Employing multiple render passes, the complete volume is reconstructed in a layer-by-layer fashion by processing the depth structure pair-by-pair. The case of volume cutting (with single depth values at the front and back of the depth structure) can be handled by clipping only at a single boundary, as described in the Section on "Clipping against a Single Boundary".

Summary

Depth-based clipping is an image-space approach with a one-to-one mapping between pixels on the image plane and texels in the 2D depth layers. Therefore, clipping has a per-pixel accuracy and provides high-quality images. Another advantage of depth-based clipping is its built-in support for dynamic clip objects because the depth structure is re-built from the clip geometry for each frame. The main disadvantage is the complex treatment required for non-convex clip objects. Multi-pass rendering can become very slow, especially if many depth layers are present.

Clipping via Tagged Volumes

In contrast to the depth-based clipping algorithms from the previous chapter, volumetric clipping approaches model the visibility information by means of a second volume texture whose voxels provide the clipping information. Here, the clip geometry has to be voxelized and stored as a binary volume. Background information on this approach can be found in [105, 106].

Conceptually, the clipping texture and the volume data set are combined to determine the visibility, i.e., the clipping texture is tagged to the volume data set. During rendering, a fragment program multiplies the entries from the data set by the entries from the clip texture—all the voxels to be clipped are multiplied by zero. Clipped fragments be removed by a conditional `texkill` within a fragment program or, alternatively, discarded by an alpha test.

When a 3D texture is used for the voxelized clip object, any affine transformation of the clip geometry can be represented by a corresponding transformation of texture coordinates. Switching between volume probing and volume cutting is achieved by inverting the values from the clipping texture. All these operations can be performed very efficiently because they only require the change of texture coordinates or a reconfiguration of the fragment program. However, a complete change of the shape of the clip geometry requires a revoxelization and a reload of the clipping texture.

The above method is based on a binary representation and requires a nearest-neighbor sampling of the clipping texture. If a trilinear (for a 3D texture) interpolation was applied, intermediate values between zero and one would result from a texture fetch in the clipping texture. Therefore, a clearly defined surface of the clip geometry would be replaced by a gradual and rather diffuse transition between visible and clipped parts of the volume. Unfortunately, a missing interpolation within the clipping texture introduces jaggy artifacts. This problem can be overcome by replacing the binary data set by a distance volume: Each voxel of the clipping texture stores the (signed) Euclidean distance to the closest

point on the clip object. The surface of the clip object is represented by the isosurface for the isovalue 0. A trilinear interpolation in the 3D clipping texture is applied during rendering. Based on the comparison of the value from the clipping texture with 0, the clipped fragments can be removed (by `texkill` or an alpha test in combination with setting alpha to zero).

A voxelized clipping representation can be extended to more generic tagged volumes that allow for a space-variant modification of the visual representation. For example, Hastreiter et al. [39] modify texture-based volume rendering with pre-classification by applying different transfer functions to different regions of the segmented data set. Tiede et al. [98] use a similar approach for the visualization of attributed volumes by ray casting. And OpenGL Volumizer [5] uses a clipping mechanism based on a volumetric description of clipping objects. More details on an advanced use of an additional volumetric representation can be found in Part XI on "Non-Photorealistic Volume Rendering" and Part XII on "Segmented Volumes".

The advantages of the voxelized clipping approach are the support for arbitrary clip objects (with unrestricted choice of topology and geometry), and an extensibility towards generic tagged volumes. Disadvantages are a potentially large memory footprint for the voxelized clip geometry and additional texture fetch operations to access the voxelized representation. Furthermore, dynamic clip geometries need a re-voxelization with a subsequent download of the modified texture to the GPU, which can be very time-consuming.

Clipping and Consistent Shading

Volume shading extends volume rendering by adding illumination terms based on gradients of the scalar field. The combination of volume shading and volume clipping introduces issues of how illumination should be computed in the vicinity of the clip object. Illumination should not only be based on the properties of the scalar field, but should also represent the orientation of the clipping surface itself.

Figure 29.1 (b) demonstrates how illuminating the volume and the clipping surface improves the perception of the spatial structure of the clip object. Figure 29.1 (a) shows the same data set without specific lighting on the clipping surface. Here, the spherical clip surface is hard to recognize due to the inappropriate lighting.

In this chapter, depth-based and volumetric clipping techniques from the two previous chapters are extended to take into account a consistent shading of the clipping surface. These extensions make use of a modified optical model for volume rendering [106].

An Optical Model for Clipping in Illuminated Volumes

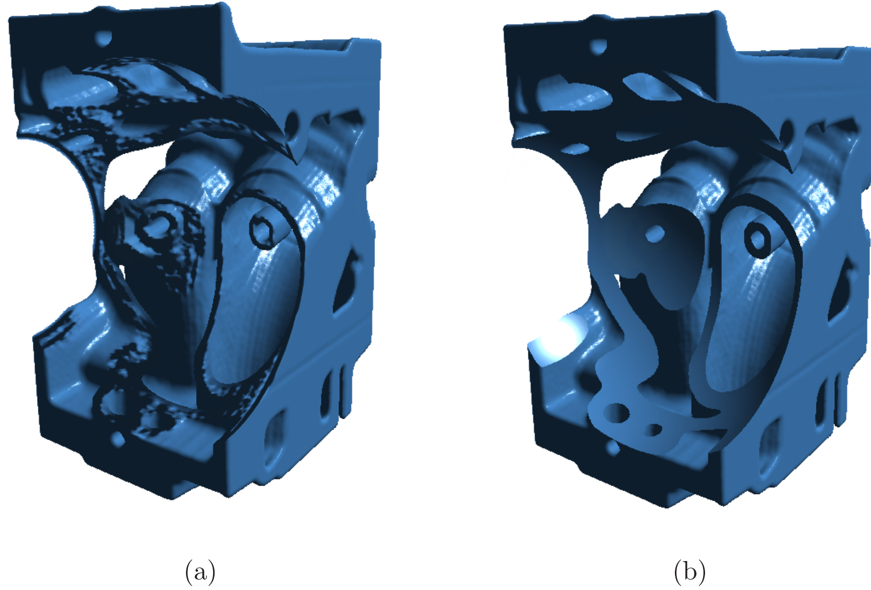
The volume rendering integral,

$$I(D) = I_0\, T(s_0) + \int_{s_0}^{D} g(s) T(s)\, \mathrm{d}s \quad , \tag{29.1}$$

along with the definition of transparency,

$$T(s) = e^{-\int_{s}^{D} \tau(t)\, \mathrm{d}t} \quad ,$$

is the widely used optical model for volume rendering (see Part II on "GPU-Based Volume Rendering"). The term I_0 represents the light entering the volume from the background at the position $s = s_0$; $I(D)$



(a) (b)

Figure 29.1: Combining volume clipping and volume shading. A spherical clip object cuts into an engine data set. No specific illumination computation is used in (a); (b) reveals lighting on the clip surface by combining surface-based and volumetric shading.

is the radiance leaving the volume at $s = D$ and finally reaching the camera.

Slice-based volume rendering approximates the integral in Eq. (29.1) by a Riemann sum over n equidistant segments of length $\Delta x = (D - s_0)/n$:

$$I(D) \approx I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_i \quad , \tag{29.2}$$

where

$$t_i = e^{-\tau(i\Delta x + s_0)\Delta x} \tag{29.3}$$

is the transparency of the ith segment and

$$g_i = g(i\Delta x + s_0)\Delta x \tag{29.4}$$

is the source term for the ith segment. Note that both the discretized transparency t_i and source term g_i depend on the sampling distance Δx, i.e., the transparency and source terms have to be modified whenever Δx is changed. In texture-based volume rendering, fragments on slice i are assigned opacities $\alpha_i = 1 - t_i$ and gray values or colors g_i.

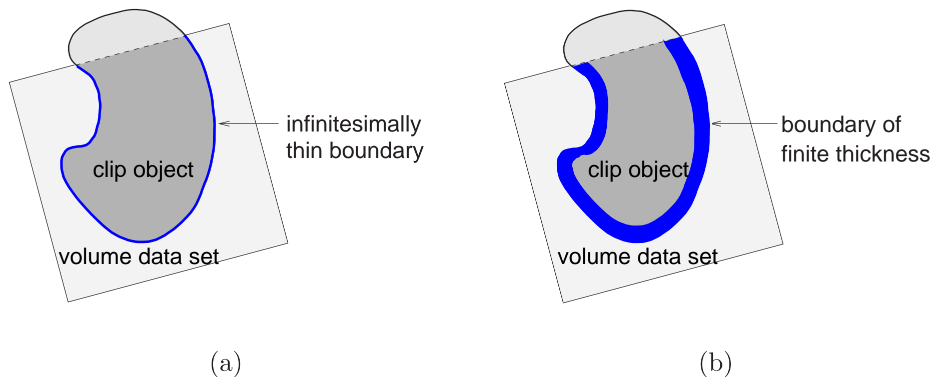


Figure 29.2: Combining volume shading for a volumetric data set with surface-based shading on the boundary of the clip geometry: an infinitesimally thin "skin" around the clip object in (a), a thick "impregnating" layer in (b).

In a first approach to combining clipping and volume shading, unmodified clipping techniques (from the two previous chapters) are first applied the shaded volume. Volume illumination is based on the gradient of the scalar field and the transfer function. In a second step, the illuminated clipping surface is layered around the volume. The lighting of the clipping surface is based on the normal vector of the surface. The surface of the clip object can be pictured as a "skin" surrounding the actual volume, as illustrated in Figure 29.2 (a).

Unfortunately, this first approach is not invariant if the sampling rate Δx changes. In the discretized form of Eq. (29.4), the source term g_i converges to zero for $\Delta x \to 0$, i.e., the color contribution of a 2D surface (of infinitely small thickness) goes to zero. Analogously, the transparency t_i from Eq. (29.3) goes to one for $\Delta x \to 0$.

This problem can be overcome by modifying the first approach in a way that the "skin" is widened to a finite thickness, i.e., the volume is covered by a thick layer whose illumination is based on the normal vectors of the clipping surface. The influence of the illuminated surface reaches into the volume for some distance, which can be illustrated as "impregnating" the shaded volume. Figures 29.2 (a) and (b) compare both the original "skin" approach and the modified "impregnation" approach.

Combining Clipping and Volume Shading in an Object-Space Approach

The "impregnation" approach leads to the following object-space algorithm for combining clipping and volume shading. The clipping object is defined as the isosurface for the isovalue zero on a signed distance volume (as already used for volumetric clipping with tagged volumes). The transition between surface-oriented and volume-oriented shading (in the "impregnation" layer) is based on the values from the distance field.

Surface-based illumination makes used of the gradients of the distance volume and the optical properties of the scalar field. The gradient is identical to the normal vector of a respective isosurface and, thus, represents the normal vector on the clipping surface.

Depth-Based Clipping and Volume Shading

The above object-space approach needs some modifications before it can be applied to depth-based clipping. Now, the clipping boundary is not modeled as a thick layer but as a 2D surface. Surface-based shading on the clip geometry yields modified contributions to transparency $t_{\mathrm{srf}} = e^{-\tau(s_{\mathrm{srf}})\Delta x_{\mathrm{srf}}}$ and source term $g_{\mathrm{srf}} = g(s_{\mathrm{srf}})\Delta_{\mathrm{srf}}$, where s_{srf} describes the location of the clipping surface and Δ_{srf} represents the thickness of the original "impregnation" layer, which is not dependent on the sampling rate. The optical properties are determined by the volume data set and the transfer function. In other words, the geometry of the clip boundary is treated as 2D surface, while the contribution to the rendering integral comes from a thick virtual layer.

To achieve consistent shading for depth-based clipping techniques, the original slice-based volume rendering is extended to hybrid volume and surface rendering. The volume and the surface parts are interleaved according to the depth structure of the clip geometry. A multi-pass rendering approach is used to combine the surface and volumetric contributions: Clipped volumetric regions (clipping is based on "Multi-Pass Rendering for Concave Clip Objects" from Chapter 27) and shaded surfaces are alternatingly rendered in a back-to-front fashion.

Example Images

Figures 29.3 and 29.4 demonstrate the consistent combination of clipping and volume shading. Depth-based clipping is applied to a medical CT

data set in Figure 29.3. The transfer function is chosen in a way to achieve opaque material boundaries.

The visualization of the orbital data set in Figure 29.4 reveals both transparent and opaque structures. Figure 29.4 (a) shows the original data set, Figure 29.4 (b) demonstrates depth-based clipping.

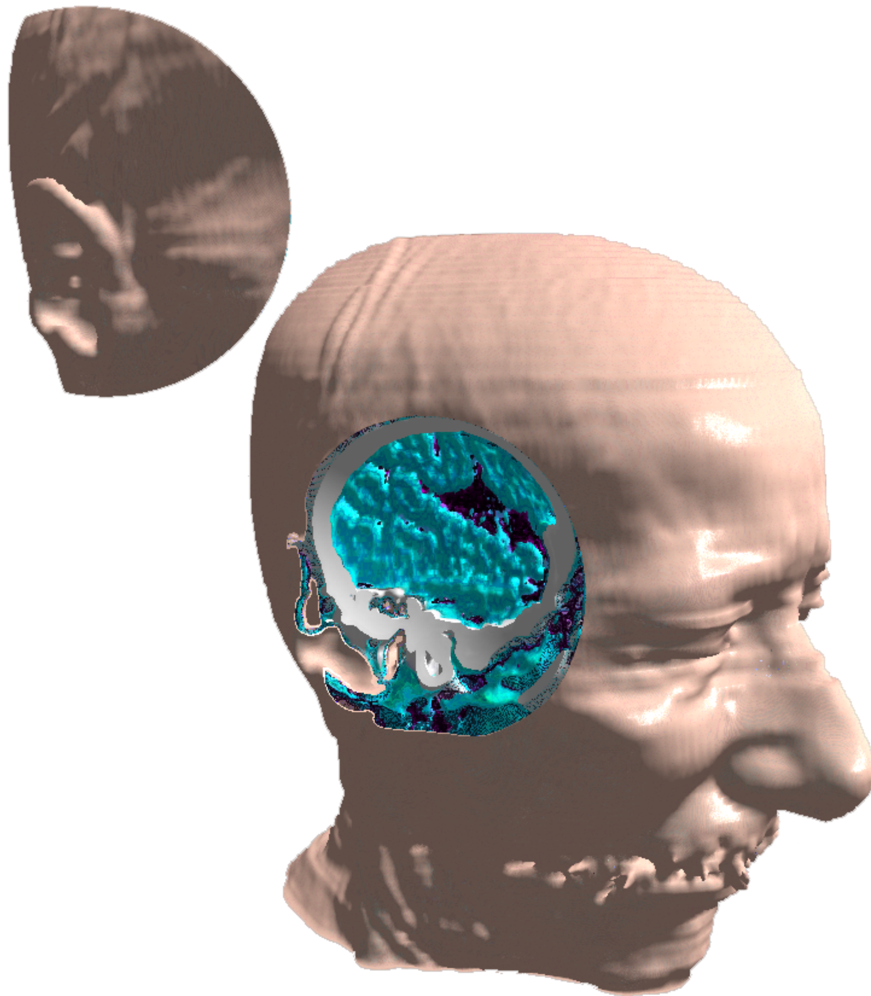
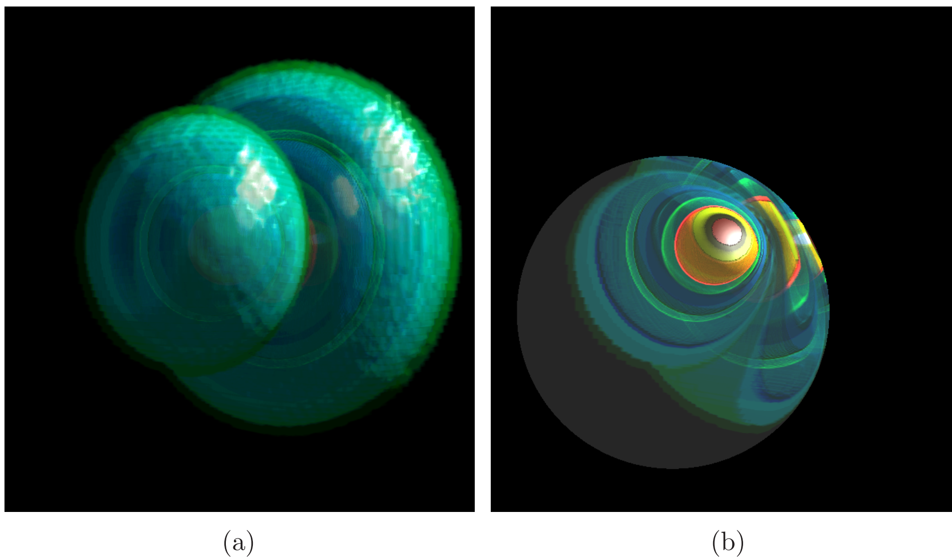


Figure 29.3: Depth-based clipping in an illuminated CT data set.



(a) (b)

Figure 29.4: Clipping in an illuminated orbital data set. Image (a) is without clipping, image (b) is with depth-based clipping.

Clipping and Pre-Integration

The aforementioned clipping approaches are based on a slice-by-slice representation of the volume data set. Therefore, the visibility computation can be reduced to checking whether a fragment on a slice is visible or not. Pre-integrated volume rendering (see Part VII on "Pre-Integration") is a modification of this slice-by-slice rendering. Pre-integration makes use of slabs between two neighboring slices, i.e., the basic building blocks have a finite thickness. To combine pre-integration with volumetric clipping three steps are necessary [84]:

- Modify the scalar data values within a clipped slab
- Adapt the length of a ray segment

In what follows, we use a volumetric description of the clip geometry via a signed distance field (as in Chapter 28 on "Clipping via Tagged Volumes"). By shifting the distance values, the clipping surface is assumed to be located at isovalue 0.5. Let d_f and d_b be the distance values of the clip volume at the entry and exit points of the ray segment (i.e., of the slab). If both distance values are below or above 0.5, the complete slab is either invisible or visible and no special treatment is necessary. Considering the case $d_f < 0.5$ and $d_b > 0.5$ (as in Figure 30.1), only the dark gray part of the volume has to be rendered. In this case we first have to set the front scalar data value s_f to the data value s'_f at entry point into the clipped region. Then we perform a look-up into the pre-integration table with the parameters (s'_f, s_b) rather than with (s_f, s_b). In general, the parameter s'_f is obtained by

$$r = \left[\frac{[0.5 - d_f]}{d_b - d_f}\right], \quad s'_f = (1-r)s_f + r\,s_b \quad .$$

The brackets denote clamping to the range $[0, 1]$. Similarly, the parameter s_b is replaced by s'_b

$$g = 1 - \left[\frac{[0.5 - d_b]}{d_f - d_b}\right], \quad s'_b = (1-g)s_f + r\,s_b \quad .$$

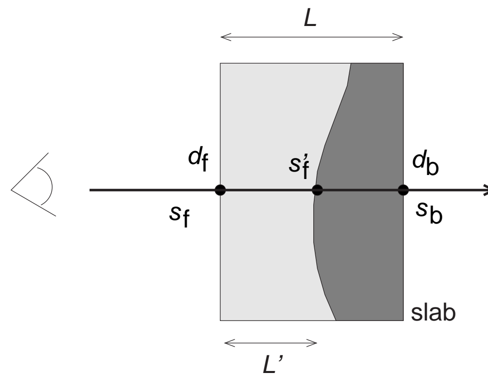


Figure 30.1: Using slabs instead of slices for pre-integrated volume clipping. The scalar data values at the entry and the exit point of the viewing ray are denoted by s_f and s_b, respectively. The corresponding distance values from the clip texture are d_f and d_b. The thickness of the slab is denoted by L; L' is the length of the visible part of the slab. The dark gray region remains after volumetric clipping.

If both clip values are below 0.5, the slab is completely invisible and the scalar values do not matter.

The second issue is the reduced length L of the clipped ray segment. The numerical integration depends on the parameters s_f, s_b, and L. The volume rendering integral (see Equation 29.1) can be re-written, based on the chromaticity vector κ and the scalar optical density ρ in the optical model of William and Max [109], to obtain the ray integral for a ray segment,

$$\begin{aligned}
s_L(x) &= s_f + \frac{x}{L}(s_b - s_f) \\
c(s_f, s_b, L) &= \int_0^L e^{-\int_0^t \rho(s_L(\tau)\,\mathrm{d}\tau} \kappa(s_L(t))\rho(s_L(t))\,\mathrm{d}t \\
\theta(s_f, s_b, L) &= e^{-\int_0^L \rho(s_L(\tau)\,\mathrm{d}\tau} \\
\alpha &= 1 - \theta \quad .
\end{aligned}$$

The reduced ray segment length L' can be taken into account in the following way. The pre-integration table is still based on a constant ray segment length L. The visible fraction of the slab is denoted by $b = L'/L$. Then the transparency θ' of the clipped ray segment is the pre-integrated transparency θ (associated with the original segment length L) raised to

the bth power because

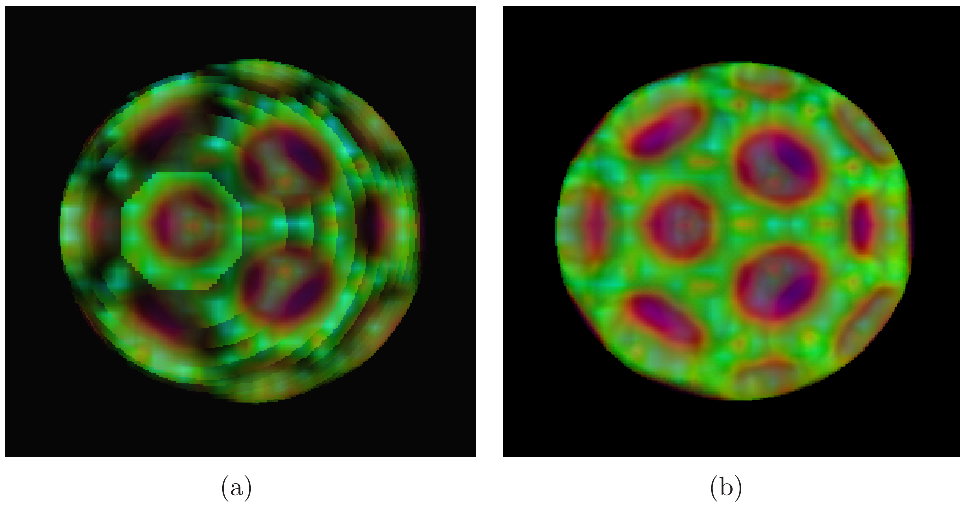
$$\int_0^{L'} \rho(s_{L'}(t'))\,\mathrm{d}t' = b \int_0^{L} \rho(s_L(t))\,\mathrm{d}t \quad ,$$

$$\theta' = e^{-b\int_0^L \rho(s_L(t))\,\mathrm{d}t} = \left(e^{-\int_0^L \rho(s_L(t))\,\mathrm{d}t} \right)^b = \theta^b \quad .$$

A first order approximation is sufficient if the thickness of the slabs is reasonable small. Furthermore, if self-attenuation is neglected, the emission of the clipped ray segment is given by $c' = bc$.

Instead of calculating the factors for the adjustment of the scalar values, the emission, and the opacity in the fragment program, the factors for all combinations of the clip values d_f and d_b can be pre-computed and stored in a 2D dependent texture.

A comparison of the rendering quality between the slab-based approach and the slice-based method from the previous chapters is depicted in Figure 30.2. A sphere has been cut out of the Bucky Ball data set so that the holes of the carbon rings are visible as green spots. Both images have been rendered with only 32 slices / slabs. While the slices are clearly visible on the left, the slab-oriented method reproduces the clipped volume accurately, i.e., it is per-pixel accurate along the viewing direction.



(a) (b)

Figure 30.2: Comparison of volumetric clipping approaches: (a) slice-based, (b) slab-based.

Course Notes 28

Real-Time Volume Graphics

Non-Photorealistic Volume Rendering

Klaus Engel

Siemens Corporate Research, Princeton, USA

Markus Hadwiger

VR VIS Research Center, Vienna, Austria

Joe M. Kniss

SCI, University of Utah, USA

Aaron E. Lefohn

University of California, Davis, USA

Christof Rezk Salama

University of Siegen, Germany

Daniel Weiskopf

University of Stuttgart, Germany

SIGGRAPH2004

Introduction

Non-photorealistic rendering techniques, e.g., rendering styles imitating artistic illustration, have established themselves as a very powerful tool for conveying a specific meaning in rendered images, especially in renderings of surfaces [30, 95]. In recent years, the interest in adapting existing NPR techniques to volumes and creating new entirely volumetric NPR models has increased significantly [20, 15, 71].

This chapter first shows two examples of integrating simple non-photorealistic techniques into real-time volume rendering by evaluating the corresponding shading equations directly in the hardware fragment shader. It then focuses on real-time rendering of isosurfaces with more sophisticated NPR techniques based on implicit curvature information, which has previously been demonstrated for off-line volume rendering [48].

Basic Non-Photorealistic Rendering Modes

This chapter outlines two simple examples of non-photorealistic volume rendering modes. First, the concept of shading surfaces with tone shading [29] can be adapted to volume shading by incorporating color and opacity retrieved from the transfer function. Second, we outline a simple model for rendering the silhouettes of material boundaries in volumes that does not use an explicit notion of surfaces but modulates a contour intensity depending on the angle between view and gradient direction by the gradient magnitude [15].

Tone shading

In contrast to Blinn-Phong shading, which determines a single light intensity depending on the dot product between the view vector and the surface normal, tone shading [29] (sometimes also called Gooch shading) interpolates between two user-specified colors over the full $[-1, 1]$ range of this dot product. Traditionally, one of these colors is set to a warm tone, e.g., orange or yellow, and the other one to a cool tone, e.g., purple or blue. Cool colors are perceived by human observers as receding into the background, whereas warm colors are seen as being closer to the foreground. Tone shading uses this observation to improve depth perception of shaded images.

Although originally developed for surface shading, tone shading can easily be adapted to direct volume rendering by mixing the color from the transfer function with the color obtained via tone shading. One of the possibilities to do this is the following:

$$\mathbf{I} = \left(\frac{1 + \mathbf{l} \cdot \mathbf{n}}{2}\right) k_a + \left(1 - \frac{1 + \mathbf{l} \cdot \mathbf{n}}{2}\right) k_b, \tag{32.1}$$

where $\mathbf{l}$ denotes the light vector, and $\mathbf{n} = \nabla f / |\nabla f|$ is the normalized gradient of the scalar field f that is used as normal vector.

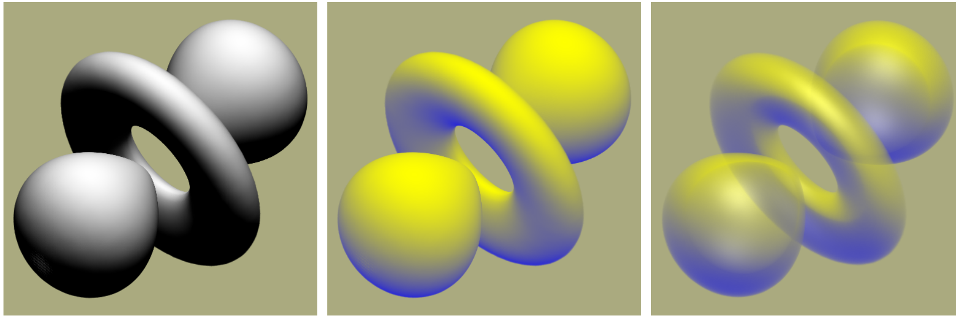


Figure 32.1: Comparison of standard volume shading (left) and tone shading (center) with an isosurface-like transfer function. Incorporating opacity from a transfer function reveals the volumetric structure of the rendered model (right).

The two colors to interpolate, k_a and k_b, are derived from two constant colors k_{cool} and k_{warm} and the color from the transfer function k_t, using two user-specified factors α and β that determine the additive contribution of k_t:

$$k_a = k_{cool} + \alpha k_t \quad (32.2)$$

$$k_b = k_{warm} + \beta k_t \quad (32.3)$$

The opacity of the shaded fragment is determined directly from the transfer function lookup, i.e., the alpha portion of k_t.

These tone shading equations can easily be evaluated in the hardware fragment shader on a per-fragment basis for high-quality results. Figure 32.1 shows example images.

Contour enhancement

Even without an explicit notion of surfaces, or isosurfaces, a very simple model based on gradient magnitude and the angle between the view and gradient direction can visualize the silhouettes of material boundaries in volumes quite effectively [15]. This model can be used in real-time volume rendering for obtaining a contour intensity I by procedural evaluation of the following equation in the hardware fragment shader:

$$\mathbf{I} = g\big(|\nabla f|\big) \cdot \big(1 - |\mathbf{v} \cdot \mathbf{n}|\big)^8, \quad (32.4)$$

where $\mathbf{v}$ is the viewing vector, ∇f denotes the gradient of a given voxel, $\mathbf{n} = \nabla f / |\nabla f|$ is the normalized gradient, and $g(\cdot)$ is a windowing function for the gradient magnitude.

Figure 32.2: Simple contour enhancement based on gradient magnitude and angle between view and local gradient direction. The gradient magnitude windowing function $g(\cdot)$ is an easy way to control contour appearance. These images simply use three different window settings.

The windowing function $g(\cdot)$ is illustrated in figure 32.3, and figure 32.2 shows three example results of using different window settings. The window can be specified directly via its center and width. Alternatively, it can also be specified through a standard transfer function interface, where the alpha component is the weighting factor for the view-dependent part, and the RGB components are simply neglected.

The obtained fragment intensity I can be multiplied by a constant contour color in order to render colored contours. If alpha blending is used as compositing mode, the fragment alpha can simply be set to the intensity I. However, a very useful compositing mode for contours obtained via this technique is maximum intensity projection (MIP), instead of using alpha blending.

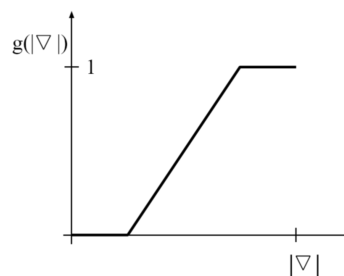


Figure 32.3: Windowing of gradient magnitude in order to restrict the detection of contours to the interfaces, i.e., boundary surfaces, between different materials. From [15].

Combination with segmented data

Using the two non-photorealistic rendering modes outlined above for rendering segmented data with per-object rendering modes is a very powerful approach to emphasizing specific object structures in volume data.

Rendering of contours, for example, is a good way to provide context for focus regions rendered with more traditional volume rendering techniques. Tone shading is naturally suited as shading mode for rendering isosurfaces or structures with high opacity, whereas objects rendered with lower opacity could be rendered with standard direct volume rendering, for example.

See chapter X for examples of combining traditional and non-photorealistic techniques in a single volume rendering in order to enhance perception of individual objects of interest and separate context from focus regions.

Rendering from Implicit Curvature

Computing implicit surface curvature is a powerful tool for isosurface investigation and non-photorealistic rendering of isosurfaces.

This section assumes that an isosurface is shaded using *deferred shading* in image space, as described in chapter IV.

When differential isosurface properties have been computed in preceding deferred shading passes (see section 13), this information can be used for performing a variety of mappings to shaded images in a final shading pass.

Curvature-based transfer functions

Principal curvature magnitudes can be visualized on an isosurface by mapping them to colors via one-dimensional or two-dimensional transfer function lookup textures.

One-dimensional curvature transfer functions. Simple color mappings of first or second principal curvature magnitude via 1D transfer function lookup tables can easily be computed during shading. The same approach can be used to depict additional curvature measures directly derived from the principal magnitudes, such as mean curvature $(\kappa_1 + \kappa_2)/2$ or Gaussian curvature $\kappa_1\kappa_2$ See figures 33.1(left), 33.5(top, left), and 33.7(top, left) for examples.

Two-dimensional curvature transfer functions. Transfer functions in the 2D domain of both principal curvature magnitudes (κ_1, κ_2) are especially powerful, since color specification in this domain allows to highlight different structures on the surface [42], including ridge and valley lines [43, 48]. Curvature magnitude information can also be used to implement silhouette outlining with constant screen space thickness [48]. See figures 33.1, 33.3, 33.5, 33.6, and 33.7 for examples.

Curvature-aligned flow advection

Direct mappings of principle curvature directions to RGB colors are hard to interpret.

However, principal curvature directions on an isosurface can be visualized using image-based flow visualization [100]. In particular, flow can be advected on the surface entirely in image space [66]. These methods can easily be used in real-time, complementing the capability to generate high-quality curvature information on-the-fly, which also yields the underlying, potentially unsteady, "flow" field in real-time. See figure 11.2(f). In this case, it is natural to perform per-pixel advection guided by the floating point image containing principal direction vectors instead of warping mesh vertex or texture coordinates.

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 [100].

See figures 33.2 and 33.3(top, left) for examples.

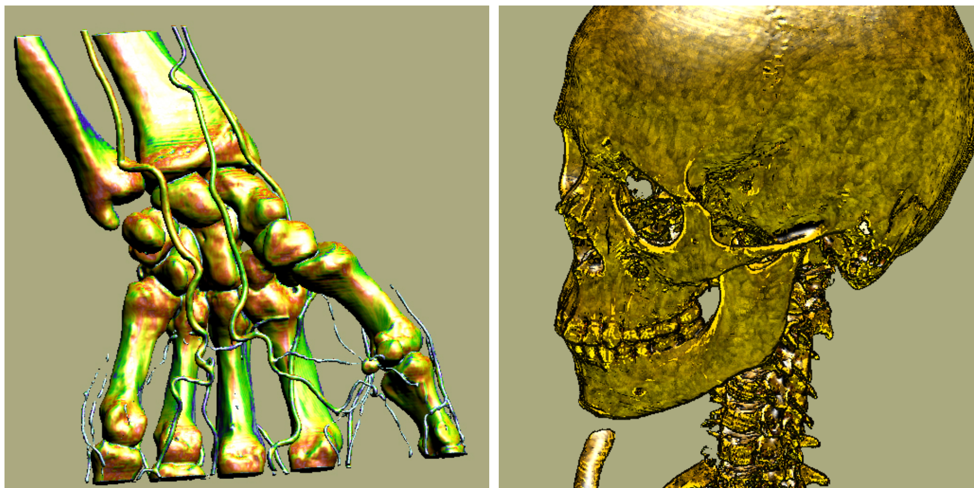


Figure 33.1: Two examples of implicit curvature-based isosurface rendering. (left) CT scan (256x128x256) with color mapping from a 1D transfer function depicting $\sqrt{\kappa_1^2 + \kappa_2^2}$; (right) CT scan (256x256x333) with contours, ridges and valleys, tone shading, and flow advection to generate a noise pattern on the surface.

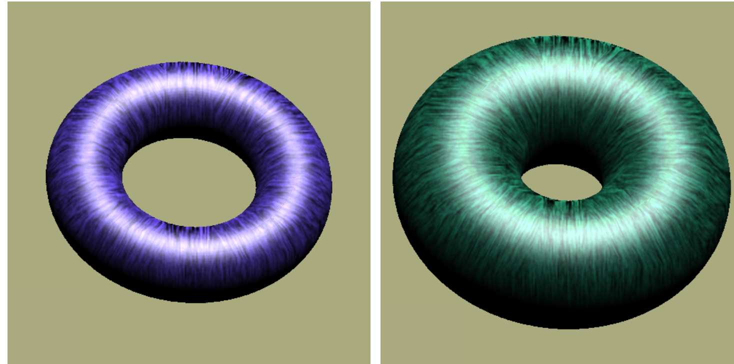


Figure 33.2: Changing the iso-value of a torus isosurface represented by a signed distance field. Maximum principal curvature magnitude color mapping and flow advection.

Figure 33.3: Curvature-based NPR. (top, left) contours, curvature magnitude colors, and flow in curvature direction; (top, right) tone shading and contours; (bottom, left) contours, ridges, and valleys; (bottom, right) flow in curvature direction with Phong shading.

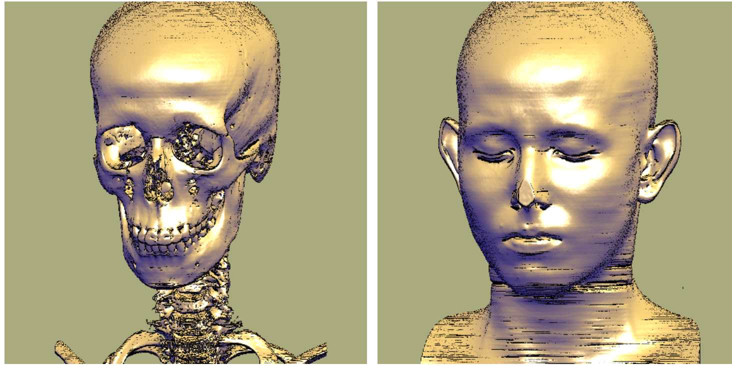


Figure 33.4: CT scan (512x512x333) with tone shading and curvature-controlled contours with ridge and valley lines specified in the (κ_1, κ_2) domain via a 2D transfer function.

Figure 33.5: Dragon distance field (128x128x128) with colors from curvature magnitude (top, left); with Phong shading (top, right); with contours (bottom, left); with ridge and valley lines (bottom, right).

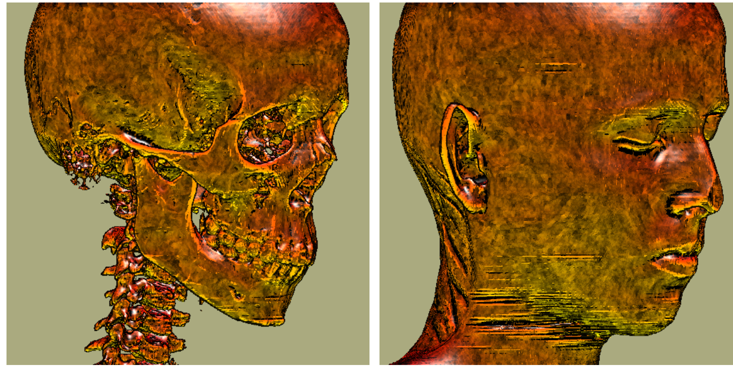


Figure 33.6: CT scan (256x256x333) with contours, ridges and valleys, tone shading, and image space flow advection to generate a noise pattern on the surface.

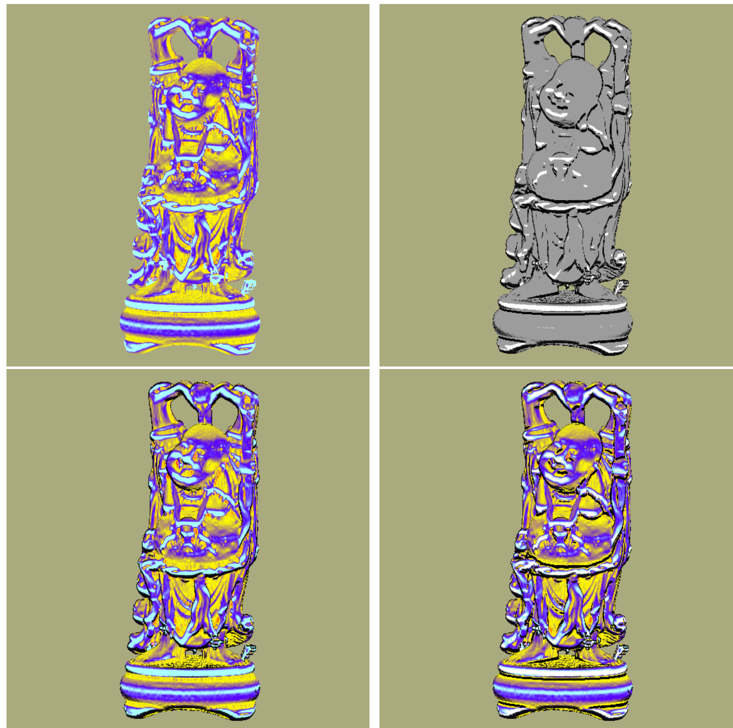


Figure 33.7: Happy Buddha distance field (128x128x128) with colors from curvature magnitude (top, left); only ridge and valley lines (top, right); with contours (bottom, left); with contours, and ridge and valley lines (bottom, right).

Course Notes 28

Real-Time Volume Graphics

Segmented Volumes

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Introduction

One of the most important goals in volume rendering, especially when dealing with medical data, is to be able to visually separate and selectively enable specific objects of interest contained in a single volumetric data set. A very powerful approach to facilitate the perception of individual objects is to create explicit object membership information via *segmentation* [99]. The process of segmentation determines a set of voxels that belong to a given object of interest, usually in the form of one or several *segmentation masks*. There are two major ways of representing segmentation information in masks. First, each object can be represented by a single binary segmentation mask, which determines for each voxel whether it belongs to the given object or not. Second, an object ID volume can specify segmentation information for all objects in a single volume, where each voxel contains the ID of the object it belongs to. These masks can then be used to selectively render only some of the objects contained in a single data set, or render different objects with different optical properties such as transfer functions, for example.

Other approaches for achieving visual distinction of objects are for example rendering multiple semi-transparent isosurfaces, or direct volume rendering with an appropriate transfer function. In the latter approach, multi-dimensional transfer functions [47, 51] have proven to be especially powerful in facilitating the perception of different objects. However, it is often the case that a single rendering method or transfer function does not suffice in order to distinguish multiple objects of interest according to a user's specific needs, especially when spatial information needs to be taken into account. Non-photorealistic volume rendering methods [20, 15, 71] have also proven to be promising approaches for achieving better perception of individual objects.

An especially powerful approach is to combine different non-photorealistic and traditional volume rendering methods in a single volume rendering. When segmentation information is available, different objects can be rendered with individual per-object rendering modes, which allows to use specific modes for structures they are well suited for, as well as separating *focus* from *context*.

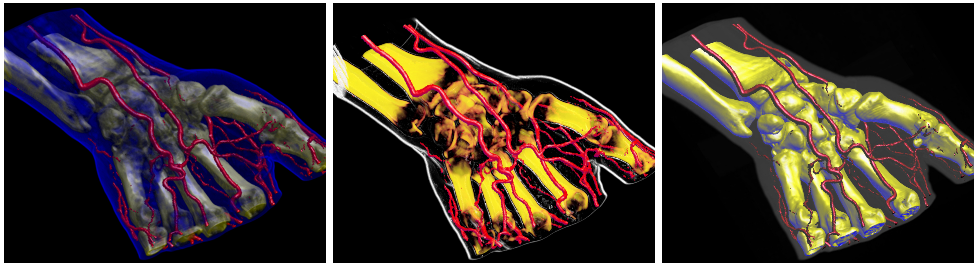


Figure 34.1: Segmented hand data set (256x128x256) with three objects: skin, blood vessels, and bone. Two-level volume rendering integrates different transfer functions, rendering and compositing modes: (left) all objects rendered with shaded DVR; the skin partially obscures the bone; (center) skin rendered with non-photorealistic contour rendering and MIP compositing, bones rendered with DVR, vessels with tone shading; (right) skin rendered with MIP, bones with tone shading, and vessels with shaded isosurfacing; the skin merely provides context.

This Chapter

Integrating segmentation information and multiple rendering modes with different sets of parameters into a fast high-quality volume renderer is not a trivial problem, especially in the case of consumer hardware volume rendering, which tends to only be fast when all or most voxels can be treated identically. On such hardware, one would also like to use a single segmentation mask volume in order to use a minimal amount of texture memory. Graphics hardware cannot easily interpolate between voxels belonging to different objects, however, and using the segmentation mask without filtering gives rise to artifacts. Thus, one of the major obstacles in such a scenario is filtering object boundaries in order to attain high quality in conjunction with consistent fragment assignment and without introducing non-existent object IDs. In this chapter, we show how segmented volumetric data sets can be rendered efficiently and with high quality on current consumer graphics hardware. The segmentation information for object distinction can be used at multiple levels of sophistication, and we describe how all of these different possibilities can be integrated into a single coherent hardware volume rendering framework.

First, different objects can be rendered with the same rendering technique (e.g., DVR), but with different transfer functions. Separate per-object transfer functions can be applied in a single rendering pass even when object boundaries are filtered during rendering. On an ATI Radeon 9700, up to eight transfer functions can be folded into a single rendering pass with linear boundary filtering. If boundaries are only point-sampled,

e.g., during interaction, an arbitrary number of transfer functions can be used in a single pass. However, the number of transfer functions with boundary filtering in a single pass is no conceptual limitation and increases trivially on architectures that allow more instructions in the fragment shader.

Second, different objects can be rendered using different hardware fragment shaders. This allows easy integration of methods as diverse as non-photorealistic and direct volume rendering, for instance. Although each distinct fragment shader requires a separate rendering pass, multiple objects using the same fragment shader with different rendering parameters can effectively be combined into a single pass. When multiple passes cannot be avoided, the cost of individual passes is reduced drastically by executing expensive fragment shaders only for those fragments active in a given pass. These two properties allow highly interactive rendering of segmented data sets, since even for data sets with many objects usually only a couple of different rendering modes are employed. We have implemented direct volume rendering with post-classification, pre-integrated classification [21], different shading modes, non-polygonal isosurfaces, and maximum intensity projection. See figures 34.1 and 34.2 for example images. In addition to non-photorealistic contour enhancement [15] (figure 34.1, center; figure 34.2, skull), we have also used a volumetric adaptation of tone shading [29] (figure 34.1, right), which improves depth perception in contrast to standard shading.

Finally, different objects can also be rendered with different compositing modes, e.g., alpha blending and maximum intensity projection (MIP), for their contribution to a given pixel. These per-object compositing modes are object-local and can be specified independently for each object. The individual contributions of different objects to a single pixel can be combined via a separate global compositing mode. This two-level approach to object compositing [40] has proven to be very useful in order to improve perception of individual objects.

In summary, this chapter presents the following:

- A systematic approach to minimizing both the number of rendering passes and the performance cost of individual passes when rendering segmented volume data with high quality on current GPUs. Both filtering of object boundaries and the use of different rendering parameters such as transfer functions do not prevent using a single rendering pass for multiple objects. Even so, each pass avoids execution of the corresponding potentially expensive fragment shader for irrelevant fragments by exploiting the early z-test.

This reduces the performance impact of the number of rendering passes drastically.

- An efficient method for mapping a single object ID volume to and from a domain where filtering produces correct results even when three or more objects are present in the volume. The method is based on simple 1D texture lookups and able to map and filter blocks of four objects simultaneously.
- An efficient object-order algorithm based on simple depth and stencil buffer operations that achieves correct compositing of objects with different per-object compositing modes and an additional global compositing mode. The result is conceptually identical to being able to switch compositing modes for any given group of samples along the ray for any given pixel.

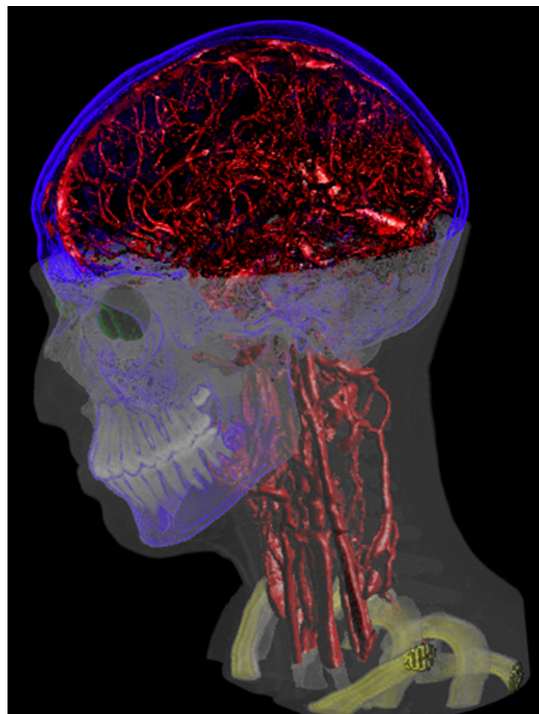


Figure 34.2: Segmented head and neck data set (256x256x333) with six different enabled objects. The skin and teeth are rendered as MIP with different intensity ramps, the blood vessels and eyes are rendered as shaded DVR, the skull uses contour rendering, and the vertebrae use a gradient magnitude-weighted transfer function with shaded DVR. A clipping plane has been applied to the skin object.

Segmented Data Representation

For rendering purposes, we simply assume that in addition to the usual data such as a density and an optional gradient volume, a *segmentation mask volume* is also available. If embedded objects are represented as separate masks, we combine all of these masks into a single volume that contains a single object ID for each voxel in a pre-process. Hence we will also be calling this segmentation mask volume the *object ID volume*. IDs are simply enumerated consecutively starting with one, i.e., we do not assign individual bits to specific objects. ID zero is reserved (see later sections). The object ID volume consumes one byte per voxel and is either stored in its own 3D texture in the case of view-aligned slicing, or in additional 2D slice textures for all three slice stacks in the case of object-aligned slicing. With respect to resolution, we have used the same resolution as the original volume data, but all of the approaches we describe could easily be used for volume and segmentation data of different resolutions.

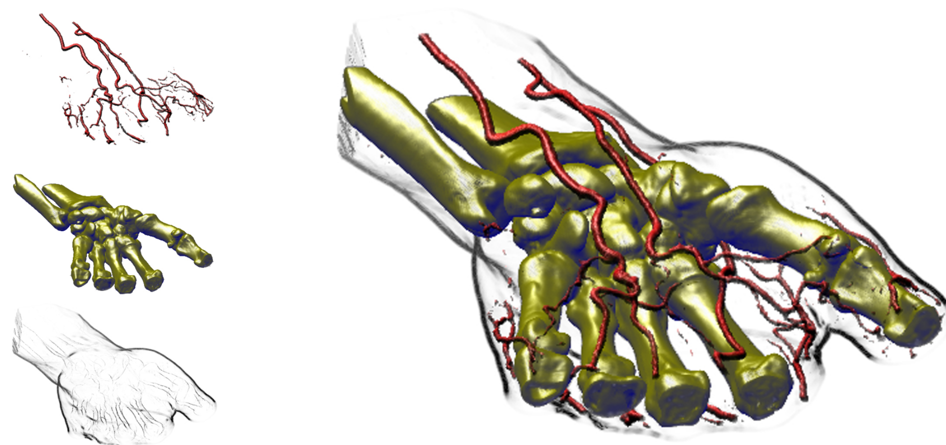


Figure 35.1: CT scan of a human hand (256x128x256) with three segmented objects (skin, blood vessels, and bone structure).

Rendering Segmented Data

In order to render a segmented data set, we determine object membership of individual fragments by filtering object boundaries in the hardware fragment shader (section 37). Object membership determines which transfer function, rendering, and compositing modes should be used for a given fragment.

We render the volume in a number of rendering passes that is basically independent of the number of contained objects. It most of all depends on the required number of different hardware configurations that cannot be changed during a single pass, i.e., the fragment shader and compositing mode. Objects that can share a given configuration can be rendered in a single pass. This also extends to the application of multiple per-object transfer functions (section 37) and thus the actual number of rendering passes is usually much lower than the number of objects or transfer functions. It depends on several major factors:

Enabled objects. If all the objects rendered in a given pass have been disabled by the user, the entire rendering pass can be skipped. If only some of the objects are disabled, the number of passes stays the same, independent of the order of object IDs. Objects are disabled by changing a single entry of a 1D lookup texture. Additionally, per-object clipping planes can be enabled. In this case, all objects rendered in the same pass are clipped identically, however.

Rendering modes. The rendering mode, implemented as an actual hardware fragment shader, determines what and how volume data is re-sampled and shaded. Since it cannot be changed during a single rendering pass, another pass must be used if a different fragment shader is required. However, many objects often use the same basic rendering mode and thus fragment shader, e.g., DVR and isosurfacing are usually used for a large number of objects.

Transfer functions. Much more often than the basic rendering mode, a change of the transfer function is required. For instance, all objects rendered with DVR usually have their own individual transfer functions. In order to avoid an excessive number of rendering passes

due to simple transfer function changes, we apply multiple transfer functions to different objects in a single rendering pass while still retaining adequate filtering quality (section 37).

Compositing modes. Although usually considered a part of the rendering mode, compositing is a totally separate operation in graphics hardware. Where the basic rendering mode is determined by the fragment shader, the compositing mode is specified as blend function and equation in OpenGL, for instance. It determines how already shaded fragments are combined with pixels stored in the frame buffer. Changing the compositing mode happens even more infrequently than changing the basic rendering mode, e.g., alpha blending is used in conjunction with both DVR and tone shading.

Different compositing modes per object also imply that the (conceptual) ray corresponding to a single pixel must be able to combine the contribution of these different modes (figure 38.1). Especially in the context of texture-based hardware volume rendering, where no actual rays exist and we want to obtain the same result with an object-order approach instead, we have to use special care when compositing. The contributions of individual objects to a given pixel should not interfere with each other, and are combined with a single global compositing mode.

In order to ensure correct compositing, we are using two render buffers and track the current compositing mode for each pixel. Whenever the compositing mode changes for a given pixel, the already composited part is transferred from the *local compositing buffer* into the *global compositing buffer*. Section 38 shows that this can actually be done very efficiently without explicitly considering individual pixels, while still achieving the same compositing behavior as a ray-oriented image-order approach, which is crucial for achieving high quality. For faster rendering we allow falling back to single-buffer compositing during interaction (figure 38.2).

Basic rendering loop

We will now outline the basic rendering loop that we are using for each frame. Table 36.1 gives a high-level overview.

Although the user is dealing with individual objects, we automatically collect all objects that can be processed in the same rendering pass into an *object set* at the beginning of each frame. For each object set, we generate an *object set membership texture*, which is a 1D lookup table that determines the objects belonging to the set. In order to further distinguish different transfer functions in a single object set, we also

generate 1D *transfer function assignment textures*. Both of these types of textures are shown in figure 36.2 and described in sections 36 and 37.

After this setup, the entire slice stack is rendered. Each slice must be rendered for every object set containing an object that intersects the slice, which is determined in a pre-process. In the case of 3D volume textures, all slices are always assumed to be intersected by all objects, since they are allowed to cut through the volume at arbitrary angles. If there is more than a single object set for the current slice, we optionally render all object set IDs of the slice into the depth buffer before rendering any actual slice data. This enables us to exploit the early z-test during all subsequent passes for each object set, see below. For performance reasons, we never use object ID filtering in this pass, which allows only conservative fragment culling via the depth test. Exact fragment rejection is done in the fragment shader.

We proceed by rendering actual slice data. Before a slice can be rendered for any object set, the fragment shader and compositing mode corresponding to this set must be activated. Using the two types of textures mentioned above, the fragment shader filters boundaries, rejects fragments not corresponding to the current pass, and applies the correct transfer function.

In order to attain two compositing levels, slices are rendered into a local buffer, as already outlined above. Before rendering the current slice,

```
DetermineObjectSets();
CreateObjectSetMembershipTextures();
CreateTFAssignmentTextures();
FOR each slice DO
  TransferLocalBufferIntoGlobalBuffer();
  ClearTransferredPixelsInLocalBuffer();
  RenderObjectIdDepthImageForEarlyZTest();
  FOR each object set with an object in slice DO
    SetupObjectSetFragmentRejection();
    SetupObjectSetTFAssignment();
    ActivateObjectSetFragmentShader();
    ActivateObjectSetCompositingMode();
    RenderSliceIntoLocalBuffer();
```

Table 36.1: The basic rendering loop that we are using. Object set membership can change every time an object's rendering or compositing mode is changed, or an object is enabled or disabled.

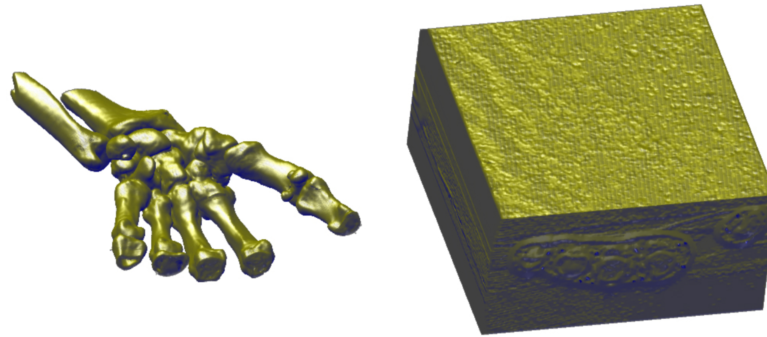


Figure 36.1: In order to render the bone structure shown on the left, many voxels need to be culled. The early z-test allows to avoid evaluating shading equations for culled voxels. If it is not employed, performance will correspond to shading all voxels, shown on the right.

those pixels where the local compositing mode differs from the previous slice are transferred from the local into the global buffer using the global compositing mode. After this transfer, the transferred pixels are cleared in the local buffer to ensure correct local compositing for subsequent pixels. In the case when only a single compositing buffer is used for approximate compositing, the local to global buffer transfer and clear are not executed.

Finally, if the global compositing buffer is separate from the viewing window, it has to be transferred once after the entire volume has been rendered.

Conservative fragment culling via early z-test

On current graphics hardware, it is possible to avoid execution of the fragment shader for fragments where the depth test fails as long as the shader does not modify the depth value of the fragment. This early z-test is crucial to improving performance when multiple rendering passes have to be performed for each slice.

If the current slice's object set IDs have been written into the depth buffer before, see above, we conservatively reject fragments not belonging to the current object set even before the corresponding fragment shader is started. In order to do this, we use a depth test of `GL_EQUAL` and configure the vertex shader to generate a constant depth value for each fragment that exactly matches the current object set ID.

Fragment shader operations

Most of the work in volume renderers for consumer graphics hardware is done in the fragment shader, i.e., at the granularity of individual fragments and, ultimately, pixels. In contrast to approaches using lookup tables, i.e., paletted textures, we are performing all shading operations procedurally in the fragment shader. However, we are most of all interested in the operations that are required for rendering segmented data. The two basic operations in the fragment shader with respect to the segmentation mask are fragment rejection and per-fragment application of transfer functions:

Fragment rejection. Fragments corresponding to object IDs that cannot be rendered in the current rendering pass, e.g., because they need a different fragment shader or compositing mode, have to be rejected. They, in turn, will be rendered in another pass, which uses an appropriately adjusted rejection comparison.

For fragment rejection, we do not compare object IDs individually, but use 1D lookup textures that contain a binary membership status for each object (figure 36.2, left). All objects that can be rendered in the same pass belong to the same object set, and the corresponding object set membership texture contains ones at exactly those texture coordinates corresponding to the IDs of these objects, and zeros everywhere else. The re-generation of these textures at the beginning of each frame, which is negligible in terms of performance, also makes turning individual objects on and off trivial. Exactly one object set membership texture is active for

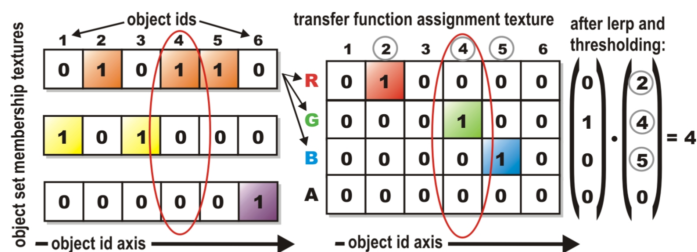


Figure 36.2: Object set membership textures (left; three 1D intensity textures for three sets containing three, two, and one object, respectively) contain a binary membership status for each object in a set that can be used for filtering object IDs and culling fragments. Transfer function assignment textures (right; one 1D RGBA texture for distinction of four transfer functions) are used to filter four object boundaries simultaneously and determine the corresponding transfer function via a simple dot product.

a given rendering pass and makes the task of fragment rejection trivial if the object ID volume is point-sampled.

When object IDs are filtered, it is also crucial to map individual IDs to zero or one before actually filtering them. Details are given in section 37, but basically we are using object set membership textures to do a binary classification of input IDs to the filter, and interpolate after this mapping. The result can then be mapped back to zero or one for fragment rejection.

Per-fragment transfer function application. Since we apply different transfer functions to multiple objects in a single rendering pass, the transfer function must be applied to individual fragments based on their density value and corresponding object ID. Instead of sampling multiple one-dimensional transfer function textures, we sample a single global two-dimensional transfer function texture (figure 36.3). This texture is not only shared between all objects of an object set, but also between all object sets. It is indexed with one texture coordinate corresponding to the object ID, the other one to the actual density.

Because we would like to filter linearly along the axis of the actual transfer function, but use point-sampling along the axis of object IDs, we store each transfer function twice at adjacent locations in order to guarantee point-sampling for IDs, while we are using linear interpolation for the entire texture. We have applied this scheme only to 1D transfer functions, but general 2D transfer functions could also be implemented via 3D textures of just a few layers in depth, i.e., the number of different transfer functions.

We are using an extended version of the pixel-resolution filter that we employ for fragment rejection in order to determine which of multiple transfer functions in the same rendering pass a fragment should actually

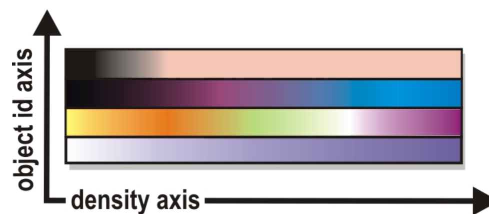


Figure 36.3: Instead of multiple one-dimensional transfer functions for different objects, we are using a single global two-dimensional transfer function texture. After determining the object ID for the current fragment via filtering, the fragment shader appropriately samples this texture with $(density, object_id)$ texture coordinates.

use. Basically, the fragment shader uses multiple RGBA transfer function assignment textures (figure 36.2, right) for both determining the transfer function and rejecting fragments, instead of a single object set membership texture with only a single color channel. Each one of these textures allows filtering the object ID volume with respect to four object boundaries simultaneously. A single lookup yields binary membership classification of a fragment with respect to four objects. The resulting RGBA membership vectors can then be interpolated directly. The main operation for mapping back the result to an object ID is a simple dot product with a constant vector of object IDs. If the result is the non-existent object ID of zero, the fragment needs to be rejected. The details are described in section 37.

This concept can be extended trivially to objects sharing transfer functions by using transfer function IDs instead of object IDs. The following two sections will now describe filtering of object boundaries at sub-voxel precision in more detail.

Boundary Filtering

One of the most crucial parts of rendering segmented volumes with high quality is that the object boundaries must be calculated during rendering at the pixel resolution of the output image, instead of the voxel resolution of the segmentation volume. Figure 37.1 (left) shows that simply point-sampling the object ID texture leads to object boundaries that are easily discernible as individual voxels. That is, simply retrieving the object ID for a given fragment from the segmentation volume is trivial, but causes artifacts. Instead, the object ID must be determined via filtering for each fragment individually, thus achieving pixel-resolution boundaries.

Unfortunately, filtering of object boundaries cannot be done directly using the hardware-native linear interpolation, since direct interpolation of numerical object IDs leads to incorrectly interpolated intermediate values when more than two different objects are present. When filtering object IDs, a threshold value s_t must be chosen that determines which object a given fragment belongs to, which is essentially an iso-surfacing problem.

However, this cannot be done if three or more objects are contained in the volume, which is illustrated in the top row of figure 37.2. In that case, it is not possible to choose a single s_t for the entire volume. The crucial observation to make in order to solve this problem is that the segmentation volume must be filtered as a successive series of binary

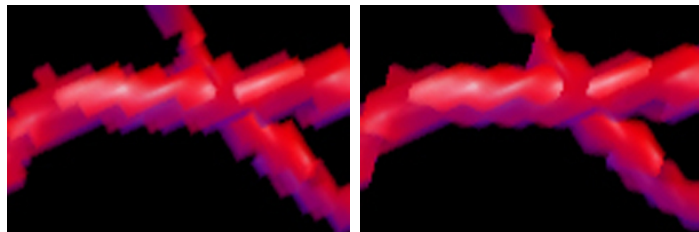


Figure 37.1: Object boundaries with voxel resolution (left) vs. object boundaries determined per-fragment with linear filtering (right).

volumes in order to achieve proper filtering [97], which is shown in the second row of figure 37.2. Mapping all object IDs of the current object set to 1.0 and all other IDs to 0.0 allows using a global threshold value s_t of 0.5. We of course do not want to store these binary volumes explicitly, but perform this mapping on-the-fly in the fragment shader by indexing the *object set membership texture* that is active in the current rendering pass. Filtering in the other passes simply uses an alternate binary mapping, i.e., other object set membership textures.

One problem with respect to a hardware implementation of this approach is that texture filtering happens before the sampled values can be altered in the fragment shader. Therefore, we perform filtering of object IDs directly in the fragment shader. Note that our approach could in part also be implemented using texture palettes and hardware-native linear interpolation, with the restriction that not more than four transfer functions can be applied in a single rendering pass (section 37). However, we have chosen to perform all filtering in the fragment shader in order to create a coherent framework with a potentially unlimited number of transfer functions in a single rendering pass and prepare for the possible use of cubic boundary filtering in the future.

After filtering yields values in the range [0.0, 1.0], we once again come to a binary decision whether a given fragment belongs to the current

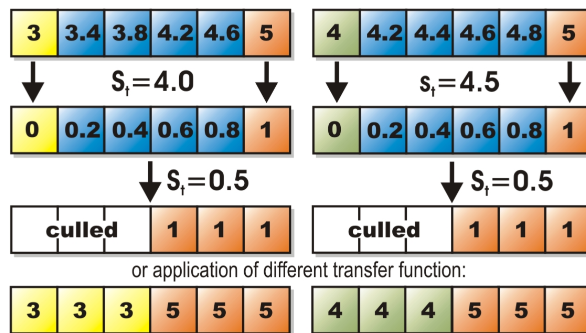


Figure 37.2: Each fragment must be assigned an exactly defined object ID after filtering. Here, IDs 3, 4, and 5 are interpolated, yielding the values shown in blue. Top row: choosing a single threshold value s_t that works everywhere is not possible for three or more objects. Second row: object IDs must be converted to 0.0 or 1.0 in the fragment shader before interpolation, which allows using a global s_t of 0.5. After thresholding, fragments can be culled accordingly (third row), or mapped back to an object ID in order to apply the corresponding transfer function (fourth row).

object set by comparing with a threshold value of 0.5 and rejecting fragments with an interpolated value below this threshold (figure 37.2, third row).

Actual rejection of fragments is done using the `KIL` instruction of the hardware fragment shader that is available in the `ARB_fragment_program` OpenGL extension, for instance. It can also be done by mapping the fragment to RGBA values constituting the identity with respect to the current compositing mode (e.g., an alpha of zero for alpha blending), in order to not alter the frame buffer pixel corresponding to this fragment.

Linear boundary filtering. For object-aligned volume slices, bi-linear interpolation is done by setting the hardware filtering mode for the object ID texture to nearest-neighbor and sampling it four times with offsets of whole texels in order to get access to the four ID values needed for interpolation. Before actual interpolation takes place, the four object IDs are individually mapped to 0.0 or 1.0, respectively, using the current object set membership texture.

We perform the actual interpolation using a variant of texture-based filtering [34], which proved to be both faster and use fewer instructions than using `LRP` instructions. With this approach, bi-linear weight calculation and interpolation can be reduced to just one texture fetch and one dot product. When intermediate slices are interpolated on-the-fly [83], or view-aligned slices are used, eight instead of four input IDs have to be used in order to perform tri-linear interpolation.

Combination with pre-integration. The combination of pre-integration [21] and high-quality clipping has been described recently [85]. Since our filtering method effectively reduces the segmentation problem to a clipping problem on-the-fly, we are using the same approach after we have mapped object IDs to 0.0 or 1.0, respectively. In this case, the interpolated binary values must be used for adjusting the pre-integration lookup.

Multiple per-object transfer functions in a single rendering pass

In addition to simply determining whether a given fragment belongs to a currently active object or not, which has been described in the previous section, this filtering approach can be extended to the application of multiple transfer functions in a single rendering pass without sacrificing filtering quality. Figure 37.3 shows the difference in quality for two objects with different transfer functions (one entirely red, the other entirely yellow for illustration purposes).

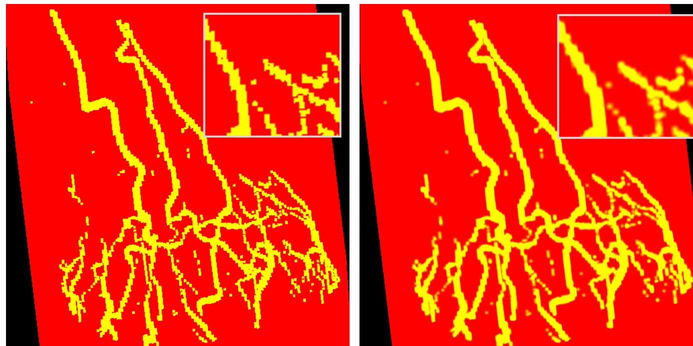


Figure 37.3: Selecting the transfer function on a per-fragment basis. In the left image, point-sampling of the object ID volume has been used, whereas in the right image procedural linear interpolation in the fragment shader achieves results of much better quality.

In general hardware-accelerated volume rendering, the easiest way to apply multiple transfer functions in a single rendering pass would be to use the original volume texture with linear interpolation, and an additional separate point-sampled object ID texture. Although actual volume and ID textures could be combined into a single texture, the use of a separate texture to store the IDs is mandatory in order to prevent that filtering of the actual volume data also reverts back to point-sampling, since a single texture cannot use different filtering modes for different channels and point-sampling is mandatory for the ID texture. The hardware-native linear interpolation cannot be turned on in order to filter object IDs, and thus the resolution of the ID volume is easily discernible if the transfer functions are sufficiently different.

In order to avoid the artifacts related to point-sampling the ID texture, we perform several almost identical filtering steps in the fragment shader, where each of these steps simultaneously filters the object boundaries of four different objects. After the fragment's object ID has been determined via filtering, it can be used to access the global transfer function table as described in section 36 and illustrated in figure 36.3. For multiple simultaneous transfer functions, we do not use object set membership textures but the similar extended concept of *transfer function assignment textures*, which is illustrated in the right image of figure 36.2.

Each of these textures can be used for filtering the object ID volume with respect to four different object IDs at the same time by using the four channels of an RGBA texture in order to perform four simultaneous binary classification operations. In order to create these textures, each

object set membership texture is converted into $\lceil \#objects/4 \rceil$ transfer function assignment textures, where $\#objects$ denotes the number of objects with different transfer functions in a given object set. All values of 1.0 corresponding to the first transfer function are stored into the red channel of this texture, those corresponding to the second transfer function into the green channel, and so on.

In the fragment shader, bi-linear interpolation must index this texture at four different locations given by the object IDs of the four input values to interpolate. This classifies the four input object IDs with respect to four objects with just four 1D texture sampling operations. A single linear interpolation step yields the linear interpolation of these four object classifications, which can then be compared against a threshold of $(0.5, 0.5, 0.5, 0.5)$, also requiring only a single operation for four objects. Interpolation and thresholding yields a vector with at most one component of 1.0, the other components set to 0.0. In order for this to be true, we require that interpolated and thresholded repeated binary classifications never overlap, which is not guaranteed for all types of filter kernels. In the case of bi-linear or tri-linear interpolation, however, overlaps can never occur [97].

The final step that has to be performed is mapping the binary classification to the desired object ID. We do this via a single dot product with a vector containing the four object IDs corresponding to the four channels of the transfer function assignment texture (figure 36.2, right). By calculating this dot product, we multiply exactly the object ID that should be assigned to the final fragment by 1.0. The other object IDs are multiplied by 0.0 and thus do not change the result. If the result of the dot product is 0.0, the fragment does not belong to any of the objects under consideration and can be culled. Note that exactly for this reason, we do not use object IDs of zero.

For the application of more than four transfer functions in a single rendering pass, the steps outlined above can be executed multiple times in the fragment shader. The results of the individual dot products are simply summed up, once again yielding the ID of the object that the current fragment belongs to.

Note that the calculation of filter weights is only required once, irrespective of the number of simultaneous transfer functions, which is also true for sampling the original object ID textures.

Equation 37.1 gives the major fragment shader resource requirements of our filtering and binary classification approach for the case of bi-linear

interpolation with LRP instructions:

$$4\mathbf{TEX_2D} + 4\left\lceil\frac{\#objects}{4}\right\rceil\mathbf{TEX_1D} + 3\left\lceil\frac{\#objects}{4}\right\rceil\mathbf{LRP}, \qquad (37.1)$$

in addition to one dot product and one thresholding operation (e.g., DP4 and SGE instructions, respectively) for every $\lceil\#objects/4\rceil$ transfer functions evaluated in a single pass.

Similarly to the alternative linear interpolation using texture-based filtering that we have outlined in section 37, procedural weight calculation and the LRP instructions can once again also be substituted by texture fetches and a few cheaper ALU instructions. On the Radeon 9700, we are currently able to combine high-quality shading with up to eight transfer functions in the same fragment shader, i.e., we are using up to two transfer function assignment textures in a single rendering pass.

Two-Level Volume Rendering

The final component of the framework presented in this chapter with respect to the separation of different objects is the possibility to use individual object-local compositing modes, as well as a single global compositing mode, i.e., *two-level volume rendering* [40]. The local compositing modes that can currently be selected are alpha blending (e.g., for DVR or tone shading), maximum intensity projection (e.g., for MIP or contour enhancement), and isosurface rendering. Global compositing can either be done by alpha blending, MIP, or a simple add of all contributions.

Although the basic concept is best explained using an image-order approach, i.e., individual rays (figure 38.1), in the context of texture-based volume rendering we have to implement it in object-order. As described in section 36, we are using two separate rendering buffers, a local and a global compositing buffer, respectively. Actual volume slices

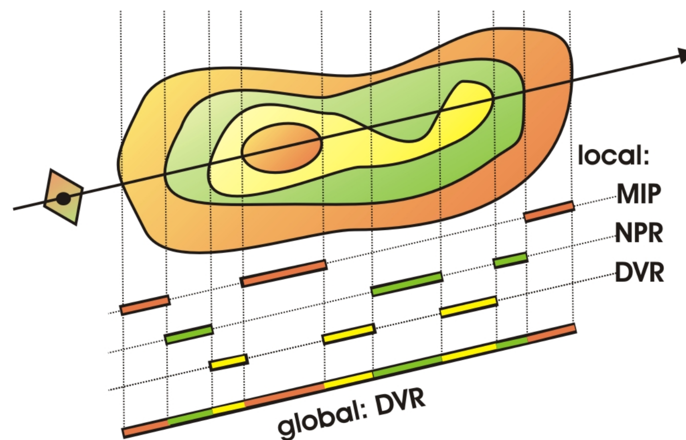


Figure 38.1: A single ray corresponding to a given image pixel is allowed to pierce objects that use their own object-local compositing mode. The contributions of different objects along a ray are combined with a single global compositing mode. Rendering a segmented data set with these two conceptual levels of compositing (local and global) is known as *two-level volume rendering*.

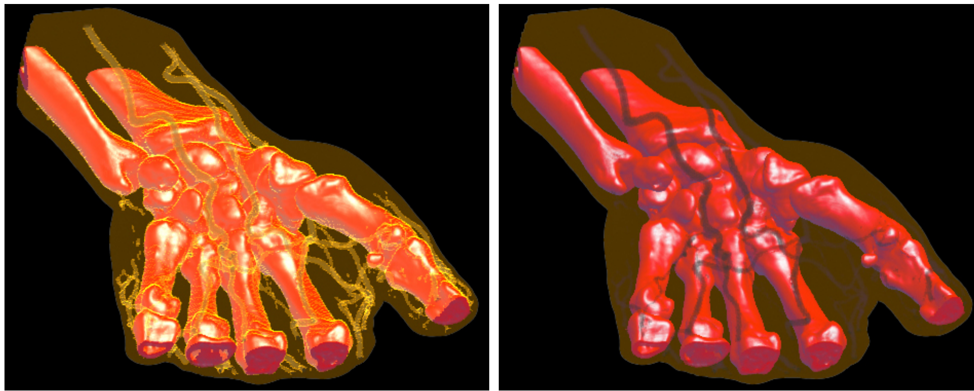


Figure 38.2: Detecting changes in compositing mode for each individual sample along a ray can be done exactly using two rendering buffers (left), or approximately using only a single buffer (right).

are only rendered into the local buffer, using the appropriate local compositing mode. When a new fragment has a different local compositing mode than the pixel that is currently stored in the local buffer, that pixel has to be transferred into the global buffer using the global compositing mode. Afterward, these transferred pixels have to be cleared in the local buffer before the corresponding new fragment is rendered. Naturally, it is important that both the detection of a change in compositing mode and the transfer and clear of pixels is done for all pixels simultaneously.

In order to do this, we are using the depth buffer of both the local

```
TransferLocalBufferIntoGlobalBuffer() {
  ActivateContextGlobalBuffer();
  DepthTest( NOT_EQUAL );
  StencilTest( RENDER_ALWAYS, SET_ONE );
  RenderSliceCompositingIds( DEPTH_BUFFER );
  DepthTest( DISABLE );
  StencilTest( RENDER_WHERE_ONE, SET_ZERO );
  RenderLocalBufferImage( COLOR_BUFFER );
}
```

Table 38.1: Detecting for all pixels simultaneously where the compositing mode changes from one slice to the next, and transferring those pixels from the local into the global compositing buffer.

and the global compositing buffer to track the current local compositing mode of each pixel, and the stencil buffer to selectively enable pixels where the mode changes from one slice to the next. Before actually rendering a slice (see table 36.1), we render IDs corresponding to the local compositing mode into both the local and the global buffer's depth buffer. During these passes, the stencil buffer is set to one where the ID already stored in the depth buffer (from previous passes) differs from the ID that is currently being rendered. This gives us both an updated ID image in the depth buffer, and a stencil buffer that identifies exactly those pixels where a change in compositing mode has been detected.

We then render the image of the local buffer into the global buffer. Due to the stencil test, pixels will only be rendered where the compositing mode has actually changed. Table 38.1 gives pseudo code for what is happening in the global buffer. Clearing the just transferred pixels in

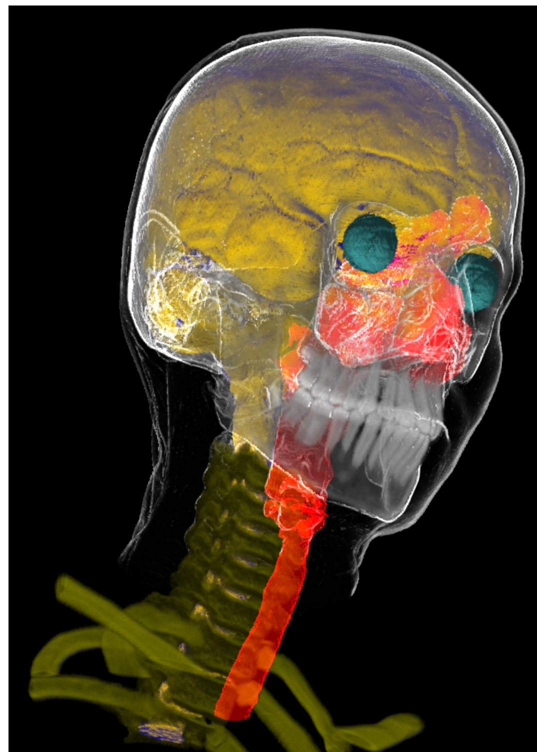


Figure 38.3: Segmented head and neck data set (256x256x333) with eight different enabled objects – brain: tone shading; skin: contour enhancement with clipping plane; eyes and spine: shaded DVR; skull, teeth, and vertebrae: unshaded DVR; trachea: MIP.

the local buffer works almost identically. The only difference is that in this case we do not render the image of another buffer, but simply a quad with all pixels set to zero. Due to the stencil test, pixels will only be cleared where the compositing mode has actually changed.

Note that all these additional rendering passes are much faster than the passes actually rendering and shading volume slices. They are independent of the number of objects and use extremely simple fragment shaders. However, the buffer/context switching overhead is quite noticeable, and thus correct separation of compositing modes can be turned off during interaction. Figure 38.2 shows a comparison between approximate and correct compositing with one and two compositing buffers, respectively. Performance numbers can be found in table 39.1. When only a single buffer is used, the compositing mode is simply switched according to each new fragment without avoiding interference with the previous contents of the frame buffer.

The visual difference depends highly on the combination of compositing modes and spatial locations of objects. The example in figure 38.2 uses MIP and DVR compositing in order to highlight the potential differences. However, using approximate compositing is very useful for faster rendering, and often exhibits little or no loss in quality. Also, it is possible to get an almost seamless performance/quality trade-off between the two, by performing the buffer transfer only every n slices instead of every slice.

See figures 39.1 and 39.2 for additional two-level volume renderings of segmented volume data.

Performance

Actual rendering performance depends on a lot of different factors, so table 39.1 shows only some example figures. In order to concentrate on performance of rendering segmented data, all rates have been measured with unshaded DVR. Slices were object-aligned; objects were rendered all in a single pass (*single*) or in one pass per object (*multi+ztest*).

Compositing performance is independent of the rendering mode, i.e., can also be measured with DVR for all objects. Frame rates in parentheses are with linear boundary filtering enabled, other rates are for point-sampling during interaction. Note that in the unfiltered case with a single rendering pass for all objects, the performance is independent of the number of objects.

If more complex fragment shaders than unshaded DVR are used, the relative performance speed-up of *multi+ztest* versus *multi* increases further toward *single* performance, i.e., the additional overhead of writing object set IDs into the depth buffer becomes negligible.

#slices	#objects	compositing	single	multi+ztest	*multi*
128	3	one buffer	48 (16.2)	29.2 (15.4)	*19.3 (6.8)*
128	3	two buffers	7 (3.9)	6.2 (3.2)	*5 (1.9)*
128	8	one buffer	48 (11.3)	15.5 (10)	*7 (2.1)*
128	8	two buffers	7 (3.2)	5.4 (3)	*2.5 (0.7)*
256	3	one buffer	29 (9.1)	15.6 (8.2)	*11 (3.4)*
256	3	two buffers	3.5 (2)	3.2 (1.8)	*2.5 (1.1)*
256	8	one buffer	29 (5.3)	8.2 (5.2)	*3.7 (1.1)*
256	8	two buffers	3.5 (1.7)	3.1 (1.6)	*1.2 (0.4)*

Table 39.1: Performance on an ATI Radeon 9700; 512x512 viewport size; 256x128x256 data set; three and eight enabled objects, respectively. Numbers are in frames per second. Compositing is done with either one or two buffers, respectively. The *multi* column with early z-testing turned off is only shown for comparison purposes.

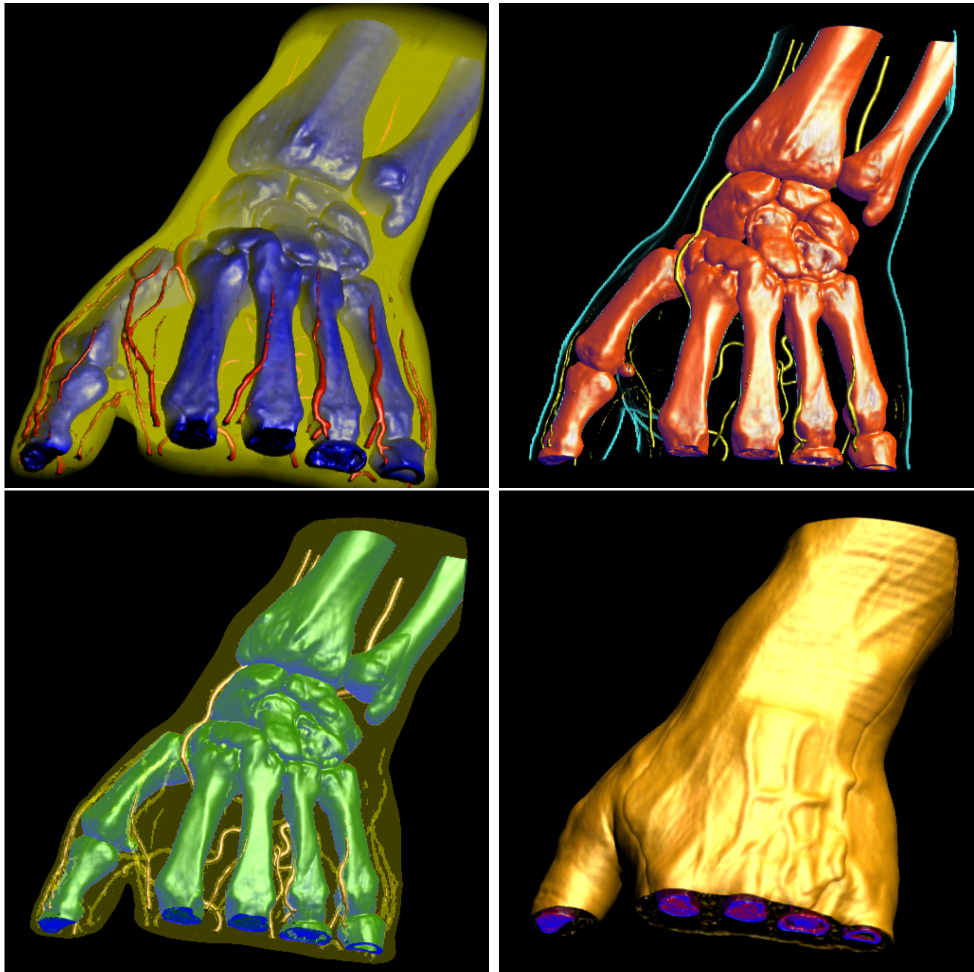


Figure 39.1: Hand data set (256x128x256) examples of different rendering and compositing modes. (top, left) skin with unshaded DVR, vessels and bones with shaded DVR; (top, right) skin with contour rendering, vessels with shaded DVR, bones with tone shading; (bottom, left) skin with MIP, vessels with shaded DVR, bones with tone shading; (bottom, right) skin with isosurfacing, occluded vessels and bones with shaded DVR.

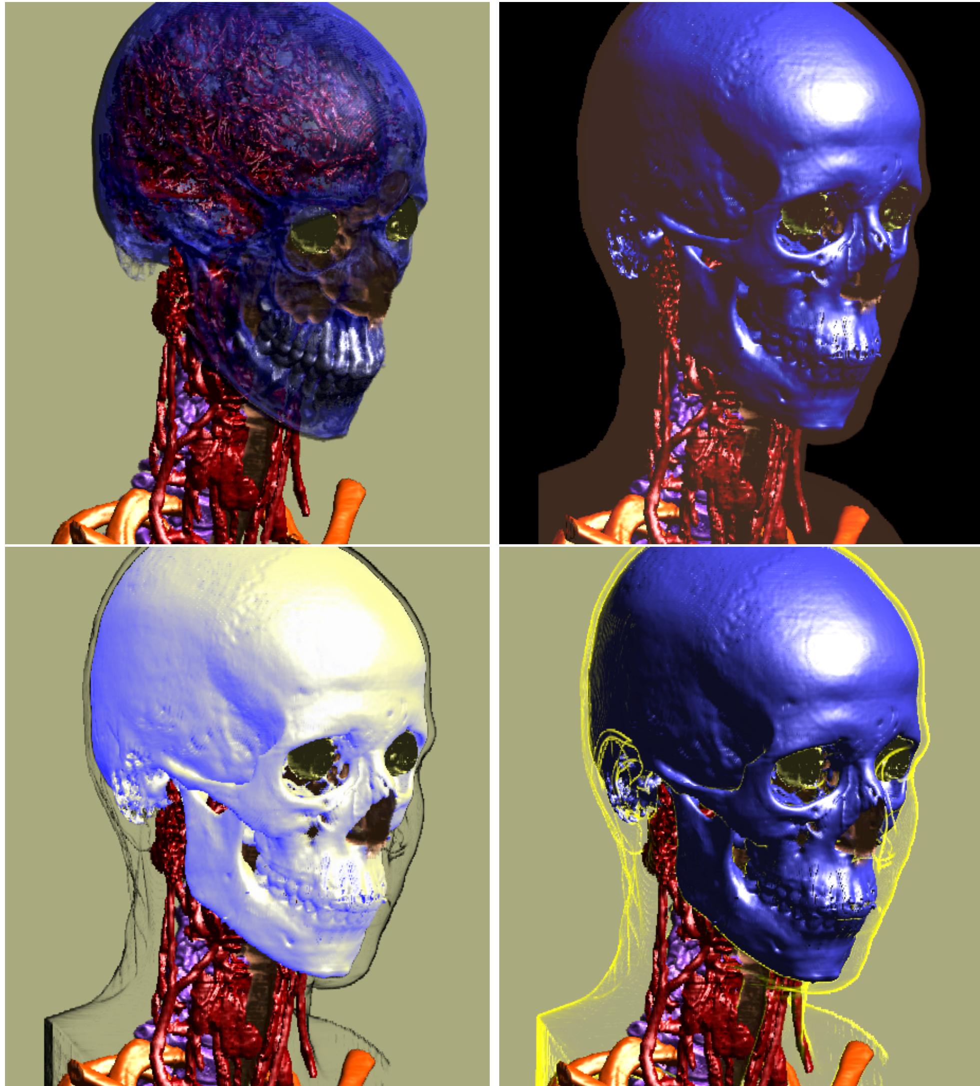


Figure 39.2: Head and neck data set (256x256x333) examples of different rendering and compositing modes. (top, left) skin disabled, skull with shaded DVR; (top, right) skin with MIP, skull with isosurfacing; (bottom, left) skin with contour rendering, skull with tone shading; (bottom, right) skin with contour rendering, skull with isosurfacing.

Acknowledgments

The signed distance fields of well-known computer graphics models shown in chapter IX have been created by Christian Sigg. The medical data sets shown in chapters IX and X are courtesy of Tiani Medgraph. Chapters IV, IX, and X have been prepared in the project *Effective Medical Visualization* at the VRVis Research Center, which is funded in part by the Kplus program of the Austrian goverment.

Course Notes 28

Real-Time Volume Graphics

Volume Deformation and Animation

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Introduction

In scientific visualization volumetric objects often need to be deformed to account for non-linear distortions of the underlying object. A prominent example is computer assisted surgery, where tomography data from therapy planning must be deformed to match non-rigid patient motion during the intervention.

With the evolution of hardware-accelerated volume rendering techniques, volumetric objects are also becoming more and more important as supplement to traditional surface models in computer animation.

Before we examine volumetric deformation in detail, let us reconsider how deformation is performed with surface descriptions.

41.1 Modeling Paradigms

In traditional modeling the *shape* of an object is described by means of an *explicit* surface while its *appearance* is defined by material properties and texture maps, that form a complex shading model. If we are animating such objects we usually want to deform the shape, but not the appearance. In consequence, we are displacing vertices (or control points), while maintaining the original texture coordinate binding. As an example, if we displace one vertex of a triangle (modifying its position without changing the texture coordinate), the assigned texture map will stretch to fit onto the modified area of the triangle.

It is important to notice, that *texture coordinates* are interpolated in **barycentric** coordinates within triangles, while *texture samples* are obtained with **bilinear** interpolation from a 2D texture map (or trilinear interpolation in case of 3D textures). This is the reason that a rectangular texture image does not map evenly onto a deformed quadrangle (see Figure 41.1). For polygonal surfaces, this is an imperfection that we can either simply neglect or alleviate by adjusting the texture coordinates for different animation poses.

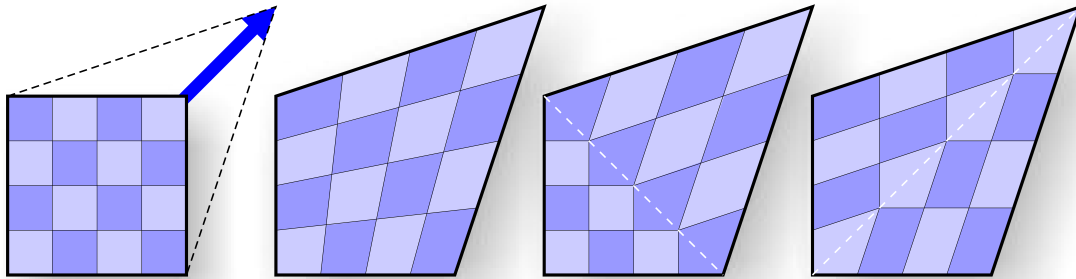


Figure 41.1: *Far left:* A texture mapped quadrangle is deformed by displacing one vertex. *Left:* Bilinear interpolation of texture coordinates within the quadrangle would result in an evenly mapped texture. *Right and far right:* Splitting the quadrangle into two triangles with barycentric interpolation results in different distortions depending on the actual tessellation *(white dotted line)*.

41.2 Volumetric Deformation

Apart of the large variety of deformable surface models [90, 14, 72, 12], only few approaches exist on volume deformation [63, 108, 23]. If we want to adapt surface deformation techniques to volumetric objects, we first notice that the strict separation of *shape* from *appearance* does not fit in properly. The drawn proxy geometry is usually not related to the shape of the object contained in the volume data. Both shape and appearance of the object are defined by the 3D texture map in combination with an appropriate transfer function. The shape of the volumetric object can then be thought of as an implicit surface or isosurface.

As a result there are two ways of deforming volumetric objects in general: Modifying either the proxy geometry in model coordinates (the *shape* in traditional modeling) or distorting the mapping of the 3D texture in texture space (the *appearance*). Both methods will result in a deformation of the shape of the volumetric object. We will examine them in the following sections.

Deformation in Model Space

As we have seen in the previous chapters, texture based approaches decompose the volume data set into a proxy geometry by slicing the bounding box into a stack of planar polygons. Unfortunately, applying a deformation by simply displacing the vertices of the volume bounding box before the slice decomposition does not lead to consistent visual results after the slice decomposition (not even if we ensure that the faces remain planar). This is due to the same interpolation problems as outlined in Figure 41.1.

In order to achieve a consistent mapping of a 3D texture image to a deformed hexahedron, the hexahedron must be tessellated into several tetrahedra before computing the proxy geometry. Subdivision into tetra-

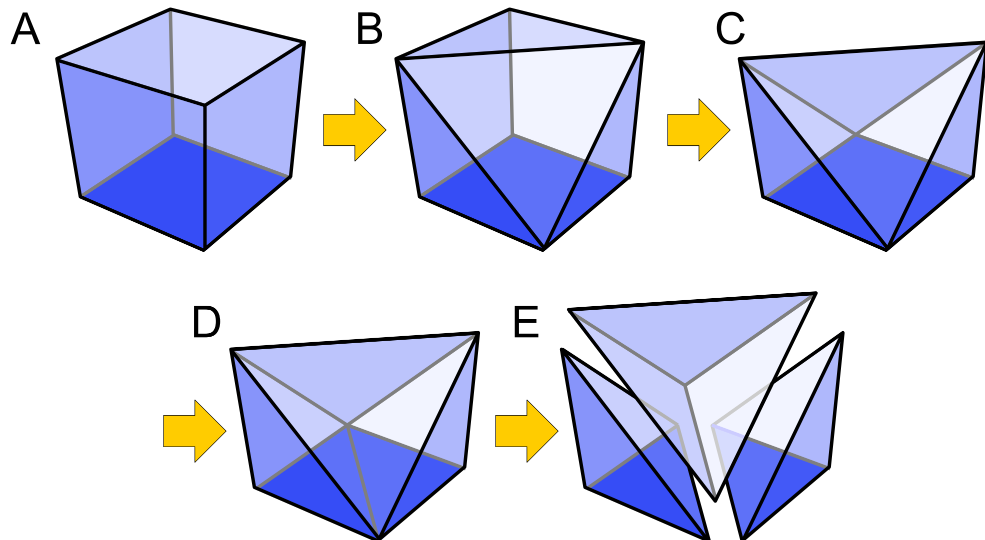


Figure 42.1: A hexahedron (A) is split into 5 tetrahedra. The first two tetrahedra are created by cutting away the foremost (B) and the rearmost vertex (C) of the top face. The remaining trunk (D) is divided into three tetrahedra (E) by splitting the bottom face along its diagonal.

hedra also ensures consistent results for slice intersection if the faces of the hexahedron become non-planar due to the deformation.

The easiest way of subdividing a hexahedron is to split it into 5 tetrahedra as outlined in Figure 42.1. The first tetrahedron is created by removing the foremost vertex of the top face and the three edges connected to it (B). The second tetrahedron is calculated the same way for the rearmost vertex (C) on the top face. The remaining simplex (D) can then be divided similarly into three tetrahedra by cutting away the leftmost and the rightmost vertex of the bottom face(E).

The deformation of a single tetrahedron can be described as a simple affine transformation

$$\Phi(\vec{x}) = \mathbf{A}\vec{x} \tag{42.1}$$

in homogenous coordinates. The deformation matrix $A \in \mathbb{R}^{4\times 4}$ is fully determined by specifying four translation vectors at the tetrahedron's vertices. The deformation of the entire volumetric object is then composed from piecewise linear transformation. The deformed tetrahedra are finally decomposed into an view-aligned slices and rendered back-to-front via alpha blending. Approaches for performing the slicing of tetrahedra cells completely in a vertex shader do exist, but have not yet been published by the respective research groups, unfortunately.

42.1 Depth Sorting

Back-to-front compositing of tetrahedral data usually requires depth-sorting to obtain the correct visibility order of the cells. Cell sorting of tetrahedra data in general is not a trivial task, especially not for non-convex data or meshes that contain visibility cycle [58] for certain viewpoints. Possible solutions can be found in [75, 13, 45, 111] Most of the complex sorting algorithms can be circumvented if the tetrahedra cells are generated by splitting hexahedra as outlined above. In this case only the hexahedra must be depth sorted using the distance from the eye point. For each hexahedron, the respective tetrahedra are finally sorted separately. This however only works properly if the common faces of adjacent hexahedra are kept planar and if the cells do not intersect each other (a condition that is required by most sorting algorithms).

Deformation in Texture Space

The other alternative for volumetric deformation is keeping the vertices of the geometry static and modifying the texture coordinates only. Since the shape of the object is defined as an implicit surface in the 3D texture, distorting the texture space results in a deformation of the object.

To achieve higher flexibility for the deformation, we first subdivide the original cuboid into a fixed set of sub-cubes by inserting additional vertices (Figure 43.1*left*). A deformation is specified in this refined model by displacing only the texture coordinates for each vertex. The displacement of the texture coordinate $\vec{u}$ for a point $\vec{x}$ in the interior of a patch is determined by trilinear interpolation of the translation vectors $\vec{t}_{ijk}$ given at the vertices. The result is a trilinear mapping

$$\Phi(\vec{u}) = \vec{u} + \sum_{i,j,k\in\{0,1\}} a_{ijk}(\vec{x}) \cdot \vec{t}_{ijk}, \tag{43.1}$$

with the interpolation weights $a_{ijk}(\vec{x})$ determined by the position $\vec{x}$ in (undeformed) model space.

If we now setup the proxy geometry, we want to preserve the benefit of our model being based on a static geometry, because the intersection calculation for all the small sub-cubes contributes a considerable computational load. We use object-aligned slices (see Figure 43.1 *right*), which keeps us from having to recompute all the cross-sections for each frame. Object-aligned slices can also be easily computed in a simple vertex shader.

Again, the straight-forward approach of slicing each sub-cube and assign texture coordinates at the resulting polygon vertices will not lead to a correct trilinear mapping as specified in Equation 43.1. There are consistency problems similar to the ones described in Section 41.1. In Figure 43.2 the texture coordinate of the upper right vertex of the quad is displaced. The correct trilinear mapping *(far left)* is poorly approximated by the internal triangulation of the graphics API *(middle left and right)*. As a solution to this problem, inserting one additional vertex in the middle of the polygon usually results in a sufficiently close approximation

to the original trilinear deformation with respect to screen resolution. If higher accuracy is required, more additional vertices can be inserted. Such a manual tessellation also provides a consistent triangulation of the *non-planar* texture map, which is result of an arbitrary deformation of 3D texture space.

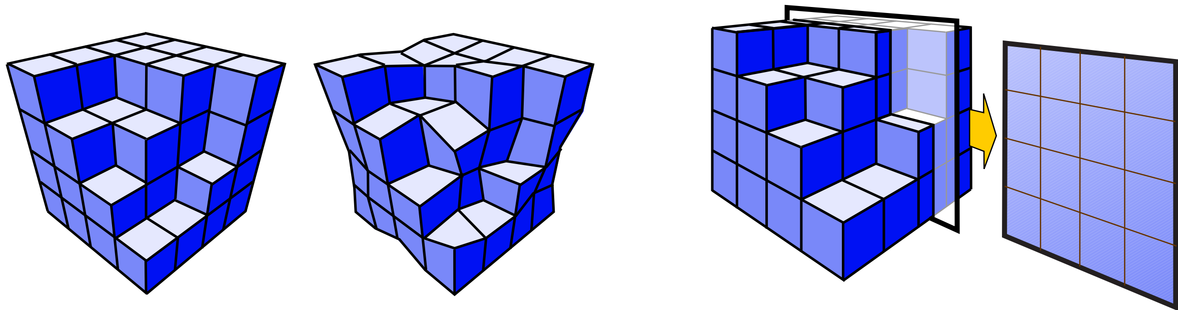


Figure 43.1: Model Space *(left)* : The volume is subdivided into a fixed set of sub-cubes. The geometry remains undeformed. Texture Space (*middle*): The deformation is modelled by displacing *texture coordinates*. *Right:* Such a deformation model allows the extraction of object aligned slices at low computational cost.

43.1 Practical Aspects

In an intuitive modelling application, the artist most likely does not want to specify texture coordinate deformation manually. Instead, the user should be provided with a mechanism which allows him to pick and drag a vertex to an arbitrary position. Such a manipulation, however, requires the inverse transformation Φ^{-1} of our trilinear mapping. The caveat here is that the inverse of a trilinear mapping in general is not again a trilinear mapping, but a function of higher complexity.

For the purpose of modeling, however, the exact inverse transformation is not necessarily required. In the usual case an intuitive modeling mechanism similar to placing control points of a NURBS patch should suffice. An approximate inverse Φ^{-1} that allows intuitive dragging of vertices can be calculated simply by negating the original translation vectors at the vertices,

$$\widetilde{\Phi}^{-1}(\vec{u}) = \vec{u} + \sum_{i,j,k \in \{0,1\}} a_{ijk}(\vec{x}) \cdot (-\vec{t}_{ijk}), \tag{43.2}$$

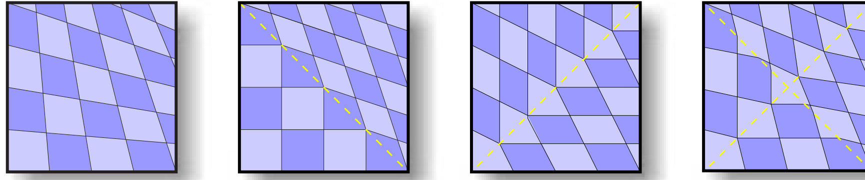


Figure 43.2: The trilinear deformation in texture space *(far left:* is poorly approximated if of the graphics API internally tessellates the textured quad into two triangles with barycentric interpolation *(middle left and right)*. Inserting an additional vertex *(far right)* usually approximates the trilinear interpolation sufficiently close.

It is easy to verify, that the approximation error for a maximum displacement vector of magnitude γ amounts to

$$\widetilde{\Phi}^{-1}(\Phi(\vec{u})) = \vec{u} + o(\gamma^2), \tag{43.3}$$

which turns out to be accurate enough for modeling purposes.

43.2 Non-uniform Subdivision

Using such a model as a basis, it is easy to increase flexibility adaptively by further subdividing single patches as required. This results in a hierarchical octree-structure as illustrated in Figure 43.3 *(left)*. In order to maintain a consistent texture map at the boundary between patches with different subdivision level, additional constraints are required. Such constraints must be setup for all vertices which are located on edges or faces, that are shared by patches of different levels. Without these constraints undesired gaps and discontinuities would emerge in texture space. In 3D, we must further differentiate between face and edge constraints.

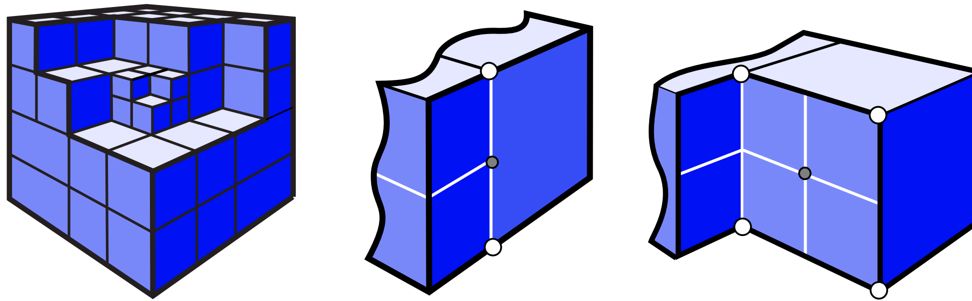


Figure 43.3: *Left:* Non-uniform subdivision is used to increase flexibility for the deformation. Edge (*middle:*) constraints and face constraints (right) are necessary to prevent gaps in texture space.

43.2.1 Edge Constraints

At common edges between patches with different subdivision levels, a simple constraint is necessary to ensure that the two half-edges of the higher level stay collinear. The inner vertex[1] in Figure 43.3*middle*, which was inserted by the higher subdivision level must stay at its fixed position relative to the two neighboring vertices. Note that this vertex is not even allowed to move in direction along the edge, as this would also result in discontinuities in texture space.

$$\vec{V_C} = (1 - \alpha)\vec{V_0} + \alpha \cdot \vec{V_1} \tag{43.4}$$

[1] Note that the constraints are setup in texture space only. If we are referring to vertices we actually mean texture coordinates (= vertices in texture space).

43.2.2 Face Constraints

At faces, which are shared by different subdivision levels, another type of constraint is required to ensure coplanarity. The middle vertex in Figure 43.3*(right)*, must stay at a fixed position relative to the four vertices, which formed the original face.

$$\vec{V_C} = \sum_{i=0...3} a_i \vec{V_i} \qquad \text{with} \qquad \sum_{i=0...3} a_i = 1; \tag{43.5}$$

To circumvent recursive constraints, we additionally follow a general rule, known from surface modeling, that says that two neighboring patches must not differ by more than one subdivision level. This means that any patch can only be further subdivided if all neighboring patches have at least the same subdivision level.

43.3 Deformation via Fragment Shaders

The texture-space deformation model can be efficiently implemented using *dependent textures* or *offset textures*. The basic idea of an offset texture is to use the RGB triplet obtained from one texture as texture coordinate offset for a second texture. On a graphics board with support for 3D dependent textures, the computation of the texture-space deformation model can be performed completely within the graphics hardware. Have a look at the code for the fragment shader:

```
// Cg fragment shader for
// texture-space volume deformation

  half4 main (float3 texcoords :  TEXCOORD0,
              uniform sampler3D offsetTexture,
              uniform sampler3D volumeTexture) : COLOR0
  {
     float3 offset = tex3D(offsetTexture, uvw);

     uvw = uvw + offset;

     return tex3D(volumeTexture, uvw);
  }
```

The idea here is to store the deformation vectors in the RGB channels of a 3D offset texture and to use the dependent texture lookup to obtain

the deformed volumetric information from a second 3D texture map, which stores the undeformed volume. Note that there is no need for the first texture to have equal size as the original volume, so it should be possible to keep it small enough to allow an interactive update of the deformation field. This technique also allows the rendering of view-aligned slices instead of object aligned slices, since a uniform trilinear mapping of voxels to transformation vectors is guaranteed by the first 3D texture.

Such a fragment shader can also be easily modified to handle linear keyframe interpolation. The following fragment shader takes two offset textures as input and interpolates the offset vectors using the `lerp` command.

```
// Cg fragment shader for texture-space volume
// deformation with blending of two keyframes

  half4 main (float3 texcoords :  TEXCOORD0,
              uniform sampler3D offsetTexture1,
              uniform sampler3D offsetTexture2,
              uniform sampler3D volumeTexture,
              uniform float time) : COLOR0
  {
     float3 offset1 = tex3D(offsetTexture1, uvw);

     float3 offset2 = tex3D(offsetTexture2, uvw);

     uvw = uvw + lerp(offset1, offset2, time);

     return tex3D(volumeTexture, uvw);
  }
```

Local Illumination

Local illumination techniques as described in Part 4 cannot directly be used with the described deformation models. Due to the deformation pre-calculated gradient vectors become invalid. In this section we want to examine possibilities to adapt pre-calculated vectors to the applied deformation.

For the deformation in model space described in Chapter 42 such an adaptation is easy, because we know the exact affine deformation matrix for each tetrahedron. We also know that if an object is transformed with a linear matrix $\mathbf{M}$, its normal vectors must be transformed with the transposed inverse matrix $(\mathbf{M}^{-1})^T$. All we have to do is multiply the precalculated normal vectors with the transposed inverse of matrix $\mathbf{A}$ from Equation 42.1, which is constant for each tetrahedron. The local illumination term can then be computed as usual.

The texture-space deformation model, however, is based on a trilinear mapping (Equation 43.1), whose inverse is a rather complex function. Calculating the exact deformation of the normal vectors becomes expensive. One working alternative is to use on-the-fly gradient estimation as described in the Illumination Part.

Another alternative is to approximate the inverse of the trilinear function using a linear transformation. The idea is to find an affine mapping, which approximates the original trilinear mapping $\Phi(\vec{x})$ and then use the transposed inverse matrix to transform the pre-computed Normal vectors. The affine transformation is simply a 4×4-matrix in homogenous coordinates, denoted

$$\overline{\Phi}(\vec{x}) = \overline{\mathbf{A}}\vec{x}, \qquad \text{with} \quad \overline{\mathbf{A}} = \left(\begin{array}{c|c} \mathbf{A} & \vec{b} \\ \hline 0\;0\;0 & 1 \end{array}\right) \in \mathbb{R}^{4\times 4}. \tag{44.1}$$

The optimal approximation $\overline{\Phi}$ is determined by minimizing the quadratic difference between the transformation of the eight static corner vertices $\overline{\Phi}(\vec{x}_i)$ and their real transformed positions $\vec{y}_i = \Phi(\vec{x}_i)$, according to

$$\frac{\delta}{\delta \mathbf{A}} \sum_{i=1}^{8} \|\overline{\Phi}(\vec{x}_i) - \vec{y}_i\|^2 = 0, \tag{44.2}$$

which leads to

$$\sum_{i=1}^{8} (\vec{x}_i \vec{x}_i^T \mathbf{A}^T - \vec{x}_i \vec{y}_i^T) = 0. \qquad (44.3)$$

Solving this equation for $\mathbf{A}^T$, results in

$$\mathbf{A}^T = \mathbf{M}^{-1} \sum_{i=1}^{8} \vec{x}_i \vec{y}_i^T, \qquad \text{with} \quad \mathbf{M} = \sum_{i=1}^{8} \vec{x}_i \vec{x}_i^T \in \mathbb{R}^{4\times 4}. \qquad (44.4)$$

It is easy to verify that the inverse of matrix M always exists. One important fact is that matrix $\mathbf{M}$ is constant for each patch, because the undeformed corner vertices $\vec{x}_i$ are static in this model. Matrix $\mathbf{M}$ can thus be pre-computed for efficiency. Taking also into consideration that the corner vertices are located on an axis-aligned grid, the computation can be further simplified, such that calculating each entry a_{ij} of the affine matrix $\mathbf{A}$ will require only eight multiplications.

The performance benefit of this approximation should become clear, if we consider the dot products that are involved in the diffuse term of the Phong illumination model

$$I_{\text{diff}} = I_L \cdot (\vec{n} \bullet \vec{l}). \qquad (44.5)$$

In this context, $\vec{n}$ is the surface normal, which coincides with the voxel gradient in our model. I_L denotes the color of the light source, weighted by a material dependent diffuse reflection coefficient. As we have seen

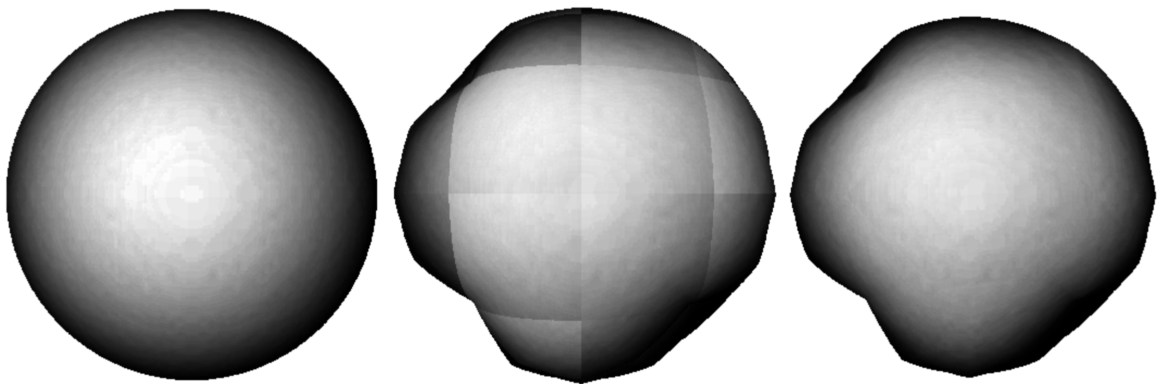


Figure 44.1: Diffuse illumination of an undeformed sphere (left). Extremely deformed sphere with discontinuities at the patch boundaries (center). Correct illumination by smoothing the deformed light vectors (right) at the vertices.

in the Illumination Part, the per-pixel dot product computation can be efficiently performed in hardware using fragment shaders.

For the undeformed volume the gradient vectors are pre-calculated and stored within a 3D normal map. In order to achieve realistic illumination results for deformable volumetric data as focused here, we have to adapt the gradient vectors to the actual deformation. According to our linear approximation, the new diffuse term after the transformation is determined by

$$\widetilde{I}_{\text{diff}} = I_L \cdot (\ ((\mathbf{A}^{-1})^T \vec{n}) \bullet \vec{l}). \tag{44.6}$$

Note that since the gradients $\vec{n}$ are obtained from a texture, this calculation requires a per-pixel matrix multiplication, which can easily be computed using fragment shaders. If we further assuming directional light, the light vector $\vec{l}$ is constant for the whole scene and there is an easy way for illumination, which circumvents these per-pixel matrix multiplication. Consider that the dot product in Equation 44.6 can also be written as

$$((\mathbf{A}^{-1})^T\ \vec{n}) \bullet \vec{l} = \vec{n} \bullet (\mathbf{A}^{-1}\vec{l}). \tag{44.7}$$

In relation to our method, this means that all the pre-computed normal vectors can be left untouched. The only thing we have to do is to calculate a new light vector to obtain an equivalent visual result.

Regardless of whether the normal deformation is exact or approximative, using a light vector constant within each patch, but different

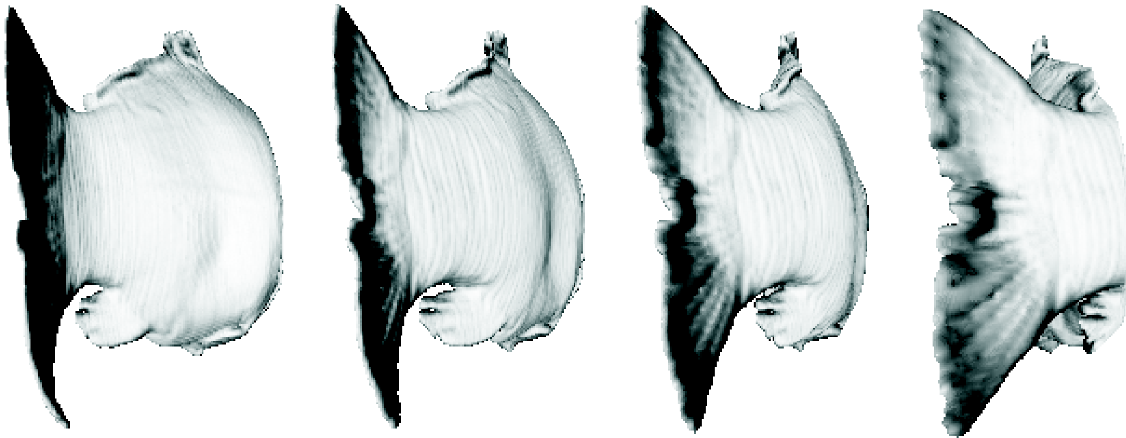


Figure 44.2: Animated tail fin of a carp demonstrates realistic illumination effects during real-time deformation.

for neighboring patches, will inevitably result in visible discontinuities as depicted in Figure 44.1 (center). To tackle this problem, there should be smooth transitions for the diffuse illumination term of neighboring patches. This can be easily achieved by assigning light vectors to the vertices instead of the patches. To each vertex a light vector is assigned, which is averaged from the light vectors of all the patches, which share this vertex. Analogously to the translation vectors, the light vectors given at the vertices are tri-linearly interpolated within each patch. To achieve this during rasterization, the light vectors must be assigned as color values to the vertices of each rendered polygon, thus allowing the interpolation to be performed by hardware Gouraud shading. As displayed in Figure 44.2, this method will lead to approximate illumination without any discontinuities.

Volume Animation

The two different approaches for volumetric deformation in model space (Chapter 42) and texture space (Chapter 43) can be utilized for volumetric animation in the same way as surface deformation approaches are used. The model space approach is well suited for large-scale motions usually modelled by skeleton animation. After subdivision of the hexahedra into tetrahedra for deformation, the free vertices can be attached to existing skeletons using smooth or rigid skin binding.

The texture based deformation approach is well suited for deformations which are small with respect to the size of the volume object itself. As described in Chapter 43, the deformation in texture space is modelled by shifting texture coordinates. The deformation can be modelled intuitively by approximating the inverse transformation for each hexahedron by negating the shift vectors. For animating these shift vectors, they can either be bound to small-scale skeleton rigs or be weighted and controlled independently, similar to blend shapes commonly used for facial animation. Example animations can be found on the course web-site.

Course Notes 28
Real-Time Volume Graphics

Dealing with Large Volumes

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

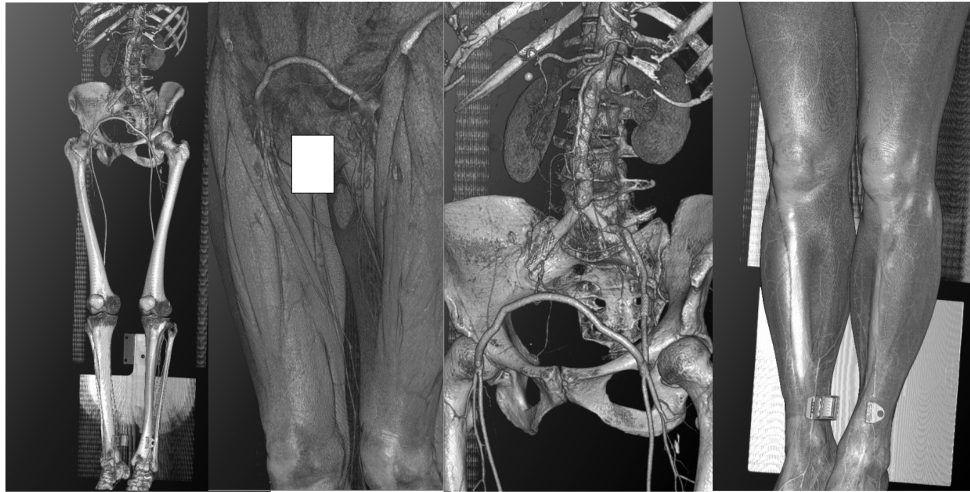


Figure 45.1: Long-leg study of a bypass operation (512x512x1800@16bit).

In recent years there has been a explosive increase in size of volumetric data from many application areas. For example, in medical visualization we now routinely acquire long-leg studies with around 2000 slices (see figure 45.1). The resolution of each slice is 512 by 512 pixels with 16 bit precision, resulting in almost a gigabyte of data. Rendering such a data set at interactive rates is a challenging task, especially considering that the typical amount of memory available on GPUs is currently 256 megabytes. 4D sequences from cardiology, which allow visualization of a beating heart, quickly approach or even surpass the virtual address limit of current PC hardware. Other application areas like geology even produce data sets exceeding a terabyte. In entertainment applications, a high level of detail is often required when adding volumetric effects to computer games or movies. To add those effects, procedural techniques often circumvent high resolution volumes as we will show in section 49.

Analysis of the current architecture of PCs and consumer graphics hardware reveals the major bottlenecks affecting the visualization of large volumetric data sets. Obviously, as noted already in the previous paragraph, the amount of memory available on today's GPUs is very limited. Therefore, other types of memory have to be employed for large volumes. The bandwidth of these different types of memory play a key role in volume rendering.

Consider the different types of memory involved: Volume data is first loaded into main memory. Current 32 bit CPUs support up to 4 gi-

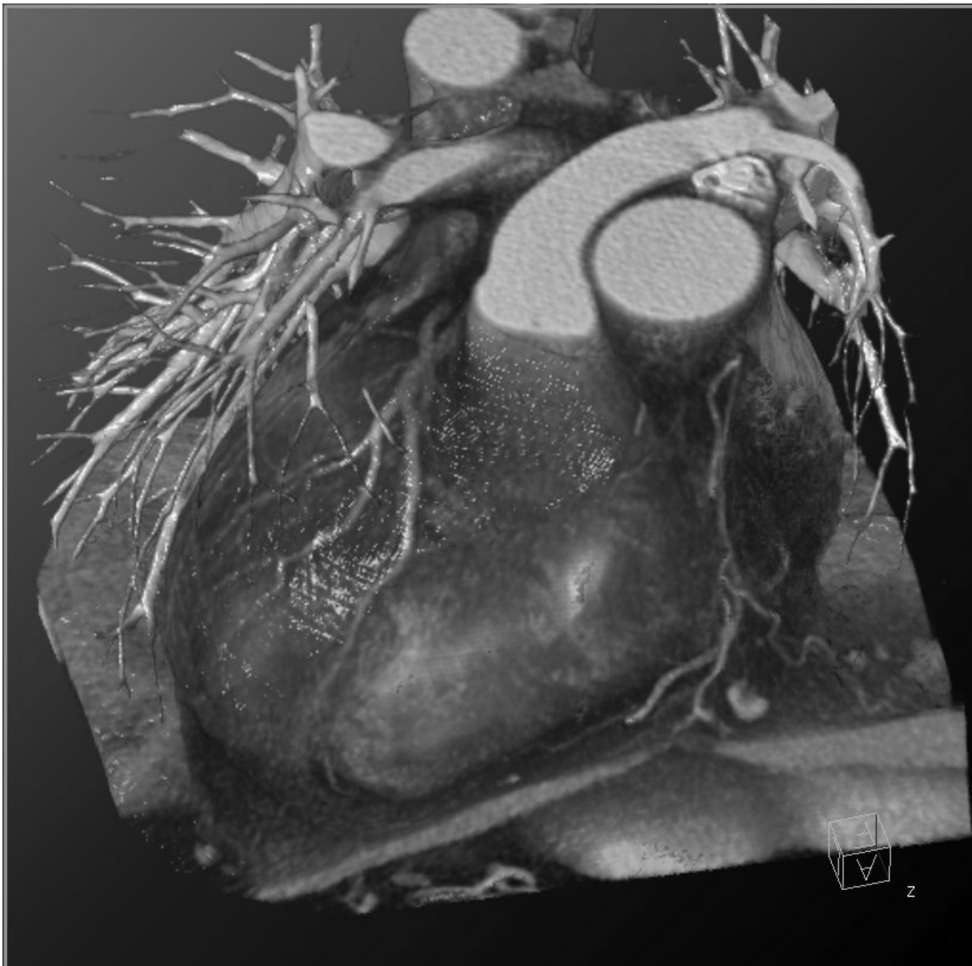


Figure 45.2: Single frame from a 4D sequence of a beating heart (512x512x240@16bit, 20 time steps).

gabytes of memory. Segments of system memory can be dynamically reserved by the OS for the graphics controller. This memory is termed AGP (accelerated graphics port) memory or non-local video memory. Before texture data can be transferred to the video memory of the GPU, the data has to be copied to AGP memory. The current peak transfer bandwidth of main memory is less than 5 gigabytes per second. Dual channel DDR400 memory provides around 6.4 gigabytes per second. From AGP memory, volume data can is transferred to video memory using AGP. AGP3.0 delivers a maximum of 2.1 GB/s bandwidthl however, sustained throughput in many applications is around 1 GB/sec, far away from the theoretical limit. Furthermore, it should also be noted that the read back performance of AGP is much smaller. Some of these restrictions will disappear or at least will be reduced with upcoming PCI Express technology.

Local video memory provides very high memory bandwidth with more than 30 GB/sec using a 256 bit wide memory interface. Data is transferred to an internal texture cache on the GPU chip with this bandwidth. Unfortunately, that GPU manufacturers do not provide any details about the amount and bandwidth of the internal texture cache, however one can assume that the memory bandwidth of the texture cache is several times higher than that of the video memory.

Very similar to the situation with many levels of cache on CPUs, we can interpret all those different types of memory at different levels of texture cache. The local texture cache on the GPU chip can be considered be as level 1 cache, local video memory as level 2 cache and AGP memory as level 3 cache. It is desirable to keep texture data as close to the chip as possible, i.e., in level 1 or level 2 of the texture caches.

We will now present different techniques that try to utilize the available amount of memory to render large volumetric data sets. Those techniques differ in how efficiently they utilize the levels of texture cache.

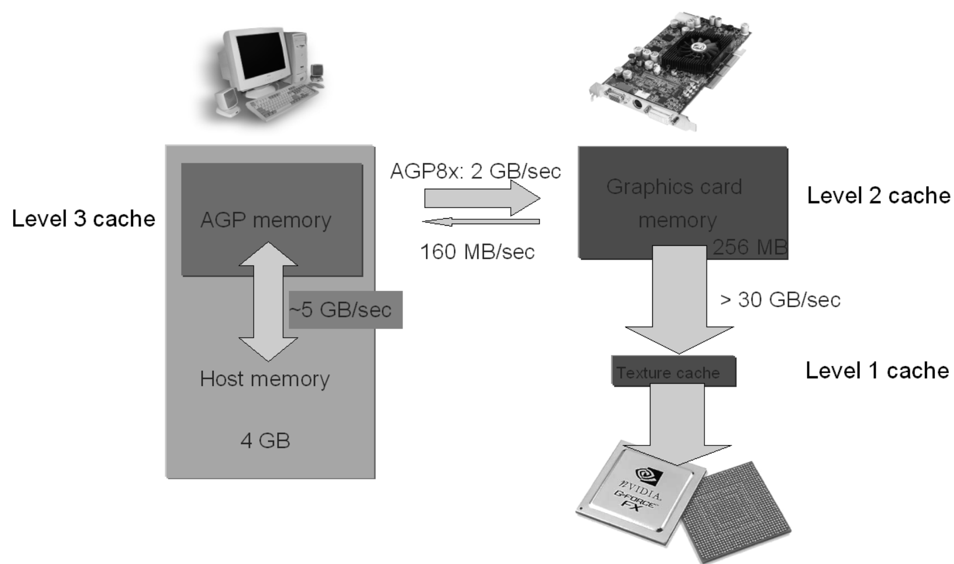


Figure 45.3: Typical data transfer bandwidths of a PC from main memory to the GPU. Different types of memory can be considered as different levels of cache.

Bricking

The most straightforward method to deal with a large volume is the divide and conquer method, which is called bricking in the context of volume rendering. The volume is subdivided into several blocks in such a way that a single sub-block (brick) fits into video memory (see figure 46.1). Bricks are stored in main memory and sorted in a front-to-back or back-to-front manner dependent on the rendering order.

Bricks are loaded into the local memory of the GPU board one at a time. Each brick is rendered using standard volume rendering techniques, i.e. by slicing or ray-casting. Note, that this technique can also be used for rendering large volumes on multiple GPU boards. In the case of multiple GPUs, an additional composting step is required to assemble the resulting images generated by the GPUs.

In order to avoid discontinuities on brick boundaries when using trilinear

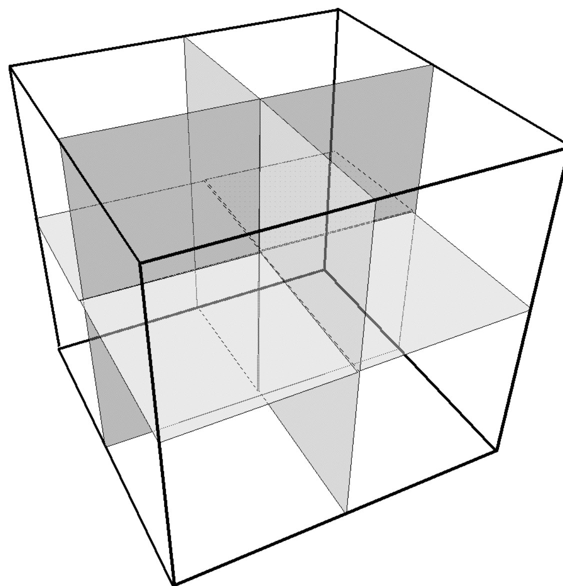


Figure 46.1: Subdivision of a large volume into several smaller bricks.

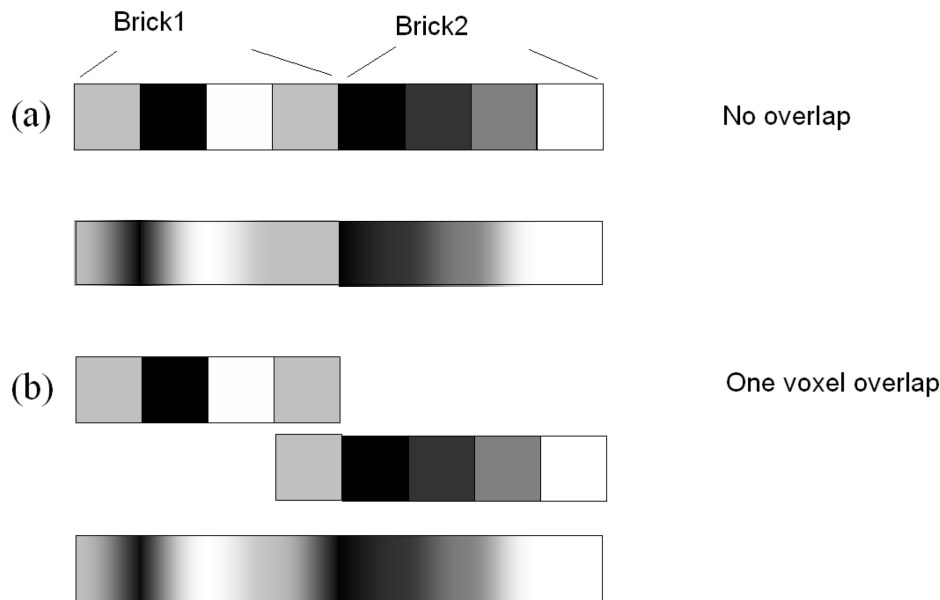


Figure 46.2: Brick boundaries without (a) and with (b) overlap.

filtering, bricks must overlap by at least on voxel size. Figure 46.2)(a) demonstrates the result of rendering two bricks with linear filtering and no overlap. By repeating one voxel of brick 1 at the brick boundary in brick 2 as shown in figure 46.2(b) we can ensure a smooth transition between bricks.
Note, that if we look-up neighboring voxels in a fragment shader it is required to increase the overlap by the same factor as the distance to the original voxel position from the neighbor voxel. Looking up neighboring voxel is done when applying high-quality filtering as presented in chapter 21 or computing gradients on-the-fly as presented in chapter 23.
Bricking does not reduce the amount of memory required to represent the original volume data. Each brick has to be transferred to the local memory on the GPU board before it can be rendered. Thus, the performance of bricking is mainly limited by the AGP transfer rate. In order to circumvent this problem, a very common technique is to use a sub-sampled version of the volume data that is entirely stored in the GPU memory during interaction and only render the full resolution volume using bricking for the final image quality. The following techniques try to prevent transfer over AGP by making better use of the available high-throughput texture caches closer to the GPU.

Multi-Resolution Volume Rendering

A subdivision of the volume that adapts to the local properties of the scalar field or some user defined criteria has many advantages to a static subdivision as previously presented. This idea was first presented by LaMar et al. [65]. They render a volume in a region-of-interest at a high resolution and away from that region with progressively lower resolution. Their algorithm is based on an octree hierarchy (see figure 47.1) where the leaves of the tree represent the original data and the internal nodes define lower-resolution versions. An octree representation of volumetric data can be obtained by either a top-down subdivision or a bottom-up merging strategy. The top-down subdivision strategy divides the volume data into 8 blocks of equal size. Each block is further subdivided recursively into smaller blocks until the block size reaches a minimum size or the voxel dimension. The bottom-up strategy merges eight neighboring voxels (or atom-blocks) into a larger block. Each block is again merged with its neighboring blocks into a larger block until the complete volume remains as a block. Each block is a down-sampled version of the volume data represented by it's child nodes. Given such a octree representation of the volume data, one can traverse the tree in a top-down manner, starting from the coarsest version of the data at the root node. At each node one can decide whether the child nodes of the specific node need to be traversed further based on some criteria. Possible criteria are the distance of the node to the viewer position, detail contained within the sub-tree, or the desired rendering resolution based on a global or local parameter such as a focus point. The sampling rate can be adapted in the levels of the hierarchy to the detail level.

The multi-representation allows memory to be saved for empty or uniform portions of the volume data by omitting sub-trees of the hierarchy. Furthermore, rendering performance may increase due to lower sampling rates for certain blocks or omitting of empty blocks.

Weiler et al. [103] further improved this algorithm by fixing artifacts that

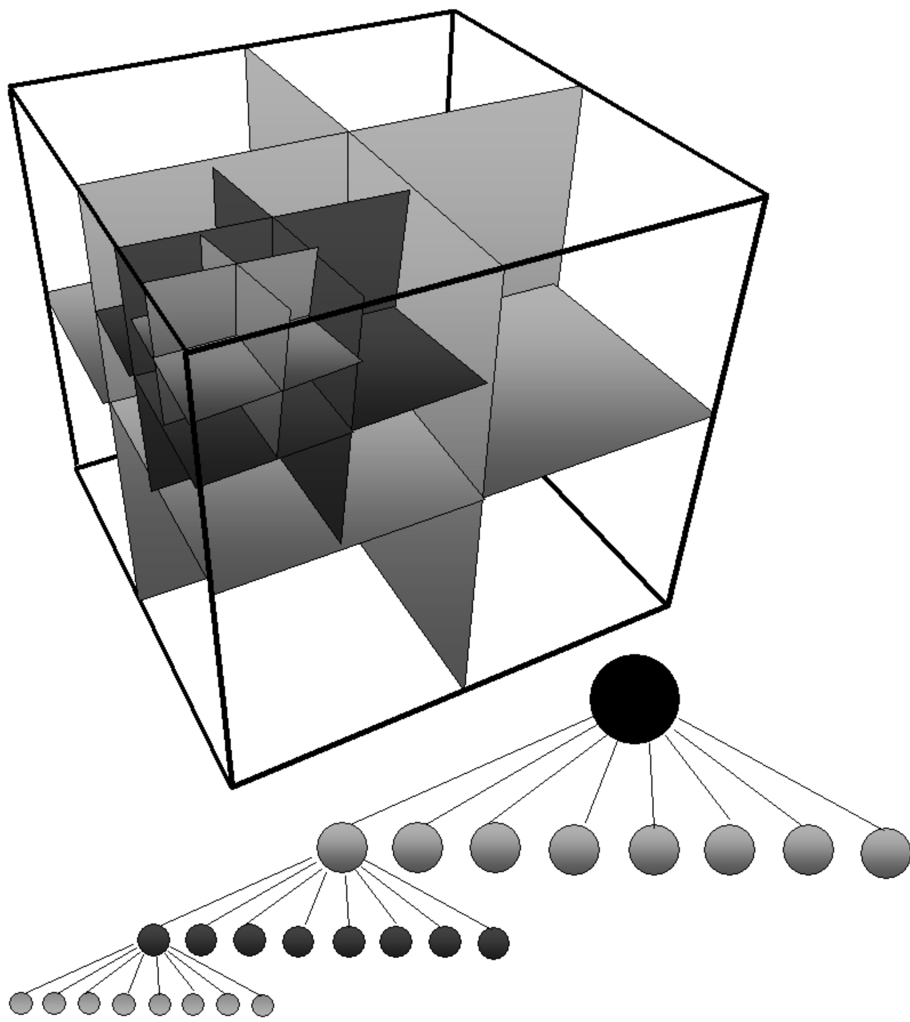


Figure 47.1: Octree decomposition of a volume.

typically occur on boundaries of different levels of the multi-resolution representation.

Compression Techniques

The multi-resolution techniques introduced in the last chapter already introduce compression of volume data if the octree is not refined to the maximum level in all branches. In this chapter we will examine compression techniques for volumetric data sets that try to utilize the available high-performance memory as efficiently as possible.
In fact, GPUs already have built-in texture compression schemes. In OpenGL, texture compression is available using the S3 texture compression standard which is accessible using the `EXT_texture_compression_s3tc` extension. In this compression method, 4x4 RGBA texels are grouped together. For each 4x4 pixel group two colors are stored, two additional colors are obtained by linear interpolation of the stored colors. For each pixel of the 4x4 block, two bits are used as lookup values to access these four colors. The `NV_texture_compression_vtc` OpenGL extension extends the S3 extension for 2D texture to the 3D texture domain.
S3 texture compression is implemented in hardware by several graphics chips; e.g. the NVIDIA GeForce and ATI Radeon series. It provides a fixed maximum compression scheme of 8:1. However, for compression of volumetric data it has some severe disadvantages. First, block artifacts can easily be observed for non-smooth data due to the block

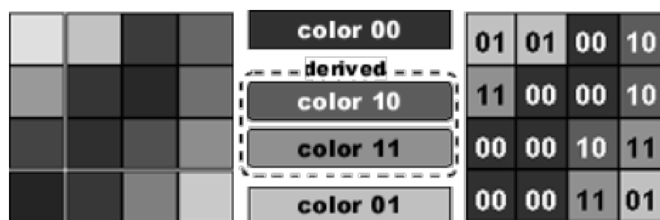


Figure 48.1: S3 texture compression stores two colors for each 4x4 texel block, two additional colors are derived by linear interpolation. The total of four colors are accessed with 2 bits per texel.

compression scheme. Secondly, this compression technique is only available for RGB(A) data. As we are mainly interested in scalar volume data, S3 texture compression can be applied. For compression of pre-computed gradients that are stored in RGB textures, S3 texture compression provides unsatisfactory quality.

48.1 Wavelet Compression

Wavelet transforms[28] provide an invaluable tool in computer graphics. This is due to the fact that, in computer graphics, we often encounter signals that are mostly smooth, but contain important regions with high frequency content. The same applies to volumetric data sets, which (in most cases) contain areas with rich detail while at the same time contain other regions that are very homogeneous. A typical data set from CT (computed tomography) for example, contains very fine detail for bone structures while surrounding air and tissue is given as very smooth and uniform areas. Figure 48.2 shows gradient-magnitude modulation of a CT data set; i.e., areas where the data values change rapidly are enhanced while homogeneous regions are suppressed. Note, that most of the data is smooth while high frequency detail is most apparent at certain material boundaries.

The wavelet transform is the projection of a signal onto a series of basis functions called the wavelets. Wavelets form a hierarchy of signals that allow to analyze and reconstruct an input signal at different resolutions and frequency ranges. This provides a basis for multi-resolution volume rendering. As wavelet transformed signals often contain many coefficient that are nearly zero, wavelets form a natural technique for building a compressed representation of a signal by omitting coefficients that are smaller than a specified threshold. A very efficient compression scheme for volumetric data has been proposed by Nguyen et al.[80]. Their scheme splits the volume into several smaller blocks of equal size that are compressed individually.

The real-time decompression and visualization of 4D volume data was proposed by Guthe et al.[33]. However, each time step of the sequence must be fully decompressed on the CPU before it can be rendered. Thus, no bandwidth is saved when transferring a single time step of the sequence over AGP therefore limiting the rendering performance to the AGP bandwidth.

Guthe et al. circumvented the problem of transferring a fully decompressed full resolution volume over AGP by introducing a multi-

Figure 48.2: Gradient-magnitude modulation volume rendering of a large CT data set.

resolution hierarchy which provides fast access to each node of the hierarchy. This allows storing very large data sets in main memory and extracting the levels of detail on-the-fly that are necessary for an interactive walk-through.
A pre-processing step is required to transform the volume data into the hierarchical wavelet representation. For that purpose, the data is divided into cubic blocks of $(2k)^3$ voxels, where $k = 16$ is a good choice. The wavelet filters are applied to each of the blocks, resulting in a low-pass filtered block of size k^3 voxels and $(2k)^3 - k^3$ wavelet coefficients representing high frequencies that are lost in the low-pass filtered signal. This scheme is applied on hierarchically by grouping eight neighboring low-pass-filtered blocks together to new blocks with $(2k)^3$ voxels. This process is repeated until only a single block remains(see figure 48.3). The results of this process is an octree where each node contains high frequency coefficients to reconstruct the volume at the given level (see figure 48.4).

By defining a threshold to discard small wavelet coefficient, lossy compression of the volume data is possible. For the encoding of the wavelet coefficients, Guthe et al. implemented two compression schemes. For details refer to their original paper.
The data is now given as a multi-resolution tree, with a very coarse representation of the data in the root-node. Each descent in the octree increases the resolution by a factor of two. To decide at which resolution a block should be decompressed by the CPU during rendering, a projective classification and a view-dependent priority schedule can be applied. Projective classification projects the voxel spacing of a node of the hierarchy to screen space. If the voxel spacing is above the screen resolution then the node must be refined, else it is passed to the renderer. The view-dependent classification scheme prioritizes nodes that are nearer to the viewer. The error introduced by rendering a node can also be used to determine if a node needs to be refined. To reduce the amount of data transferred over AGP a caching strategy can be applied that caches blocks that are frequently used in GPU memory.
Guthe et al. achieved compression rates of 40:1 for the visible female and 30:1 for the visible male data sets. Interactive walk-throughs are possible with 3 to 10 frames per second depending on quality settings.
The multi-resolution representation of volumetric in conjunction with the wavelet transform allows for rendering of data sets far beyond the virtual address limit of today's PCs. However, in all presented techniques compressed data is stored in main memory and decompressed by the CPU before it can be rendered on the GPU. An ideal solution

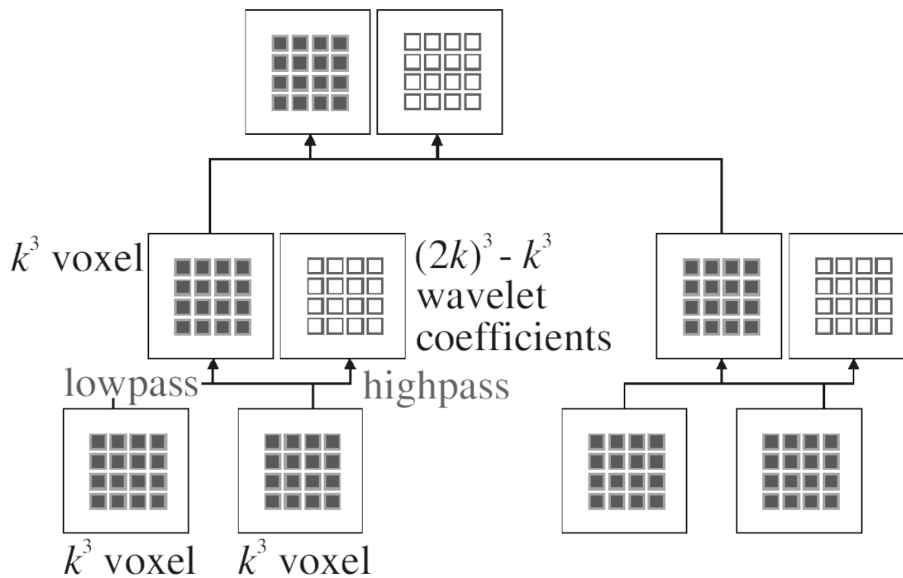


Figure 48.3: Construction of the wavelet tree.

would be to store the compressed data in the local memory of the GPU and decompress it using the GPU before rendering. However, no work has yet been published to do this and it is unclear so far if a decompression of wavelet transformed data using a GPU can be achieved in the near future.
The techniques presented in the following sections can easily be implemented on GPUs by utilizing dependent texture fetch operations. That is, the result of a previous texture fetch operation is used as a texture coordinates for a subsequent texture fetch. This provides the basis for indexed or indirect memory access required for certain packing or compression techniques.

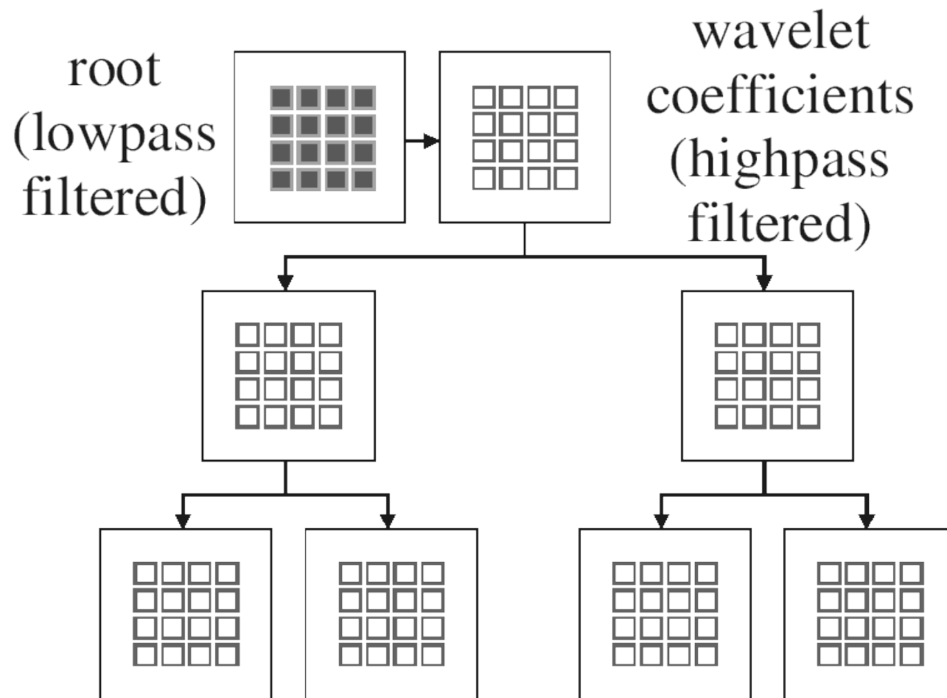


Figure 48.4: The compressed wavelet tree.

48.2 Packing Techniques

Packing techniques try to make efficient use of GPU memory by packing equal or similar blocks of an input volume into a smaller volume as compact as possible. The original volume can then be represented by an index volume referencing those packed blocks (see figure 48.5).

Kraus et al.[59] pack non-empty blocks of the input data into a smaller packed volume representation. This packed representation is then referenced using a index volume that stores the position of the origin of the indexed block in the compact representation and a scaling factor. The scaling factor accommodates non-uniform block sizes (see figure 48.6). During rendering, the decompression is performed in a fragment program. A relative coordinate to the origin of the index cell is computed first. Then the coordinate and the scaling factor of the packed data block are looked up from the texture. From the relative coordinate, the position of the packed block and the scaling factor a position in the packed data is computed which is used to lookup the decoded voxel value. In order to support linear interpolation provided

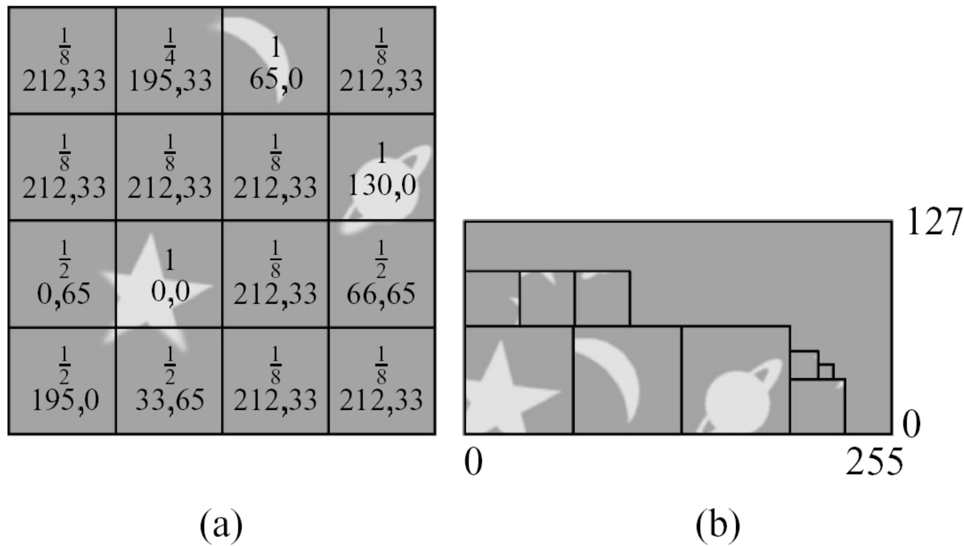


Figure 48.5: (a) Index data: scale factors and coordinates of packed data blocks are stored for each cell of a 4 x 4 grid representing the whole texture map. (b) Packed data: the data blocks packed into a uniform grid of 256 x 128 texels.

by the graphics hardware, texels on the boundaries of data blocks are replicated. As the texture coordinate to lookup the data value in the packed texture is computed based on the result of another texture lookup, the complete decoding algorithm is implemented in the fragment stage. The disadvantage is that dependent texture lookup introduce a big performance penalty, as they result in non-linear memory access patterns. In contrast to decoding the packed representation in the fragment stage, Wei Li et al.[69] decode the packed representation in the vertex stage by slicing axis aligned boxes. They allow arbitrary sized sub-textures that are generated by a box growing algorithm that determines boxes with similar densities and gradient magnitudes. Their main purpose is to accelerate volume rendering by skipping blocks that are empty after the transfer functions were applied. Furthermore, their approach also allows to pack pre-computed gradients into a compact representation (see figure 48.7).

All presented packing techniques support blocks with different sizes. Using uniformly sized blocks brings us to the concept of vector quantization which is the topic of the following section.

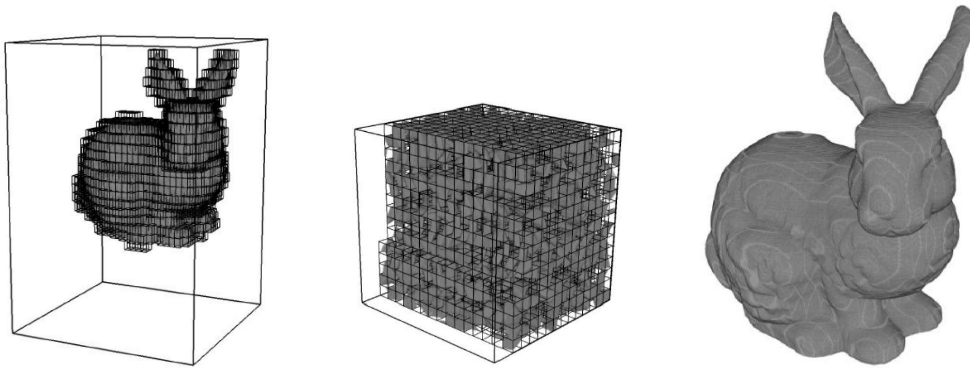


Figure 48.6: Volume rendering of a $512 \times 512 \times 360$ CT scan with adaptive texture mapping. (Left) Non-empty cells of the 32^3 index data grid. (Middle) Data blocks packed into a 256^3 texture. (Right) Resulting volume rendering.

Figure 48.7: Gradient sub-textures defined by the boxes enclose all the voxels of non-zero gradient magnitude (left). The gradient sub-textures are are packed into a single larger texture (middle). Resulting rendering of the foot with mixes boxes and textures.

48.3 Vector Quantization

Due to the availability of indirect memory access provided on GPUs by means of dependent texture fetch operations, vector quantization is an obvious choice for a compression scheme to make more efficient use of available GPU memory resources. In general, a vector quantization algorithm takes a n-dimensional input vector and maps it to a single index that references a codebook containing vector of equal length as the input vector. As the codebook should have a significantly smaller size than the set of input vectors, a codebook should be capable of reproducing an original input vector as close as possible. Hence, a codebook must be generated from the set of input vectors.
Schneider and Westermann [88] introduced vector quantization to GPU-based volume visualization. Vector quantization is applied on two different frequency bands of the input volume data. For that purpose, the data is partitioned into blocks with 4^3 voxels dimension. Each block is down-sampled to a 2^3 block and a difference vector with 64 components is obtained by computing the difference between the original and the down-sampled block. This process is repeated for the 2^3 blocks resulting in a single mean-value for the block and a second 8-component difference vector. Vector quantization is applied to two difference vectors and the two resulting indices are stored together with the mean-value into a RGB 3D-texture (see figure 48.8). The indices and the mean value are fetched during rendering and used to reconstruct the input value. Two dependent texture fetch operations lookup the 8- and 64-component difference vectors from two 2D textures.
For a codebook with 256 entries they achieve a compression factor of $64 : 3$ neglecting the size of the codebooks; i.e., a 1024^3 volume is compressed to $3 * 256^3$=48 MBytes. Thus, it fits easily into the GPU memory and no AGP transfer is required to swap in data during rendering. It should be noted, that despite the fact that the decoding stage for vector quantized data is very simple, frame rates considerably drop compared to uncompressed data due to texture cache misses produced by dependent texture fetch operations. To improve performance a deferred decompression based on early-z tests available on ATI R3xx GPUs is employed. That is, every slice polygon is rendered twice, first with a simple (and thus fast) shader and then with the full (and thus slow) decoding shader. To prevent decoding of empty regions of the data, the first pass evaluates the median value stored in the 3D texture. If the median value is zero, the execution of the complex shader for this fragment is prevented by masking the fragment with a z-value. For the

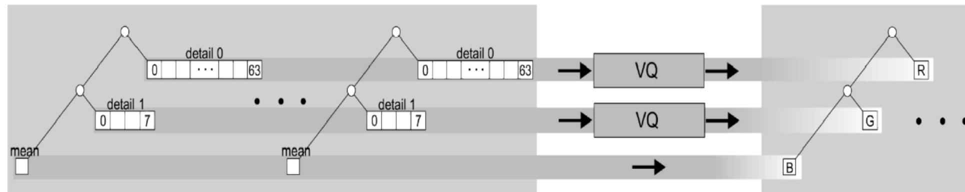


Figure 48.8: Hierarchical decompression and quantization of volumetric data sets. Blocks are first split into multiple frequency bands, which are quantized separately. This generates three index values per block, which are used to reference the computed codebook.

generation of the codebooks, Schneider and Westermann use a modified LBG-Algorithm [70] based on an initial codebook generated by principle component analysis (PCA). For details please refer to their paper [88]. Vector quantization is also applicable to 4D volume data. A shock wave simulation sequence with 89 time steps can be compressed from 1.4 GB to 70 MB, thus it can be played back entirely from fast GPU memory.

Procedural Techniques

In order to capture the characteristics of many volumetric objects such as clouds, smoke, trees, hair, and fur, high frequency details are essential. A very high resolution of the volume data is required to store those high frequency details. There is actually a much better approach to model volumetric natural phenomena. Ebert's[19] approach for modelling clouds uses a coarse technique for modelling the macrostructure and uses procedural noise-based simulations for the microstructure (see Figure 49.1). This technique was adapted by Kniss[53] for interactive volume rendering using a volume perturbation approach which is efficient on modern graphics hardware. The approach perturbs texture coordinates and is useful to perturb boundaries in the volume data, e.g. the cloud boundary in figure 49.1.

The volume perturbation approach employs a small 3D-perturbation volume with 32^3 voxels. Each texel is initialized with four random 8-bit numbers, stored as RGBA components, and then blurred slightly to hide the artifacts caused by trilinear interpolation. Texel access is then set to repeat. An additional pass is required for both approaches due to limitations imposed on the number of textures which can be simultaneously applied to a polygon, and the number of sequential dependent texture reads permitted. The additional pass occurs before the steps outlined in the previous section. Multiple copies of the noise texture are applied to each slice at different scales. They are then weighted and summed per pixel. To animate the perturbation, they add a different offset to each noise texture's coordinates and update it each frame.

Procedural techniques can be applied to most volumetric natural phenomena. Due to the fact that details can be represented a coarse macrostructure volume and small noise textures, there is no requirement to deal with large volumetric data.

Figure 49.1: Procedural clouds. The image on the top shows the underlying data, 64^3. The center image shows the perturbed volume. The bottom image shows the perturbed volume lit from behind with low frequency noise added to the indirect attenuation to achieve subtle iridescence effects.

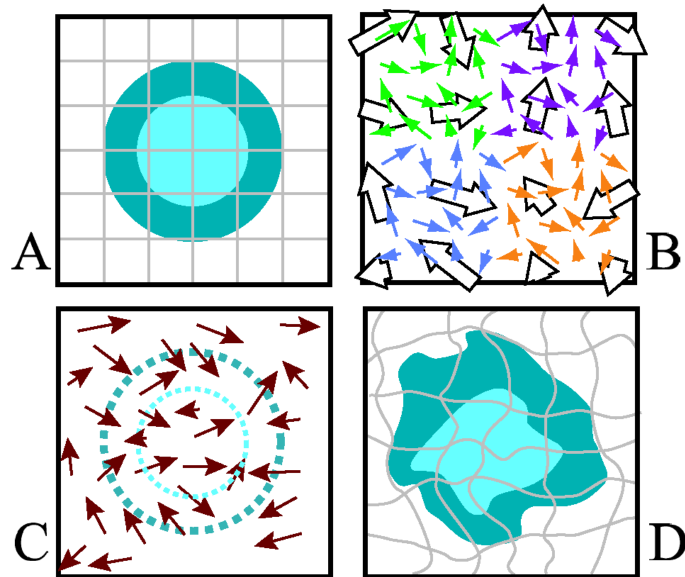


Figure 49.2: An example of texture coordinate perturbation in 2D. A shows a square polygon mapped with the original texture that is to be perturbed. B shows a low resolution perturbation texture applied to the polygon multiple times at different scales. These offset vectors are weighted and summed to offset the original texture coordinates as seen in C. The texture is then read using the modified texture coordinates, producing the image seen in D.

Optimizations

Shading volumetric data requires computing a gradient based on the scalar field at each position. One possible to option is to pre-compute gradients and store the resulting vector per voxel into a texture. During rendering interpolated gradients can be fetched from the texture and used for shading calculations. However, this is impractical in the context of large volumes because the volume data already consumes too much memory to store additional information. In the following section we show how to prevent storing additional gradient information by calculating gradient information on-the-fly.

Computing gradients on-the-fly requires access to multiple interpolated voxel values for each fragment. This increases the required memory bandwidth. Even though today's GPUs provide an enormous peak memory bandwidth of more than 30 GB per second, the visualization of large volume data requires optimization techniques to achieve satisfying frame rates. Consider a 1 GB volume: in theory a 30 GB per second memory bandwidth should allow to access the complete volume data 30 times per second; thus, neglecting rendering times, this should yield 30 frames per second frame rate. In practice however, there are several reasons why this calculation is totally academic. Firstly, the peak GPU memory bandwidth can only be achieved if memory is only access sequentially. This is not typical in many cases, especially when using 3D textures. Thus, the actual sustained bandwidth is much lower. Second, the same volume data value is accessed multiple times during rendering; for example, for trilinear interpolation of two fragments may access the same or voxels of the volume data. Third, some calculations require accessing multiple interpolated data values at one position. For example, for high-quality filtering the contribution of many voxels is summed up. Therefore, we want to prevent unnecessary memory reads. This can be achieved by leaping over empty space or termination rays that have accumulated enough opacity.

50.1 Shading

For shading large volumetric data it is impractical to store pre-computed gradients in texture maps. Instead, as outlined in section 23, gradients can be computed on-the-fly. The result of this are very high-quality gradients with no memory overhead; however, many texture fetches are required for the per fragment gradient calculation (6 for central differences). A large number of texture fetches decrease performance considerably, hence it is desirable to perform those expensive shading computations only if they are actually required.
Often when using quite complex and expensive fragment programs, a two pass approach can provide a huge benefit. Every slice polygon is rendered twice. In a first pass, a simple fragment program is used to mask those fragments that need to be processed by the complex fragment program using a z- or stencil-value in a second pass. In this second pass, only the fragments that passed the test in the first pass are processed by the complex fragment program. This requires an early-z or early-stencil test which is available on ATI R3xx and NVIDIA NV4x class hardware. Such a test is performed by the graphics hardware before the actual fragment program is run for a certain fragment, thus providing a performance benefit.
A fragment program for on-the-fly gradient computation and shading is quite complex in contrast to a simple post-classification shader. As shading effects are only apparent if the transparency of a fragment is not to small, it is sufficient to shade only fragments with a alpha value above a certain threshold. First the slice polygon is rendered with a post-classification shader to determine the opacity of the fragment. By enabling an alpha-test that only allows fragments to pass with an opacity over a certain threshold we can set a z- or stencil-value for those fragments. In the second pass z- or stencil test is enabled to process only fragments that had a opacity larger then the given threshold in the first pass.
This optimization can provide a significant speedup when many fragments are classified with a small or zero alpha value.

50.2 Empty Space Leaping

Volumetric data sets contain many features that are not required for the final rendering. Typically, they are removed by setting zero alpha values

in the transfer function. In order to not waste computation power on features that have been removed by the transfer function, empty space leaping can be employed to reduce memory access and save computation power.
For that purpose, the volume is subdivided into smaller blocks. In order to detect empty space, all blocks of the volume are classified in a pre-processing step. For each block, the minimum and maximum scalar values are stored in a separate data structure. Based on the transfer function and the minimum and maximum density values, the visibility of each block can quickly be determined after each change of the transfer function. This allows us to slice only visible blocks (see figure 50.1), thus increasing the number vertices but reducing the number of fragments.

Because the vertex processor is idle during volume rendering

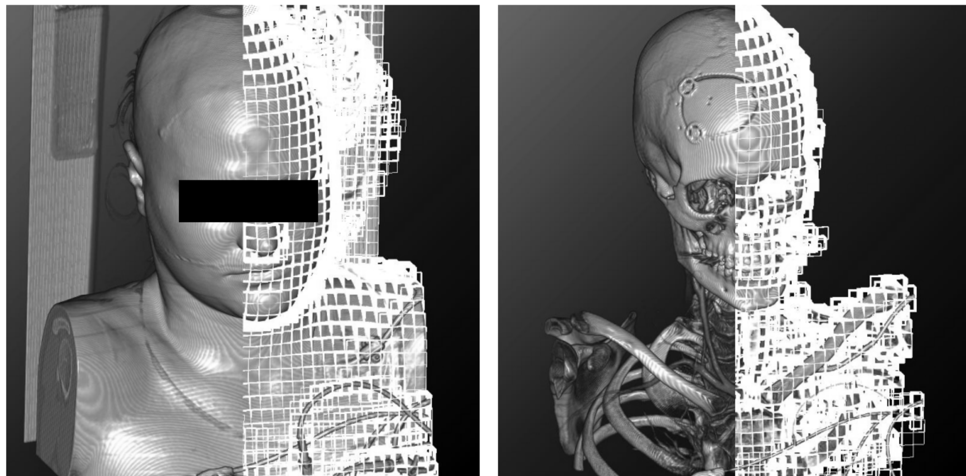


Figure 50.1: Non-empty blocks of a CT volume for two different transfer functions.

in most of the cases anyway (the number of slice polygons is rather small), this technique can provide significant performance improvements.

50.3 Ray Termination

Early ray-termination is another important optimization technique known from ray-casting. When tracing rays through a volume in a front-to-back fashion many rays quickly accumulate full opacity. This means, that features in the data set in the back are occluded and

must not be considered for rendering the image. Thus the ray can be terminated.
This technique was recently introduced to hardware-accelerated volume rendering algorithms [60, 84]. It requires front-to-back compositing and is based on early-z or early-stencil operations; i.e., z- or stencil-tests are performed by the graphics hardware for each fragment before the fragment shader code is executed. In order to mask rays (pixels) that can be terminated, every n integration steps an intermediate pass is performed. By rendering to a texture, the frame buffer can be applied as a texture map to a screen-filled quad that is rendered to the screen. Only those pixels with an alpha-value above a certain threshold, a depth or stencil values is written. Pixels with this stencil or depth value are excluded during volume rendering by means of the stencil or depth-test.
Early-ray termination is a complementary acceleration technique to empty-space skipping. If the volume contains a lot of empty space, empty-space skipping performs quite well while early-ray termination does not provide a big speedup. Vice versa, if the volume does not contain much empty space and the transfer function is not too transparent, early-ray termination provides a big speedup while empty-space skipping does not perform well.

Summary

Despite the huge gap between volume data sizes from different application areas and the amount of memory on today's GPUs, it is possible to render large volumes at acceptable frame rates. Many techniques were introduced that either try to reduce the transfer of texture data over AGP or try to make more efficient use of the available high-speed GPU memory. As many traditional techniques, like pre-computed gradients, are not feasible for large volumes, certain properties of the scalar field must be computed on-the-fly. This however, introduces additional computational cost which require optimization techniques to prevent unnecessary memory access and calculations.

Course Notes 28

Real-Time Volume Graphics

Rendering From Difficult Data Formats

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

Rendering From Difficult Data Formats

52.1 Introduction

The previous chapters of this course have discussed numerous volume rendering techniques for data stored in 3D texture memory. This chapter describes how the previously described volume rendering techniques can be applied to volumetric data stored in "difficult" formats. We define a "difficult" format to be any data space that differs from the regular-grid, 3D volumetric space in which the rendering is defined. Example difficult formats include compressed data and a stack of 2D slices.

The need to represent volumetric data in difficult formats arises in two scenarios. The first case is when the data is stored on the GPU in a compressed format. The second case arises when the 3D data is generated by the GPU (i.e., dynamic volumes). Dynamic volumes arise when performing General-Purpose Computation on Graphics Processor (GPGPU) techniques for simulation, image processing, or segmentation [10, 31, 37, 62, 67, 96, 87, 93]. Current GPU memory models require that such dynamic volume data be stored in a collection of 2D textures (or pixel buffers) [36]. As such, real-time visualization of these data require a new variant of 2D-texture-based volume rendering that uses only a *single* set of 2D slices.[1].

This chapter describes the two main challenges of volume rendering from difficult formats: reconstruction and filtering. The reconstruction stage decompresses the data into the original 3D volume domain. The filtering stage uses multiple samples of the reconstructed volume to recreate a continuous representation from the discrete data. This is the same filtering problem discussed earlier in these notes, but with the added difficulty that the neighboring data elements required for filtering are

[1]The proposed Superbuffer [73] extension to OpenGL enables the GPU to render to slices of a 3D texture. This extension has not yet been approved by the OpenGL Architecture Review Board (ARB) as of April, 2004

often not available (or are expensive to compute), due to the on-the-fly volume decompression. We describe the *deferred filtering* [67] algorithm as a solution to this problem.

52.2 Volume Domain Reconstruction

The process of reconstructing volume domain data on-the-fly at render time is currently of great interest to the computer graphics and visualization communities [3, 6, 57, 89]. Computer games can use these techniques to save valuable texture memory, and visualization applications can leverage compression strategies to interactively view data sets that are too large to fit in GPU memory. Modern programmable fragment processors make it possible to perform increasingly complex decompressions at interactive rates.

There are two basic approaches to volume-domain reconstruction. The first uses a fragment program to translate volume-domain texture addresses to data-domain addresses before reading from the data texture. The address translation occurs using fragment arithmetic operations and one or more index textures [6, 57, 89]. The second method uses the CPU and/or vertex processor to pre-compute the data-domain texture addresses [67]. This technique is especially applicable when no indexing data is available, such as when volume rendering from a set of 2D slices. Figure 52.1 shows an example of using pre-computed data-domain texture addresses.

An example of volume reconstruction using pre-computed data-domain addresses is the rendering technique described in Lefohn et al. [67]. The algorithm reconstructs each slice of the volume domain at render time using small texture mapped quadrilaterals or line primitives. The geometric primitives represent contiguous regions in the data space, and the texture coordinates of these primitives are the data-domain addresses. Figure 52.1 shows an example of this reconstruction. This technique is a modified form of 2D-texture-based volume rendering that allows full volume rendering from only a *single* set of 2D slices. The memory savings comes at the cost of processing a larger amount of small geometry.

The new 2D-texture-based reconstruction algorithm is as follows. When the preferred slice axis, based on the viewing angle, is orthogonal to the 2D slice textures, no reconstruction is needed and a slice-sized quadrilateral is drawn. If the preferred slice direction is parallel to the 2D slices, the algorithm renders a row or column from each slice using

textured line primitives, as in Figure 52.1.

Note that although the description only allows for axis-aligned view reconstructions, axis-aligned views can be interpolated to provide the illusion of view-aligned volume rendering. Additionally, if the technique is applied to compressed data stored in 3D textures, view-aligned slices can be reconstructed using many small polytopes which each represent contiguous regions of the data domain texture.

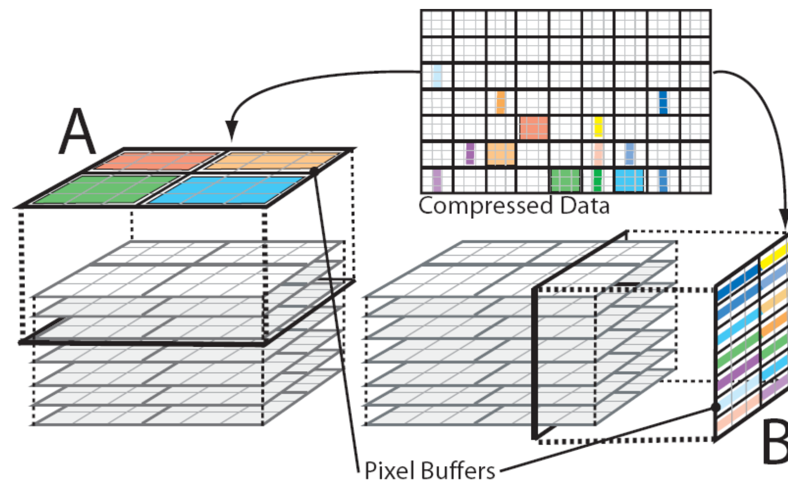


Figure 52.1: Reconstruction of a slice for volume rendering packed data (a) When the preferred slicing direction is orthogonal to the 2D memory layout, the memory tiles (shown in alternating colors) are draw into a pixel buffer as quadrilaterals. (b) For slicing directions parallel to the 2D memory layout, the tiles are drawn onto a pixel buffer as either vertical or horizontal lines. Note that the lines or quads represent contiguous regions in the data domain.

52.3 Filtering

As discussed in earlier chapters, the use of trilinear filtering (or better) is generally required to reduce objectionable aliasing artifacts in the volume rendering. Many compression-domain volume rendering techniques, however, do not support filtering because the reconstruction step precludes the use of the GPU's native filtering hardware. In this section, we describe a deferred-filtering technique that overcomes this problem for a large class of difficult-data-format rendering algorithms. The algorithm, originally published in Lefohn et al. [67], allows compression-domain renderings to use achieve fast trilinear filtering by leveraging the GPU's native bilinear filtering engine.

A standard volume renderer leverages the GPU's native trilinear filtering engine to obtain a filtered data value from texture memory. A brute force method for performing filtered, compression-domain volume rendering is to reconstruct a data value for each element in the filter kernel. Unfortunately, this approach is impractical on current GPUs because of the very large number of fragment instructions and texture accesses required (8 data reads, at least 8 index texture reads, and many arithmetic operations). While it is possible to implement this method with modern GPUs, the rendering performance is unacceptable for interactive use. A significant problem with this brute force method is that adjacent fragments perform many redundant data decompression operations.

The deferred filtering algorithm optimizes the brute-force approach by separating the reconstruction and filtering stages of the data access. The separation avoids the duplicate decompressions and leverage GPU bilinear filtering hardware.

Deferred filtering is conceptually simple (Figure 52.2. Each slice of the volume (that would normally be rendered in a single pass) is rendered in two passes. The first pass performs the volume domain reconstruction using the appropriate decompression/reconstruction algorithm for the data format. This reconstruction is performed for two consecutive slices in the volume, and the results are stored in temporary 2D textures. The second (filtering) pass reads from the two reconstructed slices using the GPU's native bilinear filtering. The final, trilinearly filtered, result is computed by linearly interpolating between the data values read from the two reconstructed slices. Lighting and transfer functions are then applied to the data in this second stage (see Figure 52.2).

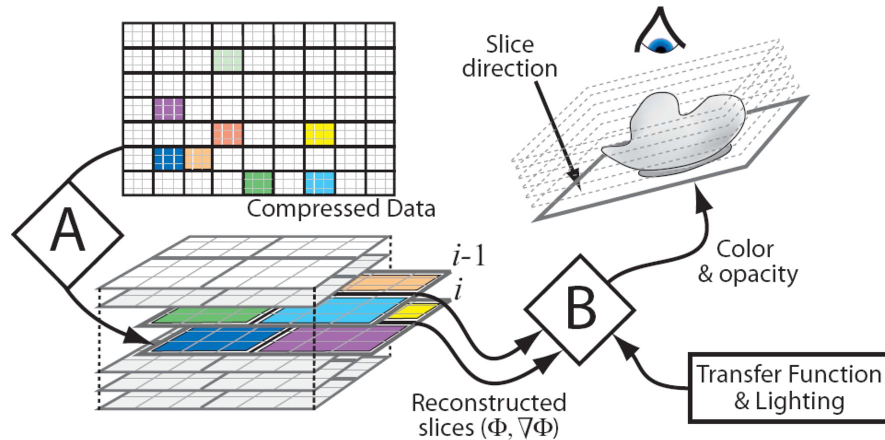


Figure 52.2: Deferred filtering algorithm for performing fast trilinear filtering when volume rendering from difficult data formats (e.g., compression domain volume rendering). The first stage reconstructs two consecutive volume-domain slices and saves the results in textures. The second stage reads data values from these two textures using the GPU's native bilinear filtering engine. The final, trilinearly filtered, result is computed by linearly interpolating between two reconstructed data values in a fragment program. The lighting and transfer function computations are applied as normal in this second stage.

52.4 Conclusions

This chapter discusses how the volume rendering algorithms presented in previous chapters can be applied to data stored in compressed or slice-based formats. Rendering from these complex formats requires separate volume reconstruction and filtering stages before the transfer function and lighting computations are applied. We discuss both fragment-processor and vertex-processor volume reconstruction (i.e., decompressions). We also describe a deferred filtering algorithm that enables fast trilinear filtering for compression-domain/difficult-data-format volume rendering.

Course Notes 28

Real-Time Volume Graphics

Literature

Klaus Engel
Siemens Corporate Research, Princeton, USA

Markus Hadwiger
VR VIS Research Center, Vienna, Austria

Joe M. Kniss
SCI, University of Utah, USA

Aaron E. Lefohn
University of California, Davis, USA

Christof Rezk Salama
University of Siegen, Germany

Daniel Weiskopf
University of Stuttgart, Germany

SIGGRAPH2004

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] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 373–378, August 1996.

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

[5] Praveen Bhaniramka and Yves Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. In *Proc. Volume Visualization and Graphics Symposium 2002*, pages 45–53, 2002.

[6] Alcio Binotto, Joo Comba, and Carla Freitas. Real time volume rendering of time-varying data using a fragment shader approach. In *IEEE Parallel and Large Data Visualization and Graphics*, October 2003.

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

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

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

[10] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM Transactions on Graphics*, volume 22, pages 917–924, July 2003.

[11] 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.

[12] C. Chua and U. Neumann. Hardware-Accelerated Free-Form Deformations. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[13] P. Cignino, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In R. Scateni, J. van Wijk, and P. Zanarini, editors, *Visualization in Scientific Computing*, pages 58–71. Springer, 1995.

[14] S. Coquillart. Extended Free-Form Deformations. In *Proc. SIGGRAPH*, 1990.

[15] B. Csebfalvi, L. Mroz, H. Hauser, A. König, and M. E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. In *Proceedings of EUROGRAPHICS 2001*, pages 452–460, 2001.

[16] 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.

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

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

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

[20] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, pages 195–202, 2000.

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

[22] Cass Everitt. Interactive Order-Independent Transparency. White paper, NVidia, 2001.

[23] S. Fang, S. Rajagopalan, S. Huang, and R. Raghavan. Deformable Volume Rendering by 3D-texture Mapping and Octree Encoding. In *Proc. IEEE Visualization*, 1996.

[24] 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.

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

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

[27] Michael P. Garrity. Raytracing Irregular Volume Data. In *Proceedings of the 1990 Workshop on Volume Visualization*, pages 35–40, 1990.

[28] Andrew S. Glassner. Wavelet transforms. In A. Glassner, editor, *Principles of Digital Image Synthesis*, pages 243–298. Morgan Kaufman, 1995.

[29] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of SIGGRAPH '98*, pages 447–452, 1998.

[30] B. Gooch and A. Gooch. *Non-Photorealistic Rendering.* A.K. Peters Ltd., 2001.

[31] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, pages 102–111, July 2003.

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

[33] S. Guthe and W. Straßer. *Real-Time Decompression and Visualization of Animated Volume Data.* IEEE Visualization '01 Proceedings, pp. 349–356, 2001.

[34] 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.

[35] 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.

[36] Mark Harris, David Luebke, Ian Buck, Naga Govindaraju, Jens Kruger, Aaron Lefohn, Tim Purcell, and Cliff Woolley. GPGPU: General purpose computation on graphics hardware. In *ACM SIGGRAPH Course Notes*, page Course 32, 2004.

[37] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Graphics Hardware 2002*, pages 109–118, September 2002.

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

[39] P. Hastreiter, H.K. Çakmak, and T. Ertl. Intuitive and Interactive Manipulation of 3D Datasets by Integrating Texture Mapping Based Volume Rendering into the OpenInventor Class Hierarchy. In T. Lehman, I. Scholl, and K. Spitzer, editors, *Bildverarbeitung für die Medizin - Algorithmen, Systeme, Anwendungen*, pages 149–154, Universität Aachen, 1996. Verl. d. Augustinus Buchhandlung.

[40] H. Hauser, L. Mroz, G-I. Bischi, and E. Gröller. Two-level volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):242–252, 2001.

[41] 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.

[42] J. Hladůvka, A. König, and E. Gröller. Curvature-based transfer functions for direct volume rendering. In *Proc. of Spring Conference on Computer Graphics 2000*, pages 58–65, 2000.

[43] V. Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *Proc. of IEEE Visualization '95*, pages 52–59, 1995.

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

[45] M. Karasick, D. Lieber, L. Nackmann, and V. Rajan. Visualization of three-dimensional delaunay meshes. *Algorithmica*, 19(1-2):114–128, 1997.

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

[47] G. Kindlmann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of IEEE Volume Visualization '98*, pages 79–86, 1998.

[48] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. of IEEE Visualization 2003*, pages 513–520, 2003.

[49] 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.

[50] Davis King, Craig M Wittenbrink, and Hans J. Wolters. An Architecture for Interactive Tetrahedral Volume Rendering. In Klaus Mueller and Arie Kaufman, editors, *Volume Graphics 2001, Proceedings of the International Workshop on Volume Graphics 2001*, pages 163–180. Springer, 2001.

[51] J. Kniss, G. Kindlmann, and C. Hansen. Multi-dimensional transfer functions for interactive volume rendering. In *Proceedings of IEEE Visualization 2001*, pages 255–262, 2001.

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

[53] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interative Translucent Volume Rendering and Procedural Modeling. In *IEEE Visualization*, 2002.

[54] 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.

[55] M. Kraus. *Direct Volume Visualization of Geometrically Unpleasant Meshes*. PhD thesis, University of Stuttgart, 2003.

[56] M. Kraus and T. Ertl. Cell-Projection of Cyclic Meshes. In *Proc. of IEEE Visualization 2001*, pages 215–222, 2001.

[57] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, pages 7–15, 2002.

[58] Martin Kraus. *Direct Volume Visualization of Geometrically Unpleasant Meshes*. Dissertation, University of Stuttgart, Germany, 2003.

[59] Martin Kraus and Thomas Ertl. Adaptive Texture Maps. In *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop*, 2002.

[60] Jens Krueger and Ruediger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[61] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proc. of IEEE Visualization 2003*, pages 287–292, 2003.

[62] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics*, volume 22, pages 908–916, July 2003.

[63] Yair Kurzion and Roni Yagel. Interactive Space Deformation with Hardware-Assisted Rendering. *IEEE Computer Graphics & Applications*, 17(5), 1997.

[64] 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.

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

[66] B. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *Proc. of IEEE Visualization 2003*, pages 131–138, 2003.

[67] A. E. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, page To Appear, 2004.

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

[69] Wei Li and Arie Kaufman. Texture partitioning and packing for accelerating texture-based volume rendering. *Graphics Interface*, pages 81–88, 2003.

[70] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications COM-28, 1(Jan.)*, pages 84–95, 1980.

[71] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. In *Proceedings of IEEE Visualization 2002*, pages 211–218, 2002.

[72] R. MacCracken and K. Roy. Free-Form Deformations with Lattices of Arbitrary Topology. In *Proc. SIGGRAPH*, 1996.

[73] Rob Mace. Superbuffers. `http://www.ati.com/developer/gdc/SuperBuffers.pdf`, 2004.

[74] 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.

[75] N. Max, P. Hanrahan, and R. Crawfis. *Area And Volume Coherence For Efficient Visualization Of 3D Scalar Functions.* ACM Computer Graphics (San Diego Workshop on Volume Visualization) 24(5), pp. 27–33, 1990.

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

[77] 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.

[78] D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer graphics. In *Proc. of SIGGRAPH '88*, pages 221–228, 1988.

[79] T. Möller, K. Müller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *Proc. of IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[80] K.G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. In *Computer Graphics Forum 20(3)*, 2001.

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

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

[83] 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.

[84] S. Röttger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238, 2003.

[85] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of VisSym 2003*, pages 231–238, 2003.

[86] 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.

[87] M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings EG/IEEE TCVG Symposium on Visualization*, pages 75–84, 2001.

[88] J. Schneider and R. Westermann. *Compression Domain Volume Rendering*. IEEE Visualization '03 Proceedings (to appear), 2003.

[89] Jens Schneider and Rudiger Westermann. Compression domain volume rendering. In *IEEE Visualization*, pages 293–300, October 2003.

[90] T. Sederberg and S. Parry. Free-Form Deformation of Solid Geometric Models. In *Proc. SIGGRAPH*, 1986.

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

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

[93] Anthony Sherbondy, Mike Houston, and Sandy Nepal. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visualization*, pages 171–196, October 2003.

[94] Peter Shirley and Allan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):63–70, November 1990.

[95] T. Strothotte and S. Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation.* Morgan Kaufmann, 2002.

[96] R. Strzodka and M. Rumpf. Using graphics cards for quantized FEM computations. In *Proceedings VIIP Conference on Visualization and Image Processing*, 2001.

[97] U. Tiede, T. Schiemann, and K. H. Höhne. High quality rendering of attributed volume data. In *Proceedings of IEEE Visualization '98*, pages 255–262, 1998.

[98] Ulf Tiede, Thomas Schiemann, and Karl Heinz Höhne. High Quality Rendering of Attributed Volume Data. In *Proc. of IEEE Visualization '98*, pages 255–262, 1998.

[99] K. K. Udupa and G. T. Herman. *3D Imaging in Medicine*. CRC Press, 1999.

[100] J. van Wijk. Image based flow visualization for curved surfaces. In *Proc. of IEEE Visualization 2003*, pages 745 – 754, 2003.

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

[102] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. of IEEE Visualization 2003*, pages 333–340, 2003.

[103] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl. *Level-Of-Detail Volume Rendering via 3D Textures*. IEEE Volume Visualization 2000 Proceedings, 2000.

[104] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-Based View-Independent Cell Projection. *IEEE Transactions on Visualization and Computer Graphics (Special Issue on IEEE Visualization 2002)*, 9(2):163–175, April-June 2003.

[105] D. Weiskopf, K. Engel, and T. Ertl. Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization. In *Proc. of IEEE Visualization 2002*, pages 93–100, 2002.

[106] D. Weiskopf, K. Engel, and T. Ertl. Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

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

[108] R. Westermann and C. Rezk-Salama. Real-Time Volume Deformation. In *Computer Graphics Forum (Eurographics 2001)*, 2001.

[109] P. L. Williams and N. L. Max. A Volume Density Optical Model. In *Computer Graphics (Workshop on Volume Visualization '92)*, pages 61–68, 1992.

[110] Peter L. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[111] C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *Proc. IEEE Visualization Late Breaking Hot Topics*, pages 21–24, 1999.

[112] 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.

[113] Craig M. Wittenbrink. R-Buffer: A Pointerless A-Buffer Hardware Architecture. In *Proceedings Graphics Hardware 2001*, pages 73–80, 2001.