Volume Rendering Techniques for General Purpose Graphics Hardware

---

Techniken der Volumenvisualisierung auf universell einsetzbarer Graphik-Hardware

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Christof Rezk-Salama

Erlangen – Dezember 2001

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	18. Dezember 2001
Tag der Promotion:	14. März 2002
Dekan:	Prof. Dr. A. Winnacker
Berichterstatter:	Prof. Dr. G. Greiner
	Prof. Dr. T. Ertl

Abstract

In recent years the value of direct volume rendering techniques for the visualization of 3D scalar fields has become evident in many application areas ranging from medicine to natural science and engineering. The applicability of high quality volume rendering, however, was mainly restricted to expensive special purpose hardware or dedicated workstations with high-end graphics subsystem and high-speed memory bus. To these ends this thesis introduces methods for interactive high-quality volume visualization on general purpose hardware such as commodity desktop computers with graphics cards designed for computer games and multimedia. The aim of this work is to achieve a high image quality comparable to traditional ray-casting solutions at interactive frame rates on inexpensive hardware platforms. In this context the benefits and drawbacks of traditional texture based implementations are analyzed with respect to image quality and rendering performance. Based on this analysis, efficient volume rendering techniques are developed targeting the advanced features of modern PC graphics boards such as multi-stage rasterization, pixel shaders and dependent texture lookup.

In the context of direct volume rendering, transfer functions are used to specify the emission and absorption values which are required for ray integration. Several implementations of transfer functions for pre- and post-classification are presented and analyzed. Automatic image- and data-driven techniques for transfer function design are examined and adapted to different application problems. Advanced features of the graphics hardware are used to include local illumination effects into direct volume rendering and techniques for non-polygonal isosurface display. The lighting effects are achieved as per-pixel illumination with dynamic light sources or as reflection maps, which cache the incident illumination at one point. The analysis of texture based algorithm is completed by a detailed performance measurement on different hardware architectures.

As supplements to the 3D-texture based method, volumetric deformation models are introduced which allow the intuitive modeling of volume objects as well as the automatic optimization of deformation parameters for registration purposes. The presented approaches comprise an efficient algorithm for slice decomposition of arbitrary deformed polygonal surfaces and a deformation model based on regular hexahedra structures.

A major goal of this work was the improvement of the availability of direct volume rendering for specific visualization problems in medicine and natural science. The application of different techniques in clinical environments are documented in several case studies including the visualization of the inner ear, the examination of tiny vascular structures as well as the functional analysis of the vertebral column.

Revision 1.0
©2001, Copyright by Christof Rezk-Salama
All Rights Reserved
Alle Rechte vorbehalten

Contents

Abstract iii

Table of Contents viii

List of Figures xii

List of Tables xiii

Listings xiv

Acknowledgements xv

Preface xvii

I Introduction 1

1 Volume Rendering 2
- 1.1 Physical Background 2
- 1.2 Volume Data 4
- 1.3 Algorithms 6
  - 1.3.1 Indirect Methods 7
  - 1.3.2 Direct Methods 10

2 Graphics Hardware 14
- 2.1 The Graphics Pipeline 14
  - 2.1.1 Geometry Processing 15
  - 2.1.2 Rasterization 16
  - 2.1.3 Per-Fragment Operations 17
- 2.2 APIs 18
  - 2.2.1 OpenGL 19
  - 2.2.2 Direct3D 19

2.3 Volume Rendering Hardware 21

II Volume Rendering **24**

3 Texture Based Volume Rendering **25**
3.1 The 2D-Texture Based Approach 25
3.1.1 Principles 26
3.1.2 Compositing 28
3.1.3 Discussion 30
3.2 The 3D-Texture Based Approach 32
3.2.1 Viewport-Aligned Slices 34
3.2.2 Bricking 35
3.2.3 Discussion 36

4 2D-Multi-Texture Based Methods **38**
4.1 Rasterization Revisited 38
4.1.1 Multi-Textures 39
4.1.2 Pixel Shaders 40
4.2 Principles 43
4.3 Interpolation of Arbitrary Slices 44
4.4 Discussion 45

5 Transfer Functions **48**
5.1 Principles 48
5.1.1 Pre-Classification 49
5.1.2 Post-Classification 50
5.2 Implementations 50
5.2.1 Pre-Classification 50
5.2.2 Post-Classification 53
5.3 Discussion 56
5.4 Multi-Dimensional Transfer Functions 58
5.4.1 Implementation 61
5.4.2 Fusion 63
5.5 Local Transfer Functions 64
5.5.1 Implementation 65

6 Transfer Function Design **67**
6.1 Interactive Adjustment 67
6.2 Image-Driven Techniques 69
6.3 Data-Driven Techniques 70
6.3.1 Automatic Adaptation 72
6.4 Conclusion 74

7 Local Illumination **75**
7.1 Principles . . . 75
7.2 Non-Polygonal Isosurfaces . . . 77
7.3 Per-Pixel Illumination . . . 79
7.3.1 Implementations . . . 79
7.4 Reflection Maps . . . 83
7.5 Discussion . . . 86

8 Performance Measurement **88**
8.1 Architectures . . . 89
8.1.1 Consumer PC Hardware . . . 89
8.1.2 SGI Graphics Workstations and Servers . . . 91
8.2 Performance Analysis . . . 92
8.3 Conclusion . . . 99

9 Extensions **101**
9.1 Multi-Texture Speedup . . . 101
9.2 Stencil Buffer Clipping . . . 105
9.3 Vector Fields . . . 109
9.3.1 Line Integral Convolution . . . 110
9.3.2 Animated 3D LIC . . . 112

10 Deformation **114**
10.1 Principles . . . 114
10.2 Volumetric Free-Form Deformation . . . 115
10.2.1 Shape and Appearance . . . 115
10.2.2 Intersection Calculation . . . 116
10.2.3 Shape Deformation . . . 118
10.2.4 Appearance Deformation . . . 119
10.2.5 Illumination . . . 120
10.3 Hexahedra Deformation . . . 122
10.3.1 Deformation Model . . . 122
10.3.2 Modeling . . . 123
10.3.3 Adaptive Subdivision . . . 124
10.3.4 Implementation . . . 125
10.3.5 Illumination . . . 126
10.4 Discussion . . . 129
10.5 Registration . . . 130

III Applications in Medicine **134**

11 Introduction **135**

11.1 Medical Image Data . . . 135
11.1.1 Computed Tomography . . . 136
11.1.2 Magnet Resonance Imaging . . . 136
11.1.3 Partial Volume Effects . . . 136

12 The Inner Ear **138**
12.1 Background . . . 138
12.2 Surface Reconstruction . . . 139
12.3 Direct Volume Rendering . . . 142
12.4 Results . . . 142

13 Intracranial Aneurysms **144**
13.1 Background . . . 144
13.2 Visualization . . . 145
13.3 Semi-transparent Isosurfaces . . . 146
13.4 Results . . . 147

14 Dural Arteriovenous Fistulae **150**
14.1 Background . . . 150
14.2 The MR-CISS Sequence . . . 151
14.3 Pre-Processing . . . 153
14.4 Results . . . 154

15 The Vertebral Column **157**
15.1 Background . . . 157
15.2 Data Acquisition . . . 158
15.3 Pre-Processing . . . 159
15.4 Results . . . 161

IV Conclusion **165**

16 Summary **166**
16.1 Future Challenges . . . 168

A Data Sets **169**

Bibliography **184**

List of Figures

1.1 Reconstruction filters for one-dimensional signals 4
1.2 Voxel model of a volumetric object 5
1.3 The visualization pipeline 6
1.4 The visualization pipeline for indirect volume rendering 7
1.5 Cell configurations of the *Marching-Cubes*-algorithm 8
1.6 The visualization pipeline for reconstruction of surfaces from segmentation 9
1.7 The visualization pipeline for volume ray casting 11
1.8 Principles of the shear-warp-algorithm for parallel projection 12
1.9 Principles of the shear-warp-algorithm for perspective projection 12
1.10 Decomposition of a volume into object-aligned slices 13

2.1 The standard graphics pipeline for display traversal 15
2.2 Geometry processing as part of the standard graphics pipeline 15
2.3 Rasterization as part of the standard graphics pipeline 16
2.4 Per-fragment operations as part of the standard graphics pipeline 17

3.1 Decomposition of the volume object into object-aligned polygon slices . . . 26
3.2 CT: Comparison between alpha blending and MIP 29
3.3 Distance between adjacent sampling points depending on the viewing angle 30
3.4 Aliasing artifacts become visible at the edges of the slice polygons 30
3.5 Sampling artifacts are caused by changing to a different slice stack 31
3.6 Decomposition of the volume object into viewport-aligned polygon slices . 33
3.7 Sampling illustrated for viewport-aligned slices 33
3.8 Sorting of edge intersection points to form a valid polygon 34
3.9 Interpolation at the brick boundaries illustrated for the 1D case 35

4.1 Multi-textures as a strict sequence of texturing operations 39
4.2 NVidia's register combiners bypass the standard texture application unit . 40
4.3 NVidia's general register combiner 41
4.4 NVidia's final register combiner 42
4.5 Register combiner setup for interpolation of intermediate slices 43
4.6 Rendering procedure for interpolating slice images in arbitrary direction. . 45
4.7 Register combiner setup for interpolation of arbitrary slice images. 45

5.1 Transfer functions for pre- and post-classification . . . 49
5.2 Dependent texture lookup . . . 54
5.3 Reconstruction of an arbitrary set of discrete values. . . . 59
5.4 Comparison of pre- and post-classification of a CTA data set . . . 60
5.5 Register combiner setup for opacity weighting with gradient magnitude . . 61
5.6 Pre-classified transfer function with linear gradient weighted opacity . . . 62
5.7 Texture shader and register combiner setup for multi-dimensional transfer functions . . . 63
5.8 The 3D anatomical atlas *VoxelMan* . . . 64
5.9 Principles of local transfer function application . . . 65
5.10 Local transfer function for pre-classification during pixel transfer . . . 65
5.11 CT scan of a frog with local transfer functions . . . 66

6.1 User interfaces for editing transfer functions . . . 68
6.2 Monotonously increasing function and its first and second order derivatives 70
6.3 Averaged derivatives and position function . . . 71
6.4 Position functions and histograms for different CTA data sets . . . 73
6.5 Automatic adaptation of transfer function templates . . . 74

7.1 The Phong illumination model . . . 76
7.2 Non-polygonal isosurface with different illumination effects . . . 77
7.3 Combiner setup for fast rendering of shaded isosurfaces using 3D-textures. 80
7.4 Combiner setup for fast rendering of shaded isosurfaces. . . . 80
7.5 Combiner setup for rendering semi-transparent volumes with local diffuse illumination. . . . 81
7.6 Examples of illumination effects for non-polygonal isosurfaces and semi-transparent volume data . . . 82
7.7 Example of a spherical environment map . . . 83
7.8 Example of an environment cube map . . . 84
7.9 Non-polygonal isosurface with diffuse and specular lightmaps . . . 85

8.1 Modern Consumer PC architecture with *Accelerated Graphics Port (AGP)*. 89
8.2 The *Unified Memory Architecture (UMA)* as implemented in the SGI O2. . 91
8.3 Performance of 2D-texture based volume rendering . . . 93
8.4 Performance of 3D-texture based volume rendering . . . 94
8.5 Comparison of 2D- and 3D-texture based volume rendering . . . 95
8.6 Performance of 2D-multi-texture based volume rendering . . . 96
8.7 Supersampling with 2D-multi-texture based volume rendering . . . 97
8.8 The influence of the transfer functions on the overall performance . . . 98
8.9 Performance of illumination techniques for texture based volume rendering 100

9.1 Rendering two slices with a single slice polygon . . . 102
9.2 Combiner setup for correct blending of two slices with one polygon . . . 103
9.3 Clipping a volume against an arbitrary polygonal object. . . . 105

9.4 The idea of *stencil buffer clipping* . . . 105
9.5 Efficient implementation of stencil buffer clipping. . . . 107
9.6 Voxelization of a polygonal mesh . . . 109
9.7 2D noise texture and resulting LIC image . . . 110
9.8 CFD simulation of turbulent flow inside the wheel casing of a car. . . . 111
9.9 Time surfaces inside a simple cavity flow field. . . . 112
9.10 Animation sequence of a data set from numerical CFD simulation generated with the stencil buffer clipping approach. . . . 112
9.11 CFD simulation of turbulent flow inside the wheel casing of a car. Animation sequence generated with the stencil buffer clipping approach. . . . 113

10.1 Example of direct volume rendering using the free-form deformation model 116
10.2 Separation of shape from appearance in the volumetric case . . . 117
10.3 The active edge data structure . . . 117
10.4 The influence of the tessellation on the visible deformation . . . 118
10.5 The displacement volume specifies the extent of a local deformation . . . 119
10.6 Volumetric free-form deformation with different levels of subdivision for the displacement volumes . . . 120
10.7 Register combiner setup for the computation of forward differences . . . 121
10.8 Example of volumetric free form deformation with diffuse illumination . . . 121
10.9 Deformation model based on hexahedra structures . . . 123
10.10 Inconsistent subdivision leads to gaps in texture space . . . 124
10.11 Edge and face constraints . . . 125
10.12 Object-aligned slices are extracted at low computational cost. . . . 126
10.13 Interpolation problems within slice polygons of the deformed volume . . . 127
10.14 Illumination artifacts for piecewise linear patches . . . 128
10.15 Realistic illumination of the animated tail fin of a carp . . . 129
10.16 Iterative procedure for image registration . . . 131
10.17 2D compound histograms for mono- and multi-modal data sets . . . 132

11.1 Partial volume effects in CT and MRI data . . . 137

12.1 The inner ear . . . 139
12.2 High-resolution CT slice images of the temporal bone . . . 140
12.3 Surface reconstruction process for the inner ear. . . . 141
12.4 CT data set of the temporal bone . . . 142
12.5 Comparison of pre- and post-classification of a high-resolution CT data set 143

13.1 Intracranial arteries and its relation to the cerebral nerves . . . 145
13.2 Dual-pass rendering for semi-transparent isosurfaces . . . 147
13.3 Example of semi-transparent isosurface rendering of CTA data I . . . 148
13.4 Example of semi-transparent isosurface rendering of CTA data II . . . 149

14.1 The MR-CISS sequence . . . 151

14.2 MR-CISS images of the spinal column 152
14.3 Sequence of image processing operations for the coarse segmentation of MR-CISS data 153
14.4 Intricate vascular structure in the area of the brain stem 155
14.5 Dural arteriovenous fistula in the area of the thoracic spine 155
14.6 Distention of venous vessels caused by a dural arteriovenous fistula 156

15.1 Anatomical structures related to the vertebral column 158
15.2 Slice image of an MR-MEDIC sequence of the vertebral column 159
15.3 Sequence of image processing operations for the segmentation of the MR-MEDIC data 160
15.4 Comparison of x-ray myelography to volume rendering of MR-MEDIC data in inclination 161
15.5 Comparison of x-ray myelography to volume rendering of MR-MEDIC data in reclination 162
15.6 Volume rendering of MR-MEDIC data with local transfer functions 163
15.7 Functional MR-MEDIC data of a patient with spinal stenosis and spondylolisthesis 164

List of Tables

2.1 Comparison between OpenGL and Direct3D graphics APIs 20
2.2 Hardware architectures for volume rendering 22

3.1 Summary of 2D-texture based volume rendering. 32
3.2 Summary of 3D-texture based volume rendering. 36

4.1 Summary of 2D-multi-texture based volume rendering. 46

5.1 Overview of different implementations of transfer function lookup 57

8.1 Specifications of the PC systems used for the experiments 91

Listings

3.1 OpenGL sample code for selecting the slice direction 27
3.2 OpenGL compositing setup for alpha blending 28
3.3 OpenGL compositing setup for maximum intensity projection 29
5.1 OpenGL setup for color mapping during the pixel transfer from main memory to the local texture memory. 51
5.2 OpenGL setup for the paletted texture extension. 52
5.3 OpenGL setup for post-interpolative color table lookup 53
5.4 OpenGL setup for the dependent texture lookup. 55
5.5 DirectX 8.0 pixel shader setup for post-classification via dependent texture lookup. 56
7.1 OpenGL setup for the alpha test. 78
7.2 OpenGL setup for the the dot product extension. 79
7.3 OpenGL setup for the reflection mapping using diffuse and reflective cube maps. 86
9.1 OpenGL setup for rendering one textured slice polygon using stencil buffer clipping. 108

Acknowledgements

First of all, I would like to express my gratitude to my supervisors, Prof. G. Greiner and Prof. T. Ertl, for their friendship and for the incredible support that finally lead to the success of this work. Prof. Ertl and his group have made it easy to bridge the geographical distance to Stuttgart and to successfully work on common scientific problems. For all his duties, Prof. Greiner has always taken the time to explain complex topics in an understandable way. The benefit of this collaboration is invaluable.

I am very much obliged to the staff of the Division of Neuroradiology of the university hospital in Erlangen, especially to Dr. habil. K. E. W. Eberhardt and Dr. B. F. Tomandl for allowing me crucial insight into clinical matters. Throughout the years of my doctorate, I have learned that such a fruitful and unbureaucratic cooperation must not be taken for granted.

Prof. R. Westermann from the *Scientific Computing and Visualization Group* at the *University of Aachen* is held in high esteem for his fellowship and for sharing his profound knowledge of all aspects of volume rendering. I'm also grateful to my colleagues at the *Visualization and Interactive Systems Group* of the *University of Stuttgart*, namely Matthias Hopf, Martin Schulz, especially Klaus Engel for sharing his ideas, Sabine Iserhardt-Bauer for joint development, and Ove Sommer for his unequalled programming skills.

I would like to extend a very special thank you to Jörg Scherer, who discussed and implemented some ideas on transfer function design. Above all, I want to express my appreciation to Michael Bauer for his invaluable contribution to texture-based volume rendering and registration, that he developed within the scope of his pre-diploma and master thesis.

My deep respect is due to Dr. C. Teitzel for the constructive discussions on flow visualization and also to Dr. P. Hastreiter for laying the groundwork for medical visualization in Erlangen and for arranging and managing so many projects.

Over and above that, I'm much obliged to David Kirk from *NVidia, USA*, to Michael Doggett, Rex Sikora and Steve Morein from *ATI, Canada* and to Eckehard Traber from the *ELSA AG, Germany* for making the latest graphics boards available to me.

I also wish to thank the members of the Graduate Research Center *3D Image Analysis and Synthesis* and all my gifted colleagues at the Computer Graphics Group, namely our

secretary Maria Baroti, the researchers Kai Hormann, Frank Reck, Grzegorz Soza, Gerd Sussner, especially Christian Vogelgsang, Peter Kipfer and Roman Sturm for uncompromising system administration, my office mate Ulf Labsik and my true follower Michael Scheuering.

Finally, and most of all, I'd like to express my love and gratitude to my family, especially to my parents and my girlfriend Iris for helping me over many difficulties, for supporting my career both ideologically and financially and for tolerating my occasionally obsessive behavior.

Christof Rezk-Salama

Preface

Information is not knowledge,
Knowledge is not wisdom,
Wisdom is not truth ...

Frank Zappa (1940–1993)
Joe's Garage

Visual artists portray their perception of reality in pictures which reflect their individual impressions and emotions. The beholder observes and draws conclusions consciously as well as unconsciously. Similar to visual arts, scientific visualization communicates information which is still in need of interpretation. Instead of subjective impressions, the underlying information here is scientific data, as it arises from measurement or simulation. However, in both cases aesthetical requirements are never neglected.

Visualization in general denotes the transformation of symbolic information into intuitive geometric representations, which are both effective and expressive. Compared to traditional rendering, preciseness and comprehensibility are of greater importance than photorealism. Human vision enables us to grasp a large amount of information and understand the connectivity within the twinkling of an eye. The generation of synthetic images which enable the analysis and interpretation of abstract scientific data is an important aid in science and engineering. Although computer graphics in the early years has often been called *"a solution in search of a problem"* , it has rapidly evolved to a renowned scientific discipline and to an integral part of computer science.

Nowadays, as a consequence of the fast evolution of technology, typical scientific data sets usually contain much more information than it is possible to communicate in a static image. Scientific visualization has become a creative and explorative process which reveals structures and their connectivity. This requires an interaction between the user and the visual representation of his data. The development of real-time algorithms to classify and display the steadily increasing amount of data is still a great challenge for computer graphics.

The major goal of the research described in this thesis was to investigate and improve the applicability of volume visualization for specific problems in medical data analysis. An enjoyable side effect was the extension of volume rendering to a larger variety of scientific disciplines apart of medicine. When I started my research activity three years ago, interactive high-quality visualization of volumetric data was restricted to expensive graphics

hardware, that was unaffordable for most research facilities. Volume rendering systems were usually operated by experts in computer graphics, who exactly knew the underlying rendering algorithm. However, these experts usually were unable to interpret the information contained in the data, which belonged to a completely different scientific area. The conclusion from this dilemma was that scientific visualization must be a cooperative task, that involves both profound knowledge in computer graphics as well as the ability to interpret the data according to its specific scientific origin.

With the development of general purpose graphics hardware for the mass market, today volume visualization is available on multiple platforms in a variety of different implementations. As an effect of this technical progress, inexpensive visualization systems are now operated by scientists, physicians and engineers with only marginal knowledge of the underlying technology. Although the images generated by volume rendering algorithms are purely virtual, for the interpretation of the data it is still important to exactly know how the original information is visually represented. In this context, the relevant parameters to evaluate the image quality of volume visualization are still not clear. In consequence, there is an increasing demand for a standardized work flow to produce high-quality results, which are reproducible on multiple systems and thus reliable for documentation purposes. Especially for an application in medicine, there is also a clear trend towards an automation of repetitive tasks, such as clipping and coloring.

The thesis is organized into three parts. The introductory part gives an overview of volume visualization in general. As a prerequisite for successive chapters this part deals with the mathematical and physical basis, with general problems of volume rendering, possible solutions and their approximations. In this context an outline of related work is presented, as well as an introduction to basic concepts of graphics hardware. The second part is a detailed description of the developed volume rendering methods based on general purpose hardware. Starting with the basic algorithms, supplements and enhancements such as classification and illumination are successively introduced. The subsequent chapters discuss strategies for transfer function design and models for volumetric deformation. The third part of this thesis describes the practical aspects of this work. The application of volume rendering to several problems in clinical practise and research is presented. A study of the inner ear using high-resolution CT data exemplifies the comparison between direct and indirect visualization methods. Tissue classification via transfer functions is demonstrated by examination of tiny vascular structures, followed by advanced clinical visualization problems of high complexity.

Part I

Introduction

Chapter 1

Volume Rendering

In the context of this thesis the term *volume rendering* refers to the visualization of static 3D scalar fields. The interpretation of such data fields is considerably difficult because of their intrinsic complexity. A variety of different approaches have been developed in the past. This chapter tries to draw an overall picture of significant concepts and ideas. In Section 1.1 the physical basis for image synthesis is explained. Section 1.2 takes a closer look at volumetric data with respect to signal processing theory. In Section 1.3 the basic algorithms for volume visualization are outlined.

1.1 Physical Background

The fundamental concept of all physically based rendering methods is the transport theory of light. The equation of radiative transfer completely describes the radiation field in a participating medium, which involves emission, absorption and scattering of light [68].

The intensity radiated from a given point $\vec{x}$ in direction $\vec{n}$ is determined by the *radiance* $I(\vec{x}, \vec{n}, \nu)$, which is dependent on the frequency ν. Radiance is directly proportional to the photon density. *Absorption* χ of light consists of two terms,

$$\chi(\vec{x}, \vec{n}, \nu) \;=\; \kappa(\vec{x}, \vec{n}, \nu) \;+\; \sigma(\vec{x}, \vec{n}, \nu). \tag{1.1}$$

True absorption κ transforms radiant energy from the visual spectrum into thermal energy. *Scattering* σ causes a change in direction $\vec{n}$ of the radiation. Analogously, the *emission* of light η also consists of a source term q and a scattering part j,

$$\eta(\vec{x}, \vec{n}, \nu) \;=\; q(\vec{x}, \vec{n}, \nu) \;+\; j(\vec{x}, \vec{n}, \nu). \tag{1.2}$$

The equation of radiative transfer can be written as a differential equation,

$$\frac{\partial}{\partial s} I(\vec{x}, \vec{n}, \nu) \;=\; -\chi(\vec{x}, \vec{n}, \nu)\, I(\vec{x}, \vec{n}, \nu) \;+\; \eta(\vec{x}, \vec{n}, \nu). \tag{1.3}$$

Scattering of light is a complex process which changes both frequency ν and direction $\vec{n}$ of the radiant energy. Integrating radiance I along rays is only valid if scattering can be completely neglected. Thus, volume rendering approaches in general use an *emission-absorption model*, that leaves scattering out of account. As a consequence, frequency dependance (variable ν) can also be safely ignored. The analytic solution of differential equation 1.3 is then given by

$$I(s) \;=\; I(s_0)\, e^{-\tau(s_0,s)} \;+\; \int_{s_0}^{s} q(\tilde{s})\, e^{-\tau(\tilde{s},s)}\, d\tilde{s} \tag{1.4}$$

with the *optical depth*

$$\tau(s_1, s_2) \;=\; \int_{s_1}^{s_2} \kappa(s)\, ds. \tag{1.5}$$

The numerical solution of this equation requires a discretization along the ray. The integration range is divided into n intervals. Radiance I at a discrete position s_k can then be computed iteratively according to

$$I(s_k) \;=\; I(s_{k-1})\, e^{-\tau(s_{k-1},s_k)} \;+\; \int_{s_{k-1}}^{s_k} q(s)\, e^{-\tau(s,s_k)} ds. \tag{1.6}$$

The term

$$\vartheta_k \;=\; e^{-\tau(s_{k-1},s_k)} \tag{1.7}$$

is called the *transparency* of the medium. The *emission term*

$$b_k \;=\; \int_{s_{k-1}}^{s_k} q(s)\, e^{-\tau(s,s_k)} ds. \tag{1.8}$$

describes the increase in radiant energy that is caused by active emission within the range $[s_{k-1}, s_k]$ along the ray. According to the *density-emitter* model [144] a volumetric object can be understood as a space filled with light emitting particles, which are described by means of a density function $\rho(s)$. The source term q and the true absorption coefficients κ are then proportional to the particle density, according to

$$\kappa(s) \;=\; \kappa_0 \cdot \rho(s) \qquad \text{and} \qquad q(s) \;=\; q_0 \cdot \rho(s) \tag{1.9}$$

with κ_0 and q_0 being constants. Substituting this into the emission term in Equation 1.8 results in

$$b_k \;=\cdots=\; \frac{q_0}{\kappa_0}\,(1 - e^{-\tau(s_{k-1},s_k)}) \;=\; \frac{q_0}{\kappa_0}\,(1 - \vartheta_k). \tag{1.10}$$

Using these abbreviations, the iterative solution to the equation of transfer in Equation 1.6 reads

$$I(s_k) \;=\; I(s_{k-1}) \cdot \vartheta_k + b_k \;=\; I(s_{k-1}) \cdot \vartheta_k + \frac{q_0}{\kappa_0}(1 - \vartheta_k). \tag{1.11}$$

This iterative approach to solving the *emission-absorption*-model is the fundamental equation for almost all methods of direct volume rendering. Before we examine volume rendering algorithms in detail, we will have a closer look at volumetric data and its representations.

1.2 Volume Data

Compared to surface data which solely determines the outer shell of an object, volume data is used to describe the internal structures of a solid object. Volume data also allows the modeling of fluid and gaseous objects as well as natural phenomena, such as clouds, fog, fire or water.

A scalar volume can be interpreted as a continuous three-dimensional signal

$$f(\vec{x}) \in \mathbb{R} \quad \text{with} \quad \vec{x} \in \mathbb{R}^3. \tag{1.12}$$

Provided that the original 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 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 1.1a) in the spacial domain. For the one-dimensional case, the sinc function reads

  $$sinc(x) \; = \; \frac{\sin(\pi x)}{\pi x}. \tag{1.13}$$

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

- 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.

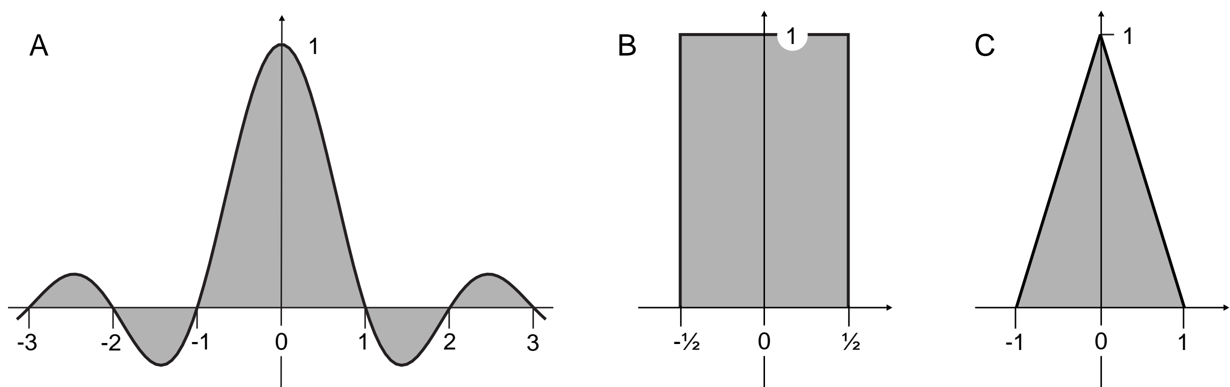


Figure 1.1: Reconstruction filters for one-dimensional signals. Ideal *sinc*-Filter *(A)*, box filter *(B)* and tent filter *(C)*. The 3D versions of these filters are simply obtained by tensor product.

Discrete volume data sets differ in the structure of the underlying sampling grid. Measured data sets as they arise from computed tomography (CT), magnet resonance imaging (MRI) or ultrasound (US) are usually acquired on a uniform *rectilinear* grid. In contrast, grid generation algorithms for finite element simulation (FEM) usually produce *unstructured* grids based on tetrahedra and prisms. With respect to the fact that the vast majority of volume data sets arise from measurement, we will focus our interest on algorithms for uniform rectilinear grid.

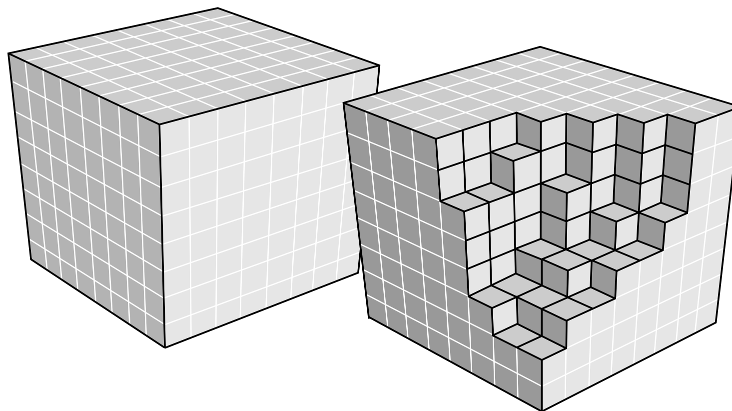


Figure 1.2: Voxel model of a volumetric object.

In this case, a discrete volume data set is simply a three-dimensional array of cubic elements (voxels) [79], each representing a unit of space (Figure 1.2). In order to reconstruct a continuous signal from this array in practise the ideal 3D *sinc* filter is usually replaced by either a box filter (Figure 1.1b) or a tent filter (Figure 1.1c). 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. A more detailed overview of different reconstruction filters for volume data can be found in [119].

A variety of different data sets are generated from *tomographic* measurement. Whether tomographic data sets represent adequately sampled signals according to signal processing theory strongly depends on the acquisition and reconstruction process. Up until now for computed tomography (CT) in general it is not always possible to guarantee proper band-limitation. For magnet resonance tomography (MRT) some techniques are available that ensure band-limitation. However, in clinical practise these acquisition techniques are rarely used. As a consequence, aliasing artifacts are visible in the images which do not result from the rendering algorithm, but from inadequate data acquisition.

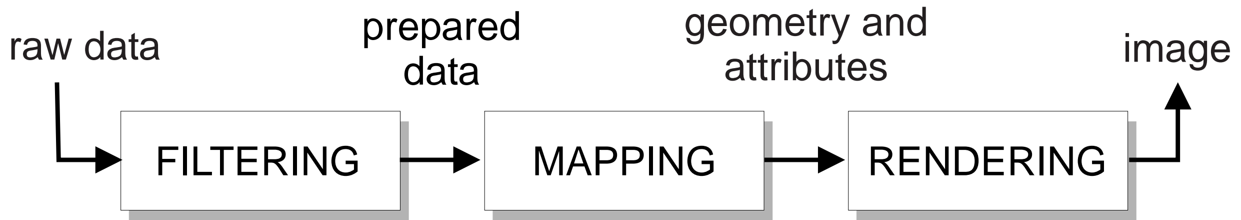


Figure 1.3: The visualization pipeline.

1.3 Algorithms

In literature, the general process of image generation is often described as a *visualization pipeline* [148, 53]. The most elementary version of this pipeline consists of three tiers as displayed in Figure 1.3. The input of such a system is a field of raw data. In most cases not the entire information contained in the original data field is relevant. Usually only a subset or a partial aspect is interesting. The initial *filtering* step selects the data to be analyzed and extracts it from a possibly huge amount of raw data. We assume, that validation and error-correction is performed as pre-processing step and is not part of this filtering step. The main purpose of this tier is resampling and interpolation as well as data reduction according to internal or user-specified selection criteria. The *mapping* stage is the core of every visualization system. This step comprises the detection of structures and their correlation within the data. This can be as simple as a color table lookup, but also as complex as tracing a particle in a vector field. The aim of the mapping step is to convert the abstract information into renderable shapes, which are further decomposed into geometric primitives (points, lines, triangles) and their visual attributes (size, color, transparency etc.). The *rendering* step finally uses the generated primitives to synthesize virtual images. This requires the scan conversion (*rasterization*) and the compositing of each geometric primitive. Parts of this process can be accelerated by the graphics hardware. For efficient visualization, all stages of this pipeline must be carefully adapted to the specific problem.

From the experience in real application scenarios, a basic set of requirements has been derived in order to compare different volume rendering solutions:

- **Interactivity:** Interactivity is required in every aspect of volume rendering. Although real-time performance (frame rate >30 Hz) is desirable, in some cases a minimum frame rate of 1 Hz or more might be sufficient depending on the application area. For traditional (non-stereoscopic) 2D displays interactivity also considerably contributes to the perception of depth.

- **Image Quality:** In order to evaluate the image quality of a volume rendering application two aspects must be considered. On the one hand the information contained in the data set must be represented accurately according to the underlying voxel model. On the other hand, visual artifacts caused by the individual rendering algorithm must be avoided. Some applications provide a mechanism to trade image quality for speed. While a user interaction takes place, e.g. a rotation, the volume

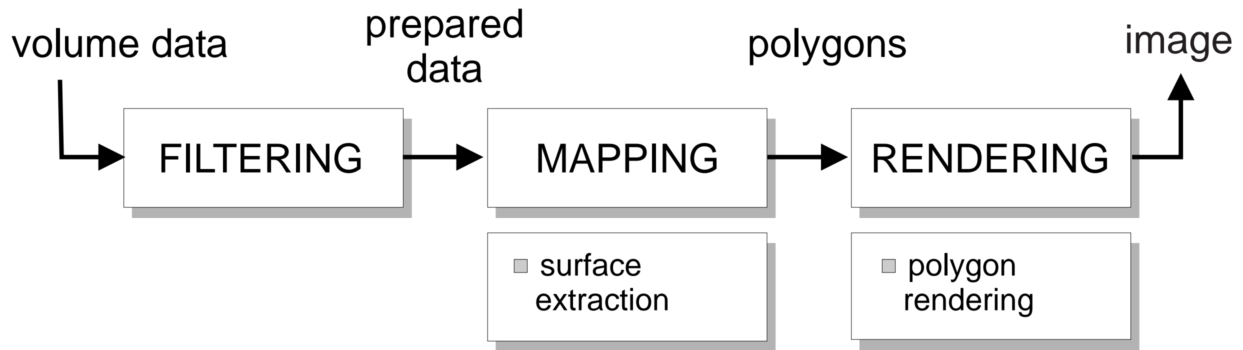


Figure 1.4: The visualization pipeline for indirect volume rendering.

object is displayed at lower resolution. When the interaction has finished, the object is again displayed at its original high resolution. Although this might an be option for enhancing the usability, in many applications this trade-off is not desirable. Augmented reality applications in medicine that have a permanent user interaction are a prominent example.

- **Availability:** A few years ago, interactive direct volume rendering was restricted to expensive graphics workstations and special purpose hardware. Today availability has become one of the main issues in volume visualization. A general solution for increasing the availability of very expensive graphics computers are client-server frameworks [37, 39] in which multiple clients communicate with a single high-end volume rendering server via local or global area networks. In contrast, the aim of the work presented in this thesis is to increase the acceptance of direct volume rendering by bringing it to inexpensive hardware platforms.

Keeping these basic concepts in mind, we will have a closer look at different algorithms in the following sections. According to the way the information content is represented geometrically, volume rendering algorithms can be roughly categorized into *direct* and *indirect* approaches. Indirect methods in general compute level surfaces from the data, while direct methods display the volume as a three-dimensional semi-transparent medium.

1.3.1 Indirect Methods

Indirect methods extract homogenous regions of equal or similar attributes and display the boundary surface of such regions. This category comprises all approaches that transform the voxel data into surface representations in a pre-processing step. The resulting surface is finally displayed by the use of traditional polygon rendering techniques. Thus, with respect to the visualization pipeline the major part of the computation of indirect approaches takes place in the *mapping* step (see Figure 1.4). The complex task of converting volume data into polygonal surfaces is completely performed prior to the rendering step. Indirect methods differ mainly in the way this surface extraction is achieved.

1.3.1.1 Isosurface Extraction

The most popular indirect method is the *Marching-Cubes*-algorithm, developed by Lorensen et al. [96]. This algorithm constructs the level surface of a given iso-value by collecting the contribution of all grid cells of eight neighboring voxels. For each of these cubic cells a binary classification of the eight corner vertices is computed, which results in one of $2^8 = 256$ possible cell configurations, four of which are displayed in Figure 1.5. The number of possible configurations can be reduced by eliminating symmetric cases. The remaining configurations are stored in a lookup table for efficiency. This table also guarantees that faces from neighboring cells exactly fit together and form a consistent compound surface. The exact position of the triangle vertex on an edge of a grid cell is computed by linear interpolation of the voxel values.

The original *Marching-Cubes*-algorithm usually results in a large number of very small and narrow triangles. In practice such triangles can be completely removed without significant loss in accuracy by merging vertices from different cells which lie very close to a shared grid point into a single vertex. As a result, very narrow triangles are collapsed into a single edge and very small triangles into a single vertex. This supplement was first introduced by Moore and Warren [110] and is known as *Compact-Cubes*-algorithm. Compared to the original *Marching-Cubes*-algorithm, *Compact-Cubes* is known to reduce the number of triangles by approximately 40 percent.

Further supplements to the *Marching-Cubes* algorithm have been introduced in recent years, including a client-server framework for remote visualization [40] and a method for feature detection [85]. However, in many cases—especially in medicine—the desired surface cannot be extracted from volume data by a simple iso-value threshold. Usually more complex segmentation procedures must be applied in order to determine the target object. In consequence the surface generation process must be adapted to these internal representations.

1.3.1.2 Surface Reconstruction From Segmentation

Segmentation denotes the process of object classification on a per-voxel basis. Each voxel is assigned to an unambiguous structural entity. Explicit segmentation of volumetric data

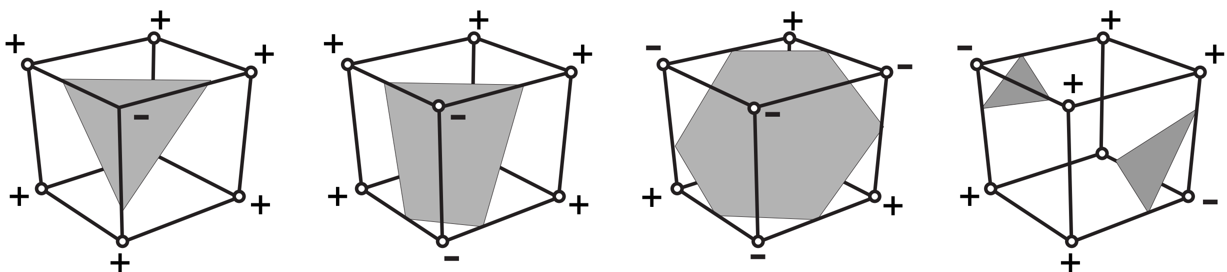


Figure 1.5: Four of the 256 cases of cell configurations used by the *Marching-Cubes*-algorithm.

is a highly complex task, going far beyond the scope of this thesis. Recently developed segmentation techniques range from manual approaches, semi-automatic tools such as active contour models (*Snakes* [78], *Live Wire* [111], *Intelligent Scissors* [112, 113]) and threshold-based methods (e.g. *volume growing* [183, 157], *watershed transformation* [151]) to model-based segmentation [42, 26, 48].

Exact segmentation is definitely the most time-consuming procedure. It is either performed within the filtering step of the visualization pipeline (see Figure 1.6) or as a separate pre-processing step. The mapping step subsequently computes the boundary surface of the segmented objects and the results are again displayed by traditional rendering techniques.

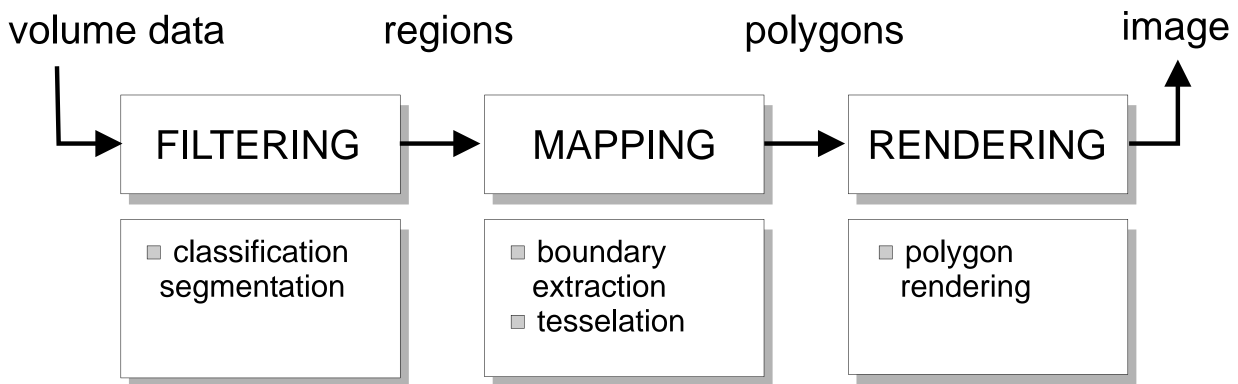


Figure 1.6: The visualization pipeline for reconstruction of surfaces from segmentation.

Many segmentation procedures work on 2D slice images instead of the entire volume. The segmentation result in this case is a stack of polygonal contour lines, which must be adequately joined to form the segmented object. Possible solutions for the specific problems of triangulating contours from neighboring slices are manifold. A detailed survey of surface reconstruction techniques has been prepared by Hastreiter [59].

The polygonal surfaces generated by indirect volume rendering approaches can be displayed in real-time by exploiting the capabilities of modern graphics hardware. This allows interactive exploration of the objects, changing the camera position and zooming in and out on features of interest. As surface generation is usually performed in a time-consuming pre-processing step, the adjustment of an iso-value or any other surface parameter is extremely difficult. Due to the fact that important interior structures might be completely hidden by the generated surfaces, indirect volume rendering always comes with a significant loss of information, that further increases the demand for interactive surface generation. Although many applications for modeling and simulation still require explicit surface descriptions, in recent years indirect methods have been more and more replaced by interactive direct methods that do not require the extraction of surfaces.

1.3.2 Direct Methods

In comparison to the indirect methods presented in the previous section, direct methods display the voxel data by solving the equation of radiative transfer for the entire volumetric object. In direct volume rendering, the scalar value given at a sample point is virtually mapped to physical quantities that describe the emission and absorption of light at that point. This mapping is also often termed *classification*. It is usually performed by means of a transfer function that maps data values to color (emission) and opacity (absorption). These quantities are then used for a physically based synthesis of virtual images.

Similar to a *divide-and-conquer*-strategy, algorithms for direct volume rendering differ in the way the complex problem of image generation is split up into several subtasks. A common classification scheme differentiates between *image-order* and *object-order* algorithms.

Image-order techniques consider each pixel of the resulting image separately. For each pixel, the contribution of the entire volume to this pixels's final color is computed. *Ray casting* is a typical image-order algorithm and will be explained in the following section. Images generated by ray casting represent the reference results in terms of image quality.

Object-order algorithms start with a single voxel and compute its contribution to the final image. This task is iteratively performed for all voxels of the data set. The first object-order algorithm reported in literature was a rendering method presented by Upson and Keeler [165], which processed all voxels in front-to-back order and accumulated the values for the pixels iteratively. Further development of this idea lead to *Splatting* [179, 178, 177, 115], an algorithm which combines efficient volume projection with a sparse data representation. In splatting each voxel is represented as a radially symmetric interpolation kernel, equivalent to a sphere with a fuzzy boundary. Projecting such a structure generates a so-called *footprint* or *splat* on the screen. Splatting traditionally classifies and shades the voxels prior to projection. Mueller et al. have also introduced a method that allows classification and shading to be performed after projection [114]. *Object-order* approaches also comprise *cell projection* [180] and *3D-texture mapping* (see Section 3.2).

A third category, *fourier-space* techniques, has been introduced to also include algorithms which work in the frequency domain [101, 173]. The basic idea of these algorithms is to compute the 3D fourier transformation of the entire volume in a pre-processing step. A projection image of the volume in the spatial domain is then computed by extraction of a slice image in frequency domain. This image is transformed back into the spatial domain by the use of the 2D inverse fourier transformation, resulting in the desired projection. The main advantage of fourier-space techniques is their computational complexity $o(n^2)$, disregarding the 3D fourier transformation which is computed once in a pre-processing step. The major drawback of fourier-space techniques however is that they do not allow an interactive modification of any display parameter other than the viewing direction.

1.3.2.1 Ray Casting

Ray casting denotes the process of integrating the radiative energy which is emitted and absorbed along rays of sight. In the mapping step of the visualization pipeline (see Figure 1.7) the data set is therefore resampled at equidistant positions along a viewing ray. Each interpolated discrete sample value is then mapped to emission and absorption coefficients. According to the equation of radiative transfer and its approximation outlined in Section 1.1, in the rendering step the discrete emission and absorption values are integrated to obtain the final pixel color.

The first volume rendering method based on a ray casting mechanism was developed by Kajiya [77]. Color and opacity values are accumulated for each ray from the eye-point through an image pixel as it passes through the volume data. This accumulation is aborted when the opacity reaches a value of one, which is a concept known as *early-ray termination*. A related approach has been developed by Paolo Sabella [144], who introduced the *density-emitter*-model mentioned in Section 1.1. Marc Levoy [93, 94] examined classification functions for ray casting based on the data value and its gradient magnitude. We will focus our interest on such ideas in Chapter 5. Drebin, Carpenter and Hanrahan [31] introduced *material percentage volumes* as a means of differentiation between several materials contained in the data set. The value at a grid point of a material percentage volume describes the percentage of one material present in the corresponding voxel of the original data set. The material percentage volumes are used to display separate structures differently during ray casting.

Since ray casting accurately models the physical transport of light, the generated images represent the reference results in terms of image quality for comparison with other algorithms. A detailed comparison between ray casting and the Marching Cubes algorithm (see Section 1.3.1.1) can be found in [6]. The most significant contribution to the computational cost of ray casting is caused by the huge number of interpolation operations which is required to resample the data along the viewing rays. The most popular software implementation that tries to reduce this problem is the *shear-warp-algorithm*, which represents a hybrid form of image-order and object-order algorithms.

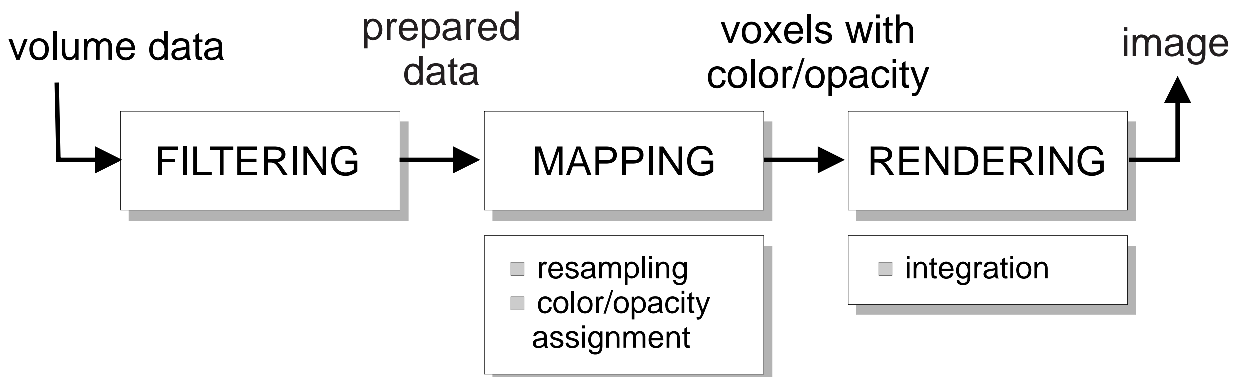


Figure 1.7: The visualization pipeline for volume ray casting.

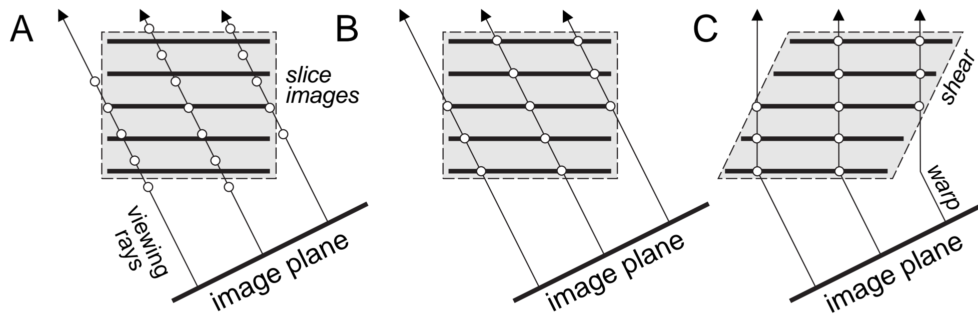


Figure 1.8: Principles of the shear-warp-algorithm for parallel projection.

1.3.2.2 The Shear-Warp Algorithm

The aim of the *shear-warp-algorithm* is to facilitate interpolation by cleverly placing the sample points along the viewing rays. Figure 1.8 illustrates this idea for the case of parallel projection. In order to substitute trilinear by bilinear interpolation, the sampling points (A) are placed exactly onto the slice images (B). For parallel projection the distance between adjacent sampling points changes. Note that the distance is not constant, but dependant on the viewing direction, and this must be considered in the integration step. The basic idea of the shear-warp algorithm is to decompose the viewing transformation into a 3D shear parallel to the slice images, a projection that computes a distorted intermediate image and a 2D warp that generates the final undistorted image (C).

To account for foreshortening in perspective projection an additional scaling must be applied to the slice images as displayed in Figure 1.9. In this case the distance between sampling points along a ray is not only dependent on the eye position, but also on the angle between the viewing rays and the image plane.

The main benefit of this factorization is that rows of voxels are exactly aligned with rows of pixels in the intermediate image, which greatly facilitates the interpolation computation. In combination with run-length encoding, empty space leaping and a data structure that accounts for spatial coherence, this algorithm currently represents the fastest pure software

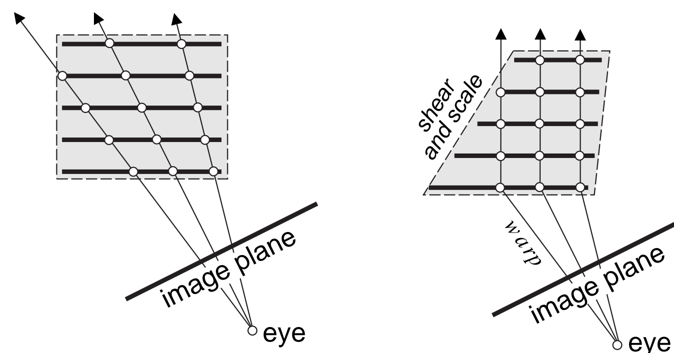


Figure 1.9: Principles of the shear-warp-algorithm for perspective projection.

solution for direct volume rendering. To enable an interactive rotation of the data set, however, three copies of the volume must be kept in main memory, one stack of slices for each slicing direction. With respect to the current position of the image plane relative to the object, the slicing direction must be chosen that minimizes the angle between the slice normal and the viewing ray. Figure 1.10 illustrates this concept by showing a gradual rotation of the volume object. With an angle between viewing direction and slice normal of 45° (D) in this 2D example, the slicing direction becomes ambiguous and can be chosen arbitrarily. Selecting the stack of slices with the minimal angle circumvents the problem that viewing rays may pass between two slices without intersecting one of them. As a result however, the intense memory requirement is one drawback of the algorithm.

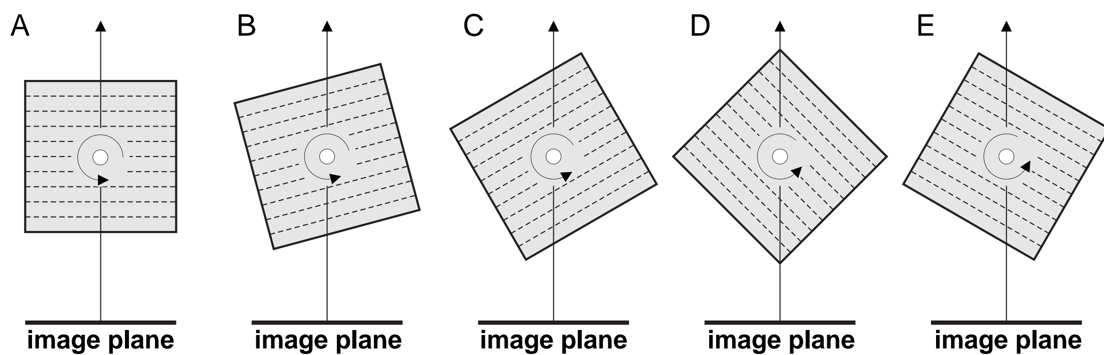


Figure 1.10: Decomposition of a volume into object-aligned slices.

The original shear-warp algorithm was introduced by Lacroute and Levoy [91] in 1994 as a pure software renderer. The basic idea of substituting bilinear by trilinear interpolation however has been adapted for 2D-texture based volume rendering (see Section 3.1). The concept of a shear-warp factorization in turn has lead to the development of special volume rendering hardware as will be outlined in Section 2.3.

In a scientific work flow, modern volume rendering applications claim to be interactive in every aspect. Apart from real-time manipulation of the viewing parameters, this must also include the ability to change iso-values, classification functions and color mappings quickly and easily. For a higher performance,the capabilities of modern graphics hardware architectures can be exploited. An introduction to the general concepts of graphics hardware is presented in the following chapter.

Chapter 2

Graphics Hardware

Nowadays, a 3D graphics acceleration board is included in almost every consumer PC. Since this inexpensive hardware is mainly used for multimedia applications, computer games and entertainment software, this thesis refers to it as *general purpose hardware*. The basic concept of such hardware is outlined in the following sections. Section 2.2 describes available software interfaces that are necessary to efficiently access the hardware using high-level programming languages. In comparison to commodity hardware, Section 2.3 gives a coarse overview of custom architectures which have been especially designed to accelerate direct volume rendering.

2.1 The Graphics Pipeline

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 [45]. 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. 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*.

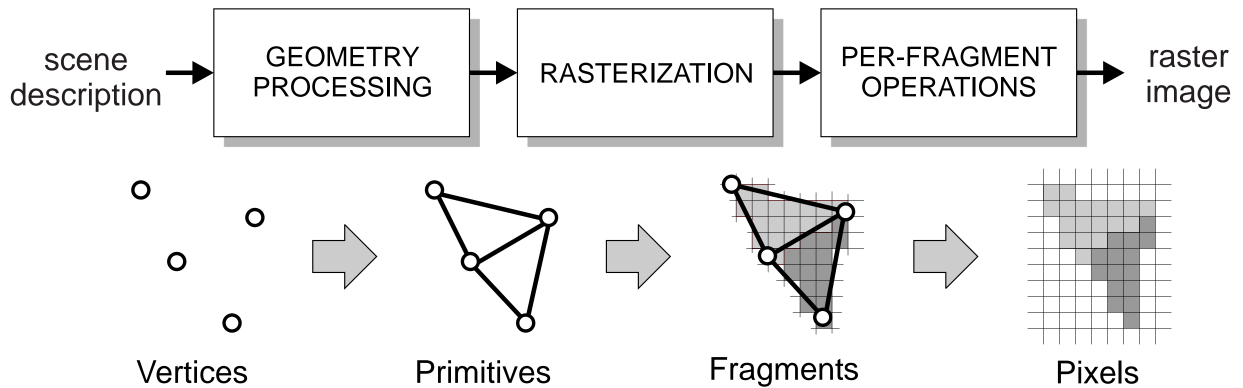


Figure 2.1: The standard graphics pipeline for display traversal.

Per-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.

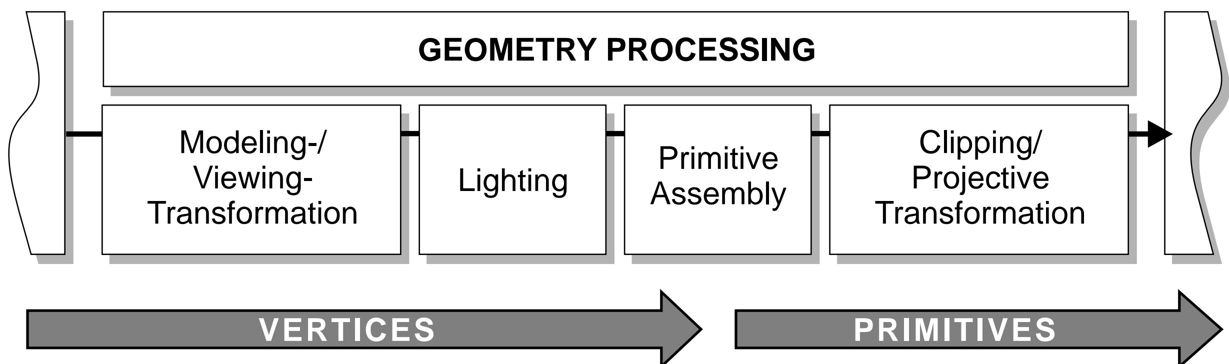


Figure 2.2: Geometry processing as part of the standard graphics pipeline.

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

Lighting: After the vertices are correctly placed within the virtual scene, the Phong model [127] for local illumination is calculated for each vertex. Since this 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 process of rasterization can be further divided into three different subtasks as displayed in Figure 2.3.

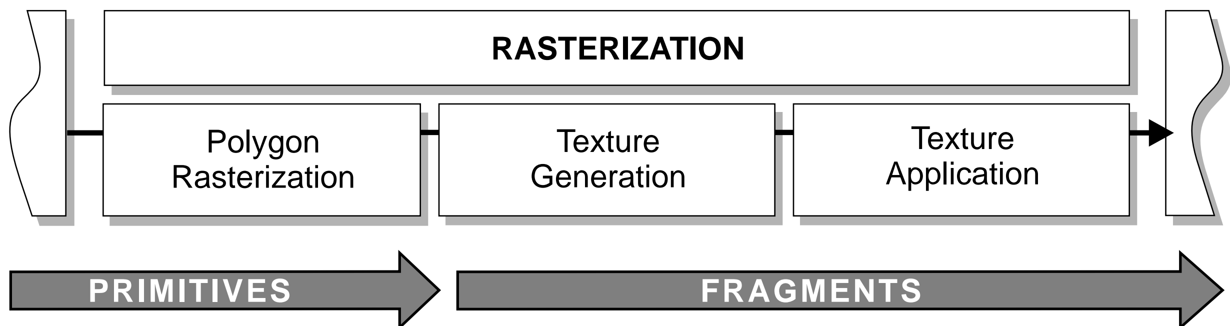


Figure 2.3: Rasterization as part of the standard graphics pipeline.

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 Generation: 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.

Texture Application: 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 Per-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 conditions. The sequence of tests and modifications is termed *per-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

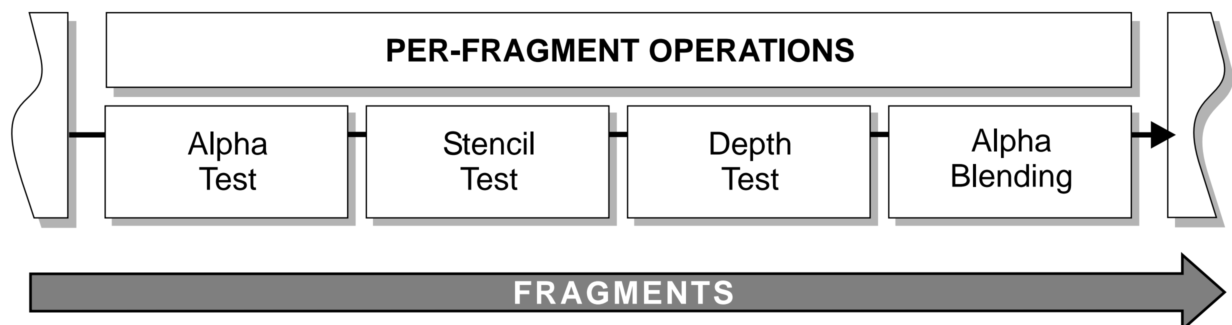


Figure 2.4: Per-fragment operations as part of the standard graphics pipeline.

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 [150, 45]. 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)*.

2.2 APIs

The programming of graphics applications which take advantage of specific hardware features requires the insertion of a standardized software layer acting as an interface between the high-level programming language (C, C++, Java etc.) and the specific device driver that ships with the hardware. This *application programming interface (API)* allows the graphics hardware of different manufacturers to be accessed in a consistent and efficient way. The API takes away the necessity of writing your own low-level, device-specific code to access hardware such as the display adapter, making it much easier for the programmer to write applications that take full advantage of the computer's graphics capabilities. Additionally, a good API also makes it easier for hardware developers to produce new devices that work well in the environment of a specific operating system. The two most important graphics APIs currently available are *OpenGL* and *Direct3D*. Both implementation use the same concept of a graphics pipeline as outlined in Section 2.1. In the following both APIs will be briefly described. From the scientific point of view the following set of criteria should be kept in mind. Programmers of entertainment software might apply another standard.

Efficiency: The API should enable the programming of high performance applications. The computational overhead introduced by providing the required software level of abstraction should be as small as possible.

Extensibility: A good API should be able to rapidly adapt to forthcoming technologies and new hardware features, which have not been available at the time the standard was defined.

Compatibility: Applications which are based on the API should be both forward and backward compatible. Programs written for an early release of the API should also run on future hardware and vice versa.

Platform Independence: For scientific research it is undesirable to restrict applications to a single hardware platform or operating system. This is especially true for prototype applications which are used to evaluate the performance in different environments.

2.2.1 OpenGL

OpenGL [123] is a software interface to graphics hardware which consists of about 200 distinct commands at its core, supplemented by multiple custom extensions. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics API.

OpenGL is an open, vendor-neutral, multi-platform graphics standard. The specification of the OpenGL standard is guided by an independent consortium, the *OpenGL Architecture Review Board (ARB)*, which also ensures backward compatibility. Language bindings are available for C, C++, Fortran, Ada, Python, Perl and Java. All licensed OpenGL implementations are derived from a single specification document and are required to pass a set of conformance tests. OpenGL runs on multiple operating system including Mac OS, OS/2, UNIX, Windows 95/98, Windows NT/2000, Linux, and BeOS.

Although the fundamental specification of OpenGL is based on the graphics pipeline described in Section 2.1, new technological innovations can be made accessible via the OpenGL extension mechanism. Hardware developers are free to adapt and expand their OpenGL implementation to a specific architectural design. OpenGL extensions allow individual routines to be executed on dedicated hardware. A number of particular OpenGL extensions will be used and explained in following chapters. The OpenGL Extension Registry is maintained by *Silicon Graphics Industries (SGI)* and contains specifications for all known extensions, written as modifications to the original specification document. While OpenGL is widely spread in the industrial and scientific community, Direct3D is an API which is mainly used for game programming and the development of multimedia applications.

2.2.2 Direct3D

Direct3D [161, 88] is a graphics API that is part of DirectX [5], a technology developed by Microsoft for programming computer games and entertainment applications. DirectX consists of several components that form two integrated software layers. The *foundation*

	OpenGL	**Direct3D**
Operating systems	UNIX, Linux, MacOs, OS/2, Windows 95/98/ME Windows NT/2000	Windows 95/98/ME Windows 2000
Programming languages	C, C++, Fortran, Ada Perl, Python, Java	all languages that support Microsoft's COM
Open standard	yes	no (Microsoft proprietary)
Extendable	yes (OpenGL extensions)	no
Pros and cons	⊕ platform independent ⊕ independent standard ⊕ open source reference ⊕ extensions ⊖ rather slow standardization	⊕ fast evolution ⊕ reference rasterizer ⊖ Microsoft proprietary ⊖ restricted to Windows

Table 2.1: Comparison between OpenGL and Direct3D graphics APIs.

layer provides general functions that manages the support of hardware devices. The *media layer* is build on top of the foundation layer and provides services for multimedia streaming and animation.

Direct3D originally consisted of two separate modes, *retained mode*, which was part of the media layer, and *immediate mode*, which belongs to the foundation layer. Retained mode provided a library of high-level, easy-to-use commands for multimedia and virtual reality applications. The development for retained mode however has been cancelled by Microsoft with the release of DirectX 6.0. Immediate mode represents a more efficient lowlevel API, that many game developers have adopted for their professional games and engines.

In contrast to the open standard OpenGL, DirectX is property of the Microsoft corporation. The DirectX API is based on Microsoft's component object model (COM) [143] and is thus accessible by all programming languages that support COM. This, however, restricts the DirectX API to Windows platforms.

A comparison between OpenGL and DirectX is outlined in Table 2.1. The difference in the core functionality provided by both APIs is only marginal. The main difference between both APIs is the way they are extended to allow for technological innovations. OpenGL follows the concept of defining a common set of standardized features and leaving special functionality to be implemented as extensions. If after some time a proposed extension turns out to be useful and is adopted by multiple implementations, it will finally be included into the OpenGL standard specification.

In contrast to this, DirectX does not allow such hardware-specific extensions. The feature set defined by the DirectX standard, however, is larger than the functionality implemented by current hardware. Functions that are not supported by the underlying

hardware devices are delegated to software emulation. This allows DirectX to include specifications for future functionality into the API. The Direct3D software development kit (SDK) also ships with a reference rasterizer, which is a virtual device that implements the complete Direct3D specification as software emulation. This might enable graphics programmers to develop software for future hardware. However, there is no guarantee for the programmer that the functionality specified by DirectX will ever be implemented in hardware. An additional drawback of this strategy is that technological innovation is restricted to implementing the features already specified by DirectX. Although this seems to be a severe limitation, critics must admit that DirectX currently is the force that ensures standardization throughout the fast evolution of hardware features that we are currently experiencing.

The implementation of the algorithms and ideas described in this thesis was done in OpenGL, as it was important to have a common code basis for the comparison of different approaches on multiple platforms and hardware architectures. The source code examples in this thesis are written in C++ using OpenGL 1.2 and its extensions. However, in most cases the algorithms can be adapted to the Direct3D API without major modification.

Apart from the general purpose hardware we have examined in the previous section, there are several special purpose volume rendering architectures, that may require special APIs, such as the *Volume Library Interface* [133] that ships with the popular VolumePro board. An overview of specialized hardware architectures for direct volume rendering is given in the following section.

2.3 Volume Rendering Hardware

Special purpose hardware for direct volume rendering has been proposed my various researchers. In [131] an overview of such hardware is presented, including a metric that allows the comparison of different designs by quantifying how efficiently data is being accessed and processed.

The *VIRIM* [57] architecture has been developed and assembled in 1994 at the University of Mannheim, Germany, to generate high-quality images of volume data sets of moderate size. *VIRIM* consists of a geometry unit for interpolation and gradient estimation and a ray casting unit. Including perspective projection and shadows, *VIRIM* is capable of displaying a volume of size $256^2 \times 128$ at a frame rate of 2.5 Hz. Multiple rendering modules can be combined to achieve interactive frame rates. This however requires the duplication of the volume data set.

In 1995, Michael Doggett introduced a hardware architecture [30, 29] for array-based ray casting that also contained a shading subsystem with a pipeline for gradient estimation. The architecture consists of a rotation array, responsible for rotating the data set and a ray array for intersecting rays with voxels. However, the system only supports parallel projection and nearest-neighbor interpolation. Simulations of this hardware design result in an estimated performance of about 15 Hz for a data set of size 256^3.

VIZARD-II [108] is the second generation of a volume rendering system developed at

the University of Tübingen, Germany. This architecture reduces memory bandwidth requirements by the use of data compression, pre-classification, pre-shading and a lookup table for quantized gradients. *VIZARD-II* consists of a control unit for intersection calculation, a memory unit, which stores the data set, an interpolation unit that performs trilinear interpolation and a shading/compositing unit that calculates illumination and classification via lookup tables. The architecture is expected to sustain a frame rate of 10 Hz in the average case.

The *Cube-4* [126] architecture developed at the State University of New York at Stony Brook in 1996 implements parallel ray-casting. Based on the original *Cube-4* architecture, EMCube[1] [124] has been implemented as a highly optimized system that improves memory and bandwidth limitations. This architecture finally lead to the most popular implementation of volume rendering hardware, the *VolumePro* [125] board for consumer PCs, that was commercially produced and distributed by *Mitsubishi Electric's Real Time Visualization Group*. The VolumePro architecture takes advantage of a factorization of the viewing matrix similar to the shear-warp algorithm (see Section 1.3.2.2). The main advantage of this architecture is that volume data can be read in planes of voxels. *VolumePro* has hardware for gradient estimation using central differences, classification via 36 bit lookup table (24 bit for RGB, 12 bit for A) and Phong illumination using a pre-computed reflectance map for the diffuse and specular term. Modulation of a voxel's opacity with its gradient magnitude can be additionally used in order to emphasize boundaries. With the *VolumePro* architecture a frame rate of 30 Hz has been achieved for a volume data set of size 256^3.

At the Graphics Hardware Workshop in 2000, Frank Dachille presented *GI-Cube* [24], a hardware architecture for acceleration of several ray-based sampling algorithms including global illumination. Harvey Ray also presented the *RACE II* Engine [132], that reduces the huge memory throughput required to render large data sets.

	VIRIM	**VIZARD-II**	**VolumePro**
Algorithm	object-order	image-order	hybrid order
Interpolation	programmable	trilinear	trilinear
Illumination	programmable	Phong	Phong
Projection	perspective	perspective	parallel
Performance	2.5 Hz (fps)	10 Hz (fps)	30 Hz (fps)

Table 2.2: Hardware architectures for volume rendering.

Most of the architectural designs described in this section have never been implemented in hardware. Exceptions are VIRIM, VIZARD II and VolumePro which are summarized in Table 2.2. Due to the considerable cost of special purpose hardware, its application is restricted to larger research facilities and hospitals. The approaches developed within the scope of this thesis, however, target consumer PC platforms and commodity laptop

[1] enhanced memory Cube-4

computers. The aim of this strategy is to make direct volume rendering available to a wide community.

Part II

Volume Rendering

Chapter 3

Texture Based Volume Rendering

In search for solutions of direct volume rendering that meet all the requirements specified in Section 1.3, it is necessary to analyze the computational problem in detail. As a first step, we put down a note that the efficiency of a volume rendering algorithms is mainly influenced by two independent factors.

- As pointed out in the introductory part, a significant contribution to the computational cost is caused by the huge **number of interpolation operations** per frame, that is required to resample the data along the viewing rays.

- The minimum size of a real data set is 256^3 voxels. Smaller data sets are purely of academic interest. CT data sets in common use have a dimension of 512^3. Data sets with a slice resolution of 1024^2 are not unusual. The scalar value of each voxel is represented by an 8 or 16 bit integer. As a result, a data set of size 512^3 requires 128 MB of main memory. With the increasing size of data sets the available **memory bandwidth** becomes a limiting factor, as in the worst case the complete data set must be accessed for generating one single frame.

The latter problem is subject of active research among hardware manufacturers as well as developers of hierarchical data structures and compression techniques. Apart from software approaches that exploit spatial coherence (see Section 1.3), the first problem is mainly attacked by exploiting the interpolation capabilities of the texturing subsystem available on current graphics hardware. In the following chapters we are going to examine these texture-based methods in detail.

3.1 The 2D-Texture Based Approach

Interpolation is performed within the rasterization unit of the graphics pipeline. However, the graphics pipeline only supports polygonal rendering primitives. Volumetric objects cannot be passed directly to the hardware as primitive objects. In consequence direct volume rendering requires a mechanism to decompose volumetric objects into planar polygons, which serve as a *proxy geometry*. There are several methods to adapt direct volume

rendering to the rasterization hardware, which differ mainly in the way this decomposition is computed. We will examine some of them within the scope of this thesis.

Nowadays, every consumer graphics board provides hardware support for 2D-texture mapping. The texturing subsystem of these boards allows bilinear interpolation of texture samples. Since interpolation contributes a significant part to the computational cost of volume rendering, it is desirable to exploit the available hardware features for acceleration. In comparison to 3D-textures with full support for trilinear interpolation (see Section 3.2), in this chapter we are going to examine a method that manages with 2D-textures and bilinear interpolation only. This approach represents a texture-based implementation of the concepts introduced with the *shear-warp algorithm* described in Section 1.3.2.2.

3.1.1 Principles

Analogous to the shear-warp algorithm, the volume object is decomposed into a stack of object aligned polygons with respect to the current viewing direction (see again Figure 1.10). For a 2D-texture based implementation, however, there is no need to explicitly decompose the viewing matrix into a shear and a warp step. The rasterization subsystem allows the polygonal slices to be rendered directly, as displayed in Figure 3.1.

The slicing direction is again chosen with respect to the minimal angle between the viewing ray and the slice normal. To allow a modification of the camera position as well as a transformation of the volume object, both the modeling and the viewing matrix must be taken into account. A code fragment for selecting the correct stack of slices in OpenGL is given in Listing 3.1. To compute the viewing direction relative to the volume object, the modelview matrix must be obtained from the current OpenGL state (line 4). This matrix represents the transformation from camera space into the local coordinate system of the volume. The rotation is extracted from the matrix (line 7) and the viewing direction (the

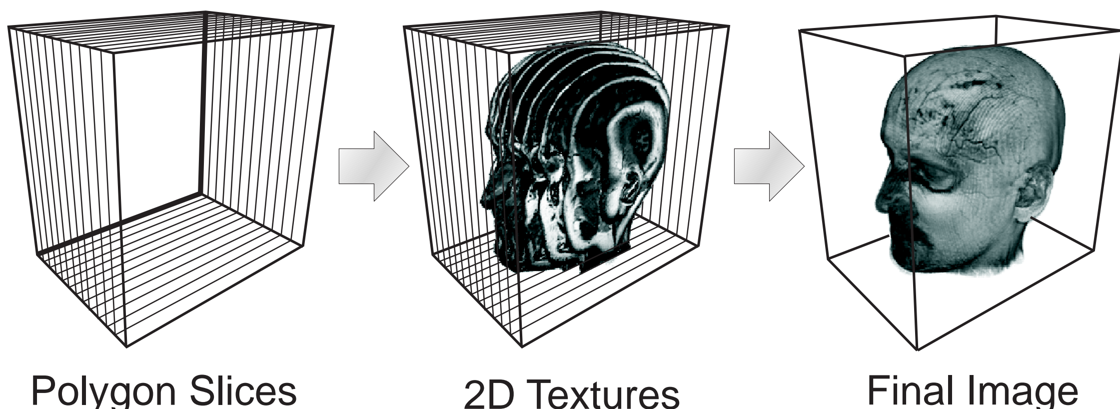


Figure 3.1: Decomposition of the volume object into object-aligned slices.

negative z-axis in OpenGL) is transformed. According to the maximum component of the transformed viewing vector (line 14), the appropriate stack of slices is chosen.

```
0   GLfloat pModelViewMatrix[16];
1   GLfloat pModelViewRotationMatrix[16];
2
3   // obtain the current modeling/viewing matrix from the OpenGL state
4   glGet(GL_MODELVIEW_MATRIX, pModelViewMatrix);
5
6   // extract the rotation from the matrix
7   GetRotation(pModelViewMatrix, pModelViewRotationMatrix);
8
9   // rotate the initial viewing direction
10  GLfloat pViewVector[3] = {0.0f, 0.0f, -1.0f};
11  MatVecMultiply(pModelViewRotationMatrix, pViewVector);
12
13  // find the maximal vector component
14  int nMax = FindAbsMaximum(pViewVector);
15
16  switch(nMax) {
17     case X:
18         if (pViewVector[X] > 0.0f) {
19             DrawSliceStack_PositiveX();
20         } else {
21             DrawSliceStack_NegativeX();
22         }
23         break;
24     case Y:
25         if (pViewVector[Y] > 0.0f) {
26             DrawSliceStack_PositiveY();
27         } else {
28             DrawSliceStack_NegativeY();
29         }
30         break;
31     case Z:
32         if (pViewVector[Z] > 0.0f) {
33             DrawSliceStack_PositiveZ();
34         } else {
35             DrawSliceStack_NegativeZ();
36         }
37         break;
38  }
39
```

Listing 3.1: OpenGL sample code for selecting the slice direction.

The selected stack of object aligned polygons is displayed by transforming each polygon with the active transformation matrices and by drawing it in back-to-front order. During rasterization, each polygon is textured with the image information directly obtained from its corresponding 2D-texture map. Bilinear interpolation within the texture image is accelerated by the texturing subsystem. As in the original shear-warp algorithm, the third interpolation step for a trilinear interpolation is completely omitted.

3.1.2 Compositing

According to the physical model described in Section 1.1, the equation of radiative transfer can be iteratively solved by discretization along the viewing ray, according to

$$I(s_k) \quad = \quad I(s_{k-1}) \cdot \vartheta_k \; + \; I_{\text{emit}}(s_k) \cdot (1 - \vartheta_k) \qquad \text{with} \qquad I_{\text{emit}}(s_k) = \frac{q(s_k)}{\kappa(s_k)}. \tag{3.1}$$

As an initial configuration, 2D-textures are used with an internal format of `RGBA`, which means that each texel allocates four fixed-point values, one value for the red (`R`), green (`G`) and blue (`B`) components respectively, plus one for the opacity (`A = Alpha`) value. For each voxel the emission coefficient I_{emit} is stored as color value (`RGB`) in the corresponding 2D-texture map. Additionally, the inverse absorption coefficient $(1 - \vartheta_k)$ is stored as opacity value `A`. Using this configuration, the radiance I resulting from an integration along a viewing ray can be approximated by the use of alpha blending.

As outlined in Section 2.1.3, blending allows the combination of the `RGBA` quadruplet of an incoming fragment *(source)* with the values already contained in the frame buffer *(destination)*. If blending is disabled, the destination value is replaced by the source value. With blending enabled, both the source and the destination `RGBA` quadruplets are combined to form a new destination value. In order to compute the iterative solution according to Equation 3.1, the inverse absorption coefficient $(1 - \vartheta_k)$ stored in the `Alpha` component of the texture map must be used as blending factor. The corresponding setup of the blending stage is displayed in Listing 3.2. For a color component $C \in \{R, G, B\}$, the described configuration results in a blending equation as follows:

$$C'_{\text{dest}} \quad = \quad C_{\text{src}} \cdot A_{\text{src}} \; + \; C_{\text{dest}} \; (1 - A_{\text{src}}). \tag{3.2}$$

As can be easily verified, substituting the emission and absorption coefficients

$$C_{\text{src}} \; = \; I_{\text{emit}} \qquad \text{and} \qquad A_{\text{src}} \; = \; (1 - \vartheta_k), \tag{3.3}$$

```
0   // enable blending
1   glEnable(GL_BLEND);
2   // setup blending equation
3   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Listing 3.2: OpenGL compositing setup for alpha blending in the per-fragment operations.

```
0   #ifdef GL_EXT_blend_minmax
1           // enable blending
2           glEnable(GL_BLEND);
3           // enable maximum selection
4           glBlendEquationEXT(GL_MAX_EXT);
5           // setup arguments for the blending equation
6           glBlendFunc(GL_SRC_COLOR,GL_DST_COLOR);
7   #endif
```

Listing 3.3: OpenGL compositing setup for maximum intensity projection in the per-fragment operations.

into this blending equation leads to the iterative solution of Equation 3.1.

3.1.2.1 Maximum Intensity Projection

As an alternative to solving the equation of radiative transfer, *maximum intensity projection* (MIP) is a common technique which does not require numerical integration at all. Instead, the color of a pixel in the final image is determined as the maximum of the all emission values sampled along the ray, according to

$$I = max\big(I_{\mathrm{emit}}(s_k)\big). \tag{3.4}$$

Unfortunately, the computation of a maximum in the alpha blending step is not possible with standard OpenGL operations. However, implementing MIP is a good example for the

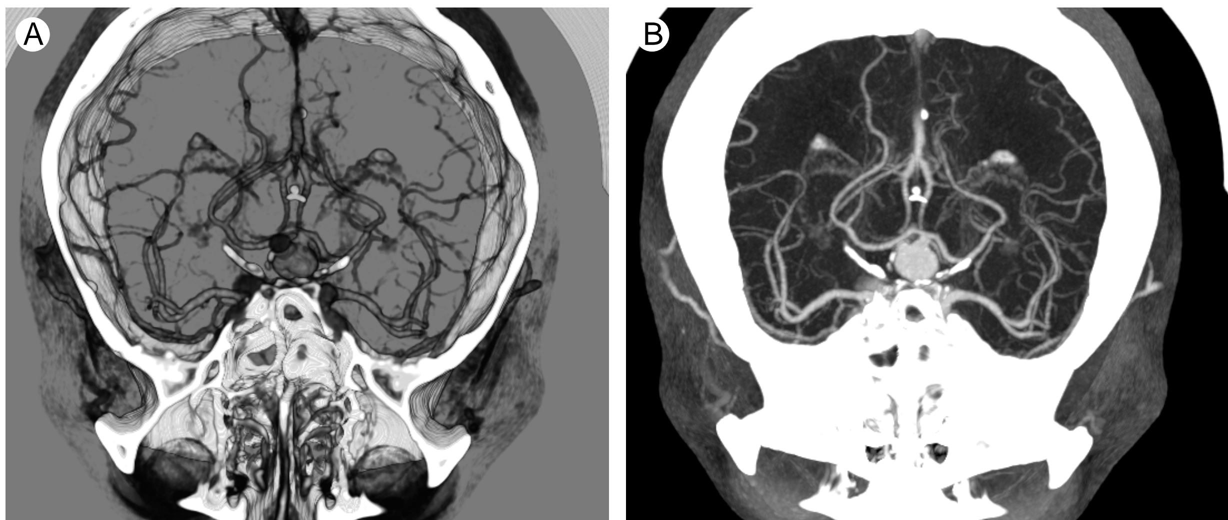


Figure 3.2: CT Angiography: A comparison between *alpha blending* (A) and *maximum intensity projection* (B).

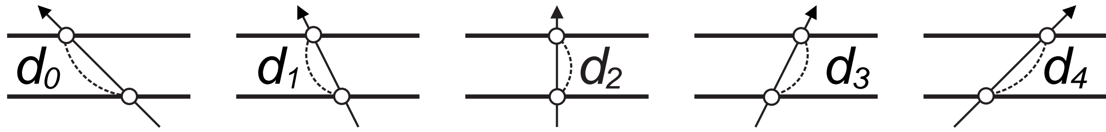


Figure 3.3: The distance between adjacent sampling points depends on the viewing angle.

use of an OpenGL extension. The extension `EXT_blend_minmax` introduces a new function `glBlendEquationEXT(...)` which enables maximum and minimum computation of the source and destination RGBA quadruplets. The OpenGL alpha blending setup for MIP is displayed in Listing 3.3.

Maximum intensity projection is mainly used for medical applications in order to visualize tomographic data with contrast dye of high signal, such as *angiography* data. A comparison of MIP and alpha blending is exemplified in Figure 3.2 by means of CTA[1] data of blood vessels inside the human head. While for alpha blending (A) a transfer function must be assigned to extract the vessels (see Chapter 5), the same vascular structures are immediately displayed in the MIP image (B). Note that in comparison to the alpha blended image the surface structure of the bone is not visible in the MIP image, since bone structures have the highest signal intensity in CT data. A major drawback of MIP is the fact that depth information is completely lost in the output images. This comes with a certain risk of misinterpreting the spatial correlation of different structures as impressively demonstrated in [59].

3.1.3 Discussion

The main benefit of the 2D-texture based algorithm presented in this chapter is that all necessary interpolation operations are performed within the graphics hardware. Since 2D-texturing capabilities belong to the standard feature set of current graphics boards, the approach can be implemented on almost every consumer PC. The rendering performance is high, provided that there is enough local video memory. Due to the high memory

[1]CTA = computed tomography angiography

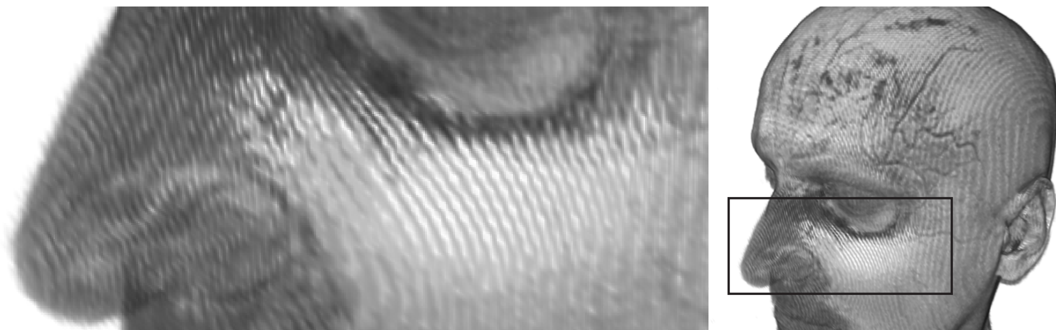


Figure 3.4: Aliasing artifacts become visible at the edges of the slice polygons.

requirements for storing the three texture stacks, the size of the data sets is limited by the available main memory.

The image quality is equivalent to a software implementation of the shear-warp algorithm, since the same computational mechanisms are applied. Magnification of the images often results in typical aliasing artifacts, as displayed in Figure 3.4. These artifacts become visible at the edges of the slice polygons and are caused by an insufficient sampling rate. Since the fixed number of slices is one of the main characteristics of the algorithm, these artifacts cannot be removed without significantly changing the algorithm.

In order to improve the image quality of this method, it is necessary to examine how numerical integration is performed in this case. Let us reconsider the physical model described in Section 1.1. The emission and absorption coefficients q_0/κ_0 and ϑ_k, which are stored in the 2D-textures are only valid if we assume a fixed distance between the sampling points for the numerical integration step. However, this is not true for the described algorithm, since the distance between adjacent sampling points depends on the viewing angle (see Figure 3.3). Thus, the result of the numerical integration is only exact for parallel projection and for one particular viewing direction. Considering perspective projection, the angle between the viewing ray and a slice polygon is not even constant within one image.

However, throughout the experiments with different approaches, we have observed that this lack of accuracy is hardly visible and thus does not explain the strong aliasing artifacts of Figure 3.4. These artifacts in turn can be successfully removed by inserting multiple intermediate slices, so it becomes evident that we need a mechanism for increasing the sampling rate in order to enhance the image quality. This idea is followed by the multi-texture based approach introduced in Chapter 4.

In addition to these sampling artifacts, disturbing visible effects may also occur when the algorithm switches between different stacks of polygon slices. The reason for such effects is an abrupt shift of the sampling positions. Figure 3.5 illustrates this problem. Figures (A) and (B) show the viewing direction at which the slicing direction is ambiguous. If we

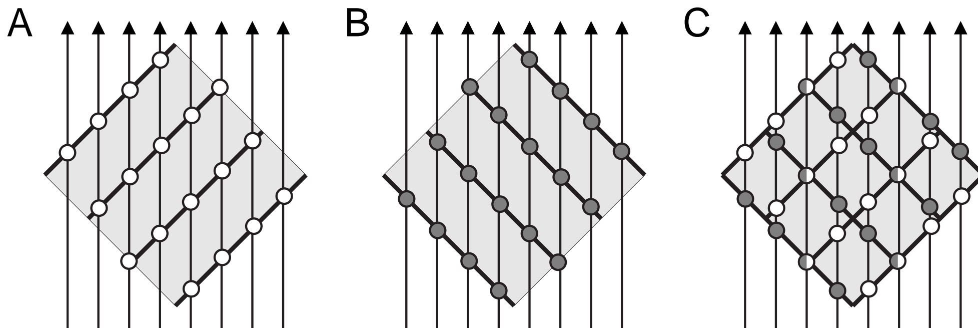


Figure 3.5: Sampling artifacts are caused by changing between different slice stacks (A) and (B). The superposition (C) shows that the location of the sampling points abruptly changes, which results in visible switching effects.

2D-Texture-Based Approach	
Pros	Cons
⊕ very high performance ⊕ high availability	⊖ high memory requirements ⊖ bilinear interpolation only ⊖ sampling artifacts ⊖ switching effects ⊖ inconsistent sampling rate

Table 3.1: Summary of 2D-texture based volume rendering.

examine the location of the sampling points by superimposing both configurations (C), it becomes clear that the actual position of the sampling points changes abruptly, although the sampling rate remains the same. According to the sampling theorem, the exact position of the sampling points should not have any influence on the reconstructed signal, however, this assumes an ideal reconstruction filter as outlined in Section 1.2 and not a linear interpolation. These shifting effects are clearly visible and can only be alleviated by the use of 3D-textures as will be explained in the following section. A summary of the advantages and the drawbacks of 2D-texture based method is presented in Table 3.1.

3.2 The 3D-Texture Based Approach

Several problems of the 2D-texture based approach described in the previous chapter are caused by the fixed number of slices and their static alignment within the object's coordinate system. The reason why we had to put up with these restrictions was that the hardware did not provide the trilinear interpolation capabilities required for volume rendering. With hardware support of three-dimensional textures [181, 15] the situation is completely different.

The support of 3D-textures does not remove the necessity of decomposing the volume object into planar polygons. Although volumetric texture objects are now available in hardware, the graphics pipeline still does not support volumetric rendering primitives. However, compared to the 2D-texture based approach, we now have greater flexibility on how to compute the decomposition. As we have seen in the previous section, one drawback of using object-aligned slices is the inconsistent sampling rate that results from the static proxy geometry. Since 3D-textures allow the slice polygons to be positioned arbitrarily, a consistent sampling rate for different viewing directions could be achieved by adapting the distance of the object aligned slices to the current viewing angle. This is actually done in the 2D-multi-texture based approach described in Chapter 4. Adjusting the slice distance, however, does not remove the shifting artifacts that occur when the algorithm switches between different slice stacks.

Both problems are efficiently solved by the use of viewport aligned slices as displayed in Figure 3.6. The proxy geometry, however, must be recomputed whenever the viewing

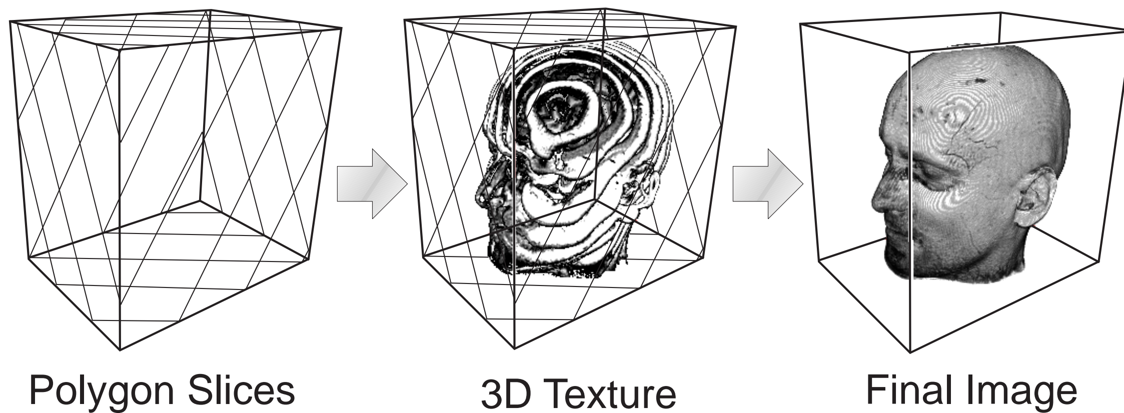


Figure 3.6: Decomposition of the volume object into viewport-aligned polygon slices.

direction changes. In case of parallel projection, the decomposition into viewport-aligned slices ensures a consistent sampling rate for all viewing rays as illustrated in Figure 3.7 A. In the perspective case, the sampling rate is still not consistent for all rays (Figure 3.7 B). The distance of sampling points varies with the angle between the slice polygon and the viewing ray. Depending on the field-of-view angle, these effects are more or less noticeable.

The compositing process in case of 3D-texture based volume rendering is exactly the same as for the 2D-texture based algorithm described in Section 3.1.2. The intersection calculation for viewport-aligned slices algorithm however requires a more detailed description.

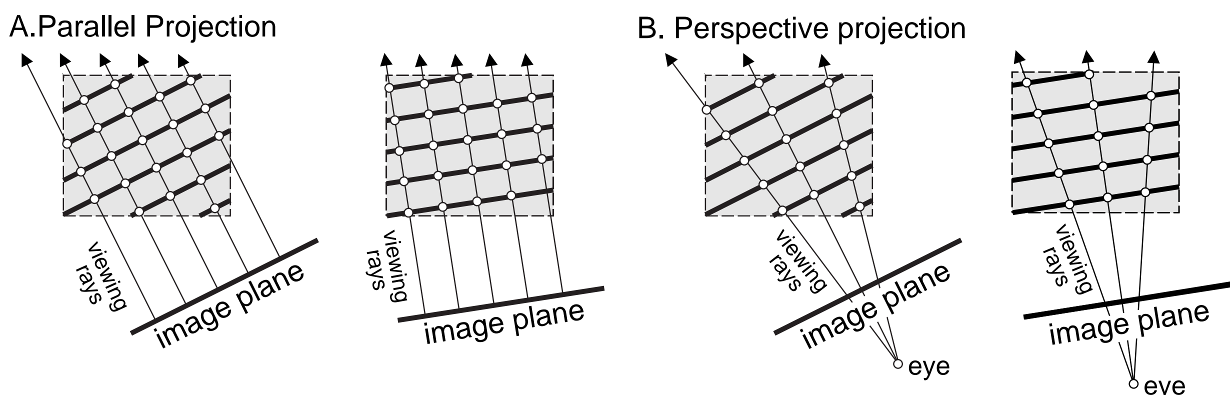


Figure 3.7: Sampling illustrated for viewport-aligned slices in the case of parallel (A) and perspective projection (B).

3.2.1 Viewport-Aligned Slices

In comparison to the object-aligned slices used in the previous chapter, the procedure of intersection calculation between the bounding box and a stack of viewport-aligned slices is of much higher computational complexity. To make matters worse, these slice polygons must be recomputed whenever the viewing direction changes. Since the whole computation must be performed several times per second to achieve an interactive frame rate, a very efficient algorithm is required. The slicing method used in our implementation is based on a sequence of three steps:

1. Compute the intersection points between the slicing plane and the straight lines that represent the edges of the bounding box.

2. Eliminate double and invalid intersection points.

3. Sort the remaining intersection points to form a closed polygon.

The intersection between a plane and a straight line in step 1 can easily be solved analytically. Almost any higher-level API provides some functions which perform such a calculation very efficiently. To determine whether an intersection point actually lies on an edge of the bounding box, a simple bounding sphere test is applied in step 2. Points that are located outside the bounding sphere do not lie on an edge and are thus discarded from the list of valid intersection points. Additionally, double points which coincide with a corner vertex of the bounding box are merged together.

In order to facilitate the sorting of the remaining edge intersection points in step 3, a set of six flags is stored for each edge, one flag for each of the six faces of the bounding box. As outlined in Figure 3.8 a flag is set if the edge belongs to the corresponding face and cleared otherwise. The sequence of intersection points that form a valid polygon is found when the flags of two adjacent edge intersection points have one flag in common.

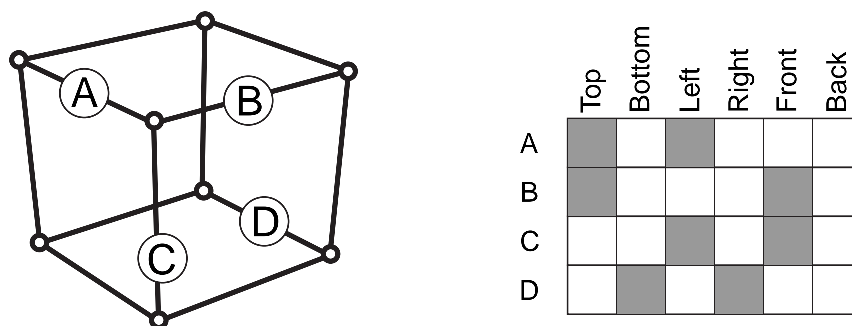


Figure 3.8: Sorting of edge intersection points to form a valid polygon: Each edge holds a set of 6 flags (left), one flag for each face of the bounding box. A flag is set if the edge belongs to the corresponding face and cleared otherwise. Edges that share a common face are easily determined by ORing the edge flags. If the result is nonzero, a common face exists. Four examples for edge flags are displayed (A-D).

This property can be easily verified by computing a bitwise OR operation of the edge flags. If the result is nonzero for every pair of adjacent points, the sequence is valid and the resulting polygon is exactly the cross-section between the plane and the bounding box. Further optimization of the slicing algorithm can be achieved by computing the intersection points for the subsequent slice plane incrementally.

3.2.2 Bricking

Compared to the previous approach using 2D-textures, the memory management for a 3D-texture is more difficult. Since the whole volume is defined as a single 3D-texture, it must entirely fit into the texture memory at one time. With the increasing size of volume data sets the available texture memory becomes the limiting factor. A popular idea to tackle this problem is to subdivide a large data set into smaller chunks (usually called *bricks*) that entirely fit into local video memory, one at a time.

The naive approach of simply subdividing the data set into bricks and rendering each brick separately will introduce additional artifacts at the brick boundaries. To explain these artifacts, we have to look at how texture interpolation is performed at the transition between neighboring bricks. Figure 3.9 (left) illustrates this problem. At the boundary between texture tiles the interpolation is incorrect, since the texture unit does not have enough information to consistently interpolate across texture boundaries. The solution to this problem arrives by duplicating a plane of voxels at each brick boundary. If two neighboring bricks share a common plane of voxels, the texture units can be set up to deliver the correct interpolation results, as displayed in Figure 3.9 (right).

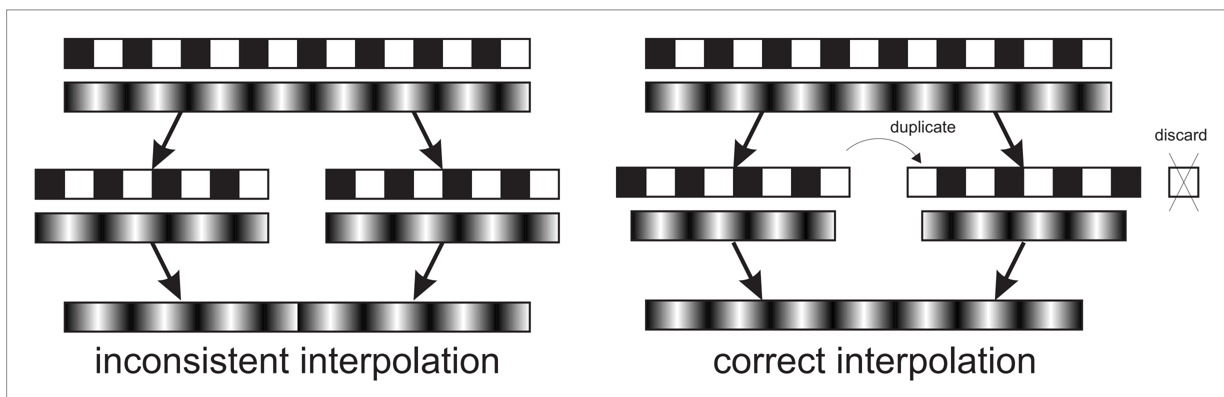


Figure 3.9: Bricking illustrated for the 1D case: Simply splitting the texture leads to inconsistent interpolation at the transition (left). Duplicating a voxel at the boundary between bricks (a plane of voxels in 3D) leads to correct interpolation results (right).

3.2.3 Discussion

In comparison to the 2D-texture based solution, the 3D-texture based approach has proven superior in terms of image quality, removing some of the significant drawbacks while preserving almost all the benefits. Hardware support for trilinear interpolation provides us with a natural means of increasing the sampling rate to obtain images of high quality. Adjusting the sampling rate is especially important to accurately account for a transfer function of high frequency as will be explained in Chapter 5. For large volume data sets the bricking strategy however turns out to be extremely inefficient. As a result, the frame rate for large data will be significantly lower compared to 2D-textures. We will analyze the performance of all algorithm in Chapter 8.

The 2D-texture based approach requires three copies of the data set to be stored in local memory. With 3D-textures this is no longer necessary, since trilinear interpolation allows the extraction of slices with arbitrary orientation. In this context, viewport-aligned slices guarantee a sampling distance which is consistent among adjacent frames for parallel projection. The problem of variable sample rate for perspective projection, however, still remains. As long as the virtual camera views the volume object from an exterior position, the effect of the inconsistent sampling rate is hardly visible. For virtual fly-through applications the inaccurate sampling rate, however, is clearly noticeable as disturbing visual artifacts. Since in this case the field-of-view angle is relatively large, the presented approach is less applicable for navigation within the volume. Constant sampling rate for perspective projection can only be achieved by the use of spherical rendering primitives instead of slices [92]. This would in turn significantly degrade the performance due to the complex intersection calculation and the required tessellation of the spherical sections. As an aside, an interesting algorithm especially designed for navigation within the volume has been proposed in [12]. It is based on software ray casting accelerated by 2D-texture mapping.

The advantages and drawbacks of the 3D-texture based method are again summarized in Table 3.2. The applicability of the approach described in this chapter greatly depends on the availability of graphics boards with hardware support for 3D-textures. On the consumer market there are currently only a few boards available which meet these requirements. 3D-textures now belong to the standard feature set of DirectX 8.0 compliant boards. In OpenGL 3D-textures have been available for quite a long time as an extension

3D-Texture-Based Approach	
Pros	Cons
⊕ high performance ⊕ trilinear interpolation	⊖ availability still limited ⊖ inefficient memory management ⊖ inconsistent sampling rate for perspective projection

Table 3.2: Summary of 3D-texture based volume rendering.

(`EXT_texture_3D`) provided for example by SGI's high-end workstations. They have been recently included into the OpenGL 1.2 standard. The first consumer product with support for 3D-textures was the Radeon GPU released by *ATI* in the spring of 2000. A hardware implementation of 3D-textures has also been provided by *NVidia* with the final release of the GeForce 3 GPU.

Chapter 4

2D-Multi-Texture Based Methods

Up until now we have seen two alternative approaches for texture based volume rendering. The 2D-texture based method was capable of rendering a volume data set at high frame rate. The mathematical accuracy, the subjective image quality and the memory requirements however were still far from being optimal. It is true that the 3D-texture based approach removes some of these limitations, the bricking strategy, however, is less suitable for large volume data sets. Let us make a note on the main advantages of 3D-texture based volume rendering:

- trilinear instead of bilinear interpolation,
- constant instead of variable sampling rate among adjacent frames for ray integration in case of parallel projection,
- lower memory requirements by removing the necessity to duplicate the volume data in memory.

On the consumer market the availability of 3D-textures is still restricted to a very limited number of hardware implementations. In this chapter we will examine a third alternative approach, which supplements the original 2D-texture based approach by removing at least two of the above mentioned limitations while preserving the benefit of more efficient memory management.

4.1 Rasterization Revisited

Although 3D-textures are not yet widely supported on consumer graphics boards, there are numerous boards which provide very flexible rasterization and texturing units with a high degree of programmability. The idea of the 2D-multi-texture based approach is to leverage these new features in order to enhance the accuracy and the image quality of the 2D-texture based approach, while preserving the benefit of efficient memory management. Before we have a look at this idea, it is necessary to get accustomed to the way multi-textures are used to achieve particular texturing effects.

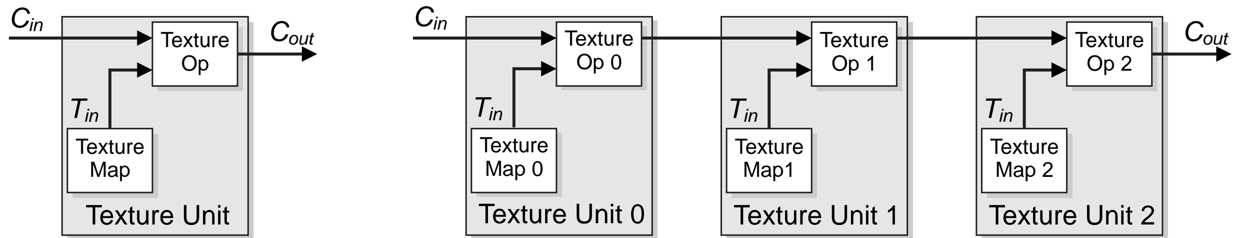


Figure 4.1: Multi-textures can be implemented by duplicating the conventional texture unit *(left)*. A number of such units are arranged in sequence, forming a static pipeline *(right)*.

4.1.1 Multi-Textures

As we have seen in Section 2.1.2, a 2D-texture is a raster image (or bitmap) that is mapped onto a planar polygon. The color information for each fragment is then determined as a combination of the polygons's primary color and the color value of the texture sample obtained from the texture map (see Figure 4.1 *left*). In OpenGL a texture value for example can be used to modulate the primary color, denoted

$$C_{\text{out}} = C_{\text{in}} \cdot T_{\text{in}}. \tag{4.1}$$

C_{in} and C_{out} are the color values of the incoming fragment before and after texture application respectively. T_{in} is the color value of the interpolated texture sample. It goes without saying that the equation is calculated component wise for each color channel.

With this concept of a texture unit, it is easy to build a system that support multi-textures as a strict pipeline by simply connecting multiple units in series (Figure 4.1 *right*). The final fragment color in this case amounts to

$$C_{\text{out}} = T_{\text{in}}^{(n)} \cdot (\, T_{\text{in}}^{(n-1)} \cdot (\quad \ldots \quad \cdot (\, T_{\text{in}}^{0} \cdot C_{\text{in}}))). \tag{4.2}$$

The input for each individual texturing unit is determined by the output value of the previous unit. A typical application example for multi-texturing is found in 3D game programming [171]. The traditional texture map, which contains the object's color, is supplemented by two multi-textures, which store the pre-computed illumination (light map) and fog density (fog map). The benefit of such a technique is obvious. While the color texture is constant, the reflection map must be updated by the dynamic light sources and the fog map is dependent on the camera position. On the other hand the color texture will be highly detailed, while for the light and the fog map low resolution might be sufficient. Keeping all these components in separate textures allows illumination and fog density to be updated interactively while preserving high resolution for the color texture.

The strict pipeline of texturing units provided by the OpenGL specification, however, turns out to be too limiting for many desired texturing effects. Per-pixel illumination, normal maps and bump maps require a more flexible texture combination than the one given in Equation 4.2. For this purpose programmable rasterization and texturing units have

been developed. These hardware features are referred to as *pixel shaders*. Unfortunately there is no standardized way to access pixel shader hardware in OpenGL yet. The two major manufacturers of 3D graphics boards NVidia and ATI both have proposed different OpenGL extensions. In general, OpenGL source code written for NVidia's GeForce family boards must be modified and adapted to support graphics boards from ATI and vice versa. The expenditure of porting the code to different OpenGL extensions for similar hardware, however, is unnecessarily high. A *shading language* that can be used to access the hardware in a consistent way has been developed at Stanford [129] in combination with the corresponding compiler [102]. A similar concept of a shading language has been recently included in the proposal for the new OpenGL 2.0 standard. In the following section the principles of pixel shaders will be exemplified on the basis of NVidia's register combiners. ATI's architecture is based on similar principles, although in the ATI's OpenGL extension the explicit register setup if hidden from the programmer.

4.1.2 Pixel Shaders

Traditional rasterization hardware computes color and illumination only at the polygon vertices and interpolates the resulting values in the interior of the polygon (*Gouraud Shading* [55]). The term *pixel shader* denotes a mechanism to perform such computations on a per-pixel basis[1]. This idea is very similar to *Phong Shading* [127], although not exactly the same. In Phong's shading model the normal vector is interpolated instead of the final color. Pixel shaders interpolate the color as well as the normal vector and the illumination terms and allow flexible combination of the interpolated values on a per-fragment basis.

An implementation of pixel shaders modifies the texture generation unit as well as the texture application unit (see Section 2.1.2). There are separate OpenGL extensions

[1] more precisely on a per-fragment basis

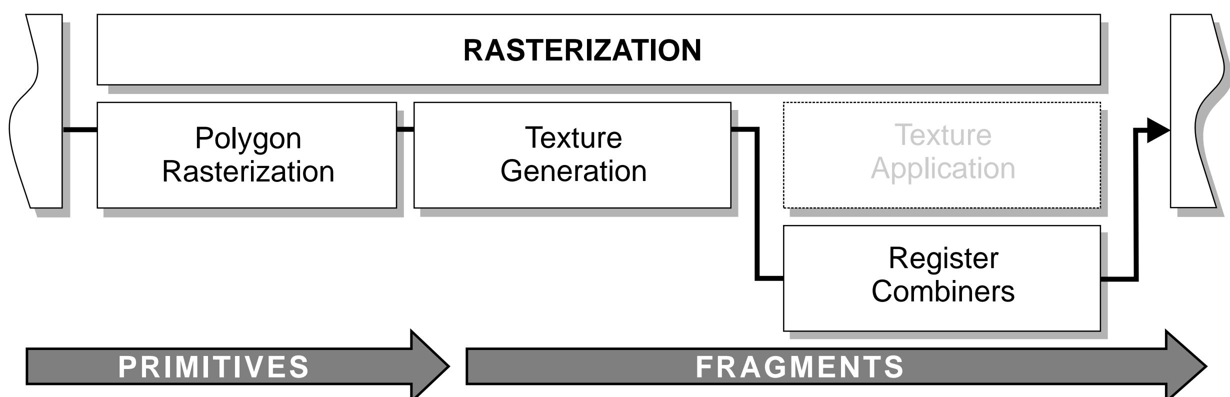


Figure 4.2: As traditional texture application turns out to be not flexible enough, NVidia's GeForce family GPUs provide register combiners that completely bypass the standard unit for texture environment application.

to access both parts of the implementation. The modification to the texture generation unit supports texture *coordinate* specification on a per-fragment basis. This mechanism is called *dependent texturing* and will be discussed in Chapter 5. NVidia's modified texture application unit is known as *register combiners* and the corresponding OpenGL extension

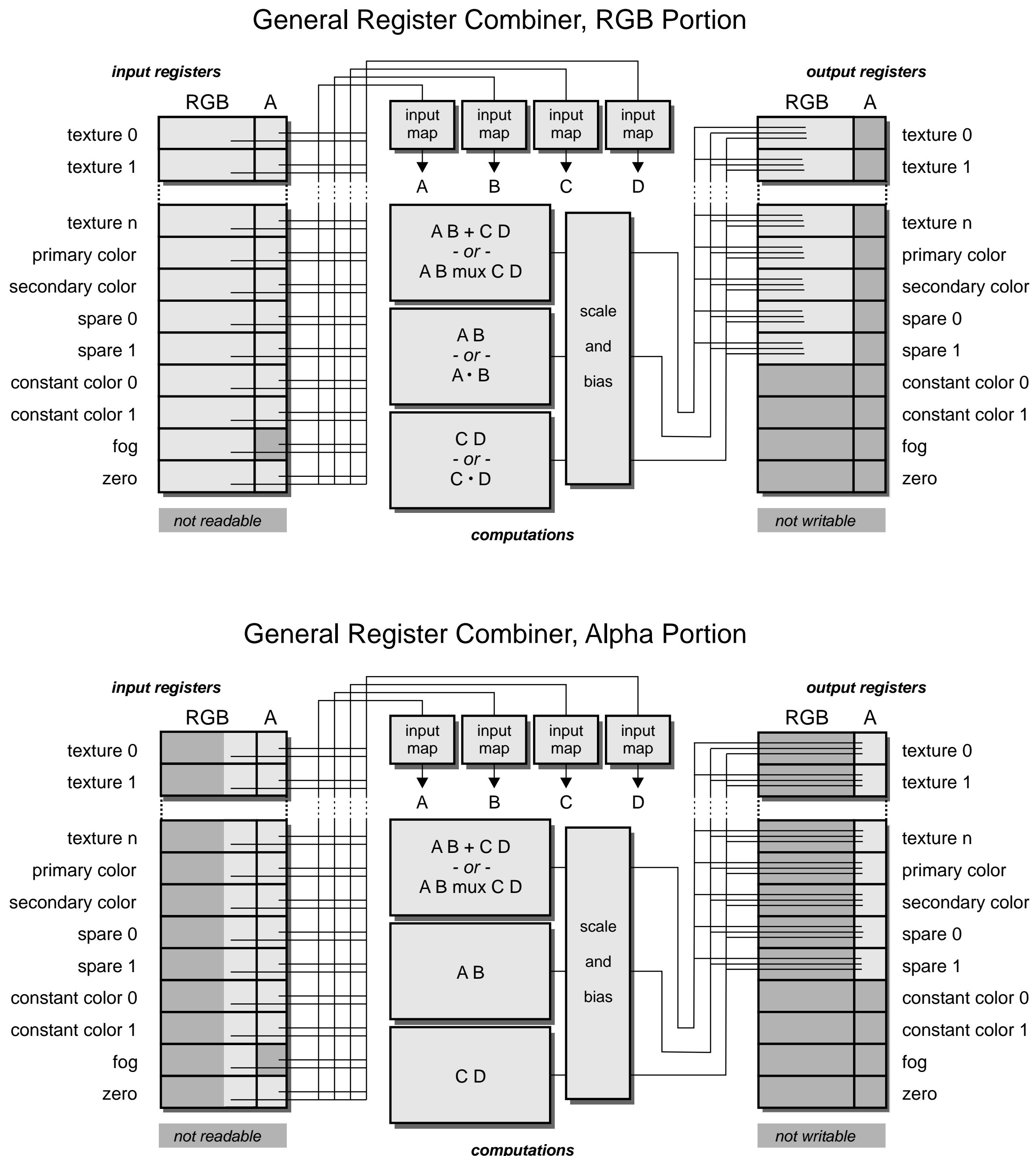


Figure 4.3: NVidia's register combiners: RGBA (top) and Alpha portion (bottom) of the general combiner.

is named NV_register_combiners.

4.1.2.1 NVidia Register Combiners

To provide the programmer with a mechanism to explicitly control the per-fragment information, NVidia has introduced the OpenGL extension NV_register_combiners. With this extension enabled, the standard OpenGL texturing application unit is completely bypassed and substituted by a register-based rasterization unit (see Figure 4.2). This rasterization mechanism is implemented as a pipeline consisting of two or more *general* combiner stages arranged in a fixed sequence, followed by a single *final* combiner stage, which generates the output. A general combiner stage is again divided into an RGB portion, which computes the fragment's color and an Alpha portion, which calculates the opacity value. The name *register combiner* is derived from the idea to store per-fragment information in a set of input registers (see Figure 4.3). The contents of these registers can be arbitrarily mapped to the four variables A, B, C and D. Multiple input mappings are supported, e.g. signed or unsigned inversion and expansion of unsigned range $[0, 1]$ to signed range $[-1, 1]$. Using the assigned input variables, the RGB portion of the general combiner stage performs three different computations in parallel:

1. Either a component-wise weighted sum $(AB + CD)$ or a conditional component-wise

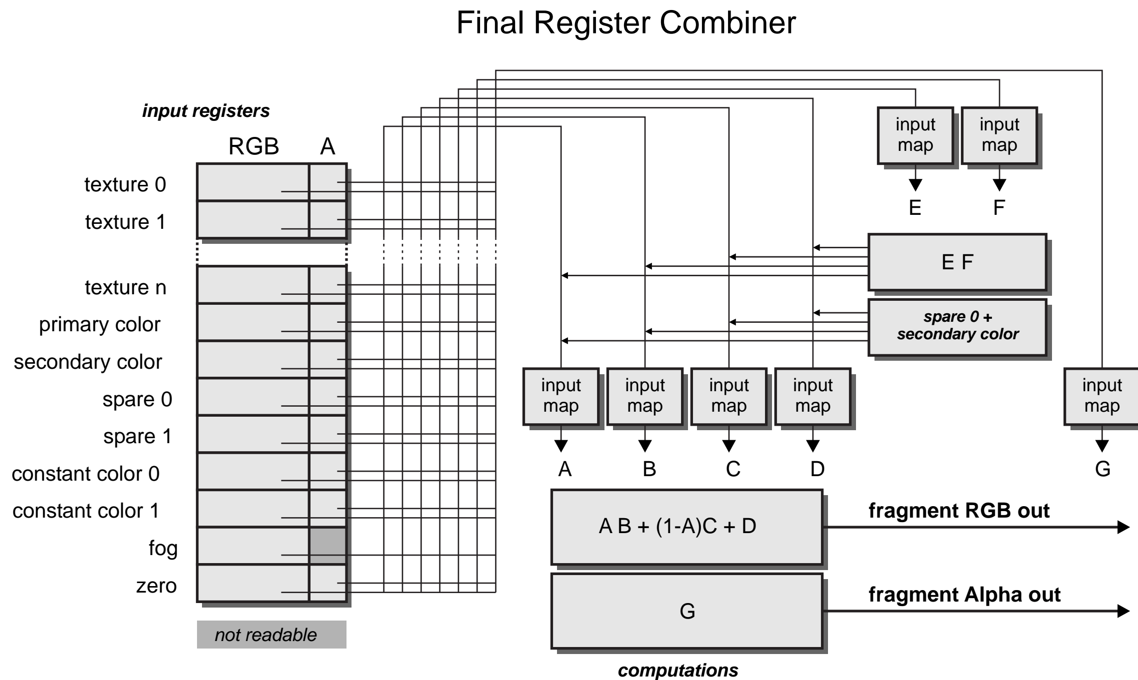


Figure 4.4: The final register combiner computes the final color of the fragment which is subsequently forwarded to fragment processing.

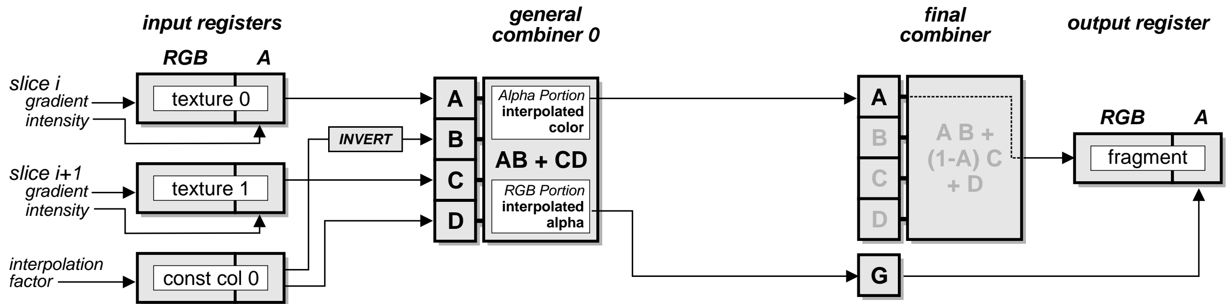


Figure 4.5: Register combiner setup for interpolation of intermediate slices.

product (AB mux CD)[2].

2. Either a component-wise product (AB) or a 3D vector dot product ($A \bullet B$).

3. Either another component-wise product (CD) or another dot product ($C \bullet D$).

The results of each of these three parallel computations are scaled and biased independently and written to arbitrary output registers. Depending on the underlying hardware two or more of these general combiner stages are connected in series. The input values of each stage are copied from the output registers of the previous stage. At the end of the pipeline the output values of the last general combiner stage are forwarded to the final register combiner. As displayed in Figure 4.4, this final stage generates two output values (RGB and Alpha). Arbitrary input registers are again assigned to the variables A–G. The final RGB value is determined by the fixed equation $AB + (1 - A)\,C + D$. The final Alpha value is simply copied from variable G. Additionally an intermediate component-wise product (EF) can be computed and assigned to one of the variables A–D. After the final register combiner the per-fragment operations are performed as usual. For more details on this extremely flexible rasterization unit, see the OpenGL extension specification provided by NVidia [120]. Register combiners are currently supported by the NVidia GeForce 256 and GeForce2 GTS chips with two general combiner stages and by the NVidia GeForce3 GPU with a sequence of up to eight general combiner stages.

4.2 Principles

The advantage of 3D-textures over 2D-textures is that trilinear interpolation is directly supported by the graphics hardware. Trilinear interpolation of a sample value I can be formulated as a function of the data values I_{ijk} given at the eight surrounding grid points, denoted

$$I \;=\; \sum_{i,j,k \in \{0,1\}} w_{ijk}\, I_{ijk}; \quad \text{with} \quad \sum_{i,j,k \in \{0,1\}} w_{ijk} = 1. \tag{4.3}$$

[2] see [120] for a description of the *mux* operation

The interpolation weights w_{ijk} are determined by the position of the sampling point in relation to the rectilinear grid. Trilinear interpolation can be easily decomposed into two bilinear interpolation operations,

$$I^{(0)} = \sum_{i,j \in \{0,1\}} a^0_{ij} \, I_{ij0} \quad \text{and} \tag{4.4}$$

$$I^{(1)} = \sum_{i,j \in \{0,1\}} a^1_{ij} \, I_{ij1}, \tag{4.5}$$

and one linear interpolation

$$I = (1-b) \cdot I^{(0)} + b \cdot I^{(1)}. \tag{4.6}$$

The new interpolation weights a_{ij} and b can be easily derived from Equation 4.3. Bilinear interpolation is efficiently performed by the 2D-texture unit. The idea to accomplish trilinear interpolation with 2D-multi-textures is simply to compute the missing linear interpolation step directly within the texture application unit. In comparison to the 2D-texture based approach, this idea will allow intermediate slices to be interpolated on the fly.

Computing an intermediate slice $S_{i+\alpha}$ can be described as a blending operation of two adjacent fixed slices S_i and S_{i+1}:

$$S_{i+\alpha} = (1-\alpha) \cdot S_i + \alpha \cdot S_{i+1} \qquad \text{with} \qquad \alpha \in]\,0,1\,[. \tag{4.7}$$

As each slice image is stored in a separate 2D-texture, bilinear interpolation is automatically performed by the texture generation unit. The third interpolation step is computed subsequently by blending the resulting two texels during texture application. As displayed in Figure 4.5, the blending step can be computed by a single general combiner stage, if the fixed slices S_i and S_{i+1} are specified as `texture0` and `texture1` using the multi-texture extension. The combiner is set up to compute a component-wise weighted sum $AB + CD$ with the interpolation factor α stored in one of the constant color registers. The contents of this register is then mapped to input variable A and at the same time inverted and assigned to variable C. In the RGB-portion, variables B and D are assigned the RGB components of `texture0` and `texture1` respectively. Analogously, the Alpha-portion interpolates between the alpha-components. For rendering semi-transparent volumes, the output of this first combiner stage is directly used for back-to-front alpha blending in the fragment processing unit without any further modification by the final combiner stage. Since multi-texture interpolation and combination is performed within one clock cycle of the graphics CPU, an intermediate slice is rendered at almost the same performance as a fixed single-textured slice. Of course multiple intermediate slices can be inserted this way without any additional memory allocation.

4.3 Interpolation of Arbitrary Slices

In addition to direct volume rendering, many applications require the interpolation of slice images from the volume data set in arbitrary direction. In medicine this procedure is usually referred to as multi-planar reformatting (MPR).

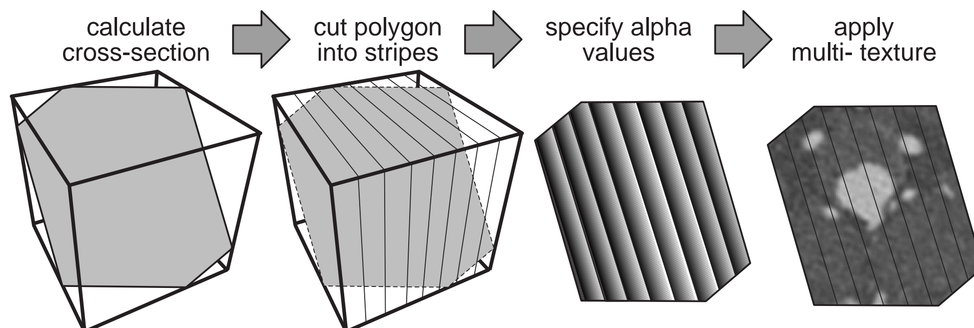


Figure 4.6: Rendering procedure for interpolating slice images in arbitrary direction.

An interesting 2D-texture based technique to extract arbitrary slices was introduced in [36] and is easily adapted to multi-texturing hardware. The basic idea of this algorithm is displayed in Figure 4.6. At first the cross-section of the slicing plane with the bounding box of the volume is calculated. The resulting intersection polygon is then cut into a set of polygon strips at the intersection line with the object-aligned texture slices. Subsequently for each of these polygon strips the image information is obtained by interpolating the two adjacent texture images. This is achieved by specification of alpha values for the polygon vertices. In this case an alpha value of 0 indicates that the corresponding vertex should be textured with the image information from the first texture. Accordingly, if a value of 1 is specified the second texture image is applied. Within the polygon, Gouraud shading is used to interpolate the alpha values. The interpolation between the two texture images is finally performed by the register combiners as displayed in Figure 4.7. In this setup, general combiner 0 is programmed to blend both textures (mapped to variables A and C) using the primary color alpha (mapped to variable B and inverted to variable D). As mentioned above, primary alpha is interpolated between the values specified at the vertices.

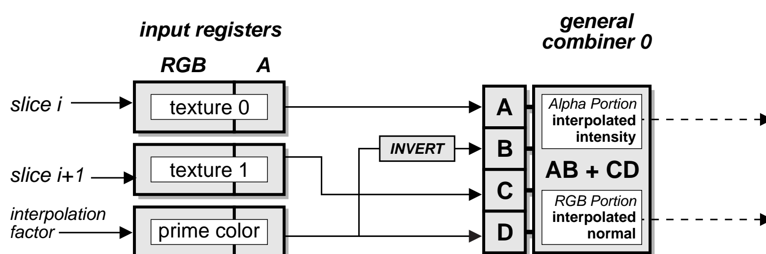


Figure 4.7: Register combiner setup for interpolation of arbitrary slice images.

4.4 Discussion

The 2D-multi-texture based approach fills the gap between the traditional 2D- and 3D-texture based methods. With the possibility to trilinearly interpolate intermediate slices

within the rasterization hardware, two drawbacks of the traditional 2D-texture based approach have been removed as promised at the beginning of this chapter.

- Trilinear interpolation can performed by multi-texture blending. This allows the extraction of axis-aligned slices at arbitrary positions.
- Consistent sampling rate in parallel projection can be achieved by adjusting the distance between the slice images to the viewing angle.

The third critical point was the high memory footprint that arises from the need to keep three copies of the data set in memory. With the approach to interpolating arbitrary slices discussed in Section 4.3, this drawback can also be removed, however at the expense of an increased CPU load. Although the described technique is usually applied to interpolate single slice images, it is potentially applicable for volume rendering with viewport-aligned slices. The significant computational overhead for intersection calculation in combination with the large number of texture binding operations results in a poor rendering performance. Using viewport aligned slices only 5—8 frames per seconds were achieved for a very small data set of size 64^3 on a GeForce 256. The interpolation of axis aligned slices, however, is also possible with this approach and can be efficiently pre-computed. Since a slice image which is exactly orthogonal to the stored slice stack is tessellated into stripes that have the width of a single texel, visual artifacts will occur, if multiple texels are mapped to a single pixel. As this will be the case especially for large data sets, this idea becomes less attractive.

Besides the different memory requirement, there are some considerable differences between the 3D-texture and the 2D-multi-texture based approaches:

- The switching artifacts that have been observed when the 2D-texture based algorithm switches between orthogonal slice stacks are still evident in the 2D-multi-texture based method. However due to the ability to adapt the slice distance arbitrarily, the effect appears less disturbing.
- The multi-texture based interpolation allocates two texturing units to interpolate one slice. These texturing units cannot be used for classification and illumination calculations, as we will see in the following chapters.

2D-Multi-Texture Approach	
Pros	Cons
⊕ high performance ⊕ trilinear interpolation ⊕ available on low cost hardware ⊕ efficient memory management	⊖ switching effects ⊖ inconsistent sampling rate for perspective projection [⊖ high memory requirements]

Table 4.1: Summary of 2D-multi-texture based volume rendering.

- The main advantage of the 2D-multi-texture approach over 3D-textures is the more efficient memory management. The bricking mechanism which must be applied for volumes that do not fit entirely into texture memory is extremely inefficient, since a huge part of the video memory must be swapped at a time. Using 2D-textures to represent the volume is advantageous, since the video memory is partitioned into small portions which can be replaced more efficiently.

As all described texture based approaches decompose the volume into planar slice images, the integration along the rays is not accurate in case of perspective projection, because the variable distance between sampling points is not taken into account. The advantages and drawbacks of the 2D-multi-texture based approach are summarized in Table 4.1. Refer to Chapter 8 for a detailed comparison of the performance of the different solutions. Successful application of the techniques described in this chapter have been reported in [138] for the visualization of medical image data in general. An adaptation of this approach in an augmented reality application for computer-assisted intervention was described in [147].

Up until now we have tacitly assumed that the texture images we used already contained the emission and absorption coefficients required for the ray integration. In the following chapter, we will have a closer look at how these coefficients are generated from the original sampling values by means of a transfer function.

Chapter 5

Transfer Functions

The physically based integration of radiant energy along viewing rays as explained in Section 1.1 requires the specification of emission and absorption coefficients. However, given a volumetric data set as they arise from measurement or simulation, there is no natural way to obtain these emission and absorption coefficients. In practise the user assigns emission and absorption coefficients to sample values v in an arbitrary way. This assignment process is referred to as *classification* and can be described by means of transfer functions

$$I_{\text{emit}} = T_{\text{emission}}(v) \qquad \text{and} \qquad \vartheta_k = T_{\text{absorption}}(v). \tag{5.1}$$

Although a few approaches have been developed to automatically generate transfer functions by some image- or data-driven mechanisms (see Chapter 6), the design of a transfer function in general is a manual, tedious and time-consuming procedure, which requires detailed knowledge of the spatial structures that are represented by the data set. In order to facilitate this classification process, it is crucial for the user to be provided with some direct visual feedback of his operations. In consequence, a mechanism is required that allows the transfer functions to be modified interactively while rendering the volume.

5.1 Principles

Although analytic continuous functions are thinkable, in practise the transfer function is realized as a lookup table of fixed size. The emitted radiance I_{emit} is usually represented as an RGB value to allow for the emission of colored light. The absorption coefficient is represented by a scalar value between 0 and 1. Both coefficients can be combined into an RGBA value.

As we have seen in Section 1.2, volume data is represented by a 3D array of sample points. According to sampling theory, a continuous signal can be reconstructed from these sampling points by convolution with an appropriate filter kernel. The transfer function now can either be applied directly to the discrete sampling points before the reconstruction or alternatively to the continuous signal after the reconstruction. Both methods lead to different visual results. Accordingly, there are two possible ways to perform the assignment in

hardware, which differ in the positioning of the table lookup within the sequence of processing steps of the graphics pipeline. Implementations of color table lookups strongly depend on the underlying hardware architecture. Multiple different hardware implementations are described in Section 5.2.

5.1.1 Pre-Classification

Pre-classification denotes the application of a transfer function to the discrete sample points before the data interpolation. The reconstruction of the continuous signal is performed subsequently based on the emission and absorption values. Figure 5.1 (left) outlines this concept. The transfer function is here represented as the graph of a one-dimensional function. In practice, several of these curves would be used to describe individual transfer functions for each of the RGBA components separately. The original sampling values on the x-axis are mapped to emission and absorption values on the y-axis. As displayed in the diagram, the emission and absorption coefficient for a sample point which does not lie on an exact grid position is determined by interpolating between the emission and absorption coefficients given at the neighboring grid points.

With respect to the graphics pipeline pre-classification means that the color table lookup is performed before or during the rasterization step, however in any case before the texture generation step. The transfer function is applied to every texel before interpolation. The advantage of this concept is that an efficient implementation of a pre-classification table is possible on almost every graphics hardware. Before we examine and evaluate different implementations, we will have a look at the alternative concept of post-classification.

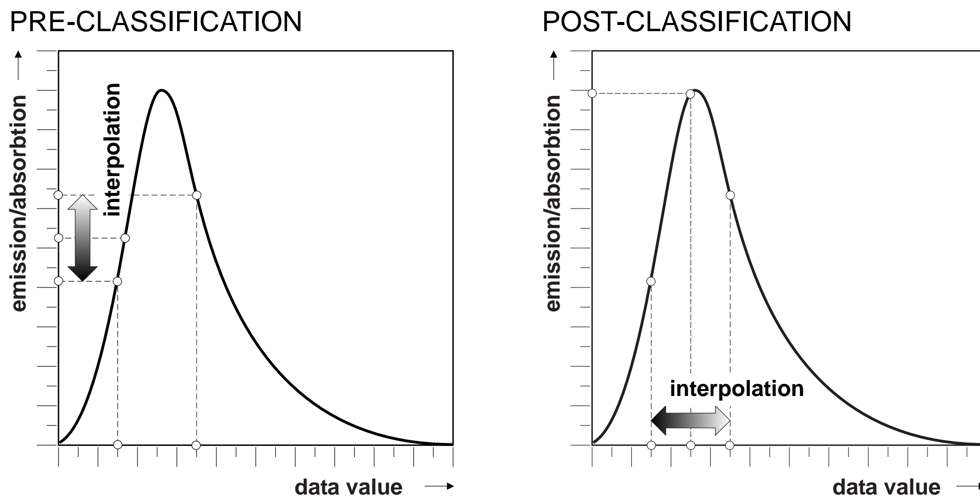


Figure 5.1: Transfer functions are used to map data values to physical quantities, which describe the emission and absorption of light. This mapping can be applied either before the interpolation (pre-classification) or after the interpolation of data values (post-classification), leading to different visual results.

5.1.2 Post-Classification

Post-classification reverses the ordering of operations. Transfer function application is moved behind the reconstruction process. The classification function is thus applied to the continuous signal instead of its discrete sampling points. This idea is illustrated in Figure 5.1 (right). For a sample point which does not lie on an exact grid position the data value itself is interpolated. Subsequently, the emission and absorption values are determined by using the interpolated data value as index into the color lookup table. It is easy to see in Figure 5.1, that pre- and post-classification lead to different results. Both alternatives will be evaluated and discussed in Section 5.3.

The implementation of post-classification is usually more complex than a simple table lookup. With a graphics pipeline implemented in hardware, it is not possible to extract intermediate values from the hardware stream, modify them in software and reinsert them into the pipeline. In consequence, post-interpolative color lookup must be explicitly supported by the graphics hardware.

5.2 Implementations

In this section we will examine multiple implementations of transfer function lookup that work with all the texture based approaches described in this thesis. Both pre- and post-classification approaches are discussed. Our main objective is to implement a fast color table update, allowing the transfer function to be modified in real-time.

5.2.1 Pre-Classification

As defined above, a transfer function for pre-classification is applied a priori to the texture images. Although there is no technical restriction that forbids the application of a color table as a pre-processing step, it is very unlikely that such an implementation will achieve interactive frame rates while updating the transfer function. The reason for this is twofold:

- A modification of the transfer function would require a reconstruction of the whole volume in main memory and a reload of the texture image into the local video memory of the graphics board. This will inevitably result in a memory bandwidth bottleneck, which significantly degrades performance.

- For storing the emission (color) and absorption (opacity) values directly in the texture an internal RGBA format is required which allocates four bytes per texel. An index into a color table however would only require one or two bytes per texel. As a result, using color indices significantly reduces both the memory footprint and the bandwidth problem.

```
0   // enable and setup pixel transfer
1   glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
2   glPixelMapfv(GL_PIXEL_MAP_I_TO_R, m_nColorTableSize, m_pColorMapRed   );
3   glPixelMapfv(GL_PIXEL_MAP_I_TO_G, m_nColorTableSize, m_pColorMapGreen );
4   glPixelMapfv(GL_PIXEL_MAP_I_TO_B, m_nColorTableSize, m_pColorMapBlue  );
5   glPixelMapfv(GL_PIXEL_MAP_I_TO_A, m_nColorTableSize, m_pColorMapAlpha );
6
7   // create texture image
8   glTexImage3D(...);
9
10  // disable pixel transfer
11  glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
```

Listing 5.1: OpenGL setup for color mapping during the pixel transfer from main memory to the local texture memory.

5.2.1.1 Pixel Transfer

The standard OpenGL specification provides a way to apply a color map during the pixel transfer from main memory to the graphics board. This is exactly what is done when a texture image is defined and transferred to the graphics board. Since changing the color table requires to upload the texture again, this is of course not a very fast way to apply the transfer function. However, for graphics hardware that does not support some of the OpenGL extensions described in the following chapters, it represents the only way to achieve the color mapping. The OpenGL code for setting up the pixel transfer is displayed in Listing 5.1.

Besides the poor performance, the main drawback of this approach is again the amount of data that must be allocated in local video memory. Although only the scalar data values are stored in main memory, the pixel transfer converts every scalar value into an RGBA quadruplet when writing it into the portion of video memory that is allocated for the texture image. As a result the size of the data that must be stored increases by a factor of four in the worst case. To work around this problem many hardware manufacturers have decided to implemented a mechanism that allows to store color indices in the texture image together with a separate color table. This concept is known as *paletted textures*.

5.2.1.2 Paletted Textures

Similar to many 2D image file formats that include a color table, texture palettes can significantly reduce the memory that must be allocated for storing the texture on the graphics board. Additionally this feature can be used to implement coloring effects by modifying the color palette without the necessity of modifying the texture object itself. Instead of storing the RGBA values for each texel, an index into a color lookup table of fixed size is used. This color table is stored together with the index texture in local video

```
0   #if defined GL_EXT_paletted_texture && defined GL_EXT_shared_texture_palette
1
2               glEnable(GL_SHARED_TEXTURE_PALETTE_EXT);
3
4               glColorTableEXT(
5                   GL_SHARED_TEXTURE_PALETTE_EXT, // GLenum  target,
6                   GL_RGBA,                       // GLenum  internal format
7                   m_nColorTableSize,             // GLsizei size of the table
8                   GL_RGBA,                       // GLenum  external format
9                   GL_UNSIGNED_BYTE,              // GLenum  data type
10                  m_pColorTable);                // const GLvoid *table
11
12  #endif // GL_EXT_paletted_texture
```

Listing 5.2: OpenGL setup for the paletted texture extension.

memory. During the texture generation step, the indices are replaced by the respective color values stored in the texture palette. It is important to notice that the color table lookup is located before the usual texture generation. The interpolation is performed after the lookup using the color values obtained from the lookup table, resulting in a transfer function for pre-classification. The amount of local video memory that must be allocated for storing an RGBA texture is significantly reduced, since only a single index value must be stored for each texel, instead of four values for the four color components. Taking into account the memory that is allocated for the texture palette itself, the required texture memory is thus reduced almost by a factor of four.

In OpenGL the access to texture palettes is controlled by two separate OpenGL extensions. The first extension `EXT_paletted_texture` enables the use of texture palettes in general. A paletted texture is created in the same way as a conventional RGBA texture. The only difference is that during texture specification the internal format of RGBA (`GL_RGBA`) must be substituted by an indexed format, such as `GL_COLOR_INDEX8_EXT`, `GL_COLOR_INDEX12_EXT` or `GL_COLOR_INDEX16_EXT` according to the intended size of the color table (see [121] for details). This extension supports texture palettes with a resolution of 1, 2, 4, 8, 12 or 16 bit respectively. In this case a unique texture palette must be maintained for each texture separately. A second OpenGL extension named `GL_EXT_shared_texture_palette` additionally allows a texture palette to be shared by multiple texture objects. This further reduces the memory footprint for a volume data set, if 2D-textures or 2D-multi-textures are used or if a 3D-texture is split up into several bricks. The OpenGL code for creating and updating a shared texture palette is displayed in Listing 5.2.

Compared to the pixel transfer method described in the previous section, the main advantage of the shared texture palettes is the ability to change the texture palette – and thus the transfer function – without having to reload the texture itself. In addition, the palette sizes of 12 or 16 bit enable high precision transfer function for tomographic data.

As a result the most efficient way to implement pre-classification for texture based volume rendering is to use paletted textures. Now that we have seen possible implementations for pre-classification, let us focus our interest on methods to realize transfer functions for post-classification.

5.2.2 Post-Classification

As mentioned above, post-classification requires a mechanism to realize a color table lookup after the interpolation of a texel. This color mapping must be applied within the graphics pipeline between texture generation and texture application. In consequence this cannot be achieved without any special hardware feature, that allows the modification of the texel directly within the texture unit. As a result the implementation of a transfer function for post-classification in texture based approaches strongly depends on the underlying graphics hardware.

5.2.2.1 Texture Color Tables

A straightforward hardware concept that allows for post-classification is provided by the high-end workstations developed by Silicon Graphics (SGI). The mechanism to setup this post-interpolative texture lookup table is very similar to the paletted textures. The OpenGL extension to access this feature is called `SGI_texture_color_table`. The extension must be simply enabled and a color table must be setup as described in Listing 5.3. Although this code looks very similar to the code presented in Listing 5.2, here the color table lookup is performed after the texture interpolation. The texture color table, which is enabled by this extension, is not restricted to a specific texture object, so it can be efficiently shared among multiple texture images.

```
0   #if defined GL_SGI_texture_color_table
1
2               glEnable(GL_TEXTURE_COLOR_TABLE_SGI);
3
4               glColorTableSGI(
5                   GL_TEXTURE_COLOR_TABLE_SGI, // GLenum  target,
6                   GL_RGBA,                    // GLenum  internal format,
7                   m_nColorTableSize,          // GLsizei width,
8                   GL_RGBA,                    // GLenum  external format,
9                   GL_UNSIGNED_BYTE,           // GLenum  data type,
10                  m_pColorTable);             // const GLvoid *table
11
12  #endif // GL_SGI_texture_color_table
```

Listing 5.3: OpenGL setup for the texture color table extension. Although this code fragment is very similar to the paletted texture setup in Listing 5.2, in this case the color table lookup is performed after the interpolation of texture samples.

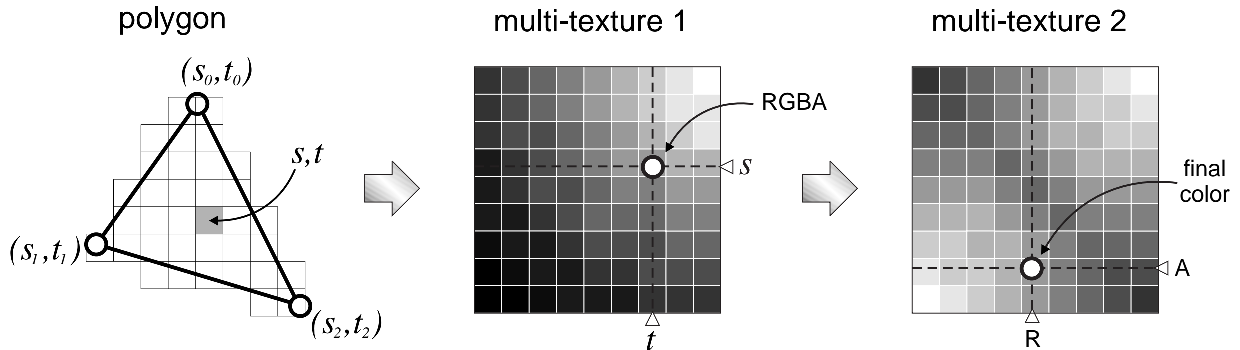


Figure 5.2: Dependent texture lookup: The texture coordinates (s, t) are interpolated as usual from the values given at the polygon vertices. An RGBA quadruplet is obtained from the first multi-texture. The red (R) and the alpha (A) component of this quadruplet are used as texture coordinates for the second multi-texture stage. The resulting final texel value is used to color the fragment.

Up until now, the post-interpolative texture lookup table is only supported by expensive graphics workstations such as the SGI Onyx (Base Reality and Infinite Reality) and the SGI Octane and Octane 2 (MXE and V series) [154]. The described OpenGL extension is also available on the SGI O2, however the hardware-accelerated implementation is replaced by a software emulation within the driver, which leads to very poor performance on this architecture. Unfortunately, post-interpolative texture lookup as described here is not supported by any of the consumer PC graphics boards currently available. The latest PC graphics processors, such as the ATI Radeon and the NVidia GeForce 3 however support a different hardware feature called *dependent textures*, which can be exploited for post-classification.

5.2.2.2 Dependent Textures

The specification of *dependent textures* is part of the pixel shader concept outlined in Section 4.1.2. It was first introduced in 2000, although a similar feature called *pixel textures* (`SGI_pixel_texture`) has been available before on SGI high-end workstations [150]. The idea of dependent textures is quite simple. Texture coordinates are usually specified at the vertices and interpolated in the interior of the polygon. However, for certain visual effects in computer graphics, such as the combination of an environment map with a bump map, it is required to specify texture coordinates on a per-fragment basis. At this point dependent textures come into play. Dependent textures are a supplement to multi-textures, which allows the texture coordinates for the second texture to be obtained from the first texture. The mechanism is illustrated in Figure 5.2. The pair of texture coordinates (s, t) for a fragment in the interior of the triangle is interpolated as usual from the values given at the three vertices. These texture coordinates are in turn used to interpolate an RGBA quadruplet at the first multi-texture stage. The red and the alpha component (or

```
0   #if defined GL_NV_texture_shader
1
2      // enable the texture shader extension
3      glEnable(GL_TEXTURE_SHADER_NV);
4
5      // activate and setup the first texture stage (3D-texture)
6      glActiveTextureARB(GL_TEXTURE0_ARB);
7      glEnable(GL_TEXTURE_3D_EXT);
8      glBindTexture(GL_TEXTURE_3D_EXT, m_nVolumeTextureName);
9      glTexEnvi(GL_TEXTURE_SHADER_NV,
10               GL_SHADER_OPERATION_NV, GL_TEXTURE_3D_EXT);
11
12     //activate and setup the second (dependent) texture stage (2D-texture)
13     glActiveTextureARB(GL_TEXTURE1_ARB);
14     glEnable(GL_TEXTURE_2D);
15     glBindTexture(GL_TEXTURE_2D, m_nColorTableTextureName);
16     glTexEnvi(GL_TEXTURE_SHADER_NV,
17               GL_SHADER_OPERATION_NV, GL_DEPENDENT_AR_TEXTURE_2D_NV );
18     glTexEnvi(GL_TEXTURE_SHADER_NV,
19               GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
20
21  #endif // GL_NV_texture_shader
```

Listing 5.4: OpenGL setup for the dependent texture lookup.

alternatively the green and the blue component) are interpreted as texture coordinates at the second multi-texture stage. This is possible because both texture coordinates and color components have a valid range between 0 and 1. The RGBA quadruplet interpolated from this second texture stage is then used as final texel value for the fragment.

The dependent texture mechanism can efficiently be used to implement a post-interpolative texture color table. To achieve this, the color indices of the voxels are stored in the red component of the first multi-texture (which can be a 3D-texture as well). The second multi-texture is defined as a one-dimensional texture, which has the same resolution as the color table[1]. During rasterization the red and the alpha component obtained from the first texture are used as texture coordinates for the second texture which contains the color table. The value of the alpha component is of course irrelevant. The resulting RGBA quadruplet now represents a color value obtained via post-interpolative index lookup.

In OpenGL dependent textures can be set up via an extension called `NV_texture_shader`, defined by NVidia for the GeForce 3 hardware. The code for activating the dependent texture lookup is displayed in Listing 5.4. At first the texture shader extension is enabled (line 3). Multi-texture stage 0 is activated and the 3D-texture that stores the color indices is bound to this unit (lines 6–8). The texture shader for

[1] Since one-dimensional textures are not supported by this extension, we actually use a two-dimensional texture with one dimension set to one.

this stage is configured for normal texture lookup (line 9–10). Subsequently, multi-texture stage 1 is selected and activated (lines 13–14). The 2D-texture which contains the color table is bound to this unit (line 15). The texture shader for this second stage is configured to perform a dependent texture lookup with the red and alpha component (line 16–17) obtained from texture unit 0 (line 18–19).

```
0   //DirectX8 Pixel shader 1.0
1    ps1.0;
2   //sample volume texture
3    tex t0
4   //use alpha and red of t0 as
5   //texture coordinate for t1
6    texreg2ar t1,t0
7   //write pixel
8    mov r0, t1
```

Listing 5.5: DirectX 8.0 pixel shader setup for post-classification via dependent texture lookup.

The same results can also be achieved on ATI Radeon family boards using a different OpenGL extension provided by ATI. Due to these different extensions, it is not possible to access the dependent texture capabilities of both the GeForce 3 and the Radeon hardware with the same OpenGL code. Up until now, consistent implementation for both hardware architectures can only be achieved using DirectX 8.0 (see Section 2.2.2) instead of OpenGL. DirectX accesses programmable rasterization via pixel shader code, written as small micro-programs which are directly executed on the GPU. A pixel shader example that delivers the same results as described above for both hardware architectures is presented in Listing 5.5. The first command (line 1) simply specifies the software version of the pixel shader specification. A texture sample `t0` is interpolated as usual from the texture unit 0 (line 3). Texture unit 1 (line 6) interpolates the dependent texture using the red and alpha components of `t0` as texture coordinates. The resulting texel `t1` is simply copied to the output register $r0$ (line 8). For more details on the DirectX 8.0 pixel shader specification refer to [109]. Although in terms of programmability the setup of dependent texture mapping seems to be rather complicated compared to the simple texture color table provided by SGI, the major benefit of dependent textures is that also two- or four-dimensional transfer functions can be realized with this concept. We will focus our interest on multi-dimensional transfer functions in Section 5.4.

5.3 Discussion

We have seen two different ways to apply a transfer function for direct volume rendering, either before (pre-classification) or after the interpolation (post-classification). A summary of the different implementations is presented in Table 5.1.

Pre-Classification		
Pixel Transfer	⊕ OpenGL Standard	⊖ high memory requirement ⊖ texture reload necessary ⊖ slow
Texture Palette	⊕ very fast	⊖ OpenGL extension required
Post-Classification		
Color Table	⊕ very fast	⊖ OpenGL extension required
Pixel Shader	⊕ fast ⊕ multidimensional	⊖ OpenGL extension required ⊖ multiple texture lookups

Table 5.1: Summary of different implementations of pre- and post-classification transfer function lookup.

A transfer function usually tries to separate different objects inside the volume data set according to their scalar value. Due to the band-limitation of the voxel data set, however, sharp boundaries between different object do not exist in the data. Thus, trying to display objects as isosurfaces with a sharp peak of infinite frequency in the transfer function is not appropriate to represent the fuzzy boundary. The transfer function of course should account for this fuzziness and simultaneously be able to separate tiny detail structures. A good transfer function will be a compromise between a sharp edge and a smooth transition between different objects.

As we have seen in Section 1.2, the discrete samples of a voxel data set represent a continuous 3D scalar field. Let us restrict our considerations to the continuous 1D signal that is obtained by casting a ray through the volume. According to sampling theory, a continuous signal $f(x)$ can be exactly reconstructed from discrete samples $f(k \cdot \tau)$ according to

$$f(x) = \sum_k f(k \cdot \tau) \cdot sinc\big(\frac{1}{\tau}(x - k\tau)\big). \tag{5.2}$$

Obviously the application of a transfer function T to the discrete sampling points instead of the continuous signal yield different results:

$$T\big(f(x)\big) \neq \sum_k T\big(f(k \cdot \tau)\big) \cdot sinc\big(\frac{1}{\tau}(x - k\tau)\big). \tag{5.3}$$

Reconstructing a collection of arbitrary discrete values will not inevitably return a valid band-limited signal, not even if an ideal reconstruction filter is used. A simple example is a discrete signal with only one single peak as displayed in Figure 5.3(A). Without loss of generality, the distance between adjacent sampling points τ is set to 1. Reconstructing such a signal results in the sinc function (B) which is represented by two peaks at $-\frac{1}{2}$ and $\frac{1}{2}$

in the frequency domain (C). The sinc function,

$$f(x) = sinc(x) \qquad \text{and} \qquad F(t) = \int f(x) \cdot e^{-2\pi ixt} dx = \chi_{[-\frac{1}{2},\frac{1}{2}]} \tag{5.4}$$

is band-limited according to

$$F(t) = 0 \qquad \text{for} \qquad |t| > F_{max} = \frac{1}{2}. \tag{5.5}$$

Since the band limit F_{max} is *equal to and not less than* the Nyquist frequency,

$$\tau = 1 \not< \frac{1}{2 \cdot F_{max}}, \tag{5.6}$$

the sampling theorem is violated. Apparently, if a collection of discrete sample points represented a valid signal with

$$\tau < \frac{1}{2 \cdot F_{max}}, \tag{5.7}$$

the frequency of the sinc filter itself must cancel out during the reconstruction. In consequence, the manipulation of a single sampling point might render the whole signal invalid. However, this is exactly what is done if a transfer function is applied before the reconstruction. The caveat is that modifying the sampling points of a given signal in an arbitrary way *before* the reconstruction might invalidate the initial assumption of band limitation and will thus strongly violate the sampling theorem.

If we first reconstruct the continuous signal and apply the transfer function afterwards, we ensure that the sampling theorem is obeyed. The transfer function will introduce another high-frequency component to the signal and we will have to adapt the sampling rate to account for the new frequency spectrum. As long as the transfer function itself is band-limited, we are on the safe side. The performance might decrease due to the higher number of slices, but the results are images of high accuracy.

Both pre- and post-interpolative transfer functions change the frequency spectrum of the original signal. While in case of pre-classification the high frequencies are simply cut off, post-classification takes them into account. These theoretical considerations are easily verified in practise. The results of both pre- and post-classification can be compared in Figure 5.4. Although one might be in doubt about the accuracy of the sharp object boundaries obtained by post-classification, it is indisputable that the blocky structure that results from pre-classification appears extremely distracting to the viewer. Another example that demonstrates the superior image quality of post-classification is shown in Figure 12.5.

5.4 Multi-Dimensional Transfer Functions

Up until now we have determined color and opacity values for a voxel as function of its scalar value. Although this is the most common way, it is not the only possibility to derive

the physical quantities required for ray integration. In addition to the traditional one-dimensional transfer function of the voxel intensity, the magnitude of the first and second order derivatives can be taken into account. In this case the transfer function is defined on a multi-dimensional parameter domain.

The principles of multi-dimensional transfer functions have been investigated before by various researchers. Mark Levoy [93] was the first who proposed to use the scalar magnitude of the gradient vector to weight the opacity value of a voxel. Homogenous regions inside the volume data are usually less interesting than regions where the intensity value changes significantly, such as boundaries and fuzzy transitions between different structures. The gradient vector is the first order derivative for a 3D scalar field $I(x, y, z)$, defined as

$$\nabla I \;=\; (I_x,\, I_y,\, I_z) \;=\; (\frac{\delta}{\delta x}\,I,\, \frac{\delta}{\delta y}\,I,\, \frac{\delta}{\delta z}\,I), \tag{5.8}$$

using the partial derivatives of I 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,

$$||\nabla I|| \;=\; \sqrt{{I_x}^2 + {I_y}^2 + {I_z}^2}. \tag{5.9}$$

There are several approaches to estimating the directional derivatives for discrete voxel data. A 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

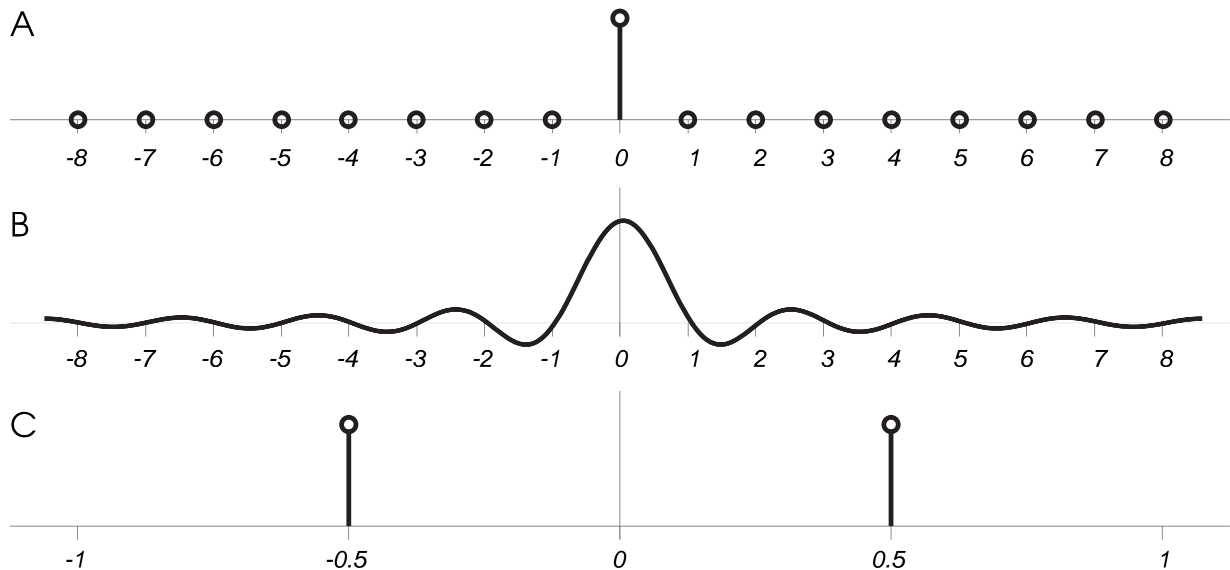


Figure 5.3: Discrete signal with only a single peak (A) unequal zero. Reconstruction of this signal yields the sinc function (B). The frequency distribution (C) of the sinc function consists of two peaks at $-\frac{1}{2}$ and $\frac{1}{2}$.

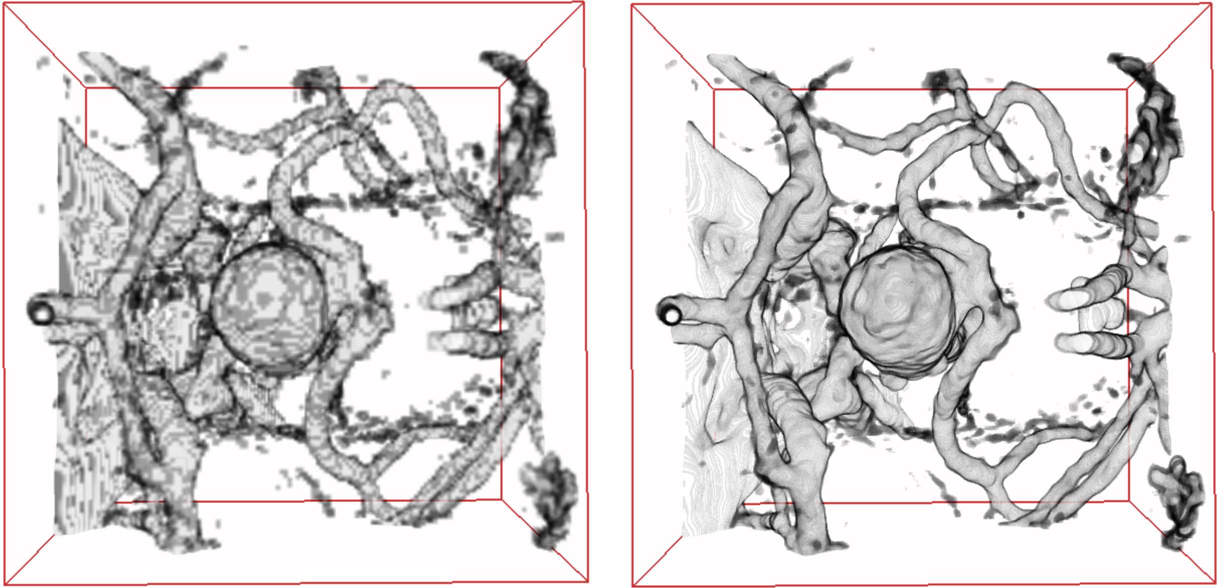


Figure 5.4: Comparison of pre- (left) and post-classification (right) of a CTA data set using a transfer function of high frequency. Both images were generated with exactly the same transfer function and with exactly the same number of slice images. Due to the blocky appearance the pre-classified image is rather disturbing. The volumetric shapes are much better represent by the high-frequency of the transfer function applied as post-classification.

as

$$I_x(x, y, z) \;=\; I(x+1, y, z) \;-\; I(x-1, y, z) \quad \text{with} \quad x, y, z \in \mathbb{N}. \tag{5.10}$$

The derivatives in other directions are computed analogously. The second order derivative of a 3D scalar field is described by a matrix called the *Hessian*, written as

$$H \;=\; \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}. \tag{5.11}$$

Sato et al. [145] used the eigenvalues of the Hessian matrix to classify local intensity structures such as *sheet*, *line* and *blob* structures. Since these local intensity structures exist at various scales, multi-scale analysis is performed using convolution with Gaussian filter kernels of different size. Other approaches use the second order derivative in direction of the gradient vector instead of the Hessian matrix. The original intensity, the magnitude of the gradient and the magnitude of the second order derivative in direction of the gradient can be used as parameter domain for a 3D transfer function as proposed by Joe Kniss et al. [83]. They also introduced an implementation of multi-dimensional transfer functions on general purpose hardware together with a framework to adjust them interactively.

A different path was followed by Hladuvka et al. [69] who introduced curvature based transfer functions. Analogous to surfaces, the local neighborhood of a point can be described by two tangent vectors, called the *principal directions* and two corresponding values

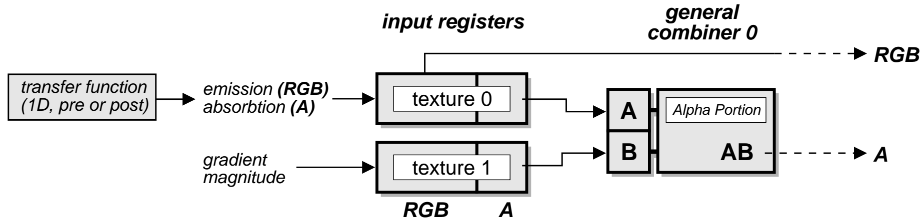


Figure 5.5: Register combiner setup for opacity weighting with gradient magnitude

of *principal curvature* κ_1 and κ_2. According to the signs of these values, we classify the surface locally as

- plane ($\kappa_1 = \kappa_2 = 0$),
- parabolic cylinder ($0 = \kappa_1 > \kappa_2$ or $\kappa_1 > \kappa_2 = 0$),
- paraboloid ($\kappa_1 \cdot \kappa_2 > 0$),
- hyperbolic paraboloid ($\kappa_1 \cdot \kappa_2 < 0$),

A transfer function based on principal curvature classifies voxels according to the local shape of the corresponding isosurface. The approach is especially useful for the extraction of turbulent regions from simulation data according to the curvature magnitude and for the detection of malformations such as vessel strictures (stenosis) in medicine.

From the visualization point of view, scalar data with multidimensional transfer functions are similar to multivariate data, where multiple scalar values are given for each voxel (such as pressure, density and temperature). In our case, the additional scalar fields are derived from the original data set e.g. by gradient computation. In consequence, the implementations of multi-dimensional transfer functions described in the following chapter can be used to display multivariate data sets as well.

5.4.1 Implementation

A multi-dimensional transfer function for pre-classification is very easy to achieve as a pre-processing step. The implementation of multi-dimensional transfer functions for post-classification however is a little bit more challenging. As a special case, the opacity value determined by a traditional one-dimensional transfer function can be weighted linearly by simply multiplying it with the magnitude of the first order derivative. In this case the absorption coefficient for a sample value $v = I(x_i, y_i, z_i)$ is given by

$$\vartheta_k \; = \; ||\nabla I(x_i, y_i, z_i)|| \cdot T_{\text{absorption}}(v). \tag{5.12}$$

Gradient weighted opacity can be implemented by the use of multi-textures as outlined in Figure 5.5. An additional 3D-texture (`texture1`) stores the pre-computed (and scaled) magnitude of the gradient vector or any other scalar field that can be used as parameter

domain for the transfer function. During rasterization the opacity value is obtained as usual from the original texture via one-dimensional transfer function. The resulting opacity value is then simply multiplied by the scalar weight obtained from the second texture. Figure 5.6 (left) shows a CT data set with a conventional pre-interpolative one-dimensional transfer function of the scalar value. Although a low transparency value is assigned for the soft tissue at the neck of the patient, this region appears completely black and opaque. This is due to the constant absorption that is accumulated for a ray that passes through this homogenous region. Linear opacity weighting with the gradient magnitude (middle) amplifies surface-like structures, while the large opaque homogenous regions of the unweighted data set completely vanish. Linear gradient weighting can also be efficiently combined with illumination calculations (right) as explained in Chapter 7.

The implementation of real multi-dimensional transfer functions can be achieved using the concept of dependent texture lookup. In Section 5.2.2.2 we have already seen an implementation of a post-interpolative transfer function using dependent textures. In this case the color table was stored in a texture, which was accessed by texture coordinates obtained from the volume texture. For a one-dimensional transfer function, one of the texture coordinates was simply discarded. This second texture coordinate can be used to accomplish a two-dimensional transfer function lookup. Analogously four-dimensional transfer functions are implemented as a sum or a product of two 2D transfer functions as outlined in Figure 5.7. Each color component of the original RGBA texture now stores a different scalar field such as the magnitude of the gradient or the second order derivative, or alternatively additional information such as pressure, density or temperature if available. The `Red` and `Alpha` component are used as texture coordinates for a dependent texture that stores a 2D transfer function lookup table. Analogously the `Green` and `Blue` components are interpreted as texture coordinates for a second dependent texture, which stores a different transfer function. Subsequently the RGBA quadruplets obtained from

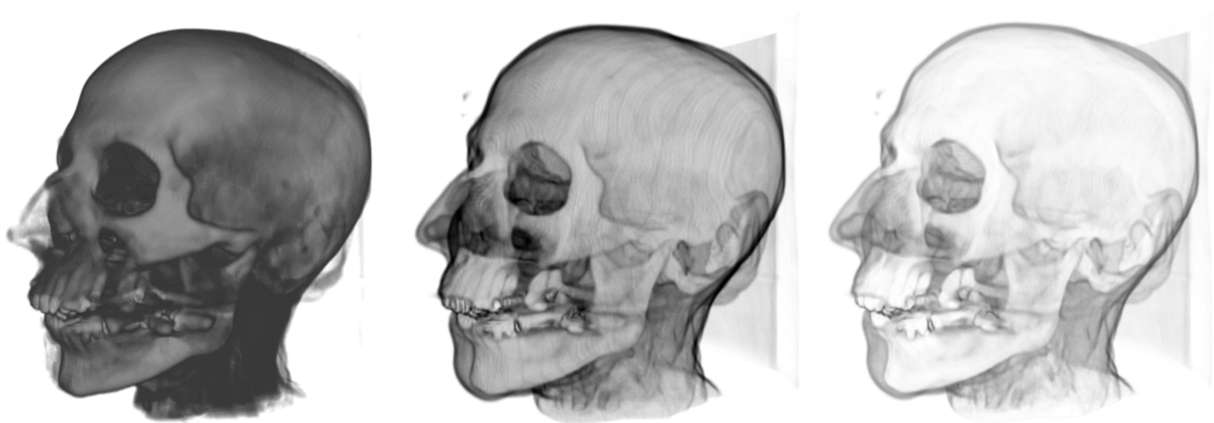


Figure 5.6: CT data set visualized with the 2D-multi-texture-based approach. Pre-interpolative transfer function (left), pre-interpolative transfer function with linear gradient weighted opacity (middle) and an additional diffuse light source (see Chapter 7) (right)

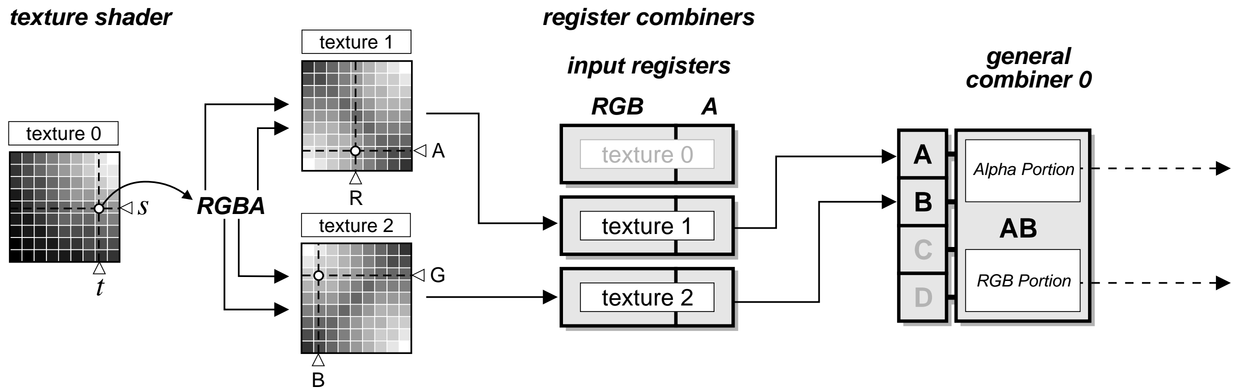


Figure 5.7: Texture shader and register combiner setup for multidimensional transfer functions.

both dependent textures are combined e.g. by component-wise multiplication or addition. The result is a four-dimensional transfer function,

$$T(v_1, v_2, v_3, v_4) = T_{12}(v_1, v_2) \cdot T_{34}(v_3, v_4) \quad \text{or} \tag{5.13}$$
$$\tilde{T}(v_1, v_2, v_3, v_4) = T_{12}(v_1, v_2) + T_{34}(v_3, v_4). \tag{5.14}$$

More complex combinations of the two independent transfer functions T_{12} and T_{34} such as dot product or blending are also possible. The classification procedure however will hardly be intuitive, if such complicated transfer functions are used.

5.4.2 Fusion

In the context of biomedical imaging, the term *fusion* refers to the combination of complementary information obtained from multiple data sets into a final image. In many applications fusion is not performed during visualization using multi-dimensional transfer functions. Instead, multiple data sets are merged together in an intelligent pre-processing step which generates a new data set that contains the joint information. Fusion techniques for multi-modal images in medicine, machine vision and remote sensing applications [1, 20, 98] are still topics of active research. Li et al. [95] have suggested a fusion algorithm that performs a sequence of forward and backward wavelet transformations. Matsopoulos et al. [104] perform hierarchical fusion based on feature extraction from morphological pyramids. Other approaches utilize pixel classification approaches [11] such as the Bayesian probabilistic method [73] or entropy based fusion [116]. More recently Mukhopadhyay and Chanda [117] introduced a fusion method based on multi-scale morphology. The advantage of these techniques over multi-dimensional transfer functions is the low memory requirements during visualization and the reduction of the problem of multi-dimensional transfer function design to the traditional one-dimensional problem. This however comes with a significant loss in flexibility for interactive exploration of the data. Since the fusion algorithms are usually performed on the voxel data without reconstruction of the continuous signal, problems similar to pre-classification will arise.

Figure 5.8: The 3D anatomical atlas *VoxelMan* [41] allows the interactive exploration of the human body based on a detailed segmentation.

5.5 Local Transfer Functions

Up until now we have applied one transfer function to the entire volume data set. The color and opacity values for all the voxels have been determined by a single color lookup table. In many application areas, such as the one described in Chapter 14, it is desirable to divide the volume data set into small voxel subsets using explicit segmentation algorithms and to specify a unique transfer function for every subset. We refer to this concept as local transfer functions assignment.

The division of the volume into disjoint subsets is performed on a per-voxel basis by assigning a unique *tag number* for each subset. A transfer function which is applied locally to such a voxel subset affects only those voxels that carry the specified tag number. As the tag numbers usually specify certain attributes, such as functional or anatomical properties, a tagged volume is also often referred to as an *attributed volume* [162]. A prominent example for such an attributed volume is the explicit segmentation of the visible human data sets [122] that was obtained by Karl-Heinz Höhne and his group. In combination with interactive video streaming (Quicktime VR [130]) this technique was used to build an interactive 3D anatomical atlas on CD-ROM, available as a commercial product for physicians and students of medicine [41].

The integration of local transfer functions into texture based volume rendering raises again the question whether to apply the color lookup before or after the interpolation. At the first glance, local transfer functions might appear similar to multi-dimensional transfer function, with the tag number being an additional scalar field that simply expands the transfer function domain. However, there is a significant difference. Tag numbers are used to differentiate between specific structural entities and thus do not represent a continuous scalar field. The interpolation and the reconstruction of a continuous signal from discrete tag numbers of course does not make sense. The principle of local transfer function application is illustrated in Figure 5.9. In addition to the original scalar value,

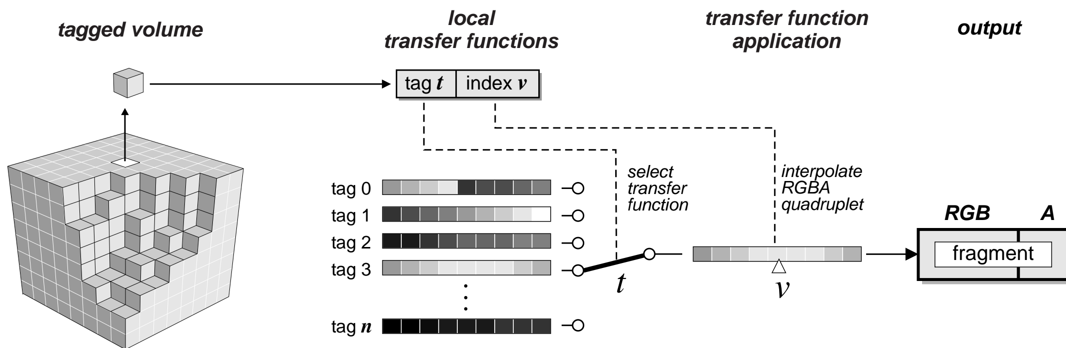


Figure 5.9: Local transfer function application: The tag number t selects one of several transfer function tables. The index value v is used to interpolate an RGBA quadruplet from the selected table.

every voxel holds an integer value representing the tag number of the subset that it belongs to. In a first step, the tag number is used to select the transfer function which was generated for the specific voxel subset. Subsequently the original scalar value is used as index into the selected transfer function table. The resulting RGBA quadruplet is used for ray integration.

5.5.1 Implementation

Similar to multi-dimensional transfer functions, the implementation of local transfer functions is relatively simple for pre-classification and considerably difficult in case of post-classification. Pre-interpolative application of local transfer functions are easily implemented using the OpenGL pixel transfer or paletted textures as described in Section 5.2. The idea here is to partition the existing one-dimensional color lookup table into multiple sections as illustrated in Figure 5.10. The OpenGL pixel transfer usually supports color lookup tables with at least 10 bit depth. Such a color table allows the allocation of four different tag numbers, each with a corresponding local transfer function of 8 bit resolution. The volume is internally stored with a resolution of 16 bit per voxel. A value of $256 \cdot tag$ is added to the original 8 bit intensity value before writing it into the texture buffer and the pixel transfer finally maps the resulting number to an RGBA quadruplet determined

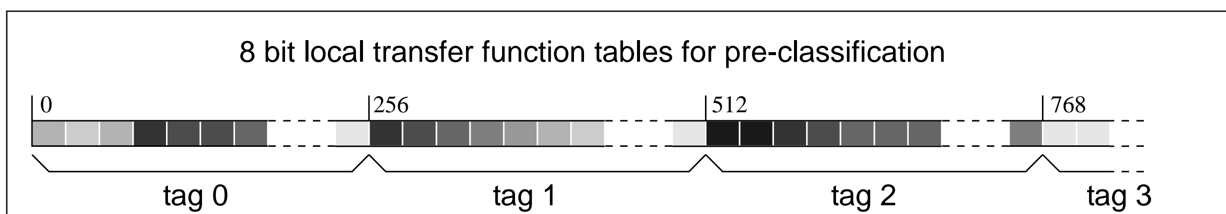


Figure 5.10: Local transfer functions for pre-classification during pixel transfer are implemented by partitioning the available hardware color table into several portions.

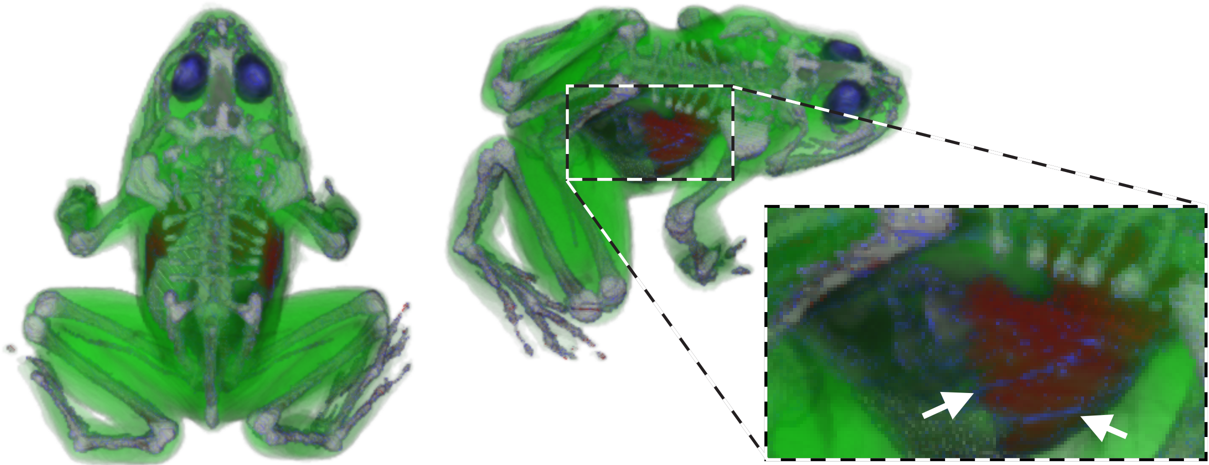


Figure 5.11: CT scan of a frog: Results of a detailed segmentation are displayed using post-interpolative local transfer functions. Visual artifacts occur at the transition between different tag regions.

by the corresponding local transfer function.

As we have seen in Section 5.3 the application of post-interpolative transfer function is favorable with respect to the image quality. Unfortunately, partitioning the color table does not work in case of post-classification. The problem here is the interpolation between neighboring voxels, that carry different tag numbers. Interpolating across the boundaries of tagged regions will result in visible artifacts. However, to completely avoid the interpolation between tag numbers, the local transfer function selection must take place before the interpolation while the transfer function should still be applied after the interpolation. This combination of pre- and post-interpolative color table lookup turns out to be extremely difficult to implement within the rendering pipeline.

Local transfer functions for *two* different tags can be implemented using dependent textures in the same way as multi-dimensional transfer functions. For more than two tag numbers, visual artifacts occur at the boundary surface between different tags. Figure 5.11 displays the results of an implementation using dependent textures and multiple tags. The only way to suppress these artifacts is to use multi-pass rendering for each slice. Each tagged object is drawn in a separate rendering pass with the transfer function for all the other tags set to fully transparent. Such an implementation, however, would hardly reach interactive frame rates.

Chapter 6

Transfer Function Design

In the previous chapter various types of transfer functions have been explained. We have also seen how transfer function tables are applied technically within the graphics pipeline. From the practical point of view however, probably the most significant challenge with direct volume rendering is finding an appropriate transfer function for a specific data set. Especially for tomography data this is a non-trivial task. The only variant of direct volume rendering which completely circumvents the problem of transfer function adjustment is maximum intensity projection (MIP) as explained in Section 3.1.2.1. The applicability of MIP, however, is extremely limited.

Now and in the future, the work of practitioners and researchers would be helped by a simplified and more intuitive procedure for transfer function adjustment as actually available. The reason why a good transfer function is difficult to accomplish is its high degree of freedom. Finding appropriate transfer functions that yield the desired image results is often a tedious process of manual tweaking of parameters. Besides interactive manual specification, there are several approaches for automatic and semi-automatic generation that we will focus our interest upon in this chapter. Apart from the specific problem of transfer function design, the automatic generation of color maps for other purposes has been investigated before my various researchers [9, 140, 170].

6.1 Interactive Adjustment

In the majority of scientific applications transfer functions are manually generated using some type of visual editor or *widget*, e.g dialog windows as displayed in Figure 6.1. Available implementations of color table editors vary in the representation and the degrees of freedom for the transfer function. They allow the composition of a transfer function using primitive objects such as linear ramps, gaussian curves and splines. Manual adjustment of transfer function requires interactive frame rates to be provided by the underlying rendering algorithm, as the direct visual feedback within the 3D viewer is indispensable for goal-directed work. The effects of changes applied in the transfer function domain must be immediately visible in the volume viewer. Recently, Joe Kniss et al.[83] have introduced

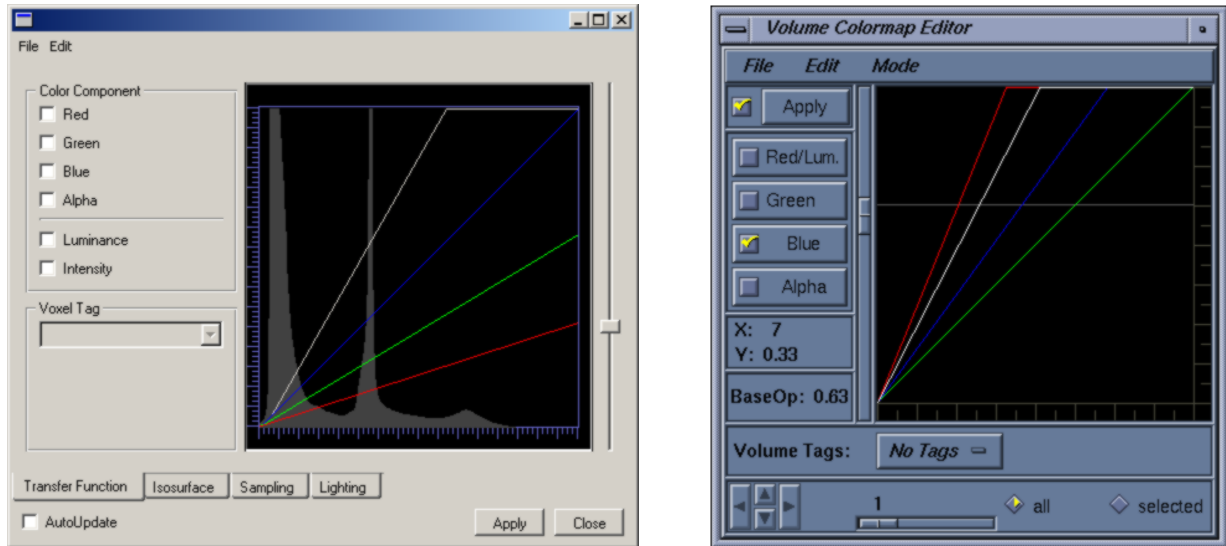


Figure 6.1: The user interface for transfer function assignment is usually implemented as a dialog window which allows editing a graphical representation of the color map. Two examples of such a dialog window are displayed.

direct manipulation widgets, a very useful concept which provides visual feedback in both directions. Interaction that is performed in the spatial domain of the volume object is used to select regions in the transfer function domain. Up until now this is the only approach for interactive assignment that integrates multi-dimensional transfer functions in an intuitive way.

Manual transfer function design is primarily based on experience, allowing the user to bring in his knowledge of the specific data as well as his personal taste. In many cases the assignment procedure is hardly reproducible even for similar data sets. Up until now manual heuristic methods are the only approaches for transfer function assignment, that include detailed knowledge about the structures inside the volume data. In order to speed up manual assignment in practise, application-specific templates are used, which are manually adapted to the individual data set. As we have seen in the previous chapter, transfer functions are usually stored and applied as color lookup tables, one-dimensional arrays of RGBA quadruplets. Templates can also be stored as piecewise linear mappings or functions of higher order, which will reduce the number of free parameters that must be adjusted. The flexibility of a transfer function template again depends on its degree of freedom.

In many application areas adequate techniques to compare examination results are required. Physicians and engineers are in demand of a reliable method to reproduce the visualization in order to discuss and re-evaluate their conclusions. As a result, there is a high demand for automatic methods to generate appropriate transfer functions in a reproducible way. Various approaches for automatic or semi-automatic assignment have been proposed recently. These approaches can be categorized into image-driven and data-

driven techniques as outlined in the following sections.

6.2 Image-Driven Techniques

The major purpose of a transfer function is to create meaningful images. In consequence, image-driven techniques for automatic transfer function design analyze the information contained in the images generated with different parameter settings. They derive a quality metric for finding an optimal setting based on this analysis. Methods for setting visual parameters reported in literature either explore the parameter space interactively (*interactive evolution* [86, 155, 163]) or search for optimal settings based on an objective quality measure (*inverse design* [156, 166, 80, 182]).

A very general concept for setting visual parameters in computer graphics and animation is the *Design Gallery* as presented by Marks et al. [103] in 1997. The basic idea of *Design Galleries* is to automatically generate and organize the broadest selection of perceptually different images that can be produced by varying a given vector of input parameters. This requires a way of finding a set of the input parameter vectors that optimally disperse the output values. Additionally, *Design Galleries* arrange the results and organize them efficiently for intuitive browsing by the user.

He et al. [67] have developed a technique for semi-automatic transfer function generation using stochastic search algorithms. In this idea the process of transfer function generation is interpreted as an optimization problem. An appropriate parameterization of the transfer function is chosen. Based on this parameterization an initial set of transfer functions is created, and for each function an image of the volume is rendered. With this initial set a stochastic search for the optimal parameter setting is started. The search algorithm is controlled by either manual thumbnail selection [87] of the best results or by evaluation of an objective metric such as entropy, edge energy or histogram variance. Possible search strategies that have been applied comprise genetic algorithms [71, 54], hill-climbing and simulated annealing [82]. Genetic algorithms are optimization techniques which are modeled on natural evolutionary processes. An initial population of transfer functions (genotypes) is created. For each of these transfer functions an image (phenotype) is generated and a fitness value is evaluated. From the set of transfer functions that have the highest fitness value, the next generation of transfer functions are created in an evolutionary process that involves crossbreeding and mutation.

Although the image-driven approaches represent a very helpful aid for unexperienced users, they are not inevitably faster than manual assignment using a color table editor. The applicability of the image-based approaches which are based on non-deterministic concepts in general is very limited since the results are hardly reproducible. Image-based quality metrics for fully automatic transfer function generation tacitly assume that the image quality is solely influenced by the parameters of the transfer function. However, in many cases other parameters during image generation, such as the viewing position, may also have great influence on the image quality. Important structures in 3D might be completely occluded for randomly generated viewpoints. A possible solution to this

problem might be the inclusion of approaches for automatic viewpoint selection, such as the methods used for image-based rendering [70, 2, 142, 159], but such a strategy has not yet been investigated.

6.3 Data-Driven Techniques

In contrast to the image-driven methods, data-driven techniques analyze the volume data itself instead of the generated images. The process of transfer function design is thus decoupled from the influence of image related parameters such as viewing position and pixel resolution. However, the data amount to be analyzed significantly increases.

Fang et al. [43] consider the problem of transfer function design from an image processing point of view. In their definition the transfer function transforms a 3D scalar volume to a 3D RGBA volume and thus can be formulated as a sequence of image processing operations. The major drawback of this method is the high cost in memory that is required to store intermediate results for the image processing operations. The large memory requirements can be circumvented by adapting the volume rendering procedure to perform the image processing operations on the fly. A related technique was presented by Sato et al. [145]. Their approach applies 3D image filters in order to accentuate local intensity structures. In their concept the transfer function domain is a multi-dimensional feature space which is based on first and second order directional derivatives. Bajaj et al [3] propose a data-driven technique which gathers statistical information about the isosurfaces by evaluating metrics such as surface area, volume and gradient magnitude. The algorithm is supplemented by a specialized user interface for parameter selection.

The most promising data-driven technique was presented by Kindlmann and Durkin [81]. They proposed an interesting data-driven method which takes into account the first and second order directional derivatives of the scalar field. The principle of this approach is based on some theoretical considerations. If we start at a particular point

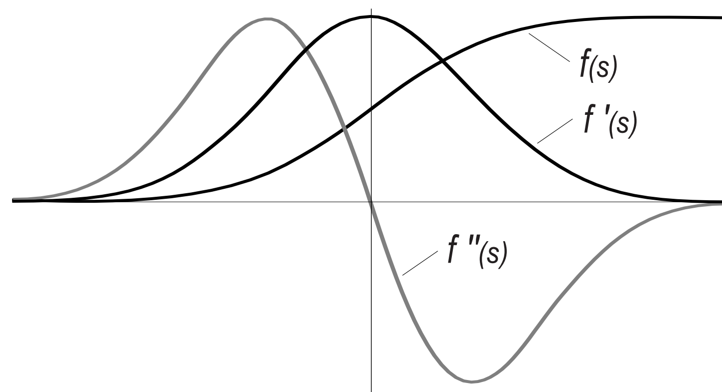


Figure 6.2: Monotonously increasing function $f(x)$ and its first and second order derivatives $f'(x)$ and $f''(x)$.

inside the volume and move through the scalar field by following the path p that is determined by its (non-zero) gradient vector, we can parameterize the scalar values $f(x, y, z)$ as a one-dimensional function of the covered distance s. Since the gradient always points in direction of the strongest increase, we will obtain a monotonically increasing function $f_p(s)$. Such a function is displayed in Figure 6.2. In image processing, the first and second order derivatives $f_p'(s)$ and $f_p''(s)$ are frequently used criteria for boundary detection. Since $f_p(s)$ is invertible, it is possible to express the derivatives $f_p'(s)$ and $f_p''(s)$ as a function of the data value $v = f_p(s)$ instead of the position s, denoted

$$\tilde{f}_p{}'(v) = f_p'(f_p^{-1}(v)) \qquad \text{and} \qquad \tilde{f}_p{}''(v) = f_p''(f_p^{-1}(v)). \tag{6.1}$$

As a result we have obtained the first and second order derivatives as function of the scalar value v. These functions however are only valid for one particular path P through the volume. To obtain first and second order derivatives for the entire data set, these curves are simply averaged for all paths $p \in P$ through the volume,

$$g(v) = \frac{1}{||P||} \sum_{p \in P} \tilde{f}_p{}'(v) \qquad \text{and} \qquad h(v) = \frac{1}{||P||} \sum_{p \in P} \tilde{f}_p{}''(v). \tag{6.2}$$

In practice for each voxel of the original data set the first and second order derivatives in gradient direction are computed. These values are averaged for all voxels with equal data value v. A boundary in the data set is described by a high magnitude of the first order derivative and a zero-crossing in the second order derivative. Combining the two functions $g(v)$ and $h(v)$ results in the *position function*

$$p(v) = \frac{-h(v)}{g(v)} \quad , \tag{6.3}$$

which is interpreted as the average distance of a data point with value v from a boundary in the data set. The *position function* $p(v)$ is used to compute a transfer function for opacity by applying a small threshold to determine the zero-crossings. The result is multiplied

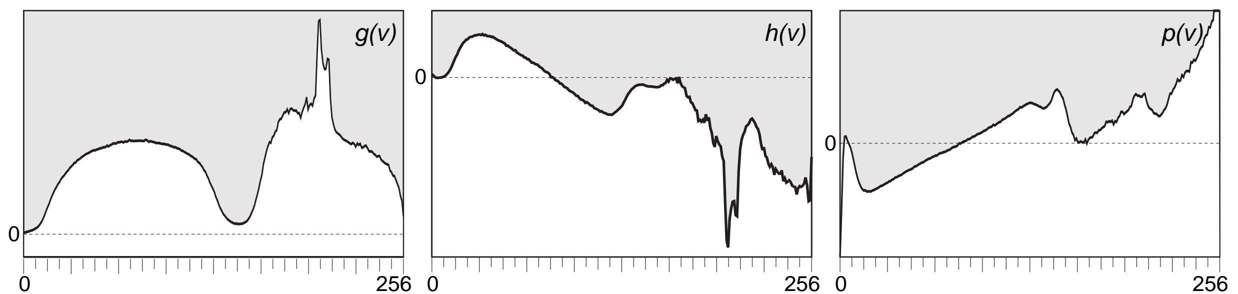


Figure 6.3: Averaged derivatives $g(v)$ and $h(v)$ and position function $p(v)$ according to the semi-automatic approach to transfer function design presented by Kindlmann and Durkin [81].

with a *boundary emphasis function* which determines the shape of the transfer function. As an example the averaged derivatives $g(v)$ and $h(v)$ and the position function $p(v)$ for the engine block data set are displayed in Figure 6.3.

The algorithm proposed by Kindlmann and Durkin is capable of accurately determining boundaries within a given data sets without any a priori knowledge of the spatial structures that it contains. It is probably the best method currently available to visualize shapes and structures in an unknown volume data set.

6.3.1 Automatic Adaptation

There are several different aspects that are important for developing algorithms for automatic transfer function design. Besides the obvious task of designing a transfer function that yields meaningful images for a specific data set, modern application also require robustness and reproducibility of the visual results for different data sets. Let us consider the following application scenario: For a clinical study such as the one described in Chapter 13, the physician has collected a large number of similar data sets from different patients. He knows exactly what particular structures are contained in the data set, and he also has some basic idea on how the resulting images should look like. Based on his profound knowledge of the anatomy, he decides that some particular structures are important for his analysis and others are less interesting or in some cases even appear distracting. In order to compare the anatomical structures of different patients, he wants to obtain exactly the same visual representation for all his data sets. This task, however, requires the inclusion of a priori knowledge about the data into the described semi-automatic approach to transfer function generation. To these ends we propose a method, that utilizes the position function to automatically adapt existing transfer function templates to new data set. Preliminary results of this idea have been published in [136].

In the solution that we aim at, the user should assign a transfer function for only one particular data set, and the resulting template should intelligently be adapted to all the other data sets in the series. For multiple data sets of equal modality the distribution of the intensity values vary within a considerable range. In order to account for these effects, the input parameter axis of transfer function template must be distorted non-linearly during the adaptation. This again leads to an optimization problem.

For a given series of similar data sets, we select a reference data set D_{ref}, and we initially design a transfer function $T_{ref}(v)$ once for this specific data set using a manual editor or any other approach. For a different data set D_{study} from the same series, we search for a non-linear transformation $t(v)$ of the intensity values v, such that the transfer function

$$T_{study}(v) = T_{ref}\big(t(v)\big) \tag{6.4}$$

yields equivalent visual results as in the reference case. Based on an appropriate parameterization of the non-linear distortion $t(v)$, the free parameters are determined by a stochastic search algorithm. Since the number of parameters for optimization is limited, exhaustive search techniques and enumeration techniques such as dynamic programming [32] are applicable.

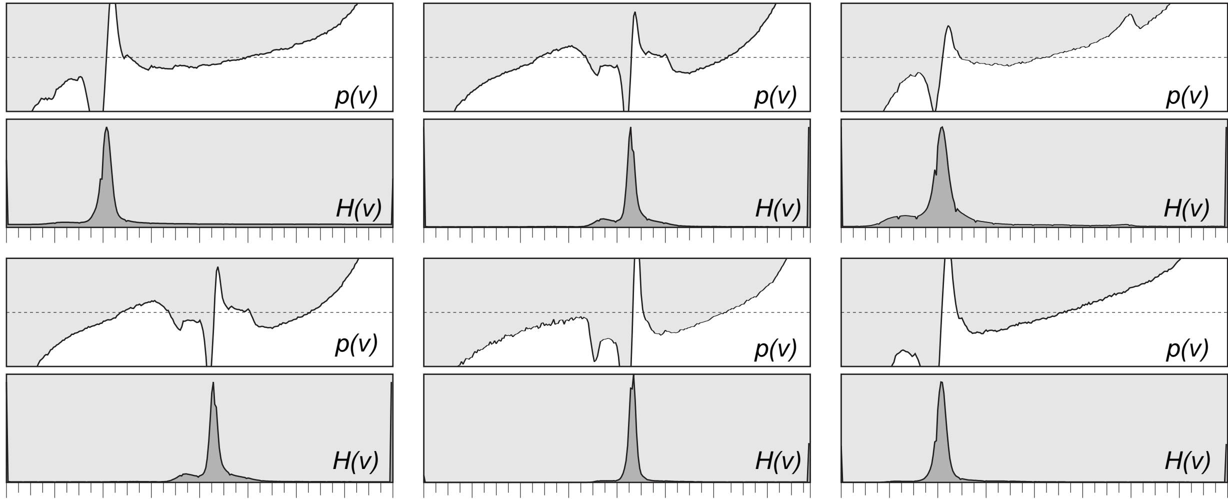


Figure 6.4: Position functions $p(v)$ and histograms $H(v)$ for six different CTA data sets.

To be able to evaluate the quality of a given transformations objectively, stochastic search algorithms require an appropriate metric. In our first experiments we used a metric M_{hist} that simply calculates the quadratic difference between the normalized histograms, denoted

$$M_{hist}(t) = \sum_{v} \Big| H_{study}(v) - H_{ref}\big(t(v)\big) \Big|^2 \tag{6.5}$$

with H_{ref} and H_{study} referring to the intensity histograms of the reference and the study data set. Note that this metric is monotonically increasing in v and thus can be used as cost function for dynamic programming.

The described optimization procedure is an efficient method to reuse empirical transfer functions designed for a specific data set and to adjust them to different data sets of the same type. Using the histogram based metric however turned out to be not accurate enough in many cases. A metric based on the position function $p(v)$ introduced in Equation 6.3, denoted

$$M_{pos}(t) = \sum_{v} \Big| p_{study}(v) - p_{ref}\big(t(v)\big) \Big|^2 \tag{6.6}$$

has proved superior to the histogram. Figure 6.4 displays the histogram $H(v)$ and the position function $p(v)$ for different data sets that belong to a large clinical study using CTA[1] of the human head. CTA is a common imaging technique in medical routine which involves injection of contrast dye to display vascular structures. The only significant feature in the histogram is a high peak which is caused by the large amount of soft tissue of the brain. This characteristic peak however has a very limited extent. In comparison to this, the position function $p(v)$ shows much more significant features. As we have seen in the previous section, zero-crossings in the position function indicate boundaries in the

[1]CTA = computed tomography angiography, angiography = imaging of vascular structures

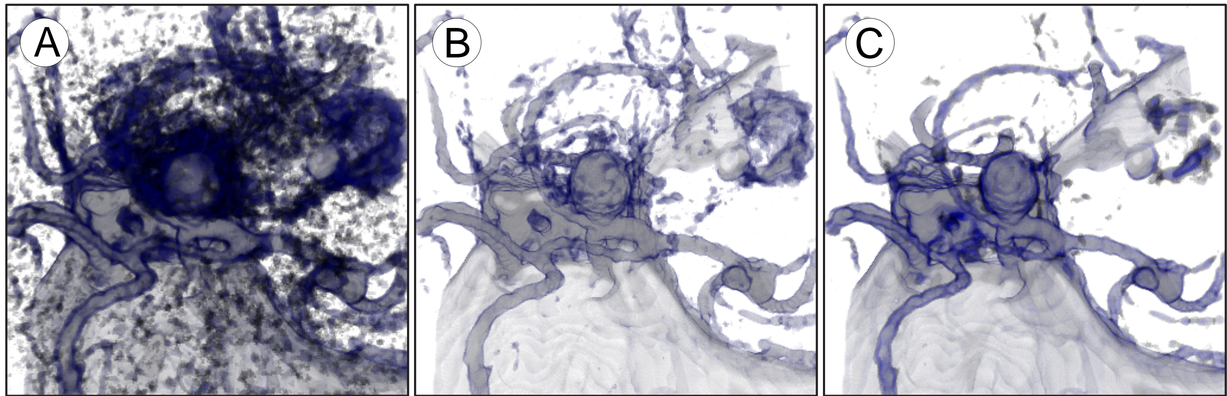


Figure 6.5: Automatic adaptation of transfer function templates: Template without adjustment (A), adjustment based on the histogram (B), adjustment based on the position function (C).

data. The transition between soft tissue and contrast dye is marked by a zero-crossing in $p(v)$ close to the peak in the histogram. To the left of this peak, there is a significant local maximum with (usually) negative value which can be interpreted as a zero-crossing describing the boundary between soft tissue and fluid. Additionally, the boundary to bone structures is clearly indicated by another zero-crossing on the right side of the peak in the histogram. Due to the higher number of significant features the position function $p(v)$ represents a more robust basis for the optimization procedure.

6.4 Conclusion

The most frequently used way of transfer function adjustment implemented in commercial applications is a dialog window which allows the visual editing of the transfer function manually. Among the users, however, there is a high demand for more intuitive ways. Transfer function design based on thumbnail selection is becoming more and more popular and has finally found its way into commercial applications.

Automatic and semi-automatic data-driven methods are still topics of active research. The functionality presented in this chapter was evaluated using CTA data of a large number of patients. The application was tested within the scope of a clinical study of intracranial aneurysms (see Chapter 13). The automatic adjustment was computed using both the histogram based approach and the metric based on the position function. As displayed in Figure 6.5 the approach based on the position function yields significantly better visual results than the histogram based metric. In very few cases, the results of both approaches were almost equivalent. The presented automatic method was integrated into a prototype framework for automated video generation [76] for documentation in clinical practise, which has been tested as part of an internet visualization service [75] provided by the University of Stuttgart.

Chapter 7

Local Illumination

In the previous chapters we have assumed that radiant energy is emitted only by the voxels. Illumination effects caused by external light sources have not yet been taken into account. In this chapter we want to include a local illumination model into our framework of texture based volume rendering. Our motivation for this is primarily the fact that lighting greatly enhances the perception of spatial structures and depth relations. As the terms *lighting* and *shading* are often used as synonyms by mistake, I will refer to the original terminology given in [45]:

Lighting: The lighting model specifies how the intensity at a given point is determined, taking into account various illumination effects from different light sources. *Lambert's*, *Phong's* and *Blinn's* illumination models are the most popular lighting equations.

Shading: The shading model specifies how the lighting model is applied to an object. The shading rules might decide to evaluate the illumination model only at certain points of a surface and interpolate between them. The most popular shading models are *flat shading*, *Gouraud shading* and *Phong shading*.

7.1 Principles

A local illumination model allows 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 [127], which computes the lighting as a linear combination of three different terms, an *ambient*, a *diffuse* and a *specular* term,

$$I_{\text{Phong}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}. \tag{7.1}$$

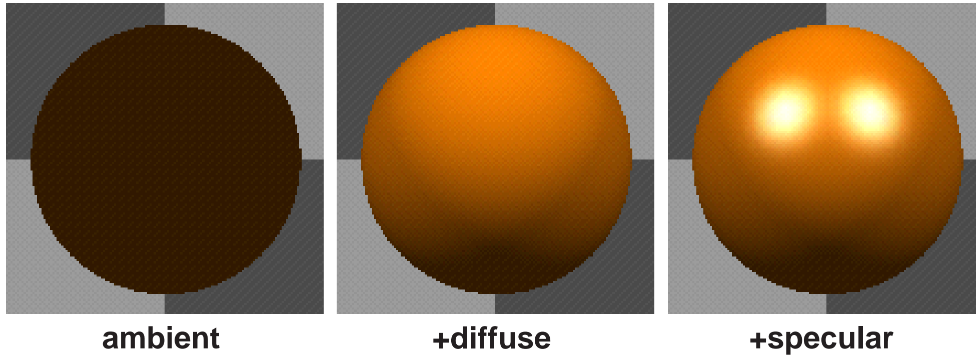


Figure 7.1: The Phong illumination model consists of an ambient, a diffuse and a specular component.

The different components of the Phong illumination model are illustrated in Figure 7.1.

Ambient Light: Ambient illumination is modeled by a constant term,

$$I_{\text{ambient}} = k_a = \mathit{const.} \tag{7.2}$$

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: Reflecting light with equal intensity in all directions is known as *diffuse* or *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

$$I_{\text{diffuse}} = I_p\, k_d\, \cos\varphi = I_p\, k_d\, (\vec{l} \bullet \vec{n}). \tag{7.3}$$

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: 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 $\vec{h}$, resulting in the Blinn-Phong illumination model [45],

$$I_{\text{specular}} = I_p\, k_s\, \cos^n\alpha = I_p\, k_s\, (\vec{h} \bullet \vec{n})^n. \tag{7.4}$$

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.

The Blinn-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 Blinn-Phong illumination model into direct volume rendering, the local shape of the volumetric data set must be described by an appropriate type of vector which substitutes the surface normal. In our concept of classification a single peak in the transfer function corresponds to an isosurface in the volume data set. We know that the normal direction of an isosurface coincides with the direction of the gradient vector of the underlying scalar field. In our illumination model, the gradient vector is thus an appropriate substitute for the surface normal. An algorithm for gradient estimation has already been discussed in Section 5.4.

7.2 Non-Polygonal Isosurfaces

As mentioned above, a sharp peak in the transfer function represents an isosurface or in special cases an isovolume. According to sampling theory, such a peak results in an infinite extent of the transfer function in the frequency domain. Rendering a volume with such a transfer function thus requires to considerably increase the sampling rate to remove slicing artifacts. Without illumination, the resulting image, however, will show nothing but the silhouette of the object as displayed in Figure 7.2 (*left*). It is obvious, that illumination techniques are required to display the surface structures (*middle* and *right*). To accomplish this, we must determine the normal vector of the isosurface for the dot-product computation. As mentioned above, this normal vector coincides with the normalized gradient vector. In consequence, we can pre-compute the gradient vector for every voxel and use it as normal vector in the Blinn-Phong illumination model.

The basic idea of this approach was presented by Westermann and Ertl [174]. In a pre-processing step the gradient vector is computed for each voxel using the central differences

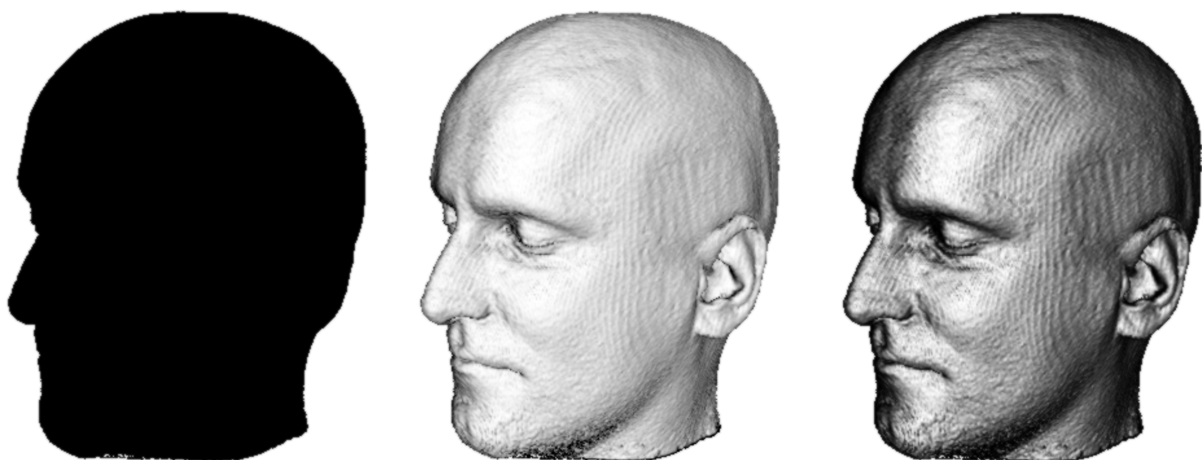


Figure 7.2: Non-polygonal isosurface without illumination (left), with diffuse illumination (middle) and with specular light (right)

```
0    glDisable(GL_BLEND);
1    // Enable Alpha Test for isosurface
2    glEnable(GL_ALPHA_TEST);
3    glAlphaFunc(GL_EQUAL, fIsoValue);
```

Listing 7.1: OpenGL setup for the alpha test.

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:

$$\begin{array}{rclcl} \nabla I & = & \begin{pmatrix} I_x \\ I_y \\ I_z \end{pmatrix} & \begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array} & \begin{array}{c} \mathtt{R} \\ \mathtt{G} \\ \mathtt{B} \end{array} \\ & & & & \\ I & & & \longrightarrow & \mathtt{A} \end{array} \tag{7.5}$$

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. A post-interpolative transfer function for opacity with a single peak from 0 to 1 can be efficiently implemented using the standard OpenGL per-fragment operations. As we have seen in Section 2.1.3, the alpha test allows the discarding of incoming fragments conditional on the outcome of a comparison of the incoming alpha value with a user-specified reference value. In our case the alpha channel contains the scalar intensity value and the 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 Listing 7.1. Due to the visual artifacts caused by the high frequency component of the transfer function, the sampling rate must be extremely increased to obtain satisfying images. Alternatively the alpha test can be set up to check for `GL_GREATER` or `GL_LESS` instead of `GL_EQUAL` (line 3), allowing a considerable reduction of the sampling rate.

After one rendering pass of the volume data set with the alpha test enabled, the frame buffer contains the correct isosurface with the normal vectors encoded in the RGB components. In the original approach, the image contained in the frame buffer is read out and piped again through the rasterization stage. In this second pass the dot product with the direction vector of a diffuse directional[1] light is computed by exploiting the color matrix OpenGL extension (`SGI_color_matrix`) provided by SGI. This original approach by Westermann and Ertl was restricted to diffuse illumination of non-polygonal isosurfaces with a single directional light source. Based on their implementation, Meißner et al. [107] have provided a method to include diffuse illumination into semi-transparent volume rendering with classification. In their approach several passes through the rasterization hardware

[1] A *directional light* is a light source, which is sufficiently far away from the illuminated object, so that the light rays can assumed to be parallel. In this case the direction of light is constant for the whole scene.

```
0   #if defined GL_EXT_texture_env_dot3
1      // enable the extension
2      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  GL_COMBINE_EXT);
3      // preserve the alpha value
4      glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_REPLACE);
5      // enable dot product computation
6      glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,   GL_DOT3_RGB_EXT);
7      // first argument: light direction stored in primary color
8      glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT,   GL_PRIMARY_COLOR_EXT);
9      glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT,  GL_SRC_COLOR);
10     // second argument: voxel gradient stored in RGB texture
11     glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT,   GL_TEXTURE);
12     glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT,  GL_SRC_COLOR);
13  #endif
```

Listing 7.2: OpenGL setup for the the dot product extension.

led to a significant loss in the overall rendering performance. Dachille et al.[25] have proposed an approach that uses 3D-texture hardware interpolation and software shading and classification. In the following chapters we examine illumination techniques based on pixel shaders for non-polygonal isosurface display as well as for rendering of semi-transparent volumes. Preliminary results of the described approaches have been published in [134].

7.3 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. As we have seen, this mechanism is provided by the pixel-shaders functionality of modern consumer graphics boards. In the original algorithm for rendering non-polygonal isosurfaces the second rendering pass was required to compute the dot product between the light direction and the local surface normal. To get rid of this second rendering pass, it is necessary to compute the dot product directly within the texture unit.

7.3.1 Implementations

Dot product computation does not require to use complex OpenGL extensions such as the NVidia register combiners. A simple functionality that supports dot product calculation is provided by the OpenGL extension `EXT_texture_env_dot3`. This extension to the standard OpenGL texture environment defines a new way to combine the color and texture values during texture applications. As displayed in Listing 7.2, 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 (line 7). In the

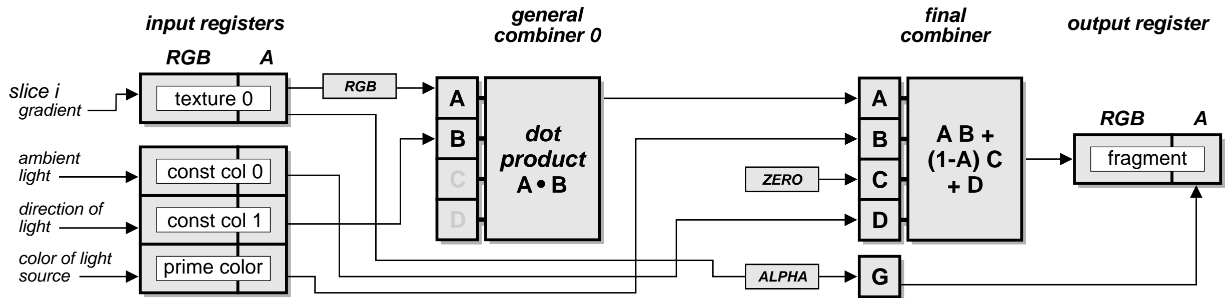


Figure 7.3: Combiner setup for fast rendering of shaded isosurfaces using 3D-textures.

sample code the RGBA quadruplets (`GL_SRC_COLOR`) of the primary color and the texel color are used as arguments. The described implementation represents a more efficient implementation of non-polygonal shaded isosurfaces as the original algorithm. However, it does neither account for the specular illumination term, nor for multiple light sources.

More flexible illumination with multiple light sources can be achieved using the NVidia register combiners or similar extensions. The register combiner setup for the diffuse term is displayed in Figure 7.3. The dot product between the direction of light is computed at the first general combiner stage. At the final combiner stage the result from the dot product computation is multiplied with the color of the diffuse light source I_p (including the diffuse reflection coefficient k_d) and the ambient color I_{ambient} is simply added. The result is passed to the alpha test. In the setup displayed in Figure 7.4, the illumination calculation is combined with the interpolation of intermediate slices for a 2D-multi-texture based approach. In this case, the first general combiner is used to interpolate an intermediate slice images and the dot product computation is performed subsequently by the second general combiner stage.

If we examine the register combiner setup for the last two configurations, we notice that

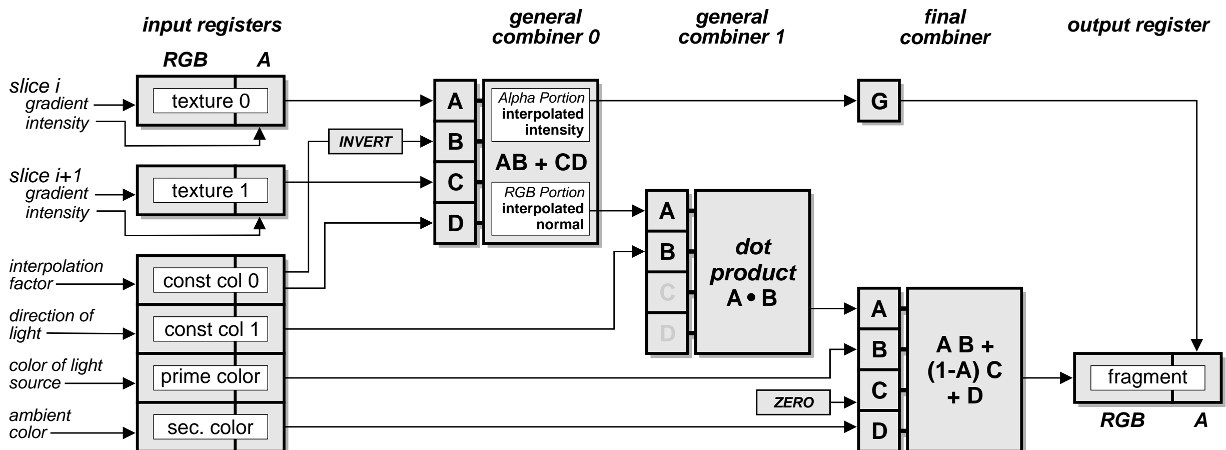


Figure 7.4: Combiner setup for fast rendering of shaded isosurfaces.

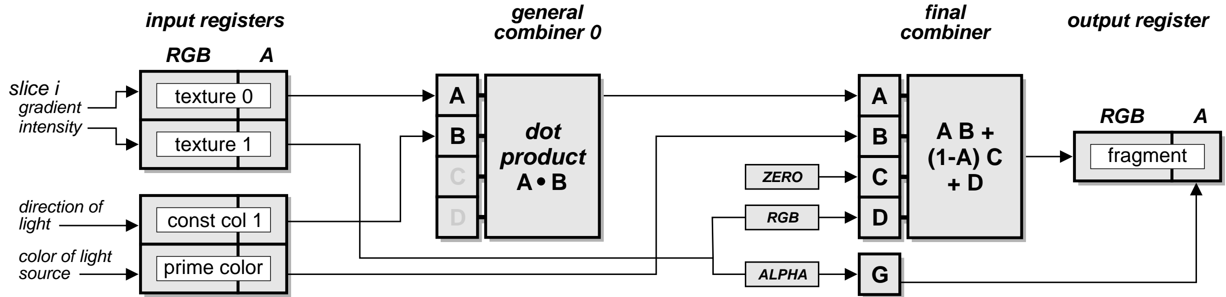


Figure 7.5: Combiner setup for rendering semi-transparent volumes with local diffuse illumination.

the variables C and D have not yet been used at the combiner stage which computes the dot product. Since a general register combiner is capable of performing two independent dot product calculations in parallel, the unused portion of the combiner can be utilized to compute diffuse illumination from another directional light source. If more than two light sources are required, an additional general combiner stage can be activated. Depending on the maximum number of general combiner stages provided by the underlying hardware architecture, the illumination calculations for multiple independent light sources are possible.

The specular illumination term requires the computation of the n-th power of the dot product with the halfway vector $\vec{h}$. Specular illumination can be accomplished using multiple register combiner stages in a similar way. In this case the direction of light is simply replaced by the halfway vector. The n-th power of the resulting dot product can be computed by a sequence of component-wise multiplications performed by succeeding combiner stages. Provided that there are enough available general combiner stages, the complete Phong illumination model for multiple light sources can be computed on the fly during rasterization. Examples of non-polygonal shaded isosurfaces are displayed in Figure 7.6.

In contrast to the original approach to rendering non-polygonal isosurfaces, the lighting techniques presented in this chapter can also be used for the illumination of semi-transparent volume data. A modification of the register combiner setup is displayed in Figure 7.5. The alpha test is again replaced by the usual alpha blending setup. A separate 3D-texture is now used to store the intensity values. Using pre- or post-classification as outlined in Chapter 5, these intensity values are mapped to color and opacity values representing the emission and absorption coefficients. The emission coefficient in this case specifies the emitted (ambient) light on a per-voxel basis. The absorption coefficient is used for numerical integration by alpha blending. The diffuse illumination term is simply added. The specular term is not included in the displayed combiner setup, but can easily be accomplished as described above. Since for semi-transparent volume rendering the intensity value is stored in a separate texture, the `Alpha` component of the gradient texture is free and can be used to store the gradient magnitude for classification with gradient

weighted opacity as described in Section 5.4.

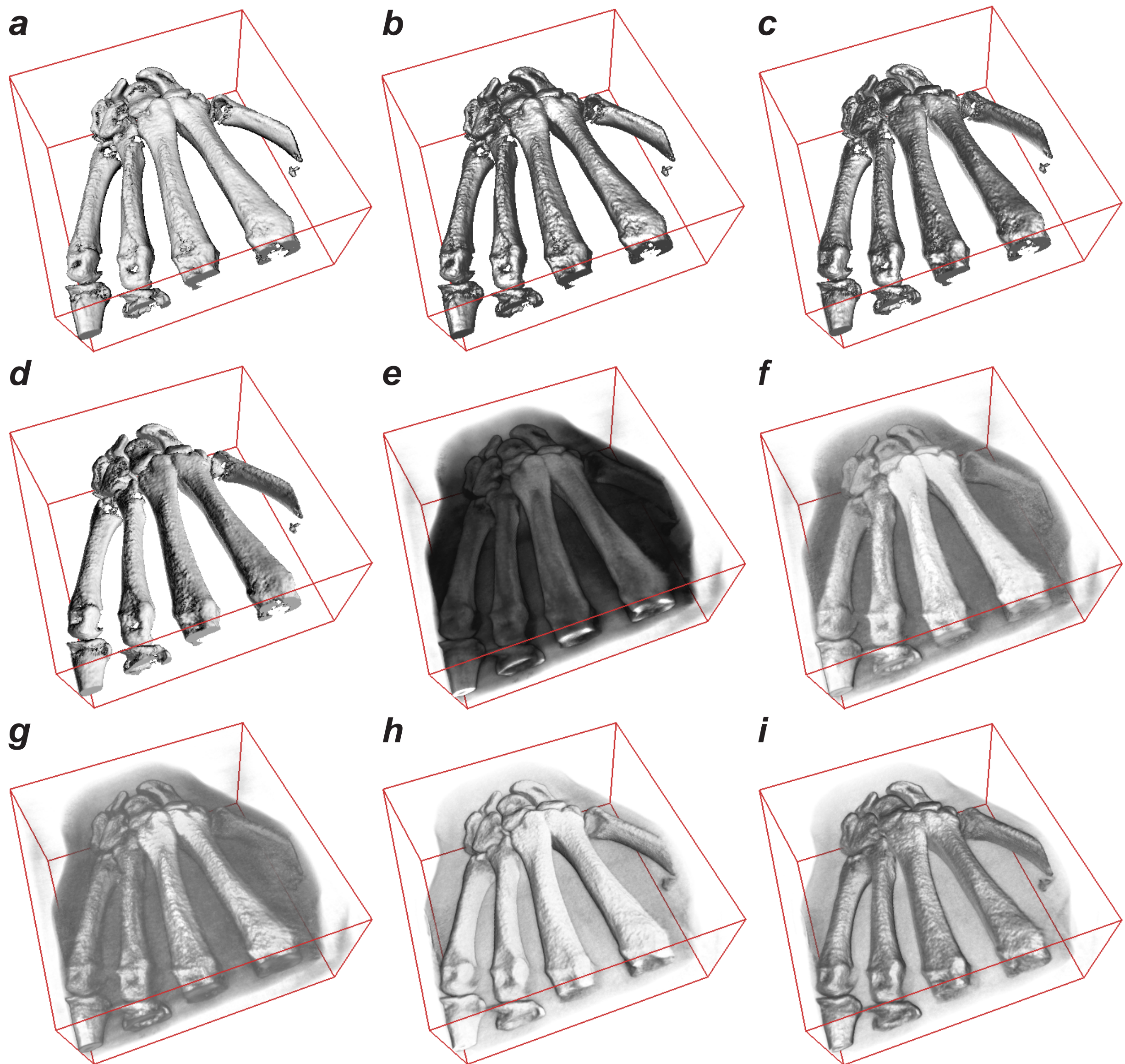


Figure 7.6: CT data of the human hand bones with examples of different illumination effects. Non-polygonal isosurface with diffuse illumination *(a)*, with specular illumination *(b)*, with both diffuse and specular illumination *(c)* and with two diffuse light sources *(d)*. Semi-transparent volume data without illumination *(e)*, with diffuse illumination *(f)* and with specular illumination *(g)*. Semi-transparent volume data with gradient weighted opacity and diffuse *(h)* or specular *(i)* illumination.

7.4 Reflection Maps

The examples of shaded semi-transparent volume data displayed in Figure 7.6 demonstrate the flexibility of per-pixel illumination calculations. However, if the illumination scenario gets 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. Reflection mapping is a two-stage process that involves the construction of a reflection map as a pre-computation step. In effect a reflection map caches the incident illumination from all directions at a single point in space.

Closely related to the diffuse and specular terms of the Phong illumination model, reflection mapping can be performed with either diffuse reflection maps or reflective *environment* maps. The indices into a diffuse reflection map are directly computed from the normal vector, whereas the coordinates for a reflective 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. Self-reflection and self-shadowing is not taken into account, resulting in technically wrong reflection maps for concave objects. The idea of reflection mapping has been first suggested by Blinn [10]. The term *environment mapping* was coined by Greene [56] in 1986. According to the parameterization of the normal direction, reflection mapping approaches can be categorized into longitude-latitude maps, spherical maps and cube maps.

As we have seen in the previous sections, the Phong illumination model uses information about the local shape of the object by means of normal vectors. Every illumination term which is a function of a normalized vector can be efficiently pre-computed in a reflection map. Normalized vectors are fully determined by two angles, longitude (from 0° to 360°) and latitude (from −90° to 90°). The idea of such a reflection map is to cache the incident illumination in a 2D-texture with the x and y axes representing the longitude and latitude

Figure 7.7: Example of a spherical environment map.

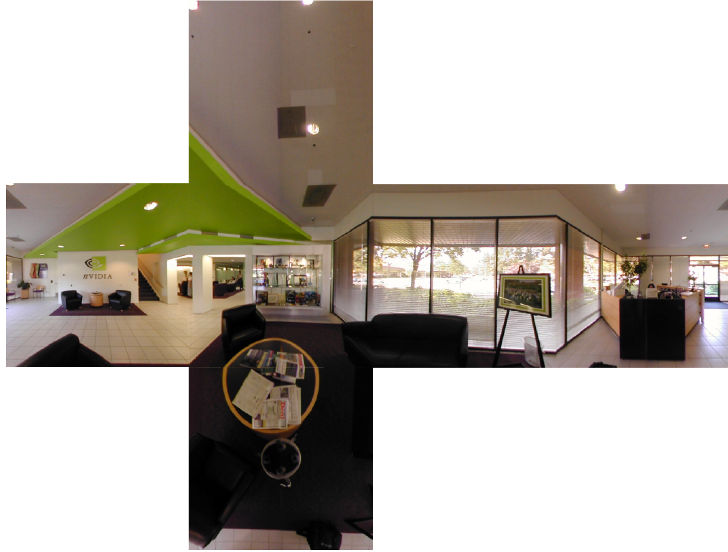


Figure 7.8: Example of a environment cube map.

angles, respectively. An alternative parameterization is used for the construction of a spherical reflection map as displayed in Figure 7.7. A circular texture image is generated by orthographic projection of the reflection as seen in the surface of a perfect mirror sphere. In practise, spherical environment maps are obtained by traditional ray-tracing or by taking photographs of shiny spheres.

A third alternative parameterization of the normal direction is used in order to construct a *cube map* as displayed in Figure 7.8. 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.

The implementation of cubic diffuse and reflective environment maps can be accomplished using the OpenGL extension `GL_NV_texture_shader`. The setup is displayed in Listing 7.3. Four texture units are involved in this configuration. Texture 0 is a 3D-texture which contains the pre-computed gradient vectors. In texture unit 0 a normal vector is interpolated from this texture (line 6). 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 reflection maps the viewing direction must also be taken into account. In the OpenGL extension, the local 3×3 modeling matrix and the camera position is specified as texture coordinates for the texture units 1, 2 and 3. From this information the GPU constructs the viewing direction and valid normal vectors in world coordinates in texture unit 1 (lines 8–15). The diffuse and the reflective cube maps are applied in texture unit 2 (lines 18–26)

and texture unit 3 (lines 29–37), respectively. Since the normal vectors are specified as color components within the unsigned range [0, 1], they must be expanded internally to a signed range [−1, 1] (lines 14, 25 and 34). As a result, the texture registers 2 and 3 contain the appropriately sampled diffuse and reflective environment map. These values are finally combined to form the final color of the fragment using the register combiner extension. According to the assigned reflection coefficients k_d and k_s of the diffuse and the reflective term, different material properties can be simulated as displayed in Figure 7.9.

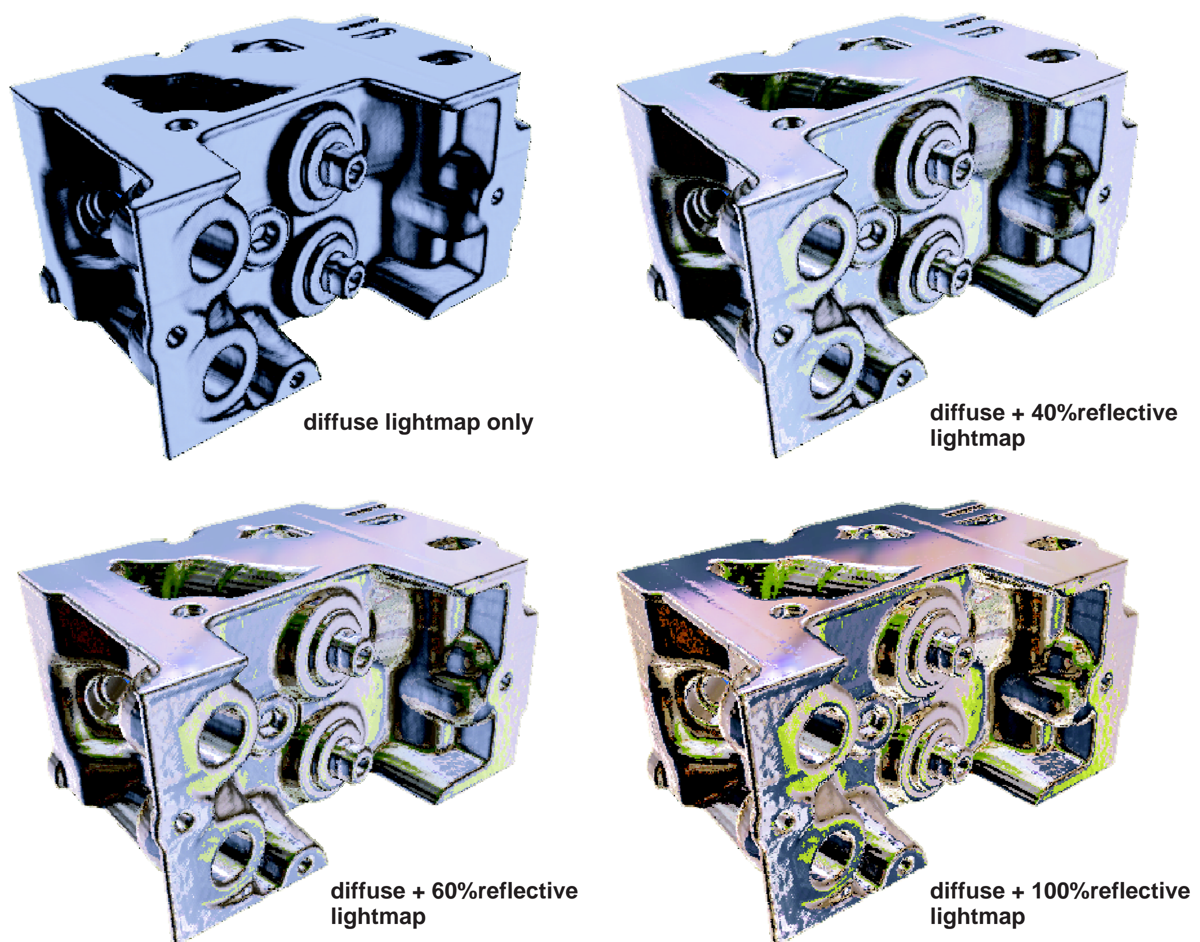


Figure 7.9: Non-polygonal isosurface with diffuse and specular lightmaps of the engine block data set. The reflectivity of the surface increases from $k_s = 0$ at top left to $k_s = 1$ at bottom right.

```
0   #if defined GL_NV_texture_shader
1      // texture unit 0 - sample normal vector from 3D-texture
2      glActiveTextureARB(GL_TEXTURE0_ARB);
3      glEnable(GL_TEXTURE_3D_EXT);
4      glEnable(GL_TEXTURE_SHADER_NV);
5      glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV, GL_TEXTURE_3D);
6
7      // texture unit 1 - dot product computation
8      glActiveTextureARB( GL_TEXTURE1_ARB );
9      glEnable(GL_TEXTURE_SHADER_NV);
10     glTexEnvi(GL_TEXTURE_SHADER_NV,
11               GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_NV);
12     glTexEnvi(GL_TEXTURE_SHADER_NV,
13               GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
14     glTexEnvi(GL_TEXTURE_SHADER_NV,
15               GL_RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV, GL_EXPAND_NORMAL_NV);
16
17     // texture unit 2 - diffuse cube map
18     glActiveTextureARB( GL_TEXTURE2_ARB );
19     glEnable(GL_TEXTURE_SHADER_NV);
20     glBindTexture(GL_TEXTURE_CUBE_MAP_EXT, m_nDiffuseCubeMapTexName);
21     glTexEnvi(GL_TEXTURE_SHADER_NV,
22               GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV);
23     glTexEnvi(GL_TEXTURE_SHADER_NV,
24               GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
25     glTexEnvi(GL_TEXTURE_SHADER_NV,
26               GL_RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV, GL_EXPAND_NORMAL_NV);
27
28     // texture unit 3 - reflective cube map
29     glActiveTextureARB( GL_TEXTURE3_ARB );
30     glEnable(GL_TEXTURE_CUBE_MAP_EXT);
31     glBindTexture(GL_TEXTURE_CUBE_MAP_EXT, m_nReflectiveCubeMapTexName);
32     glTexEnvi(GL_TEXTURE_SHADER_NV,
33               GL_SHADER_OPERATION_NV, GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV);
34     glTexEnvi(GL_TEXTURE_SHADER_NV,
35               GL_PREVIOUS_TEXTURE_INPUT_NV, GL_TEXTURE0_ARB);
36     glTexEnvi(GL_TEXTURE_SHADER_NV,
37               GL_RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV, GL_EXPAND_NORMAL_NV);
38  #endif
```

Listing 7.3: OpenGL setup for the reflection mapping using diffuse and reflective cube maps.

7.5 Discussion

In this chapter we have seen different approaches to local illumination in direct volume rendering. A new rendering technique for non-polygonal isosurfaces was introduced using the OpenGL alpha test to substitute a transfer function with a high peak. Although

the presented shading methods for rendering semi-transparent volume with classification can be used in order to obtain equivalent visual results, the approach to non-polygonal isosurfaces is more efficient since the alpha test does not require the reading of pixel values from the frame buffer as it is necessary for alpha blending.

The presented techniques for per-pixel illumination are very efficient for a small number of directional light sources whose direction can be changed interactively. Spot lights and light source attenuation is not taken into account in this approaches. The linear interpolation between normal vectors, which is performed by the texturing unit does not deliver properly normalized vectors. This problem is similar to the inaccuracy related to Gouraud shading in comparison to Phong shading. However, since the distance between neighboring voxels projected onto the image plane is usually much smaller than the size of triangles used in Gouraud shading, the effect of this inaccuracy is hardly visible.

For illumination scenarios which are too complex for on-the-fly computation, diffuse and reflective reflection maps are used. Since the hardware guarantees an internal normalization of the linearly interpolated normal vectors, reflection mapping also removes the Gouraud shading problem described above. Reflection maps account for non-directional point light sources, spot lights and light source attenuation. Spherical maps and longitude-latitude maps however suffer from nonuniform sampling. These effects are less disturbing for cubic environment maps. Apart from the high number of texture lookups that must be performed for diffuse and reflective mapping, the main drawback is that changing the illumination environment completely invalidates the reflection maps. The re-computation of a reflection map is rather expensive.

In the presented method for shading of semi-transparent volumes, the pre-computed gradient vectors may not accurately represent the local shape of the object, if the transfer function for opacity is not taken into account. For a post-interpolative transfer function, however, this would require also to perform the gradient estimation in hardware, which by now is computationally intractable. Gradient estimation after the application of a classification function can be accomplished with special purpose hardware such as the VolumePro board.

Chapter 8

Performance Measurement

In this chapter we analyze the performance of several texture-based solutions for volume rendering described in the previous chapters. We want to examine and compare the results of this measurement for different implementations on general purpose hardware. The performance for hardware-accelerated polygon rendering is technically limited by three independent factors which greatly depend on the underlying hardware architecture.

- The **geometry limit** is the maximum number of triangles per second that can be processed by the transform & light unit during geometry processing. Rendering complex geometric models consisting of a high number of small triangles is usually a geometry-limited process. Modern graphic boards have a built-in 3D graphics processor (GPU) which offloads the geometry processing from the central processing unit (CPU).

- The **pixel fill rate** is measured by counting the maximum number of fragments per second that can be piped through the texturing unit. On older PC graphics architectures only rasterization and per-fragment operations are implemented in hardware, while geometry processing is still performed by the CPU. For textured geometric models that consist of a small number of large textured polygons, performance is dominated by the efficiency of the rasterization subsystem. The pixel fill rate is also influenced by the transfer bandwidth between the GPU and its local video memory.

- The speed at which the GPU can read data from main memory instead of local video memory is often a performance bottleneck. This is generally referred to as the **memory bandwidth**. For rendering virtual scenes which consist of many different objects and textures it is thus necessary to sort the objects by texture in order to minimize texture swapping and optimize texture cache coherence.

For direct volume rendering applications the displayed geometry (slice polygons) is rather simple, so the processing speed of the geometry unit does not have significant influence on the overall performance. At the beginning of Chapter 3 however, we have put down a note that the performance of texture-based volume rendering is limited by either the number

of interpolation operations or the memory bandwidth. The number of interpolations that can be performed within a second is determined by the pixel fill rate. In consequence the fill rate is expected to be the limiting factor for volumes that entirely fit into local video memory. For larger volume data sets, performance is limited by the available memory bandwidth from main memory to the graphics boards.

8.1 Architectures

The concept of a graphics pipeline, which we used in Chapter 2 to define the term general purpose hardware, can be found in many different architectures, ranging from low-cost PC graphics boards to high-end workstations and expensive server architectures. Before we analyze the results of the performance measurement, we will have a closer look at the different hardware architectures and its specific capabilities.

8.1.1 Consumer PC Hardware

A personal computer (PC) is defined as a system that is designed for being used by only one person at a time [128]. PCs usually do not have to share resources, such as storage devices and processing subsystems with other systems. The most popular examples for PCs are Intel x86-compatible systems and Apple Macintosh computers. The architectural design of a typical PC system is outlined in Figure 8.1. One of the main characteristics of a modern PC is the *accelerated graphics port* (AGP). The AGP is a platform bus specification that enables high performance graphics capabilities.

On former PC architectures the graphics board was directly connected to the PCI[1] local bus. The main drawback of such systems was that the graphics board must share the available bus bandwidth with several other devices such as disk drives, audio devices and ethernet cards. To these ends the AGP interface was introduced as a dedicated high-speed

[1]PCI = Peripheral Component Interconnect

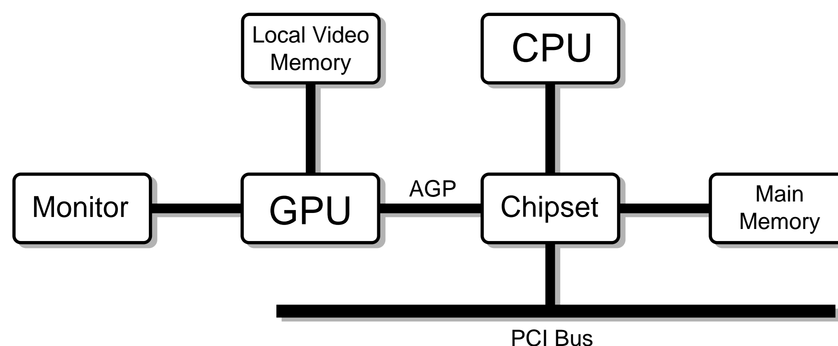


Figure 8.1: Modern Consumer PC architecture with *Accelerated Graphics Port (AGP)*.

bus directly between the chip set and the graphics controller. As a result, the bandwidth-intensive traffic to and from the graphics board was removed from the limitations of the PCI bus.

The graphics processing unit (GPU) is a high-performance 3D processor that integrates the entire 3D pipeline or parts thereof in hardware. For 3D graphics application the capabilities of the GPU are decisive for the overall performance in a typical system.

8.1.1.1 NVidia's GeForce Family GPUs

In August 1999, NVidia released a new GPU chip, the GeForce 256, which was the first graphics processor for consumer video cards with an integrated geometry processing unit. The GeForce 256 had a maximum polygon throughput of 15 million triangles per second. With four pixel pipelines working at a core clock speed of 120 MHz, the theoretical pixel fill rate of the GeForce 256 was $4.8 \cdot 10^8$ pixels/second. The performance limitation of the first GeForce boards released was the slow SDRAM[2] running at a clock speed of 166 MHz. This limitation was relieved with the introduction of Double Data Rate (DDR) SDRAM at 150 MHz resulting in an effective memory bandwidth of 300 MHz. The GeForce 256 GPU was capable of rendering two multi-textures in one rendering pass. It also introduced the concept of register combiners by providing two general combiner stages.

In April 2000, NVidia released the second generation of the GeForce family GPUs, the GeForce 2 GTS which was equipped with a second texture unit. The GeForce 2 GTS running at a clock speed of 200 MHz had a theoretical pixel fill rate of $8 \cdot 10^8$ pixels per second. The GeForce 2 Ultra is an optimized version of the GeForce 2 GTS running at a core clock speed of 250 MHz. A modified version for the use in mobile computers was released with the GeForce 2 Go GPU. As the third generation of the GeForce, the GeForce 3 was the first NVidia GPU that supported 3D-textures and dependent texture lookups. The GeForce 3 has a theoretical pixel fill rate of $2.8 \cdot 10^9$ pixels per second.

8.1.1.2 ATI's Radeon Family GPUs

In October 2000, ATI released the Radeon GPU which was the first consumer graphics processor that supported 3D textures. Some additional features such as depth-buffer compression (HyperZ) are clearly aimed at the game market. The Radeon chip has two pixel pipelines running at 183 MHz and supports 3 multi-textures in a single pass. The Radeon 8500 released in October 2001 is the high-end GPU from the second generation of the Radeon product line. It has four pixel pipelines running at 275 MHz which results in a theoretical pixel fill rate of $1.5 \cdot 10^9$ pixels per second.

The detailed configurations of the different PC systems used throughout our experiments are displayed in Table 8.1. For the performance measurement we used systems with GeForce 256, GeForce 2 Ultra and GeForce 3 hardware as well as a *Dell Inspiron* notebook with a GeForce 2 Go processor (PC System F). The performance of 2D and 3D-texture based volume rendering was also evaluated for the Radeon and Radeon 8500 GPU.

[2] SDRAM = Synchronized Dynamic Random Access Memory

	CPU	clock rate	main memory	GPU	video memory
PC System A	Intel Pentium III	500 MHz	512 MB	NVidia GeForce 256	32 MB
PC System B	AMD Athlon	1 GHz	512 MB	NVidia GeForce 2 Ultra	64 MB
PC System C	AMD Athlon	1 GHz	1 GB	NVidia GeForce 3	64 MB
PC System D	Intel Pentium III	500 MHz	512 MB	ATI Radeon	64 MB
PC System E	AMD Athlon	1 GHz	1 GB	ATI Radeon 8500	64 MB
PC System F	Intel Pentium III	1 GHz	256 MB	GeForce 2 Go	32 MB

Table 8.1: Specifications of the PC systems used for the experiments

8.1.2 SGI Graphics Workstations and Servers

Apart from the inexpensive PC systems described above, the volume rendering approaches have also been evaluated on SGI graphics workstations and high-end servers. One of the main characteristics of such systems compared to a PC is the different memory bus structure which allows the access of data by different computing subsystems in a more efficient way. The workstations that were available for the performance measurement comprise an SGI O2, an SGI Octane 2 visual workstation and an SGI Onyx system.

8.1.2.1 SGI O2

The SGI O2 is a mid-price UNIX graphics workstation based on a unified memory architecture (UMA). In such a system local memory buffers, such as a separate video memory, do not exist. A unified memory contains all data, including the frame buffer, depth-buffer, textures, executables and application data. The UMA architecture as outlined in Figure 8.2 allows all the different subsystems to share data without the necessity to copy buffers from

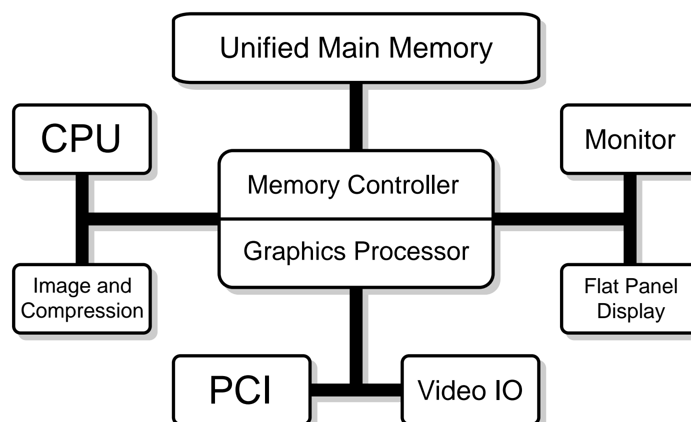


Figure 8.2: The *Unified Memory Architecture (UMA)* as implemented in the SGI O2.

one subsystem to another. The system that we used for the performance measurement was equipped with a MIPS R10000 micro processor running at 195 MHz and 256 MB of main memory.

8.1.2.2 SGI Octane 2

The SGI Octane 2 visual workstation is a high end graphics computer especially developed for scientific computing and engineering. The main characteristic of the SGI Octane 2 architecture is the memory crossbar for high-speed data transfer. A memory crossbar replaces the conventional system bus architecture and allows direct and dynamic connections between any two computing subsystems. The VPro graphics Octane 2 has a pixel fill rate of 425 million pixels per second and delivers $7.4 \cdot 10^6$ triangles per second. The system we used throughout the performance measurement was equipped with a single MIPS R12000 micro processor running at a clock speed of 400 MHz, a V12 graphics subsystem and 2 GB of main memory.

8.1.2.3 SGI Onyx

In addition to the high-end workstations, an SGI Onyx deskside server was available for the performance measurement. The Onyx architecture is based on an SGI EBus architecture, a 256-bit high-speed bus for efficient data sharing. In combination with a 40-bit address bus, the system enables fast block data transfers between multiple CPUs and memory boards. The Onyx server used for the performance measurement was equipped with 4 MIPS R10000 processors running at 194 MHz, 896 MB of main memory and a RealityEngine II graphics subsystem.

8.2 Performance Analysis

Before we start with the performance measurement, we should be aware that the validity of the results are very limited. This is due to the fast technological progress of consumer graphics hardware driven by the mass market of computer games and entertainment software. Board revisions and driver updates can have significant influence on the measured frame rate. In this context the registration of general trends and tendencies is more important than the exact values of the measurement. The different implementations of direct volume rendering used throughout the performance measurement share a common C++ source code which was recompiled for the different platforms. All measured frame rates refer to a rendering window of 580×427 pixels without decoration. Images of the data sets used throughout the performance measurement are displayed in Appendix A.

As we have seen in previous chapters, the implementations of texture-based volume rendering approaches strongly depend on the capabilities of the underlying graphics hardware. The only approach that runs on all the different platforms is the original 2D-texture based approach described in Chapter 3.1. In order to get a first impression of the general

efficiency of the different platforms, we start with an examination of the performance for 2D-texture based volume rendering.

Figure 8.3 displays the measured frame rate for the 2D-texture based approach with

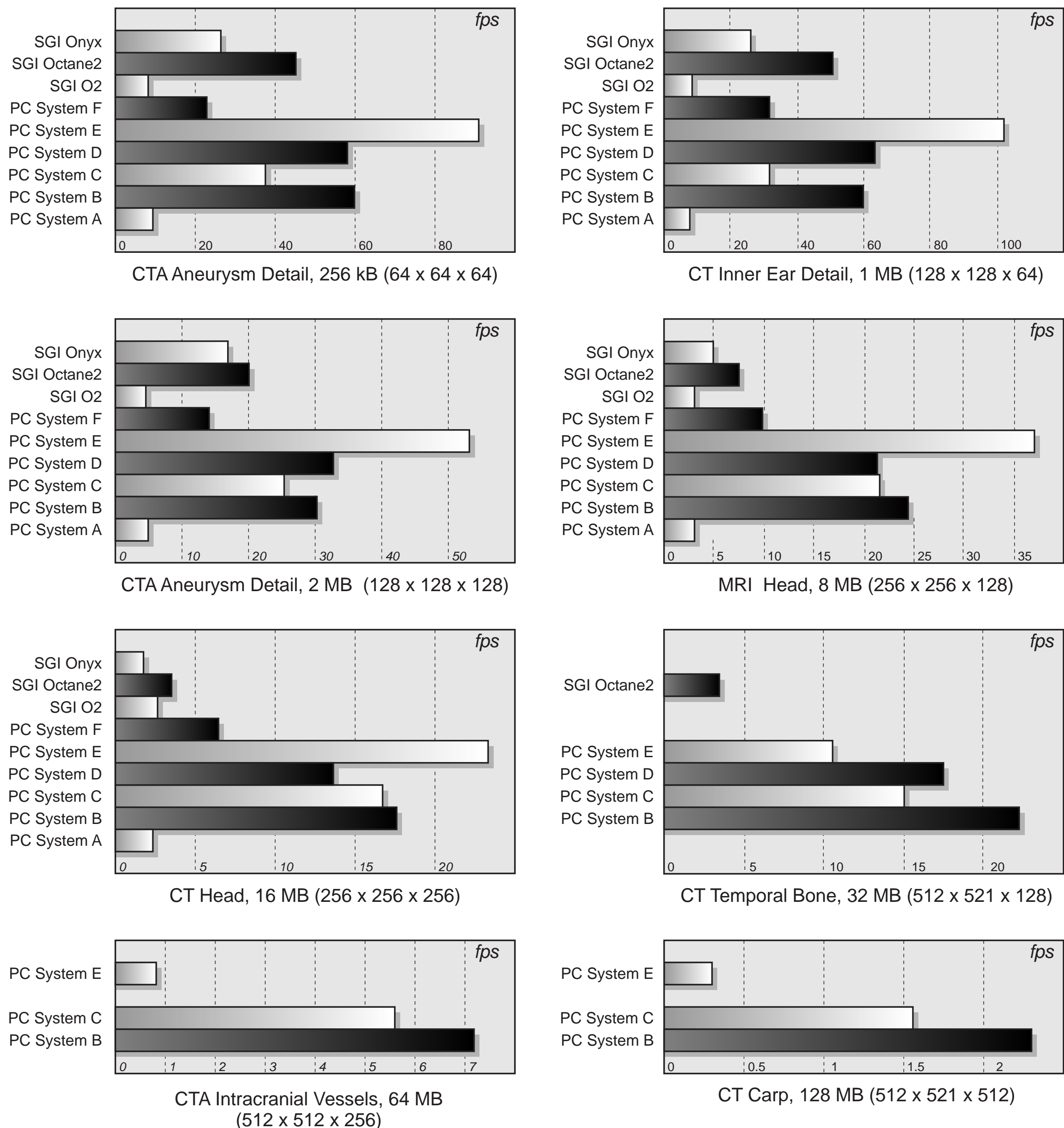


Figure 8.3: Comparison of the performance of 2D-texture based volume rendering for several data sets on different hardware architectures. The most promising hardware architectures are the second generation consumer PC boards. They prove superior to the high-end graphics workstations SGI Onyx and Octane 2.

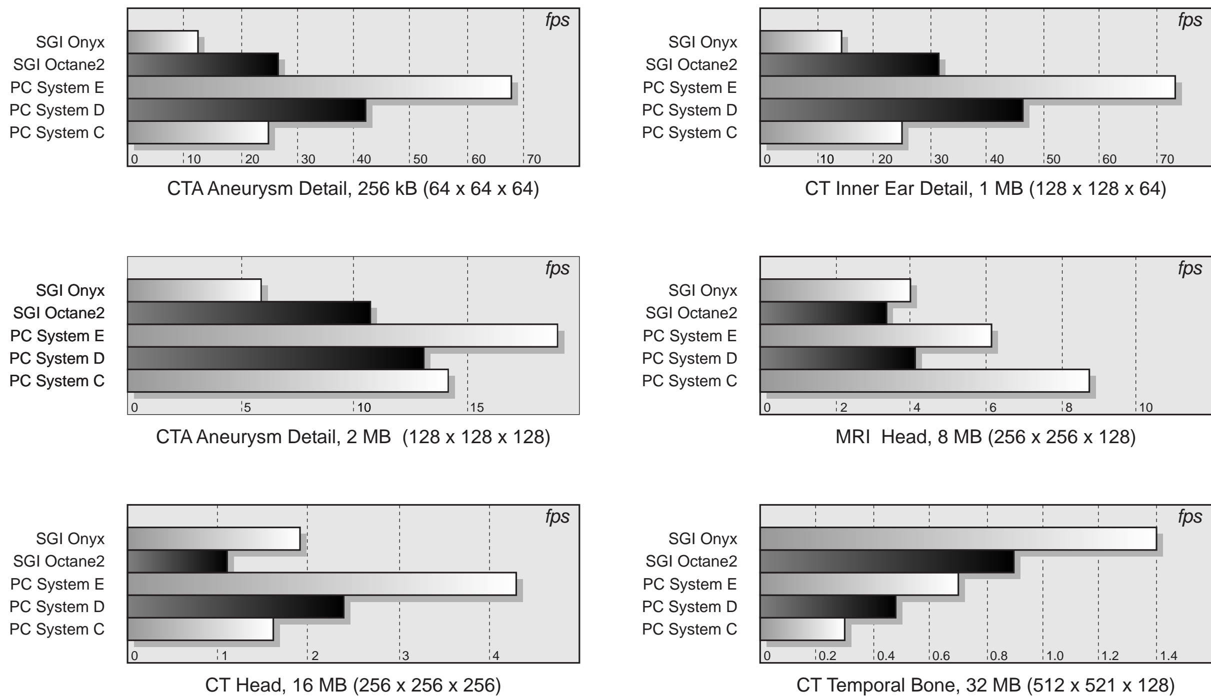


Figure 8.4: Comparison of the performance of different hardware architectures for 3D-texture based volume rendering. The ATI Radeon 8500 delivers the highest frame rate for 3D-textures. With large volume data sets however ($\geq$ 16MB) the frame rate drops down to about 1 frame per second an all architectures due to the performance penalty introduced by inefficient bricking.

different data sets. Obviously, for small data sets the pixel fill rate is the limiting factor. The influence of the memory bandwidth increases with the size of the data sets. The frame rate delivered by a GeForce 256 (PC System A) is comparable to the O2 workstation. In this case, the benefit of the unified memory architecture of the O2 cannot prevail against the optimized design of the consumer boards. The high end workstations SGI Octane 2 and the SGI Onyx deliver satisfactory results for small data sets. Although both architectures have an optimized memory bus, the limited pixel fill rate cannot cope with the increasing size of the data sets. Excellent results are delivered by the second generation consumer graphics boards such as the GeForce 2 Ultra (PC System B) and the Radeon. The high pixel fill rate of the Radeon 8500 delivers impressive frame rates for volume data of moderate size. For large data sets the performance is dominated by the higher memory bandwidth of the GeForce 2 Ultra and the GeForce 3.

3D-textures are supported by the GeForce 3 (PC System C), by the Radeon family boards (PC Systems D and E) as well as by SGI Onyx and Octane 2. Figure 8.4 shows a comparison of the measured frame rate on these architectures. For small data sets ($\leq$ 2 MB) the highest frame rate is delivered by the Radeon 8500 (PC System E). With growing size of the data the Radeon 8500 still dominates in terms of performance, however

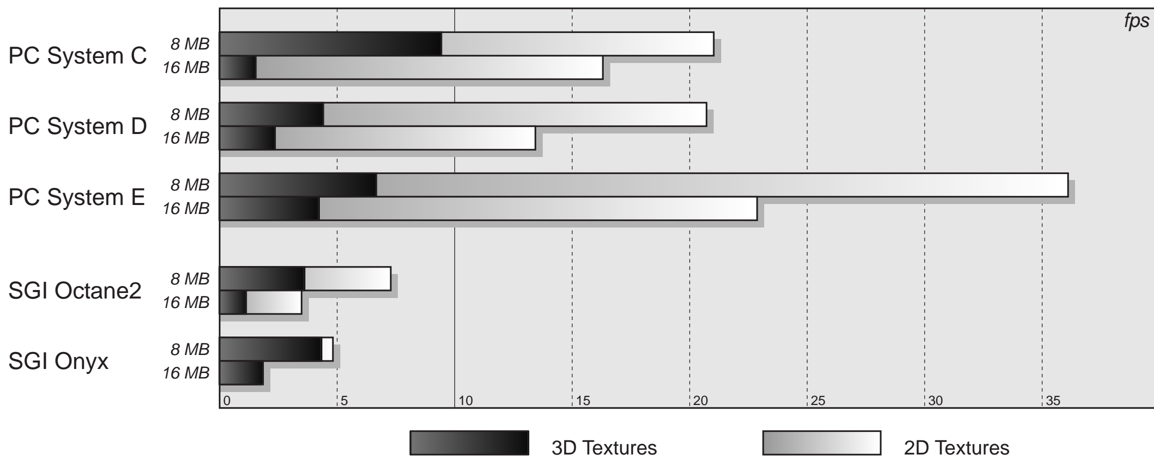


Figure 8.5: Comparison of the performance of 2D- and 3D-texture based volume rendering for different hardware architectures. Although 2D-texture based methods require to keep three copies of the data set in memory, they provide higher frame rate due to the more efficient memory management.

in some cases severe texture errors are visible, apparently due to an early driver release. For large data sets ($\geq$ 16 MB) bricking of the 3D-textures is required and the performance breaks down to less than one frame per second. In this case the more efficient memory bus of the Onyx and Octane 2 leads to a marginally higher frame rate. The bricking of data sets which do not fit entirely into the local video memory seems to be extremely inefficient. As can be seen in Figure 8.5, the 2D-texture based method delivers a higher frame rate in all cases. Note that the performance of the 2D-textures based method is significantly higher *although* it requires to keep three copies of the volume in memory. This is mainly a matter of efficient memory management and load balancing. Obviously, swapping only small portions of the video memory (e.g. single 2D-textures) is more efficient in terms of load balancing than swapping the entire texture memory at a time.

In theory, if a rendering process is solely limited by the pixel fill rate, doubling the sampling rate (the number of slices) should result in a decrease of the frame rate by a factor of one half. On the other hand if the process is completely dominated by the limited memory bandwidth, increasing the sampling rate should have little effect on the performance. In typical volume rendering approaches both the fill rate and the memory bandwidth influence the performance. The main drawback of the 3D-texture based approach in combination with large data sets is the extremely inefficient memory management. If the data set must be divided into smaller portions that fit entirely into local texture memory (*bricking*, see Section 3.2.2), the GPU is stalled until the whole memory is exchanged. In consequence methods that use 2D-textures are more efficient, since they allow the exploitation of both the available memory bandwidth and the pixel fill rate in parallel.

The volume rendering method based on 2D-multi-textures as introduced in Chapter 4 greatly enhances image quality by removing visual artifacts, while preserving the efficient

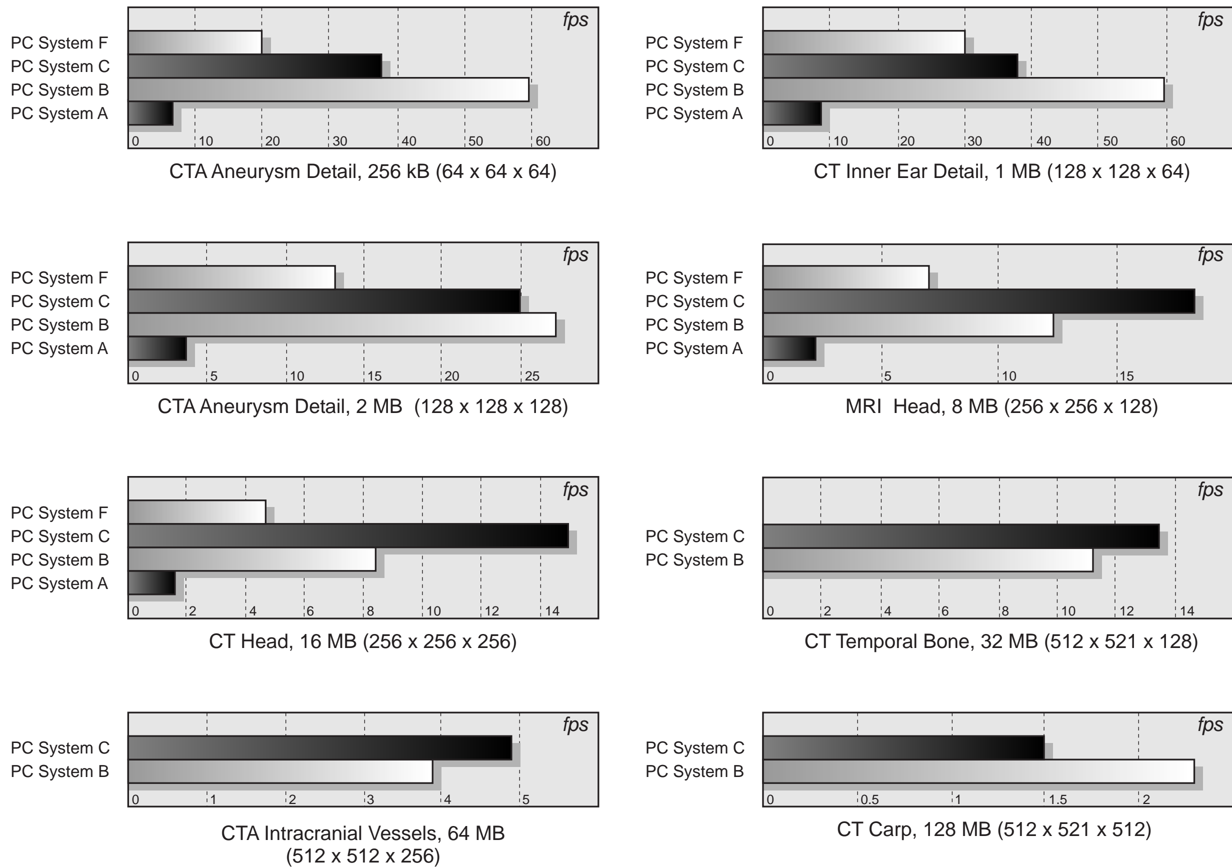


Figure 8.6: Comparison of the performance of different hardware architectures for 2D-multi-texture based volume rendering with different data sets. As a result of the more efficient memory management, the frame rates are significantly higher than 3D-texture based solutions. Note that considerable frame rates are achieved even by the mobile computer with the GeForce 2 Go (PC System F).

memory management. Figure 8.6 shows the results of the performance measurement for 2D-multi-texture based volume rendering. In the comparison of the different NVidia GPUs it is no surprise that boards with the GeForce 2 Ultra (PC System B) and the GeForce 3 boards (PC System C) deliver the highest frame rate. These architectures also allow the rendering of large data sets (128MB, 512^3) at $1-2$ frames per second. Note that for data sets of moderate size ($<$ 16MB) interactive frame rates are delivered even by the Dell Notebook with the GeForce 2 Go chip (PC System F). This represents the first solution of interactive high-quality volume rendering on mobile computers.

As we have seen throughout the discussion of transfer functions in Chapter 5, the possibility to increase the number of slices is important to allow for a post-interpolative transfer function of high frequency. Figure 8.7 displays the measured frame rate for different sampling rates. In this context, a sampling rate of 100% refers to a slice distance of half the

length of a voxel diagonal. As expected, for very small data sets doubling the sample rate results in half the frame rate. The performance is thus solely limited by the available pixel fill rate. With increasing size of the data the performance is more and more influenced by the memory bandwidth.

The type of transfer function also has some influence on the overall performance. Figure 8.8 compares three different implementations. The first implementation uses an RGBA texture in combination with pre-classification based on the OpenGL pixel transfer (Sec-

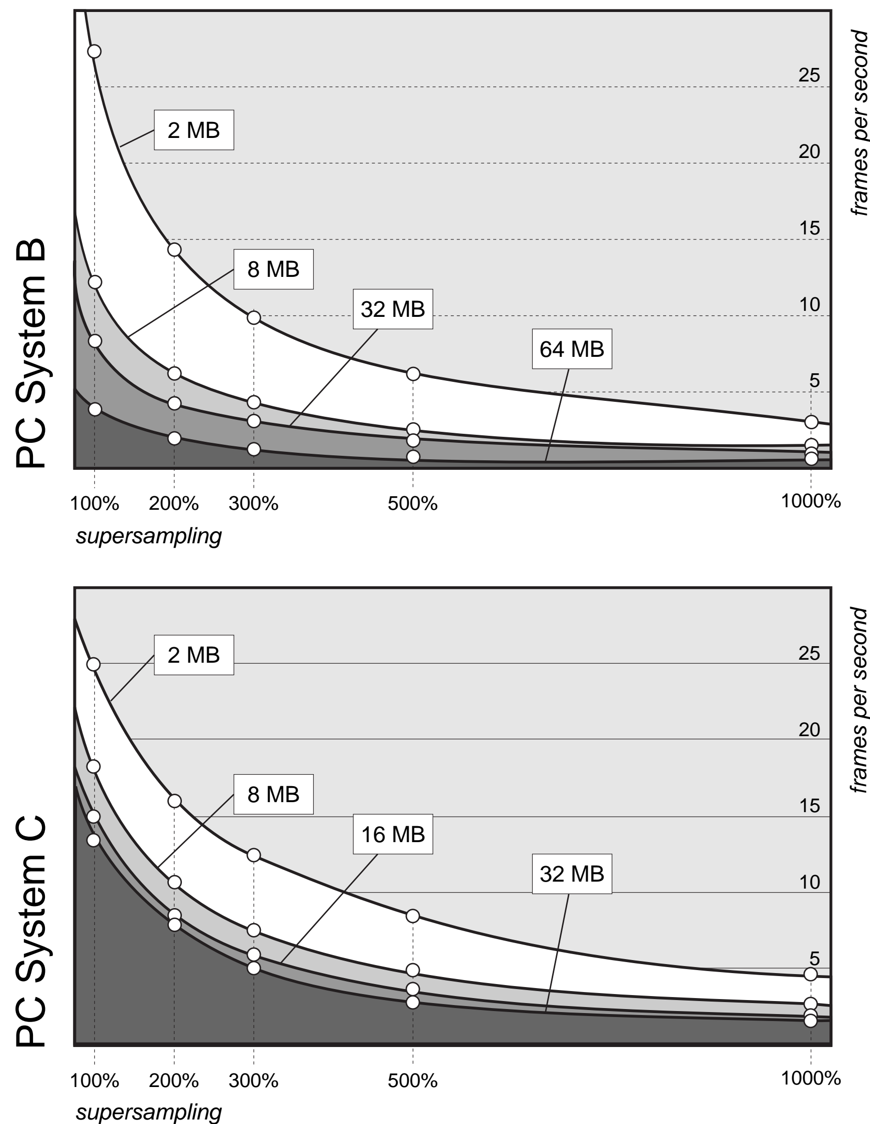


Figure 8.7: Comparison of the performance of the 2D-multi-texture based approach for different levels of supersampling. As expected for a fill rate limited process, doubling the sample rate while rendering very small data sets results in half the frame rate. With the increasing size of the data sets, the limited memory bandwidth shows its effect.

tion 5.2.1.1). The second implementation stores color indices in the texture maps and uses a dependent texture lookup for post-classification (Section 5.2.2.2). The third implementation uses two dependent texture lookups from an RGBA texture in order to accomplish a four-dimensional transfer function (Section 5.4). The measured frame rates refer to an interactive update of the rendering window *without* modification of the transfer function. Depending on the specific implementation modifying the transfer function will add some

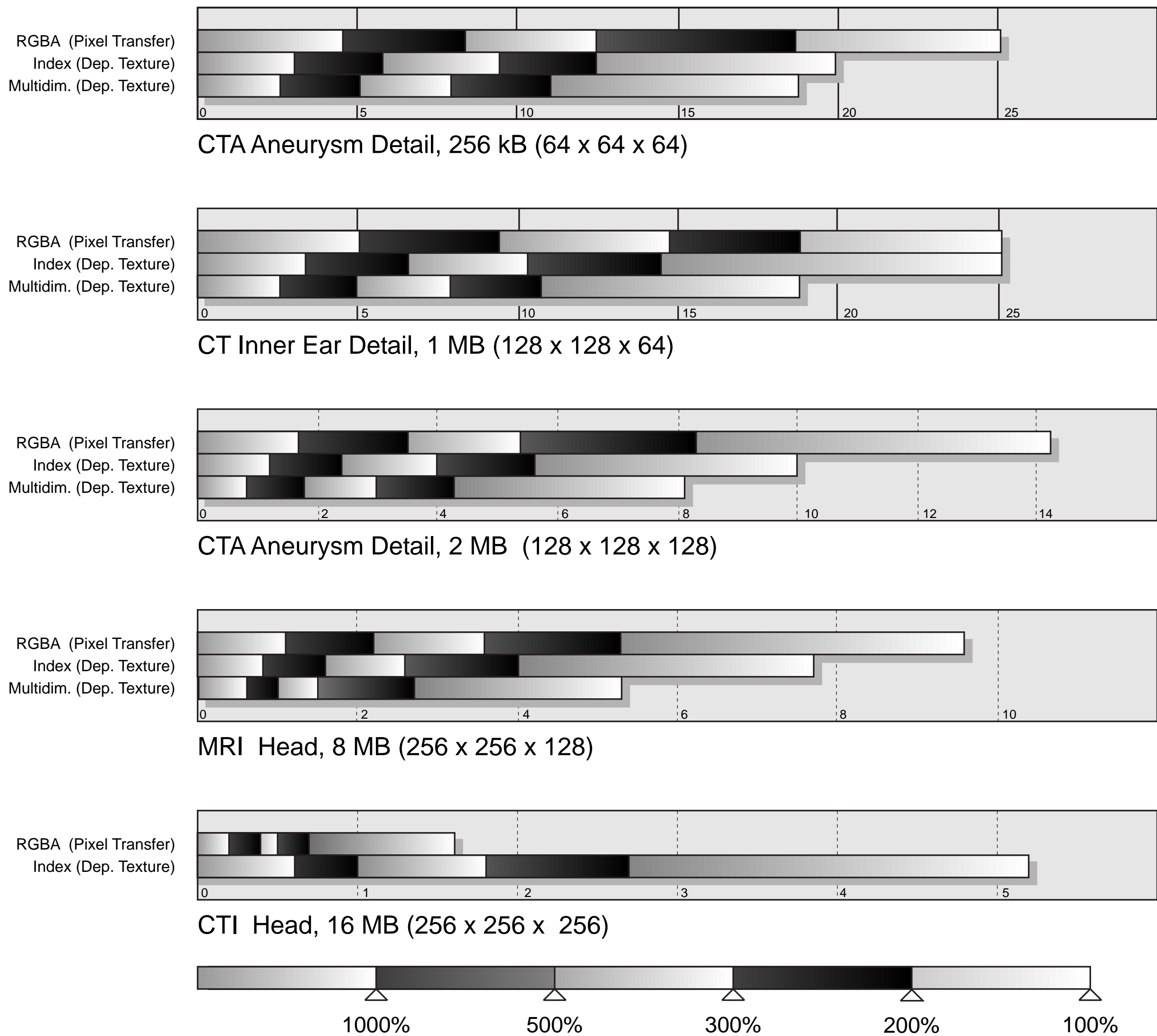


Figure 8.8: Comparison of the different implementations of transfer functions on PC System C. The implementation based on an RGBA texture generated by pre-classification using pixel transfer delivers higher frame rates compared to (multiple) dependent texture lookups using color indices. For larger data sets however, the large memory footprint of the RGBA texture significantly degrades performance.

amount of computation time. Although the RGBA texture allocates more local video memory, for a small data set the resulting frame rate is higher compared to the color index texture. This is due to the additional texture lookup that must be performed to extract the RGBA value from the dependent texture. This effect is even stronger for the four-dimensional transfer function with the two dependent texture lookups. For data sets of moderate size (> 8MB), the large memory footprint of the RGBA texture leads to a significant loss in performance.

In order to evaluate the performance for volume rendering techniques with local illumination effects as introduced in Chapter 7, we analyze the resulting frame rates for different implementations of classified and shaded volumes as well as for non-polygonal isosurfaces. The results of the measurement are displayed in Figure 8.9. The shading techniques comprise a 2D-multi-texture based implementation of per-pixel illumination (see Section 7.3) using register combiners (RC), a 2D-multi-texture based multi-pass method that renders the classified slice image and the illumination term in separate rendering passes (MP RC) and a 3D-texture based implementation using a spherical reflection map. (see Section 7.4). The techniques for non-polygonal isosurface rendering comprise a 2D-multi-texture based implementation using register combiners (RC), a 3D-texture based implementation using the dot product extension (DOT) and a 3D-texture based implementation using a diffuse and a specular cube map.

Basically, the performance measurement of shading techniques exemplifies one more time the benefit of more efficient memory management of 2D-textures compared to 3D-textures. The additional dependent texture lookup for reflection maps results in a significant loss in performance. The most efficient illumination technique is the per-pixel illumination technique based on register combiners. The multi-pass method should only be used if there are not enough multi-texture stages available.

8.3 Conclusion

In this chapter we have compared the measured frame rates for multiple implementation on different hardware platforms. The time when interactive high-quality volume visualization was restricted to expensive graphics workstations such as the SGI Onyx and Octane is definitely over. Driven by the mass market of graphics hardware, the technological progress of PC consumer boards has lead solutions that have proven superior to the high-end workstations both in terms of performance and flexibility.

Although the 2D-multi-texture based solution requires three copies of the data set to be kept in memory, the resulting performance is significantly higher than equivalent implementation using 3D textures. In spite of the higher memory requirements the 2D-multi-texture based solution has proven superior for rendering large volume data sets. This is mainly due to the more efficient memory management and to the optimized load balancing which allows the available pixel fill rate and the memory bandwidth to be used in parallel.

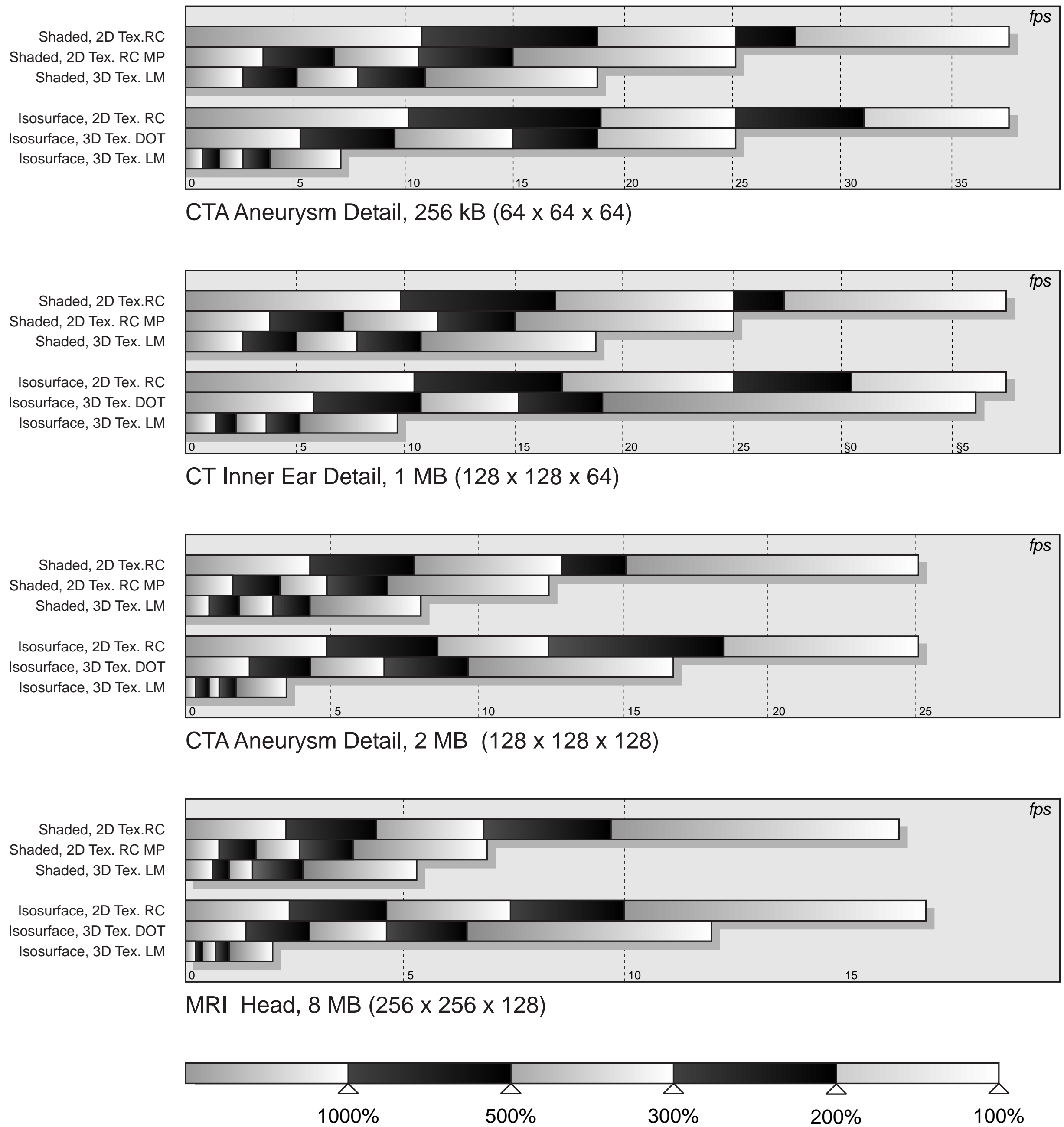


Figure 8.9: Comparison of the performance of illumination techniques on PC System C.

Chapter 9

Extensions

In this chapter I am going to report on a number of supplements and enhancements for texture based volume rendering that exploit hardware features in efficient ways. A large number of related algorithms and extensions of texture based volume rendering has been developed in recent years. Maybe the most remarkable supplement has been proposed by Klaus Engel et al. [38] who suggested to pre-compute the exact ray integration in a lookup table and to access this table by the use of dependent textures. In a different context, Lum et al. [97] have extended texture based volume rendering to time-varying volume data by using discrete cosine transformation for a temporal encoding and a decoding algorithm based on texture palettes. Apart from these approaches, a selection of interesting supplements to texture based volume rendering is presented in the following sections.

9.1 Multi-Texture Speedup

In the volume rendering approach described in Chapter 4, we utilize multi-textures for trilinear interpolation. As a result the image quality was significantly improved by reducing interpolation artifacts. In addition to the optimization of image quality, multi-texturing can be used to enhance the performance of direct volume rendering.

The idea of this approach is to reduce the necessary number of slice polygons by mapping the textures of multiple slice images onto a single surface [134]. Depending on the available number of multi-textures, the rasterization load can be significantly reduced. If there are n independent multi-textures supported in hardware, only every n-th slice polygon is drawn and textured with the image information of n consecutive slice images. During rasterization, the n texture images are combined by the texture application unit. The resulting fragment is finally blended into the frame buffer. Depending on how many clock cycles of the GPU are required for the combination of n texture images, rendering time can *theoretically* be reduced by a factor of $1/n$ in the optimal case. In a bandwidth limited procedure however this benefit will usually not be that significant. However, even for a bandwidth-limited process, the frame buffer read operations which are required for back-to-front alpha blending are reduced by the same factor.

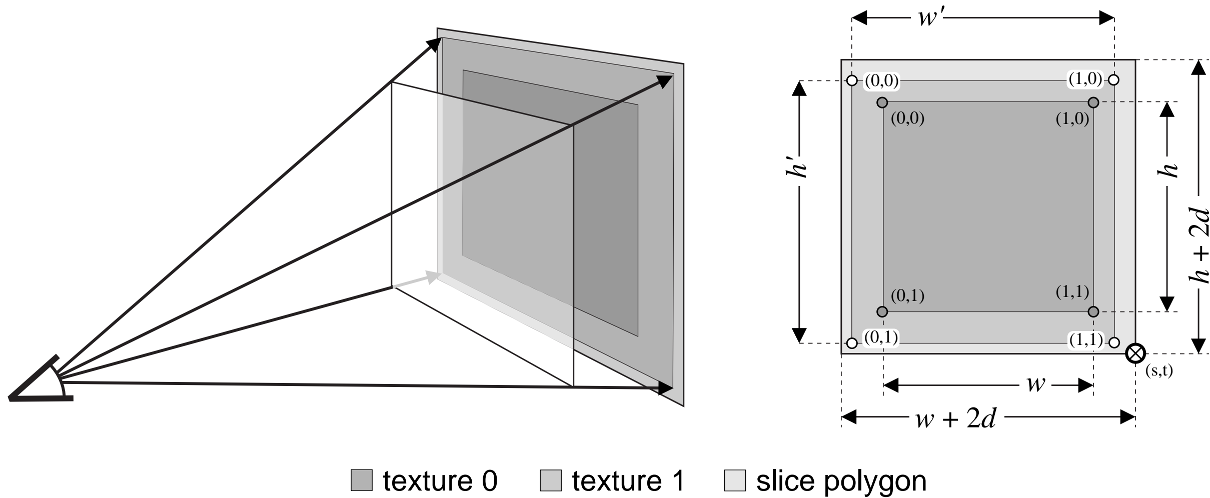


Figure 9.1: Rendering two slices with a single slice polygon: Projection of a slice image i onto the polygon of slice $i+1$.

If we simply map the texture images of the slices 0 to $(n-1)$ onto the slice polygon of texture 0, only texture 0 would be drawn at the correct position. The other textures would be mapped incorrectly due to the projective displacement of the viewing transformation. In order to compensate this incorrect mapping, we have to shift the texture coordinates of the textures 1 to $(n-1)$ to account for the original vertex positions. The correct texture coordinates are determined by projecting the slices 1 to $(n-1)$ onto slice plane 0 using the eye position as center of projection[1] as displayed in Figure 9.1 (left). Without loss of generality, we restrict our further considerations to only two textures. Texture 0 and texture 1 are mapped onto slice 0 and the rendering of slice 1 is skipped.

As a first step, we increase the width w and the height h of the original slice polygon 0 by adding twice[2] the slice distance d as outlined in Figure 9.1 (right). For a field of view of less than 90°, this ensures that no slice image will be projected onto an area outside the polygon. Subsequently, the texture coordinates of both texture units must be adapted to the modified vertex positions. For the marked vertex of the slice polygon in Figure 9.1 (right), the texture coordinates (s^0, t^0) of texture 0 are given by

$$s^0 = 1 + \frac{d}{w} \qquad \text{and} \qquad t^0 = 1 + \frac{d}{h}\,. \tag{9.1}$$

The texture coordinates of the slice 1 are determined by projecting the corners of the texture onto the polygon. A point $\vec{v}$ on slice 1 is mapped to a point $\vec{v}\,'$ on slice 0 using the eye position $\vec{v}_{\text{eye}}$ as center of projection. Assuming that the slices are parallel to the

[1] This of course applies only in case of perspective projection.

[2] For n multi-textures, $2 \cdot d \cdot (n-1)$ is added to the width and height of the polygon.

z-plane, $\vec{v}\,'$ results in

$$\vec{v}\,' = \vec{v} + \frac{d}{p_z}\,\vec{p} \qquad \text{with} \qquad \vec{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = (\vec{v} - \vec{v}_{\text{eye}}) \tag{9.2}$$

The texture coordinates (s^1, t^1) for texture unit 1 are thus given by

$$s^1 = 1 + \frac{d}{w'}(1 + \frac{p_x}{p_z}) \qquad \text{and} \qquad t^1 = 1 + \frac{d}{h'}(1 + \frac{p_y}{p_z}) \tag{9.3}$$

with the projected width and height

$$w' = (1 + \frac{d}{p_z})\,w \qquad \text{and} \qquad h' = (1 + \frac{d}{p_z})\,h. \tag{9.4}$$

Note that this technique requires the correctly handling of texture coordinates less than zero and greater than one. The opacity of texture samples for coordinates outside the range of [0,1] should be set to zero. This requires a mechanism that allows the clamping of textures to a fixed value as provided by the OpenGL extension `SGIS_texture_edge_clamp`. If this extension is not available, standard OpenGL texture clamping can be used with the border of the texture initialized with opacity values of zero.

Apart from the adjustment of texture coordinates, mapping multiple textures onto a single slice image will only lead to correct visual results, if the multi-texture samples are combined within the texture application unit in exactly the same way as at the alpha blending stage of the per-fragment operations.

As we have seen in Section 3.1.2, color values of the incoming fragment (the source) are combined with the color values at the corresponding frame buffer position (the destination) according to a function specified in the alpha blending stage. The blending function `glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)` used in Section 3.1.2 for back-to-front compositing, computes the new frame buffer value C'_{dest} as a function of texture color C_{tex0} and opacity A_{tex0} and the previous frame buffer value C_{dest},

$$C'_{\text{dest}} = C_{\text{tex0}} \cdot A_{\text{tex0}} + C_{\text{dest}}\,(1 - A_{\text{tex0}}). \tag{9.5}$$

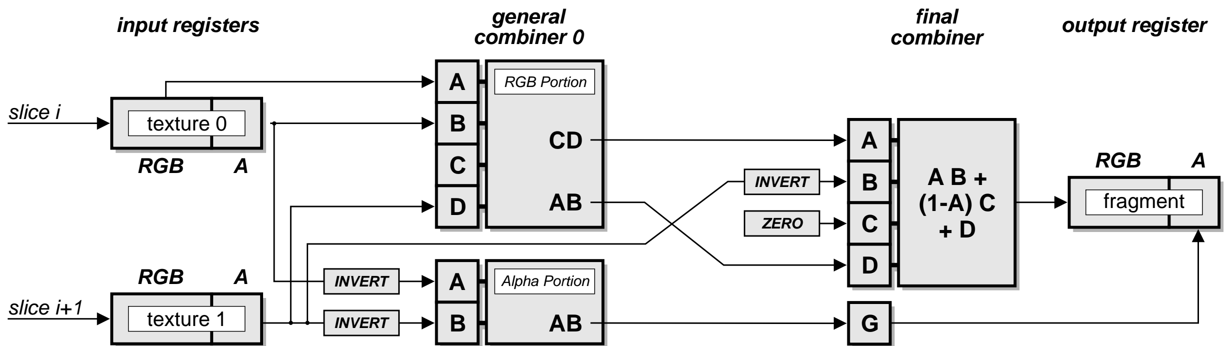


Figure 9.2: Combiner setup for correct blending of two slices with one polygon

This value C'_{dest} is written into the frame buffer replacing the previous value C_{dest}. In the next blending step it will be used in turn as the destination value C_{dest}. The calculation of two blending steps at the same time can be written as

$$\begin{aligned} C''_{\text{dest}} &= C_{\text{tex1}} \cdot A_{\text{tex1}} + C'_{\text{dest}}(1 - A_{\text{tex1}}) \quad = \\ &= C_{\text{tex1}} \cdot A_{\text{tex1}} + \big(C_{\text{tex0}} \cdot A_{\text{tex0}} + C_{\text{dest}}(1 - A_{\text{tex0}})\big)(1 - A_{\text{tex1}}) \quad = \\ &= C_{\text{tex1}} \cdot A_{\text{tex1}} + C_{\text{tex0}} \cdot A_{\text{tex0}}(1 - A_{\text{tex1}}) + C_{\text{dest}}(1 - A_{\text{tex0}})(1 - A_{\text{tex1}}). \end{aligned}$$

For correct blending results of two textures this equation must be computed within the texture application unit. Figure 9.2 shows a possible solution achieved with the NVidia register combiners. The RGB-portion of general combiner 0 is programmed to calculate

$$C^{(0)}_{\text{src}} = (C_{\text{tex0}} \cdot A_{\text{tex0}}) \quad \text{and} \quad C^{(1)}_{\text{src}} = (C_{\text{tex1}} \cdot A_{\text{tex1}}). \tag{9.6}$$

Additionally, the Alpha-portion of this combiner is used to compute

$$A^{(0)}_{\text{src}} = (1 - A_{\text{tex0}})(1 - A_{\text{tex1}}). \tag{9.7}$$

The output of the RGB-portion are routed into the final combiner stage, which calculates the resulting RGB value

$$\begin{aligned} C_{\text{src}} &= C^{(0)}_{\text{src}}(1 - A_{\text{tex1}}) + C^{(1)}_{\text{src}} \quad = \\ &= C_{\text{tex0}} \cdot A_{\text{tex0}} \cdot (1 - A_{\text{tex1}}) + C_{\text{tex1}} \cdot A_{\text{tex1}}. \end{aligned}$$

The result of the Alpha-portion is directly used as alpha value

$$A_{\text{src}} = A^{(0)}_{\text{src}} = (1 - A_{\text{tex0}})(1 - A_{\text{tex1}}) \tag{9.8}$$

of the output register. In the per-fragment operations, the alpha blending equation in Listing 3.2 is modified to

```
glBlendFunc(GL_ONE, GL_SRC_ALPHA),
```

resulting in a compositing equation according to

$$\begin{aligned} C''_{\text{dest}} &= C_{\text{src}} \cdot 1 + C_{\text{dest}} \cdot A_{\text{src}} \quad = \\ &= C_{\text{tex1}} \cdot A_{\text{tex1}} + C_{\text{tex0}} \cdot A_{\text{tex0}}(1 - A_{\text{tex1}}) + C_{\text{dest}}(1 - A_{\text{tex0}})(1 - A_{\text{tex1}}). \end{aligned}$$

Using this register combiner setup we obtain exactly the same blending results for multi-texturing as for rendering two separate polygons using single textures. As reported in [134], the presented method for multi-texture speedup leads to an increase in the frame rate by a factor of 1.8 for data sets of moderate size on a GeForce 256 with two multi-textures.

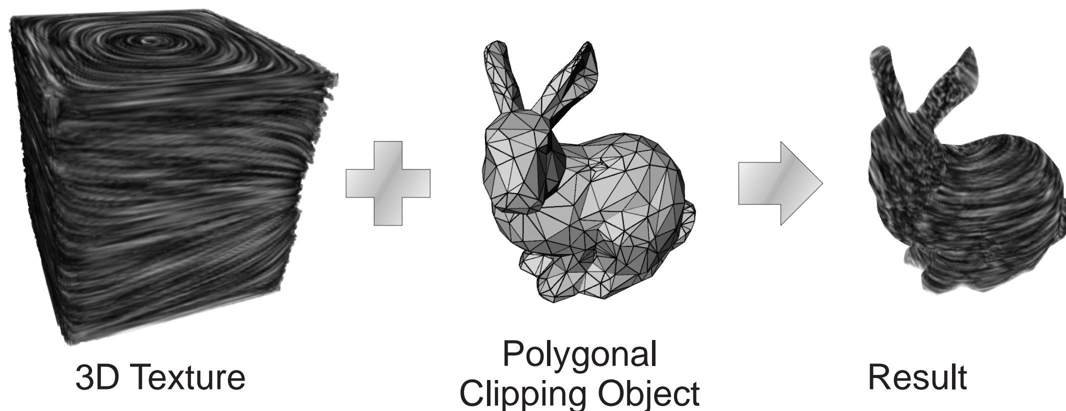


Figure 9.3: Clipping a volume against an arbitrary polygonal object.

9.2 Stencil Buffer Clipping

For the interactive examination of volume data, clipping mechanisms have proven extremely helpful. A straightforward implementation is the use of multiple clipping planes provided by OpenGL compliant graphics hardware. Clipping planes, however, only allow the construction of convex geometries. In many applications more complex clipping objects are required. Figure 9.3 shows an example of a volume object clipped against an arbitrary concave polygonal surface.

An efficient way to implement arbitrary polygonal clipping objects in combination with texture-based volume rendering was introduced by Westermann and Ertl [174]. Their idea was to exploit the OpenGL stencil buffer, a per-pixel frame buffer locking mechanism. As outlined in Section 2.1.3, the stencil test allows the discarding of incoming fragments conditional on the value at the corresponding position in the stencil buffer. The idea of stencil buffer clipping for volumetric objects is outlined in Figure 9.4. For every slice plane

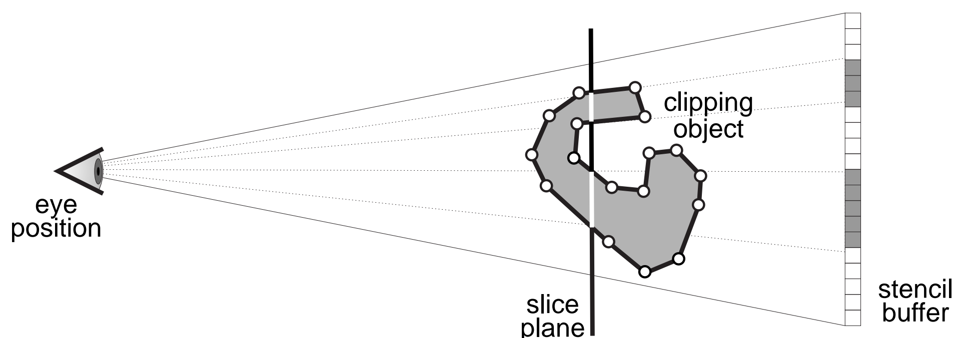


Figure 9.4: The idea of *stencil buffer clipping* is to lock those pixel which are covered by the cross section between the slice plane and the polygonal object. The stencil buffer must be updated for every slice plane.

the portion of the frame buffer, which is covered by the cross section with the polygonal object, must be marked by setting the respective pixels in the stencil buffer. When the slice polygon is finally rendered, the stencil test is passed only by voxels that lie inside the polygonal object. Alternatively, the stencil buffer method also allows inverse clipping by simply modifying the stencil test.

Calculating the cross section of the polygonal object with the slice plane can efficiently be realized as a multi-pass rendering method outlined in Figure 9.5. The algorithm works for multiple closed polygonal surfaces of arbitrary shape and for both viewport and object aligned slices. The only restriction is that the direction of a normal vector for each polygon must clearly indicate the inner and the outer region in a way consistent for the whole surface. The procedure is divided into four steps:

1. A clipping plane is set up with the same position and orientation as the current slice plane. This clipping planes removes the portion of geometry that faces the camera, so that the interior of the polygonal object becomes visible (Figure 9.5 *top*).

2. Every face of the clipping object is classified as either front or back face. The color buffers are locked, preventing the geometry from being visible in the final image. Only the back faces are rendered into the stencil buffer as displayed in Figure 9.5 *middle*. As a result, the depth buffer is also updated. The cross section contained in the stencil buffer, however, is not yet correct, since with respect to the camera position some back faces might be occluded by front faces.

3. In a second rendering pass, the erroneous portions from step 2 are removed by drawing the front faces. The stencil buffer is cleared whenever an incoming fragment passes the depth test, as displayed in Figure 9.5 *bottom*.

4. The stencil buffer now contains the correct cross section of the polygonal clipping object. Now the color buffers are unlocked and stencil test is setup to restrict the rendering to either the interior or the exterior of the polygonal object. The textured slice image is drawn and blended as usual.

OpenGL sample code for rendering one slice image is displayed in Listing 9.1. The example assumes a stencil buffer resolution of 1 bit only (lines 10–11), which requires the stencil buffer to be cleared for each slice image (line 12). If a stencil buffer with n bit resolution is available, we can increment the stencil mask (line 10) by 1 for each successive slice and clear the stencil buffer only every 2^n passes. The OpenGL back face culling mechanism is exploited in order to determine the front faces (lines 23–24) and the back faces (line 34). This assumes that the polygonal surface of the clipping object has a consistent vertex ordering. An important thing to mention is that even for the first rendering pass the depth test must be activated (line 20). Otherwise the depth buffer values of one back face might be overwritten by another back face which is occluded by the first one, as the polygons are drawn in arbitrary sequence.

In comparison to the straightforward approach of computing the cross section of the polygonal object on the CPU, the presented stencil buffer based approach will lead to a

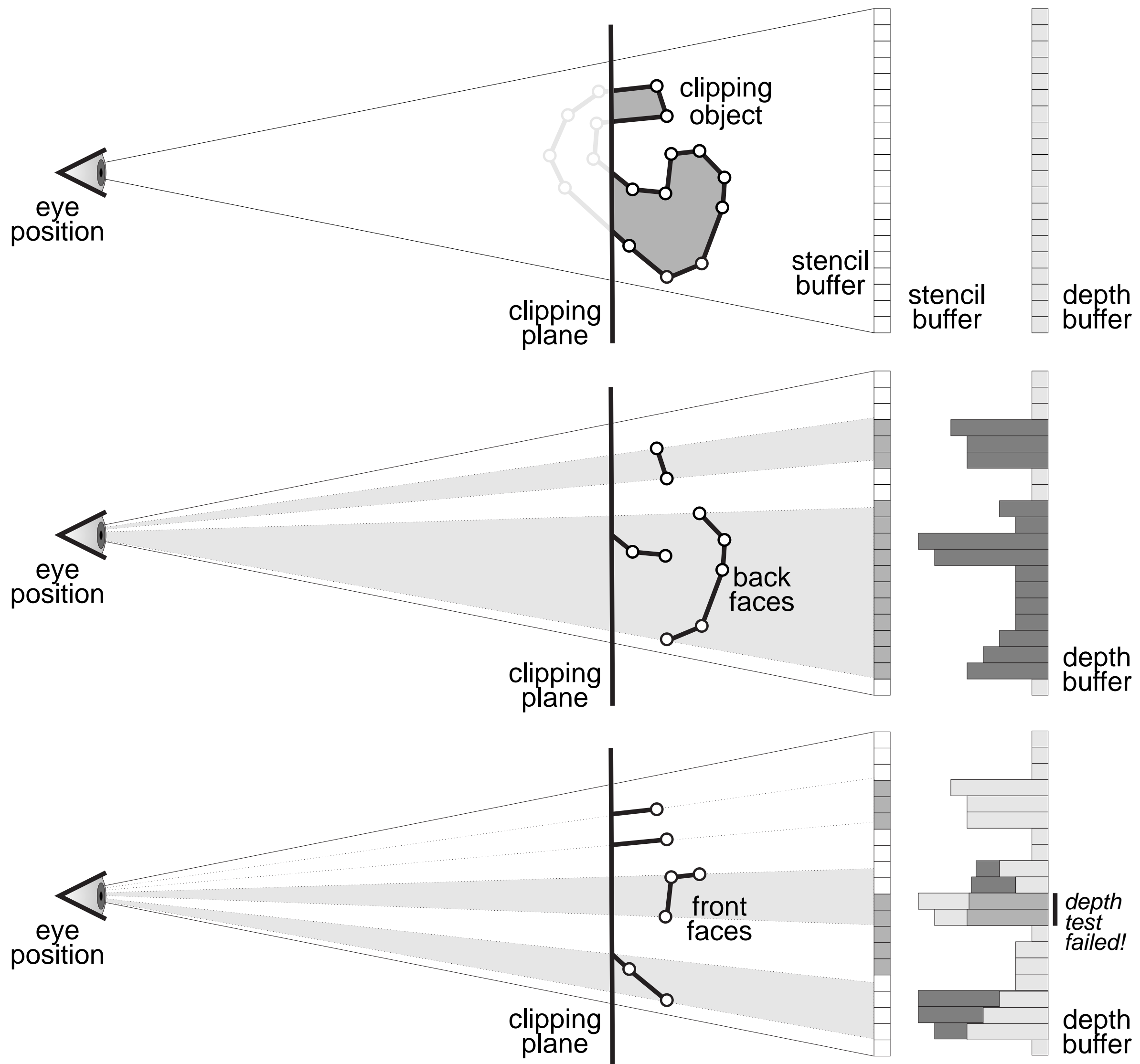


Figure 9.5: Multi-pass procedure for stencil buffer clipping: Step 1: A clipping plane is activated with the same position and orientation as the slice plane *(top)*. Step 2: Only the back faces of the polygonal object are drawn into the stencil buffer *(middle)*. The depth buffer is updated. Step 3: Finally the front faces are drawn in a separate rendering pass *(bottom)*. For fragments who pass the depth test, the stencil buffer is cleared. The resulting stencil buffer content is exactly the cross section between the polygonal object and the slice plane as displayed in Figure 9.4. The slice image can now be drawn into the frame buffer in a fourth step.

```
0    // STEP 1: --- Preparations ---------------------------------------------
1    glEnable(GL_CLIP_PLANE0);       // clipping plane setup
2    glClipPlane(GL_CLIP_PLANE0, m_pSlicePlaneEquation);
3
4    glEnable(GL_DEPTH_TEST);        // activate the depth test
5    glClear(GL_DEPTH_BUFFER_BIT);   // clear the depth buffer
6
7    glEnable(GL_STENCIL_TEST);      // activate the stencil test
8    glStencilMask(0x1);             // use value 1 for writing
9    glClearStencil(0x0);            // use value 0 for clearing
10   glClear(GL_STENCIL_BUFFER_BIT); // clear the stencil buffer
11
12   // lock the color buffers
13   glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
14
15   // STEP 2: --- First rendering pass ---------------------------------------
16   glStencilFunc( GL_ALWAYS, 0x1, 0x1); // stencil test always passes
17   glStencilOp(GL_REPLACE,  // replace if stencil test fails (never happens)
18               GL_KEEP,     // no modification if depth test fails
19               GL_REPLACE); // write to stencil buffer if depth test passes
20
21   glEnable(GL_CULL_FACE); // enable front face culling
22   glCullFace(GL_FRONT);
23
24   drawClipObject();  // draw the clipping object
25
26   // STEP 3: -- Second rendering pass ---------------------------------------
27   glStencilOp(GL_KEEP,  // keep the stencil buffer if stencil test fails
28               GL_KEEP,  // keep the stencil buffer if depth test fails
29               GL_ZERO); // clear the stencil buffer if depth test passes
30
31   glCullFace(GL_BACK); // enable back face culling
32
33   drawClipObject(); // draw the clipping object
34
35   // STEP 4: -- Render the texture slice image ------------------------------
36   glStencilFunc(GL_EQUAL, 0x1, 0x1); // activate stencil test
37   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // do not modify the stencil buffer
38
39   // unlock the color buffers
40   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
41
42   drawTextureSlice(); // draw the textured slice image
```

Listing 9.1: OpenGL setup for rendering one textured slice polygon using stencil buffer clipping.

significant gain in rendering performance for clip objects with a limited size of triangles. The drawbacks of stencil buffer clipping is the loss of the depth buffer[3] so the correct depth ordering is not ensured, if semi-transparent volume objects are combined with opaque geometry.

As an aside, the described method for stencil buffer clipping can efficiently be used for the voxelization [169, 21] of polygonal surfaces. Voxelization refers to the conversion of a polygonal surface into a volume data set and represents the transformation of a *parametric* into an *implicit* surface description [85]. Such a conversion is often used in multi-scale analysis or constructive solid geometry. For the purpose of voxelization, the described algorithm is used with an orthographic projection matrix. The contents of the stencil buffer is simply read out after the processing of each viewport-aligned slice image. Figure 9.6 displays the result of the voxelization of a polygonal mesh into volume data sets of different resolution.

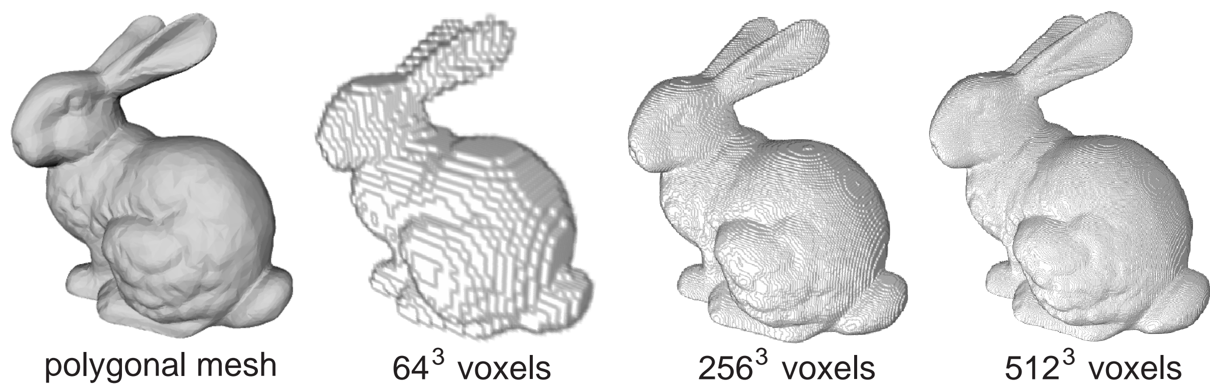


Figure 9.6: Voxelization of a polygonal mesh into volume data of 64^3, 256^3 and 512^3 resolution, respectively.

9.3 Vector Fields

In addition to scalar data, 3D vector fields are frequently used in natural science and engineering. Vector fields usually describe physical quantities such as velocities and forces, which depend on a certain direction. 3D vector data sets arise from measurement or numerical simulation in fluid mechanics, hydrodynamics, electric engineering and computational science. In recent years a number of different techniques for the visualization of 3D-flow phenomena have been developed.

Traditionally, geometric techniques are used, which represent the vector quantities by some kind of geometric primitives, such as arrows, icons or glyphs. Sophisticated approaches depict the properties of a vector field by using various techniques of particle tracing and methods like stream lines, streak lines, stream surfaces [72] and volume flow [106].

[3]For convex clipping object the depth test is not required, so the depth buffer can be preserved.

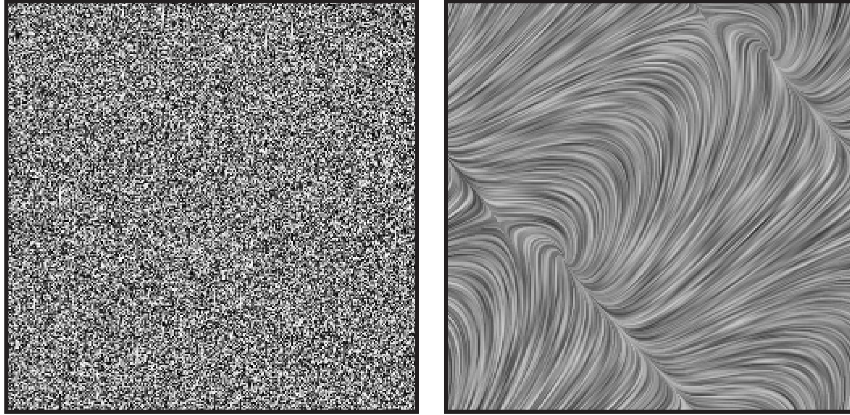


Figure 9.7: Line Integral Convolution uses a noise image *(left)* and a vector field to construct an image that clearly depicts particle paths with high density *(right)*.

The main difficulty with such approaches in practise is their restriction to a rather coarse spatial resolution. As alternative, texture-based approaches have gained increasing attention. The introduction of line integral convolution *(LIC)* [16] significantly improved the visualization of vector fields for the 2D case. LIC is an efficient technique to depict flow information in an intuitive way by transforming the vector field into a scalar field. In the 3D case however, difficulties arise in the visualization of the intricate structures inside the resulting volume data set.

9.3.1 Line Integral Convolution

The idea of LIC is to compute a 1D convolution on the stream lines in the vector field. The LIC algorithm filters a given input texture along the integral curves of a given vector field and generates a scalar field as output. For scientific visualization a noise field (Figure 9.7 *left*) is used as input texture. This ensures that the resulting LIC image is not influenced by structures in the input texture. The intensity I of a pixel at position $x_0 = \sigma(s_0)$ in the output image is determined by

$$I(x_0) = \int_{s_0-L}^{s_0+L} k(s-s_0)\, T(\sigma(s))\, ds\,, \tag{9.9}$$

with $\sigma(s)$ referring to the particle path of the vector field that runs through the point x_0 parameterized by arc length. $T(x)$ denotes the intensity of the input texture at position x and k is an appropriate filter kernel such as a Gaussian, a tent or a box filter.

For the box filter the convolution integral can be computed by sampling the input

texture T at locations x_i along the particle path $\sigma(s)$:

$$I(x_0) = k \sum_{i=-n}^{n} T(x_i)\,, \tag{9.10}$$

with the normalization factor $k = 1/(2n+1)$. The convolution causes voxel intensities to be highly correlated along stream lines, but statistically independent in direction perpendicular to the flow. In the resulting images the stream lines are clearly visible as shown in Figure 9.7 *right*.

LIC was presented by Cabral and Leedom [16] in 1993, who might have been influenced by an earlier texture-based method called spot noise [167]. A comparison of both approaches is presented in [27]. In 1995, Hege and Stalling [158] presented an implementation of LIC that was resolution independent, much faster and more accurate than the original approach. A variety of optimizations and supplements have been developed in recent years. Lisa Forssell [46] presented an extension that allows the mapping of LIC images onto curvilinear surfaces in 3D. Wegenkittl et al. [172] added information of the orientation of the flow and Risquet [141] presented a significant simplification for accelerating the imaging process. Many other authors have been working on enhancements by color coding [152] or animation [8, 47] and by adapting the algorithm to unsteady flows [153]. Victoria Interrante [74] has developed techniques to visualize 3D LIC by the use of fuzzy clipping objects and sparse noise textures with enhanced perception of depth. Sparse noise textures can also be simulated using transfer functions as displayed in Figure 9.8 *right*. In the following section, an animation technique will be outlined that utilized the approach for stencil buffer clipping to animate static 3D LIC textures. Results of this method have been previously published in [137].

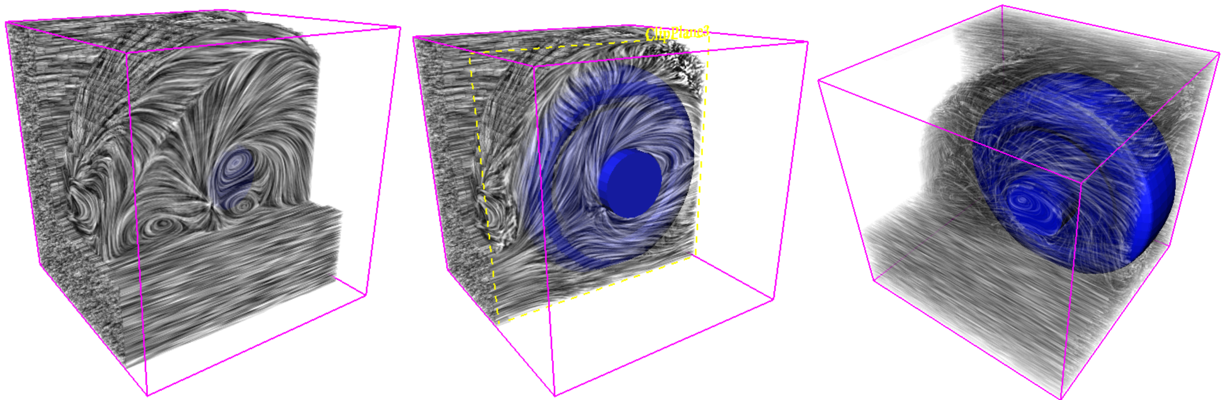


Figure 9.8: CFD simulation of turbulent flow inside the wheel casing of a car visualized with 3D LIC and different clipping planes *(left and middle)*. The use of sparse noise texture can efficiently be simulated using transfer functions *(right)*.

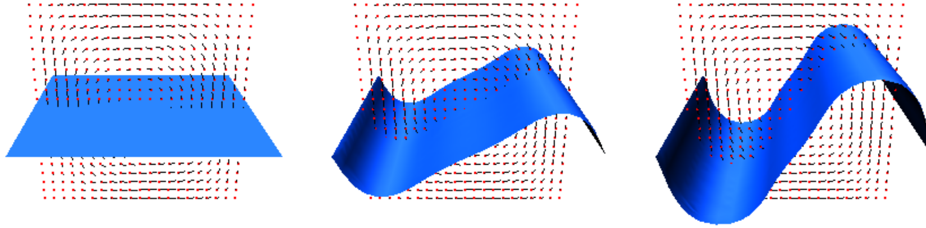


Figure 9.9: Surfaces of equal time *(time surfaces)* inside a simple cavity flow field.

9.3.2 Animated 3D LIC

The methods for stencil buffer clipping explained in Section 9.2 allows arbitrary closed polygonal meshes to be used as clipping objects. The fuzzy boundaries of a LIC texture suggested in [74] were used as volume of interest (VOI). In this context the shape of the clipping object should roughly follow the course of the stream lines. A straightforward approach is to compute stream surfaces which form a closed solid object. This represents a fast alternative to the computation of LIC on stream surfaces.

For animation purposes it is desirable to specify boundary surfaces which are intersected by the stream lines in an orthogonal angle, unlike the VOI. In consequence, a straightforward idea is to use time surfaces or time volumes as clipping geometry. In order to generate an appropriate set of time-dependent clipping objects for animation, an initial triangle mesh is placed inside the flow field in a way that stream lines intersect the surface at an angle of preferably 90°. This initial surface is evolved through time by computing particle traces for each vertex of the surface. The result is a set of surfaces of equal time as shown in Figure 9.9. Throughout the computation a simple subdivision scheme ensures that the resulting surface does not become self-intersecting. An algorithm for mesh decimation [17] is applied in order to limit the overall number of the triangles. If we use a closed polygonal mesh as initial surface and take care that the topology is not corrupted by high vorticity or vertices that leave the flow field boundaries, the resulting set of time

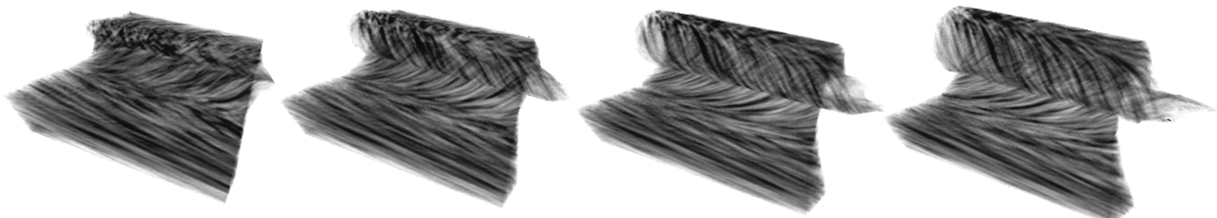


Figure 9.10: Animation sequence of a data set from numerical CFD simulation generated with the stencil buffer clipping approach.

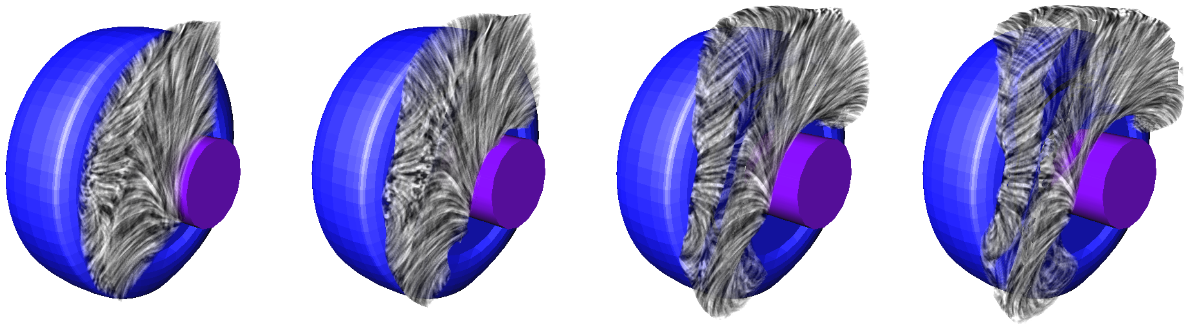


Figure 9.11: CFD simulation of turbulent flow inside the wheel casing of a car. Animation sequence generated with the stencil buffer clipping approach.

surfaces can be used as a sequence of clipping objects for the stencil buffer approach. Alternatively, closed polygonal meshes can be constructed from arbitrary time surfaces after the computation e.g. by joining two adjacent time surfaces to form a closed object.

Flow animation is performed by sequentially switching between different clipping objects. The animation sequence displayed in Figure 9.10 was generated by placing a flat box into a turbulent flow field. The box was distorted by the vector field resulting in rather complex time volumes. Figure 9.8 shows the visualization of a velocity field inside the wheel casing of a car, which resulted from numerical simulation in computational fluid dynamics (CFD). The animation sequence generated from this data is displayed in Figure 9.11.

Chapter 10

Deformation

As a consequence of the development of efficient volume rendering techniques, growing demand for volumetric deformation models has arisen in the last couple of years. Apart from obvious applications of free-form modeling in visual arts and entertainment, the ability to accurately model local deformation is extremely important in medicinal application such as minimal invasive surgery and computer assisted intervention. In a typical clinical application scenario, tomography data is acquired before the intervention for a detailed surgery planning. During the intervention, however, the pre-operatively acquired image data does not match the actual situation due to anatomical shifts and tissue resection. In consequence the spacial misalignment must be compensated by adapting the volume data to the non-linear distortion.

The deformation approaches reported in literature do no handle the problem of deforming volumetric objects sufficiently. The traditional free-form modeling tools [149, 23, 99, 19, 7] available in commercial software packages are restricted to polygonal surface descriptions, which do not account for a deformation of the interior of the object. In recent years, only a few approaches have been developed that try to bridge the gap between free-form surface deformation tools and volumetric data sets.

10.1 Principles

Kurzion and Yagel have provided the basis for many interesting space deformation algorithms by introducing ray deflectors [89]. Instead of deforming the geometry, they propose to deform the viewing rays for ray casting applications. Apart from pure software implementatons, this approach can be realized in hardware with the VIRIM ray casting architecture (see Section 2.3). The drawback of this method however is the difficulty to model different deformed objects that intersect each other. The same authors also presented an extension of the idea of ray deflectors for 3D-texture based volume rendering [90]. In this case the interior deformation is taken into account by tessellating the slice polygons into smaller triangles. Another supplement to 3D-texture based volume rendering was developed by Shiaofen Fang and his group [44]. In their approach the volumetric deformation is

computed by decomposing the volume into an octree structure and by slicing and texture mapping each sub-cube.

Other models for volumetric deformation (such as [36]) tessellate the whole volume into a set of tetrahedra. For each tetrahedron an affine transformation

$$\Phi(\vec{x}) = \mathbf{A}\vec{x} + \vec{b}. \tag{10.1}$$

is given, which results in a piecewise linear deformation of the overall object. The matrix $A \in \mathbb{R}^{3\times3}$ and the vector $\vec{b} \in \mathbb{R}^3$ are fully determined by specifying four translation vectors at the tetrahedron's vertices. Although this approach is well-defined from the mathematical point of view, its implementation suffers from multiple problems. The determination of the correct depth ordering for large set of tetrahedra contributes a significant part to the computational complexity of the algorithm. The tessellation of a simple volume cube requires at least five tetrahedra in order to account for the transformation of the corner vertices only. With the insertion of additional vertices in the interior, the overall number of tetrahedra increases rapidly.

In the following sections two different methods for the modeling of volumetric deformation will be described. Both approaches are supplements to 3D-texture based volume rendering and differ in the application scenario they were designed for. The first technique provides a mechanism to intuitively model the deformation by utilizing traditional free-form deformation tools for surface representations. This approach mainly targets modeling applications in visual arts and entertainment. The second (Section 10.3) approach has been specially designed for fast automatic deformation procedures in scientific applications such as registration of medical image data.

10.2 Volumetric Free-Form Deformation

The first approach to volumetric deformation that we will examine is mainly due to Rüdiger Westermann [175]. The aim of this approach is to adapt existing surface modeling tools to the deformation of the interior of an object. Possible applications of this approach are modeling tools as they are used by artists and industrial designers. By means of the presented deformation model, available volume objects can be manipulated and modified (see Figure 10.1) but also completely new objects might be created. Based on this algorithm a prototype deformation tool has been implemented with a simple modeling mechanism. Sophisticated surface deformation models can easily be inserted, such as free-hand manipulation tools for sculpturing or carving in virtual reality environments.

10.2.1 Shape and Appearance

The separation of *shape* from *appearance* is a common paradigm in 3D surface modeling environments. The shape of an object is usually defined by a parametric or an implicit surface description, e.g. as a polygonal mesh or a level surface. The appearance of this

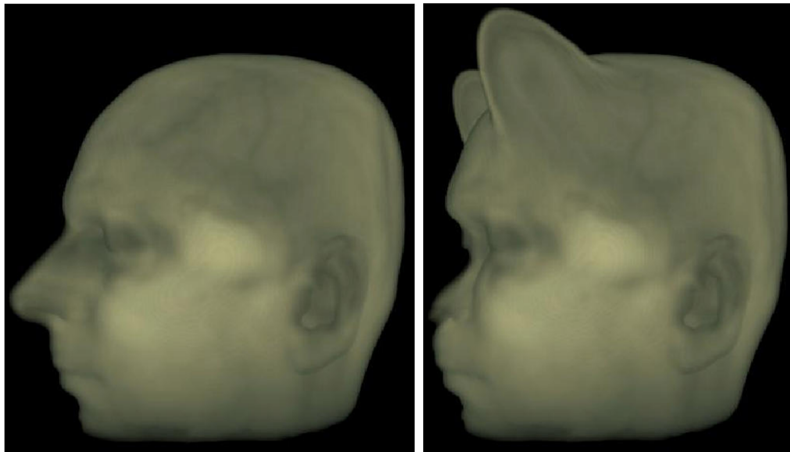


Figure 10.1: Example of direct volume rendering using the free-form deformation model.

object is defined in terms of material properties, such as color, reflection coefficients, a texture bitmap or a procedural shader.

We note that the texture-based volume rendering approaches that we have seen in the previous chapters are already based on a separation of shape from appearance in some sense. The shape of the volumetric object is determined by the polygonal representation of the bounding box, whereas its appearance is simply defined by the texture images. The bounding box can now be replaced by a more complex polygonal object as displayed in Figure 10.2. The visual result is similar to the algorithm for stencil buffer clipping (see Section 9.2), however in this case the sectional polygons with the slice planes are computed explicitly on the CPU. The *shape* of our object is now defined as the surface enclosing the volume. Without loss of generality, we assume that this surface is represented by a triangle mesh. As it is not necessary to represent the internal structures by additional geometric shapes, the interior of our object is exclusively defined in terms of *appearance*.

10.2.2 Intersection Calculation

The efficient computation of the intersection between a slice plane and the triangle mesh requires an optimized algorithm. In a previous research paper [176], Westermann et al. have proposed an *active edge data structure* to represent the triangle mesh. This data structure is illustrated in Figure 10.3. It consists of four arrays of elements that represent a list of vertices, a triangle list, an edge list and an active edge list.

Each vertex of the triangle mesh is represented by an entry in the vertex list. Such an entry consists of two coordinate triplets, one for the position and one for the 3D-texture coordinate of the vertex respectively. An element in the triangle list stores three pointers into this vertex list and also three references into the edge list. An element in the edge list in turn consists of two pointers into the vertex list and of two backward references which point to the triangles that share this edge. Although the active edge data structure contains a considerable amount of redundancy, the cross references allow an efficient search

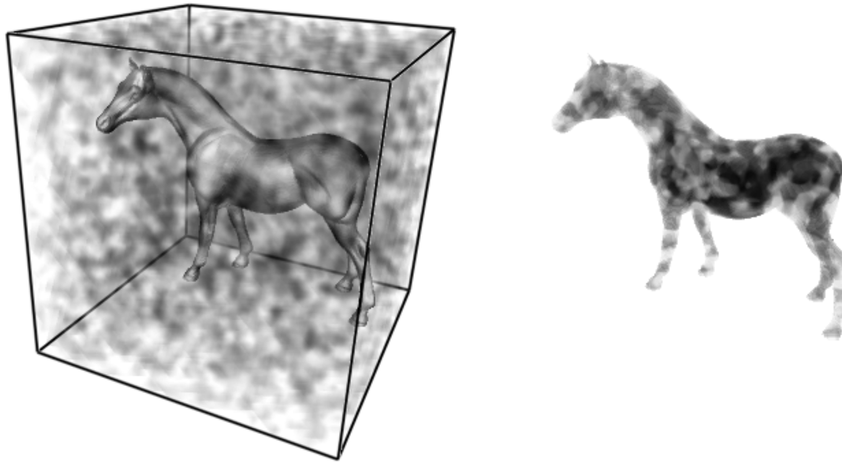


Figure 10.2: Separation of *shape* from *appearance*: An arbitrary polygonal model replaces the bounding box of the volume data set.

for the next elements that must be checked for intersection. Starting from an arbitrary intersection point between the slice plane and an element from the edge list, a sectional polygon is determined by following the references until an element is found that has already been visited.

Note that depending on the topology of the mesh, several possibly concave sectional polygons may exist. To further speed up the intersection calculation a view-dependent *active edge list* is maintained to avoid the processing of the entire edge list. In order to build such an active edge list, the polygonal mesh is partitioned into multiple slabs parallel to the view plane. Each edge then stores the first and the last slab it intersects. During back-to-front rendering, the active edge list keeps track of all the edges that may have an

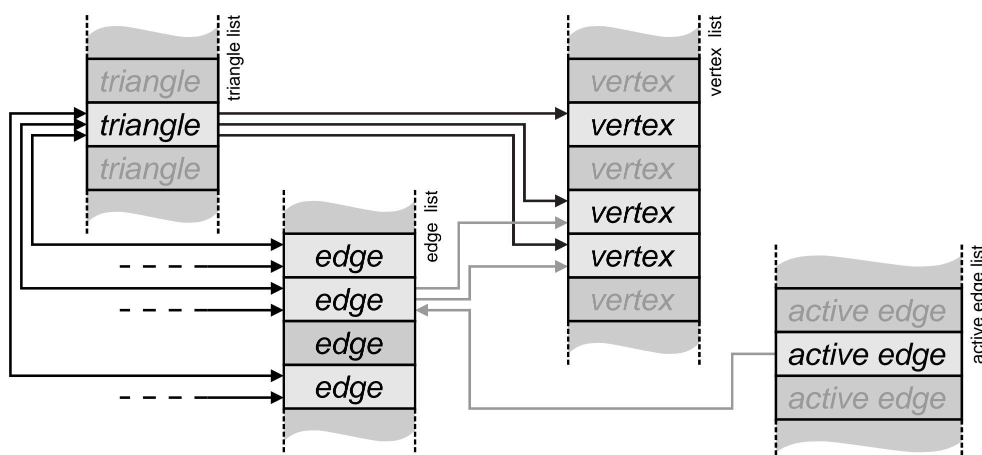


Figure 10.3: The active edge data structure is used for efficient computation of the sectional polygons.

intersection with the current slice plane. Whenever the slice plane enters a new slab, all edges that do not intersect this slab are removed from the active edge list and new edges are added.

The intersection of an arbitrary triangle mesh with a plane may yield concave polygons with holes. In consequence a sectional contour resulting from the described algorithm might be enclosed by any other one. In this case only the area between the outer contour and the inner contours must be rendered. For the tessellation of concave polygons with multiple holes the OpenGL utility library provides appropriate methods, which compute a trapezoidal decomposition.

Up until now we have not yet applied any local deformation. According to the modeling paradigm mentioned above, we separate the local deformation of a volume object into *shape deformation* which solely changes the boundary surface of the object and *appearance deformation* which adapts the 3D texture map to the deformed boundary. For the shape deformation part we are able to utilize almost every solution for surface deformation reported in literature, as long as we properly adapt our appearance deformation model.

10.2.3 Shape Deformation

Available free-deformation tools allow the manipulation of the shape in an interactive and intuitive way. In our case shape deformation is applied by changing the position of the vertices while the texture coordinates remain the same. We further assume that the applied free-form deformation approach takes care that the topology of the surface is preserved.

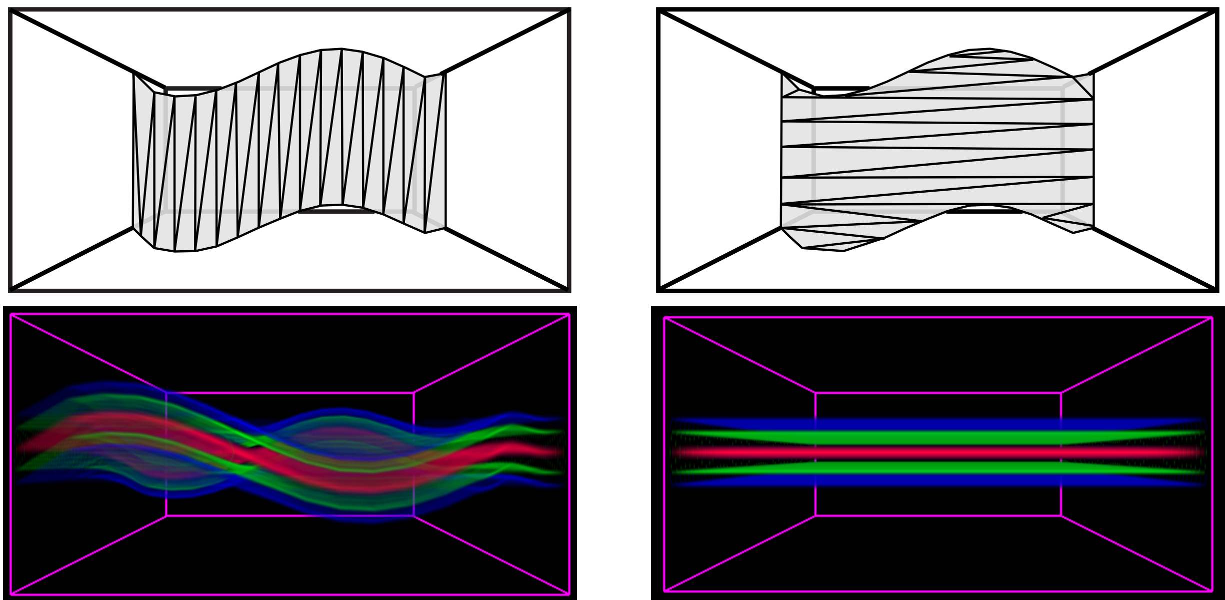


Figure 10.4: Without appropriate consideration of the interior, the deformation depends on the tessellation of the slice polygons. The same deformation is rendered with varying tessellation of the slice polygon.

As we do not change the texture coordinates at the displaced vertices the deformation is propagated into the interior of the volume. However, since the interpolation of texture coordinates within a sectional polygon greatly depends on its tessellation, this approach leads to inconsistent results as displayed in Figure 10.4. In order to obtain a consistent deformation model, it is obvious that also the deformation of the appearance must be taken into account.

10.2.4 Appearance Deformation

The modeling of volumetric deformations requires a mechanism to determine how the displacement of a vertex influences the appearance in the interior of the object. A simple way to describe the extent of a deformation caused by movement of a vertex is displayed in Figure 10.5. In addition to the translation vector that results from the vertex movement, the extent of the deformation is defined by a bounding box which is aligned with the direction of the displacement. Every point inside this volume will be displaced into the direction specified by the translation vector. The magnitude of this translation is determined by a function $V(u, v, w)$, which is computed as a 3D tensor product of quadratic B-splines,

$$V(u, v, w) \;=\; B_u(u) \cdot B_v(v) \cdot B_w(w). \tag{10.2}$$

To obtain a smooth transition between the deformed and the undeformed volume outside the bounding box, the translation magnitude is set to zero at the boundary of the displacement volume. Pre-computed lookup tables are used to minimize the computational cost for evaluating the B-spline functions.

Texture-based volume rendering is now performed by checking for intersection of the current slice plane with any of the displacement volumes. If an intersection is found, the resulting sectional contour is included as inner contour into the tessellation procedure

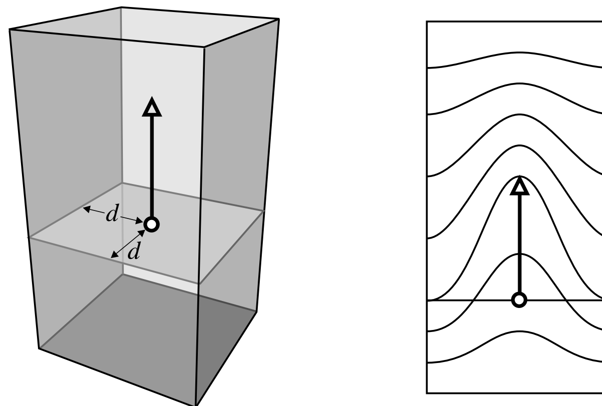


Figure 10.5: The displacement volume (left) defines the extent of a local volumetric deformation. It is specified as a bounding box which is aligned with the local deformation vector. The sectional drawing (right) outlines how displacement values are propagated inside the volume.

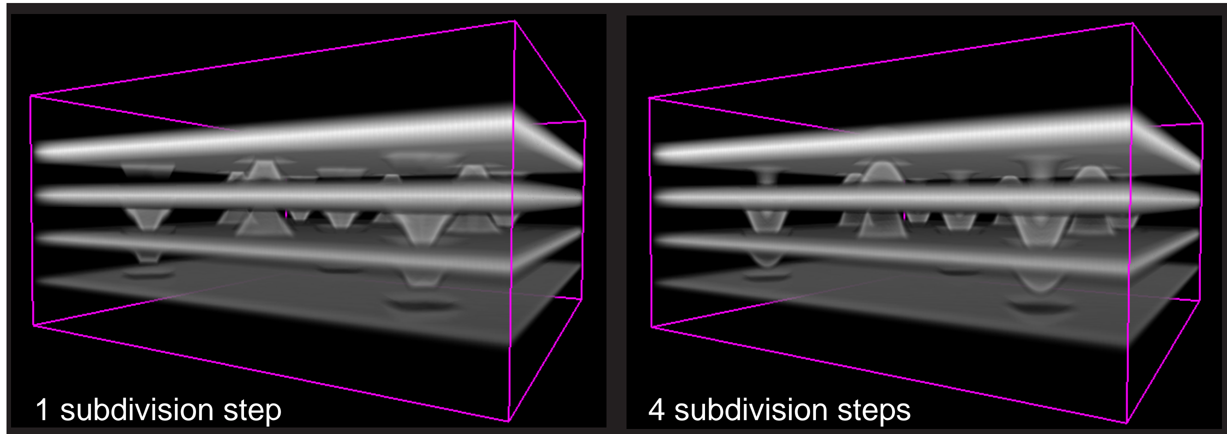


Figure 10.6: Volumetric free-form deformation with different subdivision levels for the displacement volumes. Although the deformation is already visible with only one subdivision step, the bumps are not correctly rendered. With four recursive subdivision steps the deformation is modeled with high precision.

described above. The interior of the sectional polygon is also tessellated and uniformly subdivided by recursively splitting each triangle into four new triangles. At each new vertex inside the displacement volume, the texture coordinates are shifted in negative direction according to the specified B-spline function. The subdivision terminates, if a user defined subdivision level has been reached. As a result, the non-linear deformation is modeled by an adaptive piecewise linear approximation.

The maximum subdivision level can be used to interactively control the accuracy of the deformation. For performance enhancement during user interaction a coarse subdivision level can be chosen, whereas for still images subdivision is performed with the full depth. Visual results of the deformation approach for different subdivision levels are displayed in Figure 10.6.

10.2.5 Illumination

In Chapter 7 we have seen several approaches to local illumination for texture based volume rendering. All of these methods pre-calculate the gradient vectors and store them as a normal map in an RGB texture. Due to the non-linear deformation, in our case pre-calculated gradient vectors are no longer valid.

To tackle this problem, we integrate an illumination method, which allows the approximation of gradient vectors on-the-fly during rasterization. As introduced in Section 7.1, the diffuse term of the Phong illumination model requires the computation of a dot product between the direction of light $\vec{l}$ and the normalized gradient vector $\vec{n}$,

$$I_{\text{diffuse}} = I_p \, k_d \, (\vec{l} \bullet \vec{n}) \quad \text{with} \quad \vec{n} = \frac{\nabla I(\vec{x})}{\|\nabla I(\vec{x})\|}. \tag{10.3}$$

We know from calculus that the dot product between two normal vectors is equal to the

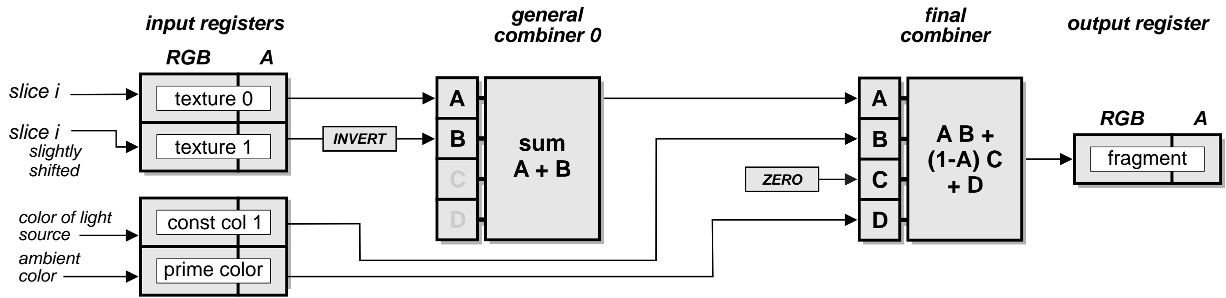


Figure 10.7: NVidia register combiner setup for the computation of forward differences.

length of one vector projected onto the axis defined by the other vector. If we take into consideration the fact that vector $\vec{n}$ is the gradient vector of a scalar field $I(\vec{x})$, we can compute the dot product by evaluating the directional derivative,

$$\left(\vec{l} \bullet \nabla I(\vec{x})\right) = \frac{\partial I(\vec{x})}{\partial \vec{l}}. \tag{10.4}$$

The normalization of the gradient vector $\vec{n}$, however, is missing in this equation. The directional derivative in turn can be approximated by a forward difference in direction of the light source,

$$\frac{\partial I(\vec{x})}{\partial \vec{l}} \approx I(\vec{x}) - I(\vec{x} + \vec{l}). \tag{10.5}$$

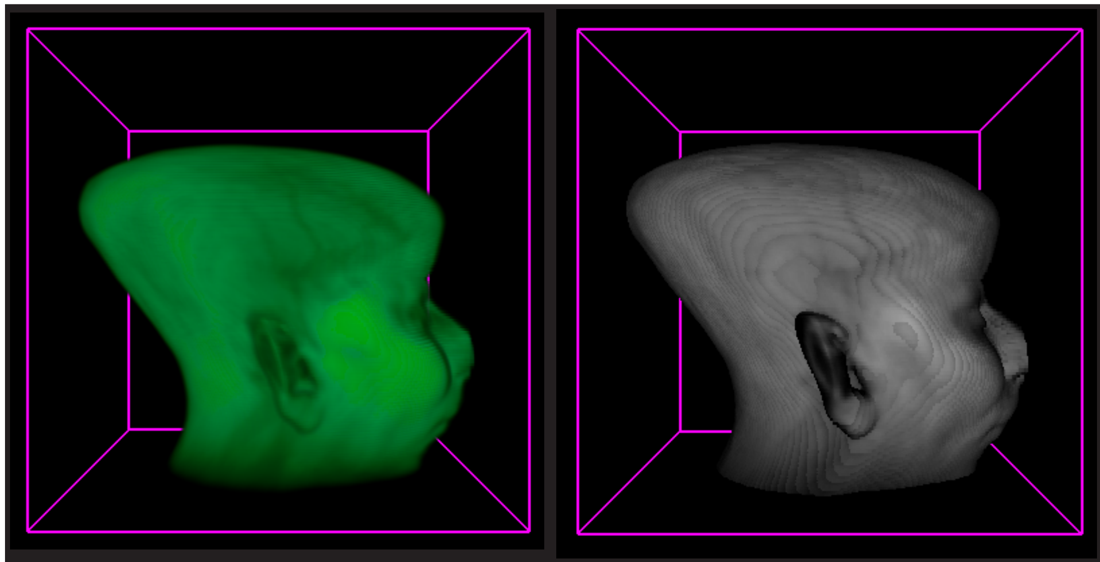


Figure 10.8: Example of volumetric free form deformation: direct volume rendering (left) and non-polygonal isosurface (right) with an approximation of the diffuse illumination term using forward differences.

This calculation is simple enough to be performed efficiently within the rasterization hardware. The appropriate setup for the NVidia register combiners is displayed in Figure 10.7. The same texture is specified for the first and the second multi-texture unit. For the second unit the texture coordinates are slightly shifted in direction of the light source. The forward difference is computed in the first general combiner stage. At the second combiner stage the result is multiplied with the color of diffuse light. Ambient light is added. This technique allows the inclusion of least the ambient and the diffuse term of the Phong illumination for a single light source model into our deformation model. The specular term is completely neglected. Visual results of the presented approach for non-polygonal isosurface rendering are displayed in Figure 10.8.

The described free-form deformation model provides a flexible way of including volumetric objects into interactive modeling tools. For an automatic deformation as required in medical imaging, the described approach is less applicable because the user must specify a considerable number of parameters. As an alternative approach we propose a different texture-based deformation model based on hexahedra structures and adaptive subdivision.

10.3 Hexahedra Deformation

The main drawback of tetrahedra based deformation models (Section 10.1) is the high computational cost for depth sorting and intersection calculation. To work around this problem we are going to examine a method based on hexahedra structures which has been especially designed with regard to a hardware accelerated implementation.

10.3.1 Deformation Model

Our deformation model based on hexahedra subdivides the volume object into a set of subcubes (patches) as depicted in Figure 10.9 (left). A piecewise linear deformation is now specified by translating the texture coordinates of the corner vertices of each hexahedron. The deformation is propagated into the interior of each patch by trilinear interpolation. As a result, the translation of the texture coordinate for a given point $\vec{x}$ in the interior of a patch is determined by

$$\Phi(\vec{x}) = \vec{x} + \sum_{i,j,k \in \{0,1\}} a_{ijk}(\vec{x}) \cdot \vec{t}_{ijk}, \tag{10.6}$$

with $\vec{t}_{ijk}$ referring to the texture coordinates of the translation vectors specified at the corner vertices. The interpolation weights a_{ijk} in this equation are obtained from the position of point $\vec{x}$ with respect to the original (undeformed) grid. An important benefit of this deformation model is that the rendered geometry is static. The deformation is applied by modification of the texture coordinates. In combination with an object-aligned slicing algorithm this approach leads to an efficient implementation, which allows the deformable model to be manipulated and rendered in real-time.

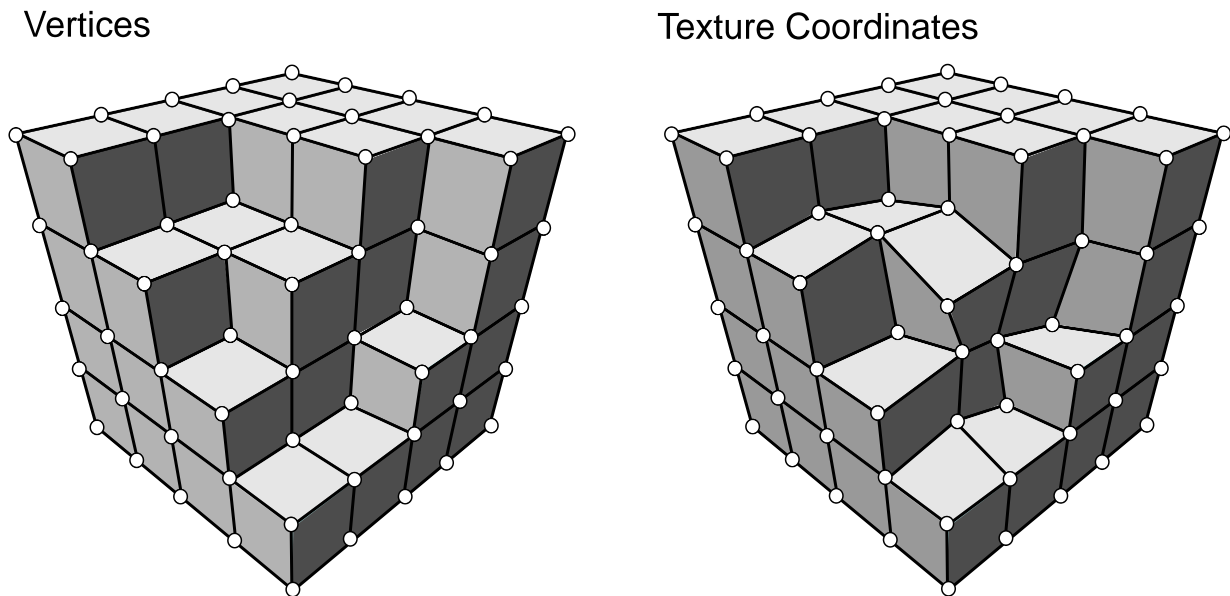


Figure 10.9: The volume object is subdivided into a set of sub-cubes (left). The deformation is modeled by transformation of only the texture coordinates at the vertices. The geometry which must be rendered is static.

10.3.2 Modeling

As mentioned above, the major application of the described hexahedra model is for automatic deformation techniques. However, for an application in modeling environments the specification of texture coordinates is neither intuitive nor user-friendly. In order to provide the user with a mechanism which allows him to specify the local deformation simply by picking and dragging of vertices, it is necessary to calculate the inverse transformation Φ^{-1}. The inverse function of a trilinear mapping, however, is not again a trilinear mapping, but a function of higher complexity. To avoid the evaluation of the exact inverse function, an approximation can be used for the purpose of modeling. Simply negating the original translation vectors,

$$\widetilde{\Phi}^{-1}(\vec{x}) = \vec{x} + \sum_{i,j,k \in \{0,1\}} a_{ijk}(\vec{x}) \cdot (-\vec{t}_{ijk}), \tag{10.7}$$

yields a sufficiently good approximation to the original inverse Φ^{-1}. The approximation error for a maximum deformation magnitude γ amounts to

$$\widetilde{\Phi}^{-1}(\Phi(\vec{x})) = \vec{x} + o(\gamma^2). \tag{10.8}$$

This approximation turns out to be accurate enough to enable intuitive modeling. High precision is not necessarily required in sculpturing applications. An intuitive mechanism to model the deformation similar to specifying control points for a B-spline surface should suffice.

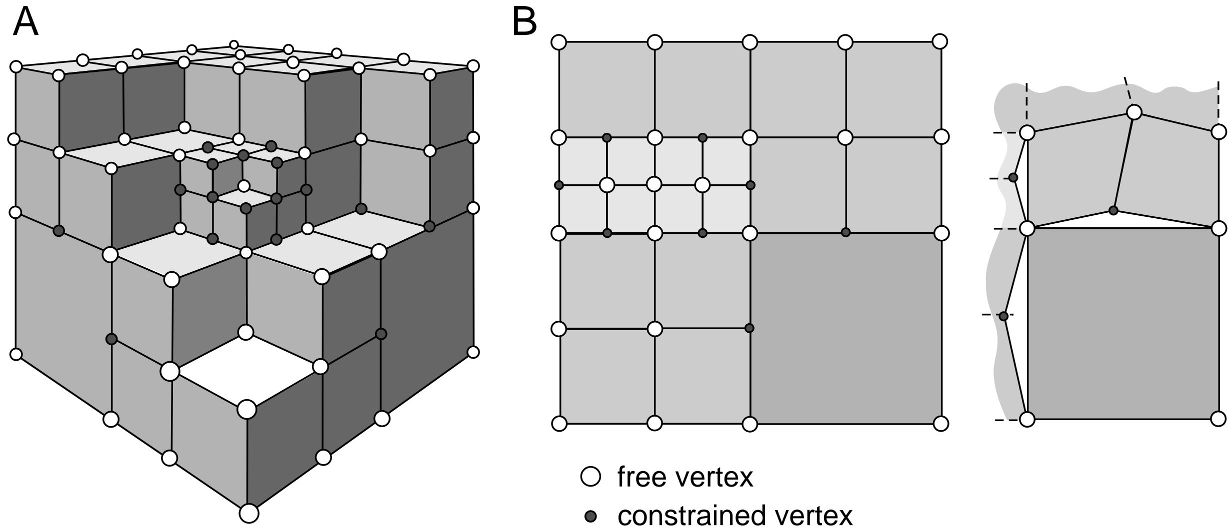


Figure 10.10: Constrained vertices are located on edges or faces between patches of different subdivision levels. Ignoring the constraints will lead to gaps in texture space *(B, right)*.

10.3.3 Adaptive Subdivision

With respect to deformation methods, which automatically search for the best parameter values according to a specified quality metric, the number of free parameters should be as small as possible in order to minimize the computational load for the optimization algorithm. Uniform subdivision of our model into a hierarchical structure of hexahedra, however, will soon result in a high number of free vertices.

Using the above concept as a basis, it is easy to circumvent this problem by adaptive subdivision of single patches in regions where higher flexibility is required. Adaptive subdivision result in a hierarchical octree structure as shown in Figure 10.10 A. Similar to subdivision surfaces, appropriate measure have to be taken in order to prevent undesired gaps in texture space as depicted in Figure 10.10 B. In order to maintain a consistent texture map, constraints have to be specified for all vertices that are shared by patches of different subdivision levels. In the 3D case two different types of constraints are required as depicted in Figure 10.11.

Edge Constraints: If two patches of different subdivision level share a common edge, a constraint has to be specified for the vertex which has been inserted in the middle of this edge. To prevent gaps the edge must stay collinear. In consequence, the inner vertex is required to stay on a fixed position relative to the two neighboring vertices (see Figure 10.11, left). Its position in texture space is determined by

$$\vec{V_C} = (1-\alpha)\vec{V_0} + \alpha \cdot \vec{V_1}, \tag{10.9}$$

with $\alpha = \frac{1}{2}$ in case of regular subdivision.

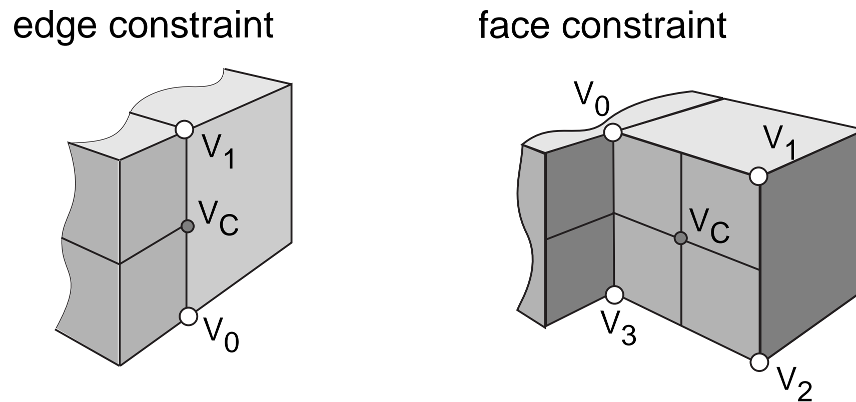


Figure 10.11: Edge (left) and face constraints (right) are necessary to prevent gaps in texture space.

Face Constraints: A second type of constraint is required to ensure that faces between patches of different subdivision levels stay coplanar. The vertex in the middle of such a face must stay at a fixed position relative to the four surrounding vertices (see Figure 10.11, right). Its position in texture space is determined by

$$\vec{V_C} = \sum_{i=0\ldots3} a_i \vec{V_i} \qquad \text{with} \qquad \sum_{i=0\ldots3} a_i = 1. \tag{10.10}$$

In case of uniform subdivision, the interpolation weights a_i must be set to $\frac{1}{4}$.

To avoid the additional computational cost for managing recursive constraints, a general rule is applied which is known from surface subdivision: Two neighboring patches[1] must not differ by more than one subdivision level. This means that any patch can only be further subdivided if all neighboring patches have an equal or higher subdivision level.

10.3.4 Implementation

As described in Chapter 3, OpenGL hardware rendering requires the decomposition of the patches into planar polygons. For an efficient implementation, we want to preserve the benefit of our deformable model being based on a static geometry. In consequence object-aligned slices are used as displayed in Figure 10.12, allowing the pre-computation of intersection polygons.

The straightforward approach to render the object by computing the slice polygons for each sub-cube and assigning texture coordinates at the resulting polygon vertices will not lead to a correct representation of the trilinear deformation according to Equation 10.6. The resulting inaccuracy is illustrated in Figure 10.13. Column A shows the correct trilinear interpolation of the texture coordinates of a slice polygon that we want to achieve. The

[1]Patches are considered neighboring if they share at least one edge.

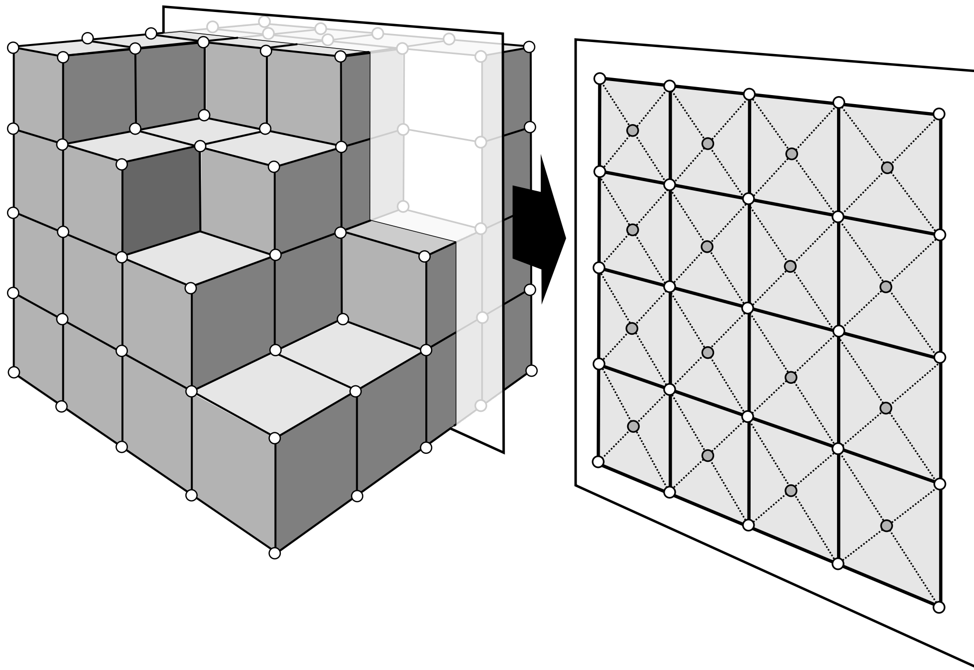


Figure 10.12: Object-aligned slices are extracted at low computational cost.

graphics hardware however internally tessellates polygons into triangles. Letting OpenGL perform the tessellation will lead to a bad approximation (column B). The grey triangle is not affected by the transformation. As a more accurate approximation, an additional vertex can be inserting in the middle of the polygon. In most cases the resulting tessellation is sufficiently close to the original trilinear mapping. The insertion of such an additional vertex also represents a correct triangulation of the non-planar texture map, which results from the 3D deformation in texture space. Possible enhancements and future possibilities for further optimization of this implementation using advanced feature of the graphics hardware are described in [139].

10.3.5 Illumination

As we have already seen in the approach for free-form deformation, illumination calculation based on pre-computed gradient vectors will result in erroneous lighting effects due to the non-linear deformation. In Section 10.2.5 we have introduced a method for on the fly gradient estimation using forward differences. Although this approach is well applicable to our hexahedra based deformation model, let us examine an alternative possibility for gradient estimation. This idea tries to adapt pre-calculated gradient vectors to the local deformation in an approximative way.

We know that if an affine transformation matrix is applied to an object, its normal vectors must be transformed with the transposed inverse of the matrix. The idea to adapt pre-computed gradient vectors to the non-linear deformation is to approximate the original

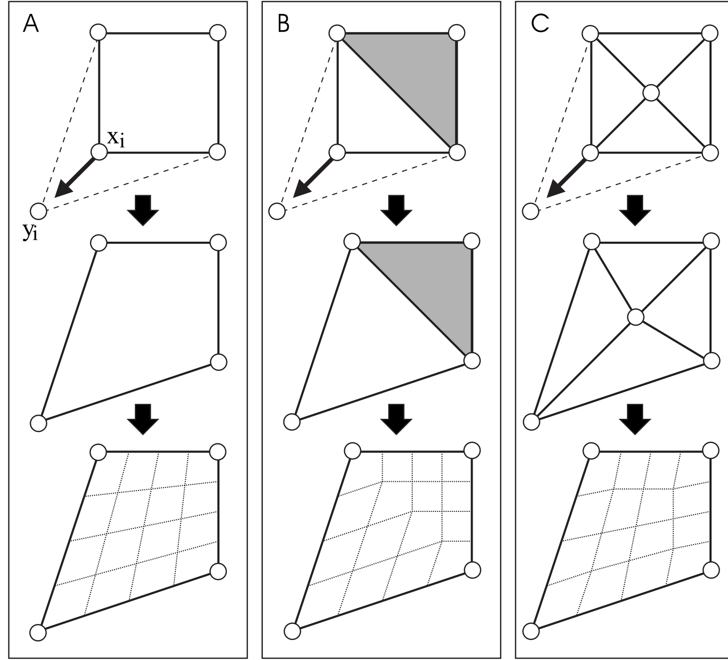


Figure 10.13: In contrast to the required trilinear interpolation (A), internal tessellation of OpenGL (B) results in linear barycentric interpolation. Inserting an additional vertex (C) approximates trilinear interpolation sufficiently.

trilinear mapping $\Phi(\vec{x})$ by an affine mapping according to Equation 10.1. For simplicity we write this equation in homogenous coordinates, denoted

$$\overline{\Phi}(\vec{x}) = \overline{\mathbf{A}}\vec{x}, \qquad \text{with} \quad \overline{\mathbf{A}} = \left(\begin{array}{c|c} \mathbf{A} & \vec{b} \\ \hline 0\;0\;0 & 1 \end{array} \right) \in \mathbb{R}^{4\times4}. \tag{10.11}$$

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

$$\frac{\partial}{\partial \mathbf{A}} \sum_{i=1}^{8} \|\overline{\Phi}(\vec{x}_i) - \vec{y}_i\|^2 = 0, \tag{10.12}$$

which leads to

$$\sum_{i=1}^{8} (\vec{x}_i \vec{x}_i^T \mathbf{A}^T - \vec{x}_i \vec{y}_i^T) = 0. \tag{10.13}$$

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

$$\mathbf{A}^T = \mathbf{M}^{-1} \sum_{i=1}^{8} \vec{x}_i \vec{y}_i^T, \qquad \text{with} \quad \mathbf{M} = \sum_{i=1}^{8} \vec{x}_i \vec{x}_i^T \in \mathbb{R}^{4\times4}. \tag{10.14}$$

It is easy to verify that the inverse of matrix $\mathbf{M}$ always exists. Also note that, since the undeformed corner vertices $\vec{x}_i$ are static, matrix $\mathbf{M}$ is a constant for each patch, thus allowing an efficient pre-computation.

In order to achieve realistic illumination results according to the Phong model, the pre-computed normalized gradient vectors must be adapted to the actual deformation. According to our affine approximation, the new diffuse term after the transformation is determined by

$$\widetilde{I}_{\text{diffuse}} = I_p \; k_d \left(((\mathbf{A}^{-1})^T \vec{n}) \bullet \vec{l} \right). \tag{10.15}$$

Note that since the gradients $\vec{n}$ are obtained from a texture, this calculation requires a per-pixel matrix multiplication, which can be computed using the pixel shaders of modern graphics boards. As alternative we propose an efficient method, which circumvents these per-pixel operations. Consider that the dot product in Equation 10.15 can also be written as

$$\left((\mathbf{A}^{-1})^T \; \vec{n} \;\right) \bullet \vec{l} = \vec{n} \bullet (\mathbf{A}^{-1} \vec{l}\,). \tag{10.16}$$

To our method this means that all the pre-computed normal vectors can be left untouched. We only have to evaluate a new light vector to obtain an equivalent visual result.

Regardless of whether the normal deformation is exact or approximative, using a light vector that is constant within each patch, but different for neighboring patches, will inevitably result in visible discontinuities as depicted in Figure 10.14 (center). This is due to the nature of piecewise linear transformations. To solve this problem, we must generate a smooth transition for the diffuse illumination term of neighboring patches. An idea similar to Gouraud shading is to specify light vectors per vertex instead of per patch. To each vertex a light vector is assigned as an average of the light vectors of all the patches that share this vertex. Analogously to the texture coordinates, the light vectors given at the vertices are interpolated within a patch. To achieve this during rasterization, the light vectors are simply assigned as color values to the vertices of each rendered polygon, thus

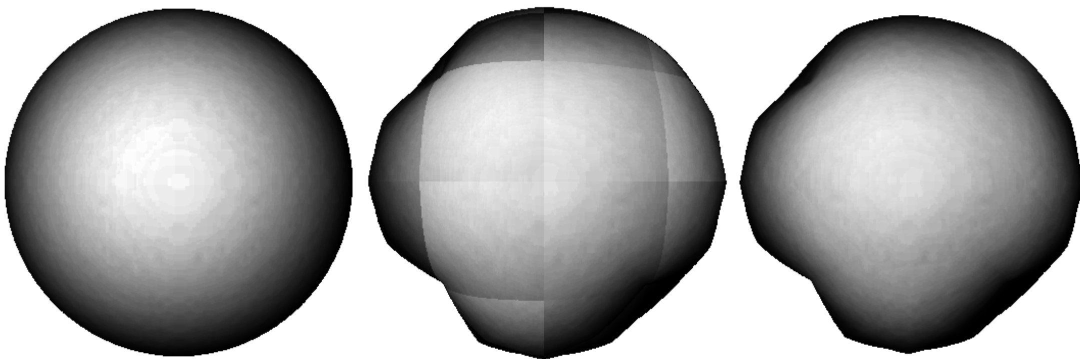


Figure 10.14: Diffuse illumination of an undeformed sphere (left). Extremely deformed sphere with discontinuities at the patch boundaries (center). Correct illumination by smoothing the deformed light vectors (right) at the vertices.

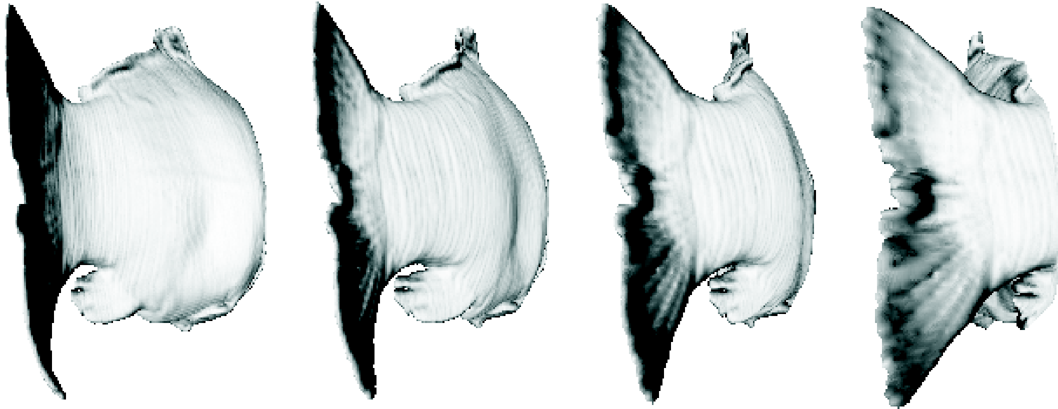


Figure 10.15: Animated tail fin of a carp demonstrates realistic illumination effects during real-time deformation.

allowing the interpolation to be performed by hardware Gouraud shading. As displayed in Figure 10.14 (right), this method will lead to satisfying illumination effects without any discontinuities.

The described gradient estimation scheme is highly approximative and should not be used for anything else than illumination. Since the Phong model does not have a strict theoretical background, the approximation error should be tolerable as long as the results are visually pleasing. An example of changing lighting effects under deformation is displayed in Figure 10.15.

10.4 Discussion

Literature reports only a few approaches for real-time volumetric deformation. Despite the increase in computation power, deformation models based on tetrahedra structures are still problematic. For any object that exhibits a reasonable complexity grid generation and rendering still cannot be performed interactively. For intuitive modeling the presented free-form deformation approach as well as the 3D-texture based approach proposed by Kurzion and Yagel [90] are well applicable and allow deformation and rendering to be performed at interactive frame rates.

The hexahedra model which has been optimized for general purpose hardware represents a fast method for automatic deformation, such as soft tissue modeling and multi-modality registration of tomographic data. The *definition* of the hexahedra transformation is based on a well-defined mathematical model. The fast implementation in hardware is an approximation to this exact mathematical model. A major benefit of the hexahedra model in comparison to free-form deformation is the possibility for accurate estimation of the approximation error. The trilinear mapping is approximated using four interpolations in barycentric coordinates. The resulting error is hardly noticeable for a deformation model with a reasonable level of subdivision. For patches of low subdivision level inconsistent

deformation might be visible when switching between orthogonal stacks of object aligned slices. Increasing the subdivision level will easily fix this problem. The most important application field of the hexahedra model is medical imaging, especially soft tissue modeling and registration.

10.5 Registration

In biomedicine, different imaging modalities provide volumetric information which is complementary in many aspects. For a wide variety of situations one individual data set is not completely sufficient in terms of the respective information content. As an example, computed tomography (CT) is especially sensitive to hard tissue and bone structures, whereas magnet resonance imaging (MRI) is capable of differentiating between various types of soft tissue. In certain cases only the combination of both sequences will meet the requirements for the clinical purpose of diagnosis and treatment.

Two volume data sets of different imaging modalities recorded under different circumstances at different points in time are usually not properly aligned with each other in the spatial domain. In order to provide the user with an image that represents the compound information obtained from different measurements, it is necessary to reconstruct the spatial alignment after the data acquisition. More specifically, *registration* denotes the computational procedure which determines a (possibly non-linear) transformation from the local coordinate system of one data set into the other one's. Registration techniques for volume data are closely related to volume rendering. The process of image generation which combines the complementary information of two registered data sets into a final compound image is referred to as *fusion* (see also Section 5.4.2). Besides medical imaging, fusion of multi-sensor images are important for machine vision and remote sensing applications in defense and atmospheric fields.

The development of registration algorithms for specific problems requires the analysis of the cause for the spatial misalignments between different data sets. In practise there are several different aspects that must be taken into account:

- The most obvious cause for spatial misalignment is the movement of the object between data acquisition. The exact location of a patient relative to the recording device is not consistent for different points in time. In this case, rigid registration approaches are sufficient if the scanned region itself is assumed to be static and undistorted.

- More complex registration techniques must be applied if the scanned object is deformed. These approaches account for changes in both the shape and the relative position of anatomical structures. Soft tissue deformation in the abdominal region such as liver shifts are a prominent example. Additionally, registration of pre- and intra-operative data must take into account the resection of soft tissue, fluid leakage and resulting shifts of anatomical structures.

- For data sets of different modality, the imaging parameters also play an important role. The contrast and the exact boundaries between different anatomical structures strongly depend on the imaging modality and the sequence parameters (see Sections 11.1.1 and 11.1.2) and may vary within a considerable range. Partial volume effects (Section 11.1.3) as well as the sensitivity of specific imaging sequences for different physical phenomena, such as fluid flow and scattering artifacts must be taken into account.

According to the most commonly used terminology, a *floating* data set is transformed into the local coordinate system of a *fixed* reference data set. The basic strategy for registration is to determine the free parameters of the transformation in a way that maximizes a specific similarity measure. The registration procedure is usually performed in an iterative process as outlined in Figure 10.16. Starting with an arbitrary initial configuration, the free parameters of the transformation are estimated. Subsequently the floating data set is transformed and compared to the fixed data set. The similarity metric is evaluated. If the result is not satisfactory, the transformation parameters in turn are modified and the whole procedure is repeated until no further improvement can be achieved. Within this general framework, the development of a specific registration procedure requires three components, which greatly influence the accuracy and the efficiency of the whole procedure:

- An appropriate coordinate **transformation**, the choice of which is based on a mathematical model of the real deformation.

- A **similarity metric** to accurately measure the *quality* of the applied transformation.

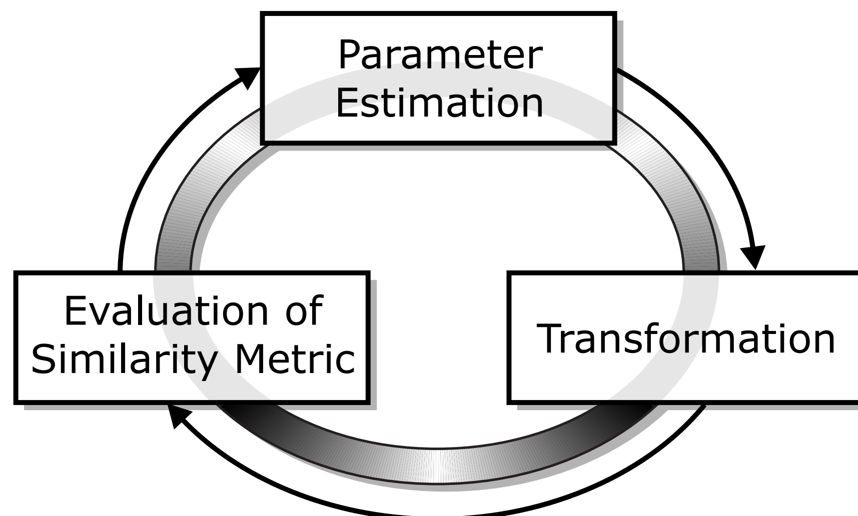


Figure 10.16: Registration as an iterative process: The free parameters of the transformation are estimated, the transformation is applied and an appropriate similarity metric is evaluated. The results of the similarity measurement are in turn used to modify the transformation parameters.

- An **optimization strategy**, which ensures that the algorithm proceeds towards the global maximum of the similarity metric.

Non-linear transformations are used to account for complex distortions and local deformations of the coordinate space. The first non-linear approach for registration of volumetric data was introduced by Bajcsy et al. [4]. They used an elastically deformable template model and a correlation–based similarity measure. Christensen et al. [18] proposed a deformation model based on viscous flow. The evaluation of this model can be significantly accelerated by the use of convolution with filters as suggested by Bro-Nielsen [13]. Thirion [160] introduced a similar model based on force fields that cause the non-linear deformation. Due to the intrinsic complexity of the registration problem, these methods usually suffer from extremely long computation times. Fast and efficient non-linear registration is still an unsolved problem in practise.

With regard to clinical application, our aim was to find a non-linear transformation that is flexible enough to model local tissue deformations and at the same time easy to evaluate with hardware acceleration. Throughout our experiments, software implementations of polynomial and Bezier tensor product patches turned out to be extremely time-consuming. On the other hand hardware implementations were restricted by the high computational complexity in case of polynomial patches and by limited hardware support in case of Bezier splines. Registration methods using the deformation model based on hexahedra, as described in Section 10.3, have been investigated and published in [135, 64]. In this implementation the floating data set was divided into a constant number of linear patches. For simplicity, we have assumed the boundary vertices to be fixed in place. The translation of every inner vertex has been computed for optimal alignment with the reference data set using multi-dimensional optimization based on mutual information [22, 168] as similarity metric. Mutual information is a voxel-based similarity metric that has its ori-

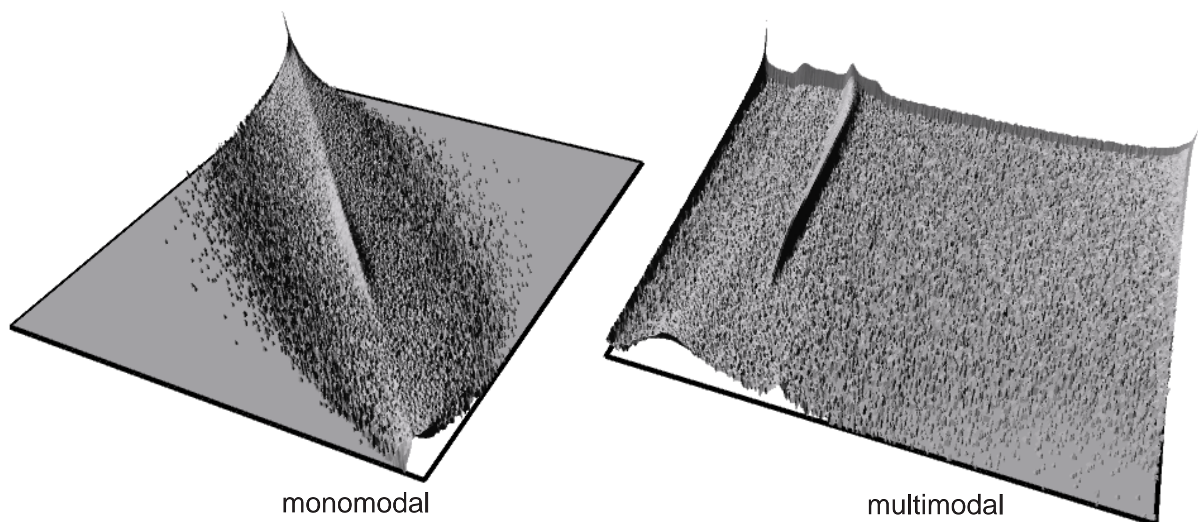


Figure 10.17: 2D compound histograms of mono-modal *(left)* and multi-modal data *(right)*.

gin in information theory. It is based on the relative entropy, a measure for the amount of disorder included in a system. In this context, the term disorder is synonymous with randomness and stochastic independency[2]. Mutual information involves the calculation of the joint probability distribution of two random variables, graphically represented by a 2D compound histogram as displayed in Figure 10.17. The optimal transformation is found, if the dispersion of significant clusters in the histogram is minimized, which coincides with mutual information reaching its maximum.

Real-time algorithms for registration and fusion of volumetric data sets are still an unsolved problem in computer science. Because of its important role in clinical applications such as computer assisted surgery, various research groups are working on tractable solutions. Surveys of registration techniques for medical image data can be found in [100, 14, 105]. Approaches that try to exploit hardware acceleration of any type are extremely rare in this field. Since from the computational point of view registration and volume rendering techniques are very similar, I am sure that future applications will leverage the great potential of graphics hardware for an improvement of the overall computation time.

[2] In thermodynamics, entropy measures the forces that cause the spontaneous mixing of different fluids and gases. These forces in turn are related to the amount of disorder at molecular level.

Part III

Applications in Medicine

Chapter 11

Introduction

"O Deep Thought computer,
The task we have designed you to perform is this.
We want you to tell us ... the Answer!"

Douglas Adams (1952–2000)
The Hitchhiker's Guide to the Galaxy

A major aim of the techniques presented in this thesis was to increase the applicability of volume visualization techniques in medicine. In this chapter I am going to report on several cases of clinical applications, that have been carried out in cooperation with the Division of Neuroradiology of the University of Erlangen-Nuremberg. Section 11.1 provides a brief introduction to medical image data and tomographic measurement. The described clinical studies comprise research projects as well as applications of volume rendering for diagnosis in clinical practise. The description of each individual project starts with a short introduction to the medical background, the clinical relevance and the specific visualization problem.

11.1 Medical Image Data

Radiology is a medical discipline which is primarily concerned with the acquisition, evaluation and interpretation of images of the internal human body. In the last decades, medical imaging and radiology has run through a revolution of technical progress. It started in 1972 with the development of computed tomography (CT). CT slice images represent the first image material in medicine purely obtained by computation.

Nowadays tomographic techniques represent invaluable methods for medical research, diagnosis and treatment planning. Major improvements have been achieved especially for the examination of the brain, the spinal column and the abdominal cavity, including the liver and the colon. The collaboration of medicine and computer science has lead to new minimal invasive examination techniques such as virtual endoscopy and virtual colonoscopy.

11.1.1 Computed Tomography

In contrast to the traditional X-ray technique, which only records two-dimensional projection images, computed tomography utilizes X-ray technology to reconstruct cross-sectional images of the internal human body. During the examination an X-ray tube rotates around the patient while emitting a fan of rays. On the opposite side of the tube an array of detectors measures the amount of radiation that passes through the tissue. From these measurements the cross-sectional slice image can be reconstructed by a computer program. Computed tomography is a widely used technique for the discovery and identification of space-occupying lesions.

The intensity of a voxel in a CT data set is proportional to the absorption of ionizing radiation, which is in some sense related to the density of the tissue. As a result CT data is extremely sensible to hard tissue types and bone structures. The *Hounsfield* scale allows the differentiation of various materials such as air, fluid, soft tissue and bone according to the respective intensity value. A data base of CT slice images can be accessed via internet at [118].

11.1.2 Magnet Resonance Imaging

In contrast to computed tomography, magnetic resonance imaging (MRI) is a completely non-invasive technique. The basis of MRI is a strong directional magnetic field. In the quiescent state the magnetic moments of the hydrogen nuclei inside the human body are randomly aligned. When a patient is placed into the magnetic field of an MRI scanner, the magnetic moments of the free hydrogen nuclei align themselves with the direction of the magnetic field. If a radio-frequency (RF) pulse is applied perpendicular to the direction of the magnetic field, the magnetic moments of the nuclei tilt away from their equilibrium. Once the pulse is removed, the magnetic moments realign themselves. During this relaxation process the hydrogen nuclei emit their own RF response signal, which is measured by the conductive coil of the MRI scanner. From this measurement a 3D intensity image can be reconstructed in the computer. The intensity of a voxel is proportional to the proton density, which is specific for a given tissue type. The contrast of the measured intensity values depends on two additional tissue-specific parameters. The *longitudinal relaxation time* T_1 indicates the time required for the magnetic moments to return to equilibrium. The *transverse relaxation time* T_2 measures the decay time of the response signal. The initial RF pulse now is applied periodically. The time between the inciting RF pulse and the measurement of the response signal is called the *echo delay time*. By adjusting both the repetition time and the echo delay time images of different contrast can be recorded.

11.1.3 Partial Volume Effects

Data sets that arise from tomographic measurement, especially CT data, often contain scanning artifacts for sample points close to the boundary of different tissue types. If

multiple objects of different intensity partly project into the scan plane, inconsistencies will occur resulting in shading effects. This inaccuracy is referred to as *partial volume effect* (PVE).

The partial volume effect is particularly noticeable in slice images that run almost parallel to object boundaries. As displayed in Figure 11.1 some regions in the image appear dim because tissue with lower intensity contributes to the voxel values. Obviously, partial volume effects can significantly degrade the image quality by blurring object boundaries. For CT scanner, the partial volume effect is influenced by the z-sensitivity (slice thickness). Images with higher z-sensitivity are favorable with regard to 3D reconstruction and volume rendering.

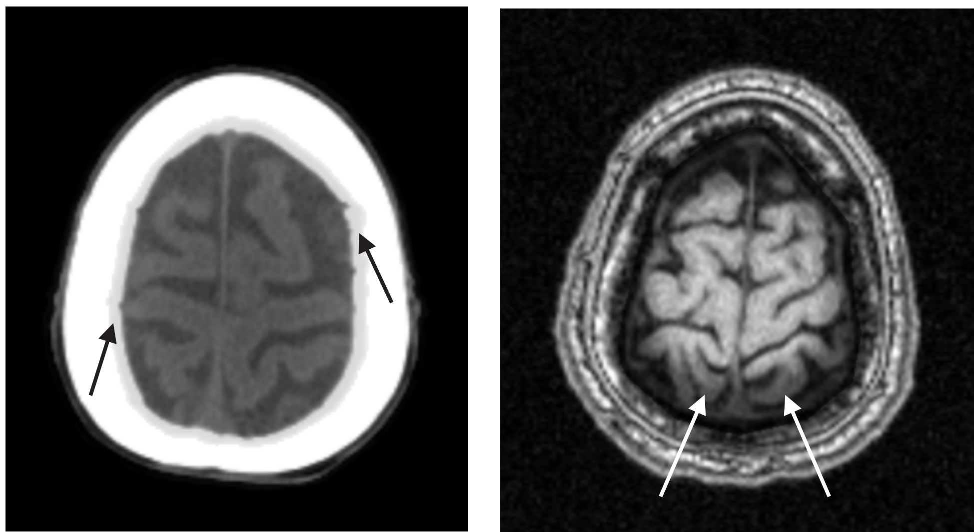


Figure 11.1: Partial volume effects in CT (left) and MRI (right) data.

Chapter 12

The Inner Ear

With high-resolution spiral-CT scanners, it is possible to record very tiny anatomical structures, such as the middle and the inner ear. The analysis of these vulnerable structures is of great importance for the planning of surgery close to the temporal bone. In this chapter we want to compare both indirect and direct volume visualization methods for this example of clinical practise. An indirect method is demonstrated by the use of threshold-based segmentation, contour extraction and surface reconstruction. The results of the surface reconstruction process are compared to direct volume rendering. Preliminary results of this work have been published in [63], including the registration of supplementary image information obtained by MRI.

12.1 Background

The internal part of the human ear can be divided into two functional units, the middle and the inner ear. The labyrinth of the inner ear mainly consists of the semicircular canals and the cochlea as displayed in Figure 12.1. These structures again consist of a bony part and a membranous part. The gap in between is filled with labyrinthine fluid (perilymph). The middle ear consists of the internal auditory canal, which accommodates the ossicles. Usually, only the bony parts of the ear are visible on CT data.

Although the small and complex structures are clearly visible on the slice images (Figure 12.2), it turns out to be extremely difficult to understand their spatial relations. This is an ideal case in which 3D reconstruction and volume rendering greatly enhances medical imaging by taking the necessity to mentally reconstruct the entire shapes away from the physician.

As outlined in Section 1.3.1.2, a polygonal model is usually obtained in a tedious and time-consuming procedure. The 3D scalar field delivered by the CT scanner is reduced to the surface description of a specified target structure. Due to the limited spatial extent of the inner ear, this is a complicated task and effusion within the temporal bone makes this process even more difficult.

In the following section the extraction of a high quality polygonal model of the middle

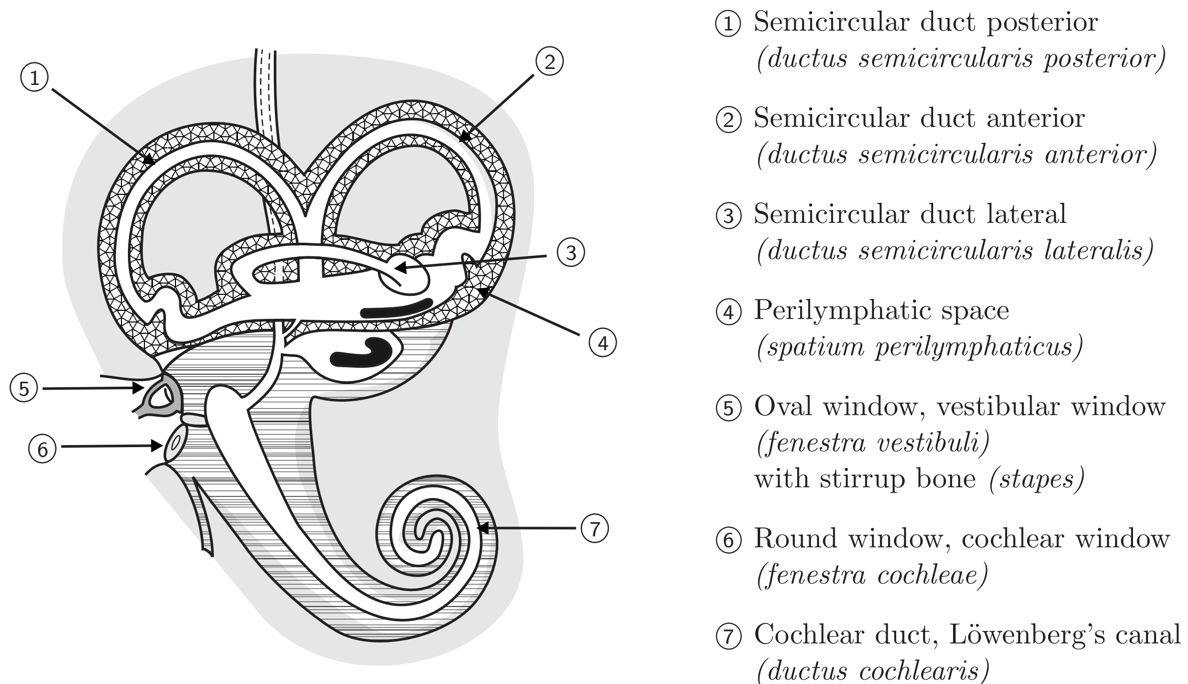


Figure 12.1: The inner ear mainly consists of three semicircular canals (①– ③) and the cochlear duct (①).

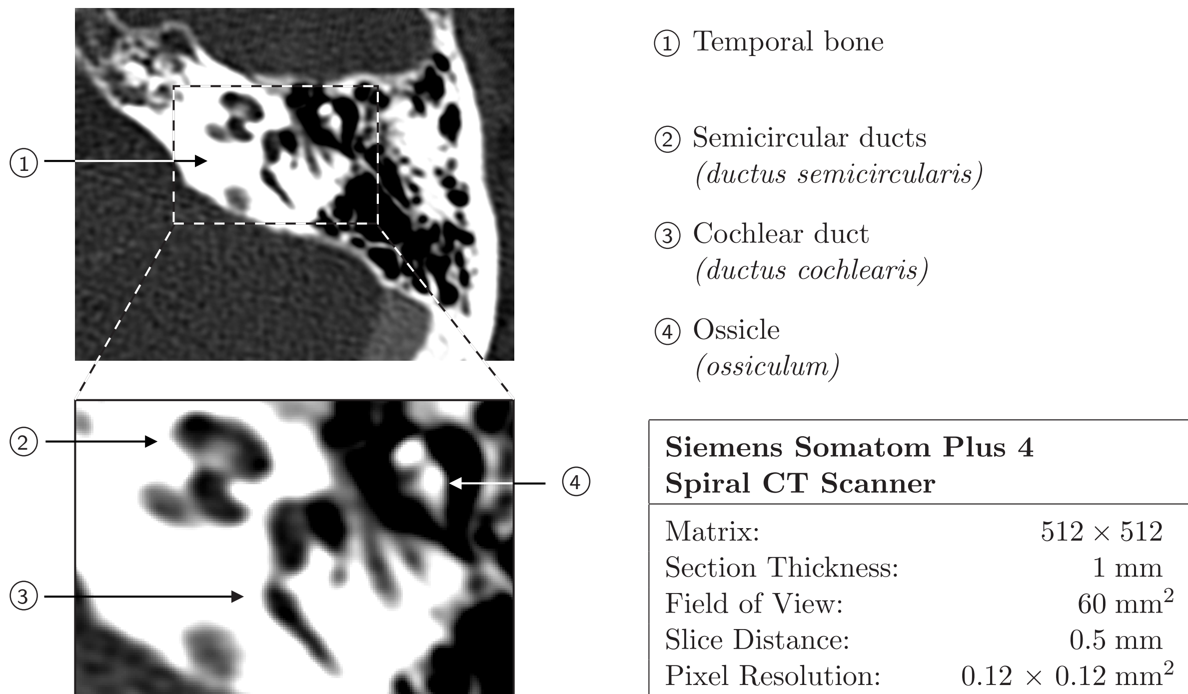
and inner ear is exemplified by describing the necessary segmentation and reconstruction steps. Neither the segmentation nor the surface reconstruction process that we apply claims to be the most efficient approach. However, the basic procedure and the difficulties that we encounter are typical for the majority of surface-based reconstruction techniques.

12.2 Surface Reconstruction

Due to noisy data and partial volume effects (see Section 11.1.3), it is hardly possible to extract the target object by a simple threshold operation. As a result, isosurface techniques are out of the question. In consequence, a voxel-based segmentation method must be used to explicitly determine the target structures. Fully automatic segmentation is usually error-prone and insufficient in terms of robustness. Thus, the reliability of the results obtained by automatic methods is doubtful.

We have chosen a user-guided semi-automatic segmentation approach, which has been described in [60]. This method is mainly based on an iterative procedure, involving volume growing, hysteresis threshold operations and manual editing, supervised by an expert user. Subsequent to this voxel-based segmentation process, the contours of the extracted objects are determined on every slice image by a simple contour detection algorithm.

From the resulting stack of contour lines, an initial polygon model is triangulated



Siemens Somatom Plus 4 Spiral CT Scanner	
Matrix:	512 × 512
Section Thickness:	1 mm
Field of View:	60 mm^2
Slice Distance:	0.5 mm
Pixel Resolution:	0.12 × 0.12 mm^2

Figure 12.2: High resolution CT slice images of the temporal bone.

using the software package *Nuages* provided by Bernhard Geiger [51, 50], which is freely available from the *French National Institute for Research in Computer Science and Control (INRIA)* [49]. *Nuages* is a package for 3D reconstruction from parallel cross-sectional data based on Delaunay triangulation. Since in our case the exact voxel boundaries of the target structures are used as contour lines, the resulting surface contains a high number of jagged edges which appear very distracting to the viewer.

As a smooth surface would much better represent the real target structures, the initial polygonal model is further post-processed. In order to remove the jagged edges, we apply *discrete fairing* as introduced by Leif Kobbelt [84]. The smoothing algorithm is based on topological mesh refinement. The original triangles are uniformly subdivided and the position of a newly inserted vertex is calculated by variational methods such as the minimization of the bending energy. With uniform subdivision, however, the overall number of triangles increases rapidly. For reducing the triangle mesh again to a manageable size, polygon reduction is applied with a technique presented by Campagna et al. [17]. In order to ensure that the resulting surface is an accurate representation of the real target structures, it is important to take care that the modifications of the surface comply with the tolerance measures that are derived from the original resolution of the volume data set. This of course applies to the surface smoothing as well as to the mesh decimation. Intermediate results from this sequence of post-processing steps are illustrated in Figure 12.3.

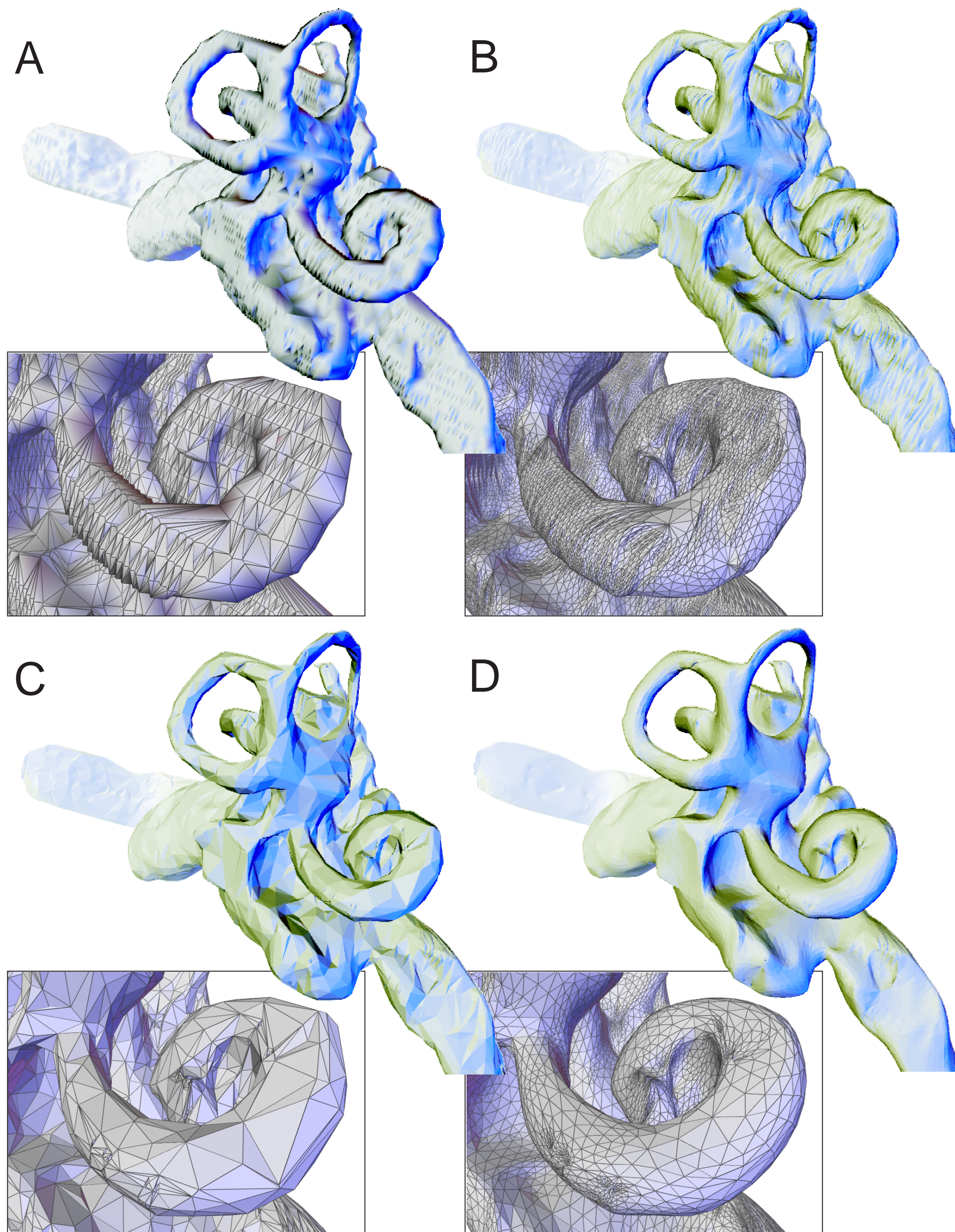


Figure 12.3: Surface reconstruction process for the inner ear. **A:** Initial surface generated by Nuages [51]. **B:** Smoothing step using uniform subdivision. **C:** Polygon reduction. **D:** Final result after additional smoothing step.

12.3 Direct Volume Rendering

Compared to the time-consuming process of surface reconstruction, texture based volume rendering allows the immediate visualization of the target structures without any pre-processing. The first step in a typical work flow of data analysis is the rendering of the whole volume data set and the assignment of an initial transfer function. If bone structures are of interest, the soft tissue is usually rendered fully transparent as displayed in Figure 12.4 (left). Subsequently the interior of the data set is explored interactively using clipping planes (right). When the interesting details are found inside the data set, sub-volumes are extracted and rendered separately. In this context it is important to apply a post-interpolative transfer function. The visualization of the cochlea or the semicircular canals of the inner ear with a transfer function for pre-classification is hopeless, as the boundary between these tiny structure will be obscured by interpolation artifacts as displayed in Figure 12.5(left). This problem is described in detail in Section 5.3. Post-classification enables implicit reconstruction of thin boundary surfaces which leads to clear visual results of high quality (right).

12.4 Results

The visual results presented in this chapter demonstrate the benefit of direct volume rendering in the context of the analysis of the middle and the inner ear. The direct volume rendering part described in this section has proven its value in a clinical application of

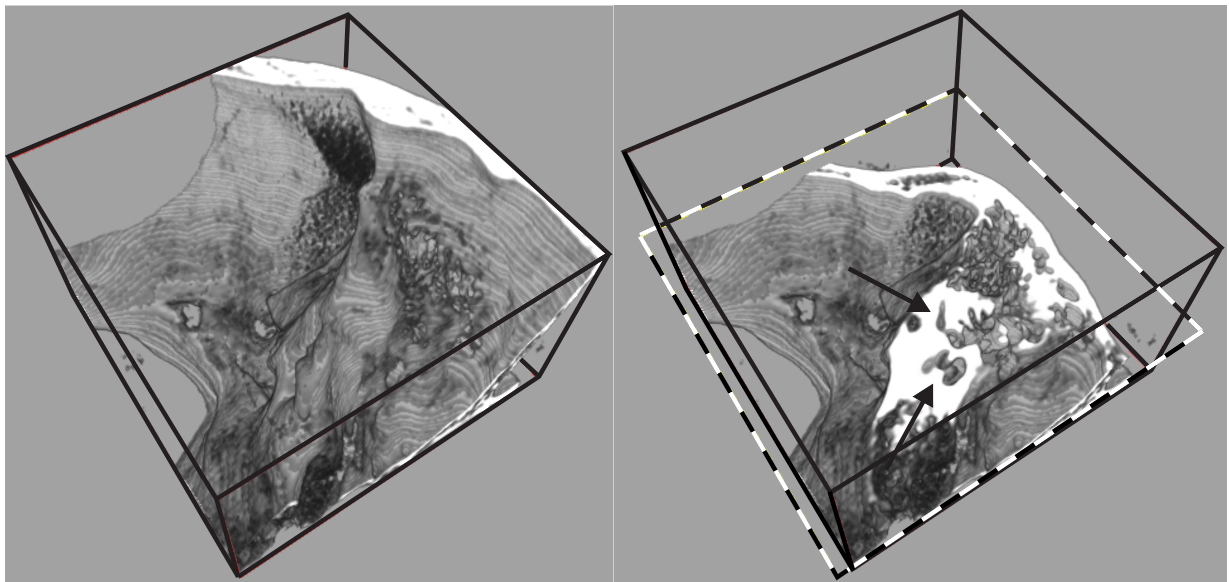


Figure 12.4: CT data set of the temporal bone: Clipping planes are used to interactively examine the data set. The structures of the inner ear (arrows) are embedded inside the temporal bone of high signal.

,

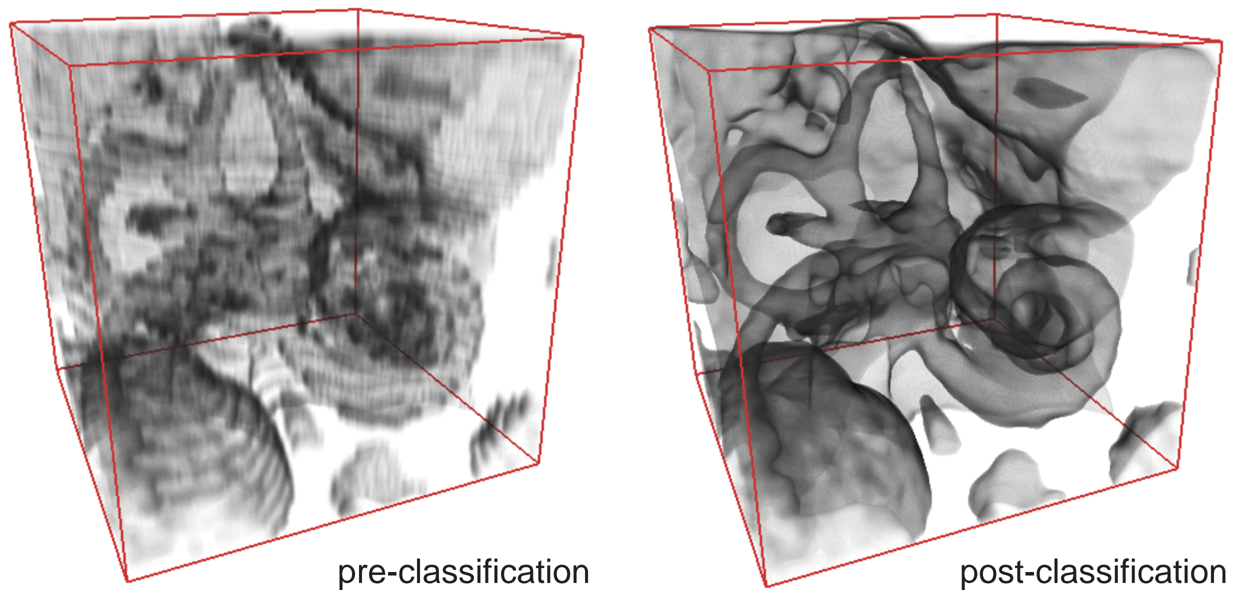


Figure 12.5: Comparison of pre- (left) and post-classification (right) of a high-resolution CT data set of the inner ear. Both images were generated with exactly the same transfer function and with exactly the same number of slice images. While the cochlea and the bony labyrinth is clearly visible in the post-classified image, due to the distracting interpolation effects the same structures are hardly visible in the pre-classified data.

virtual labyrinthoscopy [164].

In the case of the temporal bone, the application of surface reconstruction techniques for the only purpose of data visualization is not recommendable. The time consumed for accurate reconstruction is unnecessarily high compared to direct volume rendering approaches. Surface reconstruction techniques however are still important in scenarios that require an explicit surface description e.g. for surface measurement or simulation purposes.

Chapter 13

Intracranial Aneurysms

The visualization of 3D angiography data such as MRA and CTA is a typical example for interactive direct volume rendering substituting explicit voxel-based segmentation and surface reconstruction techniques. The extraction of the target structures is easily achieved using interactive transfer function assignment. The superiority of direct volume rendering for the visualization of CTA in comparison to traditional visualization techniques such as MIP or SSD[1] has already been established by a clinical study of intracranial aneurysms [65].

13.1 Background

The term angiography refers to the roentgenographic visualization of blood vessels after injection of a radiopaque substance. As a consequence of the fast evolution of tomographic measurement in radiology, there is a clear trend towards 3D imaging modalities which substitute traditional projection images. A prominent clinical example for the application of 3D angiography techniques is the diagnosis and surgery planning for the treatment of intracranial aneurysms.

An aneurysm is a localized abnormal dilatation of a blood vessel filled with fluid or clotted blood, resulting from a disease of the vessel wall. The term *intracranial aneurysm* refers to such a vessel malformation located at the cerebral arteries. As a result of a weakness of different layers of the vessels wall, there is a considerable risk that a rupture of the aneurysm will lead to severe haemorrhage in the brain.

Figure 13.1 shows the course of the cerebral arteries and its spatial relations to the cerebral nerves. Intracranial aneurysms are frequently located at the anterior communicating artery (①), the internal carotid artery (②) and the bifurcations of the medial cerebral arteries (③). The most widely used method for the treatment of intracranial aneurysms is a highly invasive procedure in which the surgeon inserts a clamp that seals the neck of the aneurysm. As an alternative approach, a platinum coil is inserted into the vessels in a minimal invasive procedure in order to prevent the turbulent blood flow inside the

[1]SSD = Shaded Surface Display

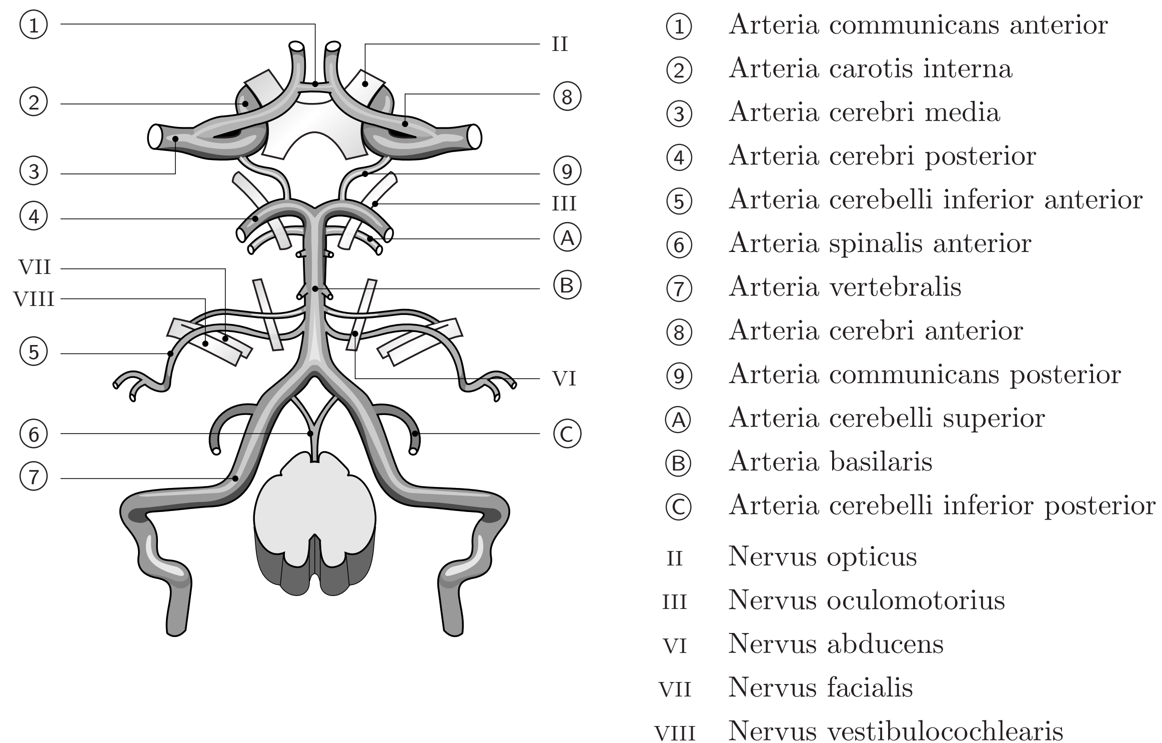


Figure 13.1: Intracranial arteries and its relation to the cerebral nerves. Intracranial aneurysms are frequently located at the arteria communicans anterior (①), the arteria carotis interna (②) and the bifurcations of the arteria cerebri media (③).

aneurysm which most likely causes the rupture. Both methods require profound knowledge about the individual vascular structures.

13.2 Visualization

The decision which treatment method is applicable in each individual case requires detailed information of the vasculature of the patient. In modern clinical scenarios non-ionic contrast dye is injected into the interesting vessels and CTA volume data is recorded.

The traditional visualization technique for angiography data is maximum intensity projection (MIP). The popularity of MIP is mainly due to its simplicity. The assignment of a transfer function is not required. However, as a consequence of the loss of depth information, the incorrect display of occlusion easily leads to erroneous interpretations of spatial relations between different vessels. Additional problems arise if vessels are located very close to the skull base, since the high intensity of the bone structures supersedes the signal of vessels filled with contrast medium.

Direct volume rendering techniques have proven superior for the interpretation of CTA data, as the physically based ray integration is much closer to natural human vision than

MIP. The results of a clinical study of intracranial aneurysms have been previously published in [65]. This application based on expensive high-end graphics server architectures includes functionality for improved navigation, distance and volume measurement and rigid registration with MRI data. Although this study provided excellent visual results which proved extremely valuable for diagnosis and surgery planning, the application was still limited by the availability of the high-end graphics workstation. In order to enable the analysis of small vascular structures on a larger number of hardware platforms an alternative approach which leads to similar visual results was implemented with respect to minimal hardware requirements

13.3 Semi-transparent Isosurfaces

As already demonstrated for the case of the inner ear, a transfer function for post-classification is mandatory for a clear visualization of tiny structures free from sampling artifacts. As we have seen in Chapter 5, the implementation of post-classification requires special hardware features such as post-interpolative texture color tables or dependent texture lookups. In order to visualize angiography data on hardware platforms that do not support extensions for pre-classification, a supplement of the techniques for rendering non-polygonal isosurfaces described in Chapter 7.2 can be used. As a result of the application of contrast medium with known CT value on the Hounsfield scale, the vascular structures are extracted by a simple isovalue threshold. An enhanced perception of depth is provided by illumination effects. For the analysis of angiography data,however, the use of non-polygonal isosurface techniques brings two drawbacks.

- The alpha test used for isosurface extraction is equivalent to a post-interpolative transfer function with a sharp edge. Besides the fact that the accurate display of such a thin surface requires a high sampling rate, a single isovalue does not accurately represent the fuzzy boundaries within the data which are caused by partial volume effects.
- Transparency has turned out to be important for understanding the spatial relations of complex vessel topologies. With the original implementation of non-polygonal isosurfaces transparent isosurfaces are difficult to accomplish. This is a result of the `GL_LESS`-metric which we used in the alpha test to avoid an unnecessarily high sampling rate.

To tackle these problems a dual-pass rendering method was developed, that allows the display of semi-transparent *thick* isosurfaces on hardware platforms with minimal requirements. The idea of this approach is to exploit the OpenGL stencil buffer to determine the interior of the isosurface. In the core of the algorithm each slice plane is rendered twice. In both rendering passes the alpha test is enabled as usual with the `GL_LESS`-metric. Note that since the alpha test belongs to the per-fragment operations, it is performed after the interpolation of the texture samples. In the first rendering pass the isovalue I is slightly

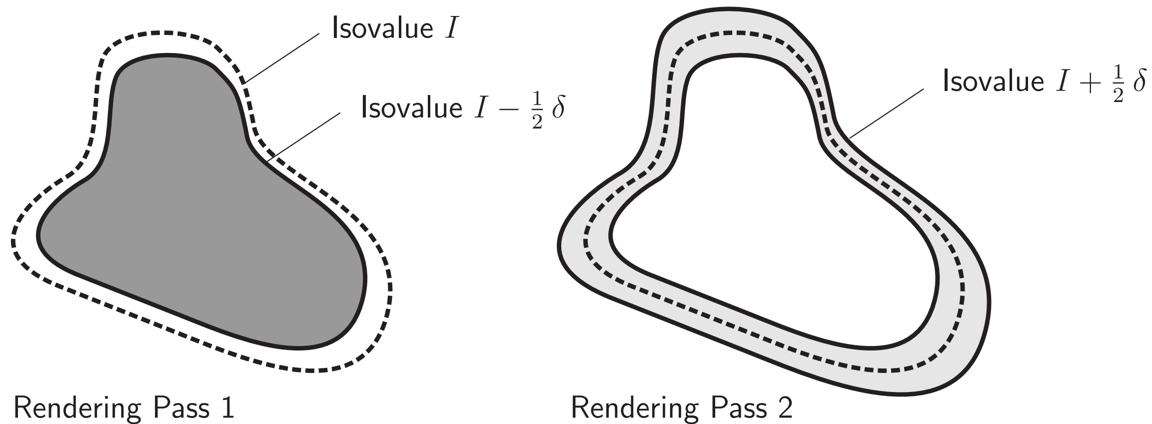


Figure 13.2: Dual-pass rendering for semi-transparent *thick* isosurfaces. In the first rendering pass the interior of the isosurface is rendered into the stencil buffer. In the second rendering pass the stencil buffer is used to discard the fragments in the interior.

shifted by a subtracting $\frac{1}{2}\delta$ as displayed in Figure 13.2. The resulting fragments are written to the stencil buffer. In the second rendering pass the isovalue I is shifted in the other direction by adding $\frac{1}{2}\delta$. Simultaneously the stencil test is used to discard all the fragments in the internal region which was determined by the first rendering pass. The result is an isosurface with a user-specified thickness δ. Additionally during back-to-front compositing of the stack of slice images alpha blending is also enabled and a constant alpha value is used to enable transparent isosurfaces.

13.4 Results

The results of the presented method for rendering of semi-transparent isosurfaces are displayed in Figure 13.3 and 13.4 for two different CTA data sets rendered on a PC system with an NVidia GeForce 2 Ultra GPU (see PC System B in Chapter 8). The dual-pass rendering procedure works in combination with both 2D- and 3D-textures. The presented approach is thus an efficient alternative visualization technique to the one presented in [65]. A major benefit of this approach is the low hardware requirements. The only specific hardware extensions required are multi-textures for the trilinear interpolation of slice images as described in Chapter 4 and a mechanism for dot-product calculation such as the popular OpenGL extension `EXT_texture_env_dot3`.

A clear visualization of the spatial relation between different vascular structures is provided by the use of transparency. Illumination effects additionally enhance the perception of depth. Compared to the original implementation of non-polygonal isosurfaces, the user-specified thickness of the isosurface is used to account for fuzzy boundaries that result from partial volume effects. With the presented approach the value of high-quality visualization of angiography data is available on inexpensive graphics hardware. By removing the

necessity of expensive high-end graphics workstations it enables high-quality visualization on the desktop PC of every physician.

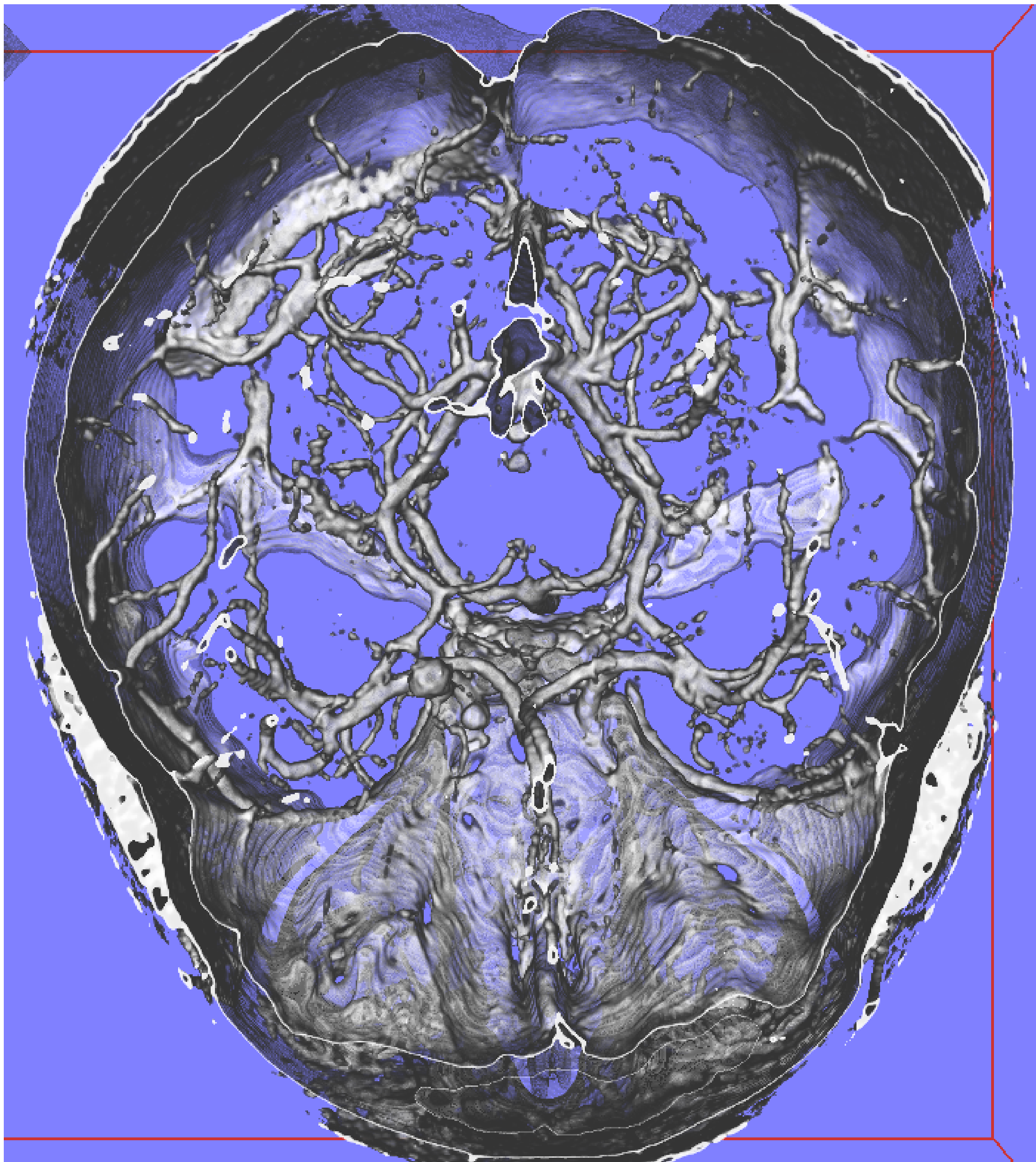


Figure 13.3: Example of semi-transparent isosurface rendering of CTA data for the visualization of intracranial vessels.

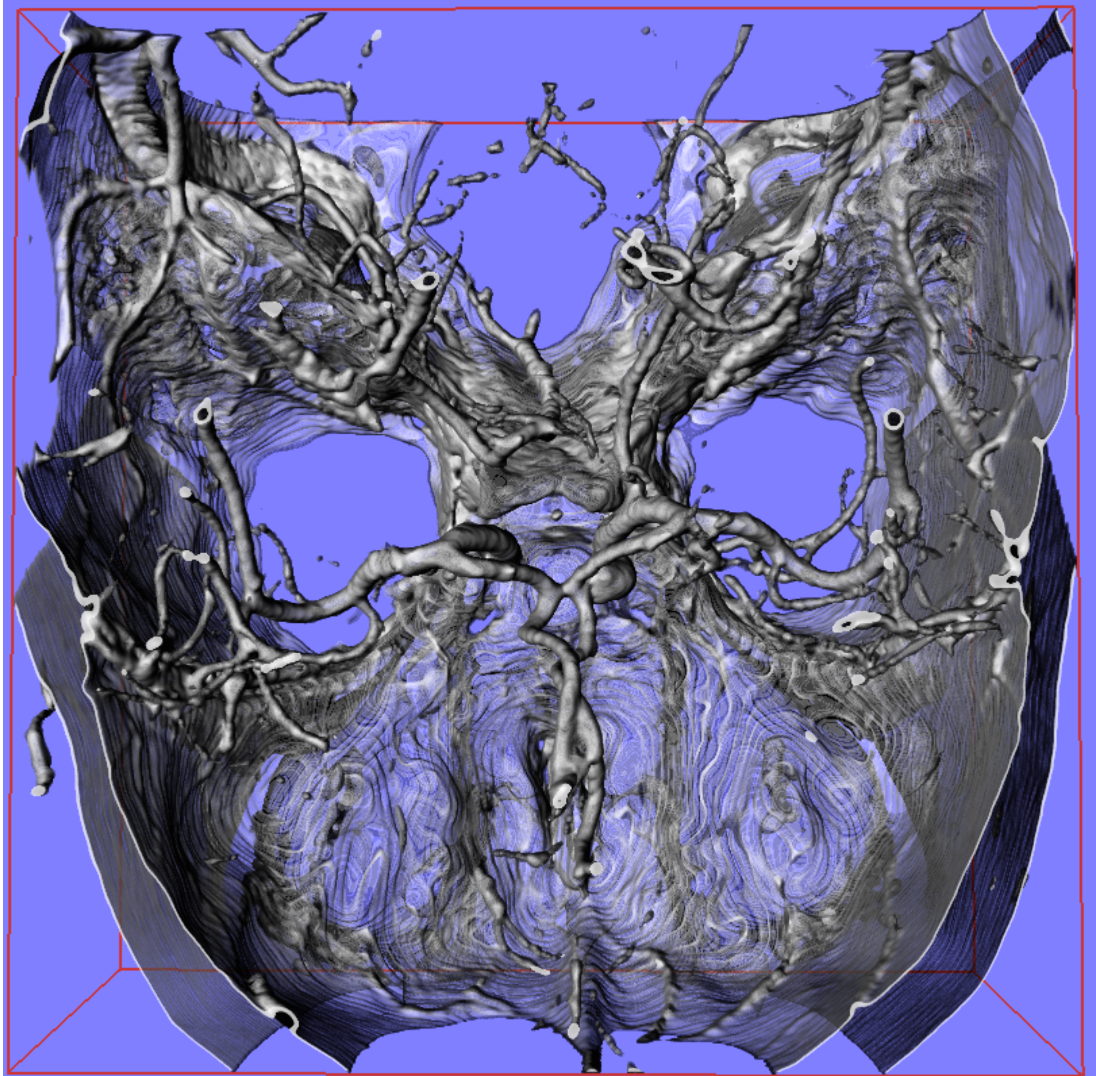


Figure 13.4: Example of semi-transparent isosurface rendering of CTA data for the visualization of intracranial vessels.

Chapter 14

Dural Arteriovenous Fistulae

In the previous section we have examined a case in which the structures of interest could be visualized simply by a transfer function or an isovalue threshold. This was possible because during data acquisition the image contrast between the target vessels and the surrounding tissue had been enhanced by the use of contrast dye. In this section we will examine a clinical problem of higher complexity, that does not allow the extraction of the target structures implicitly by a transfer function. This example demonstrates how to obtain high quality visualization results while minimizing the necessary segmentation operations.

Dural arteriovenous fistulae are vessel malformations in the area of the spinal cord. They can cause a variety of somatic diseases. The traditional way to examine such malformations is digital subtraction angiography (DSA). This medical imaging method requires the injection of contrast medium into several segmental arteries of the vertebral column in a highly difficult and time-consuming procedure. An additional drawback of DSA is the fact, that it only delivers projection images similar to traditional X-ray techniques. Due to the missing depth information it is considerably difficult to understand spatial relations of complex vessel structures.

As an alternative approach, a specialized MR sequence called MR-CISS is used to record volume data of the vertebral column. In this section we will introduce and examine a visualization approach based on such data. It supplements the traditional DSA examination by significantly reducing the number of dye injections which are required to determine the exact position of the fistula. The aim of this approach is not to substitute traditional DSA examinations completely, since DSA represents a well-known standard for such cases.

14.1 Background

The spinal cord is the large nerve trunk that runs along the vertebral column. The space between the spinal cord and the dura mater is filled with cerebrospinal fluid (CSF). In addition to the root nerves which have its origin in the spinal cord, also venous blood vessels run within the CSF. Like all veins, they are part of the low blood pressure system. Dural arteriovenous fistulae *(dAVF)* are pathologic connections between these veins and

the high-pressure segmental arteries of the vertebral column [58]. In the pathologic case such a short-circuit and the resulting pressure gradient causes a distension of the venous vessels and in turn a constriction of the nerve roots. This can lead to a variety of somatic diseases ranging from back pain to paraplegia and physical disability. Possible treatments of a dural arteriovenous fistula are coagulation of the pathologic structure or excision of the whole abnormal area during a neurosurgical intervention.

Even if the symptoms give reason for suspecting a dural arteriovenous fistula, it is often difficult to prove its existence, as this would require the detection of the feeding artery, the so-called *nidus* of the fistula. In a conventional DSA examination, the physicist injects contrast medium into several segmental arteries on both sides of the vertebral column and examines the diffusion of the dye. The feeding artery is thus determined in a trial-and-error process, which is extremely time-consuming. The injection of contrast dye into the vertebral column also comes with a certain risk of spinal cord injuries. In order to optimize this procedure it is necessary to reduce the number of dye injections to a minimum. A non-invasive way to obtain additional information of the vessel situation prior to the DSA examination is provided by magnetic resonance imaging (MRI), especially with a specialized MR sequence called MR-CISS.

14.2 The MR-CISS Sequence

As explained in Section 11.1.2, adjusting the set of MRI sequence parameters allows the differentiation between various tissue types in a very flexible way. By measuring the pulse response signal at different points in time (multi-spin echo sequences) and by combining these measured values to reduce visual artifacts, it is possible to obtain image data at high resolution.

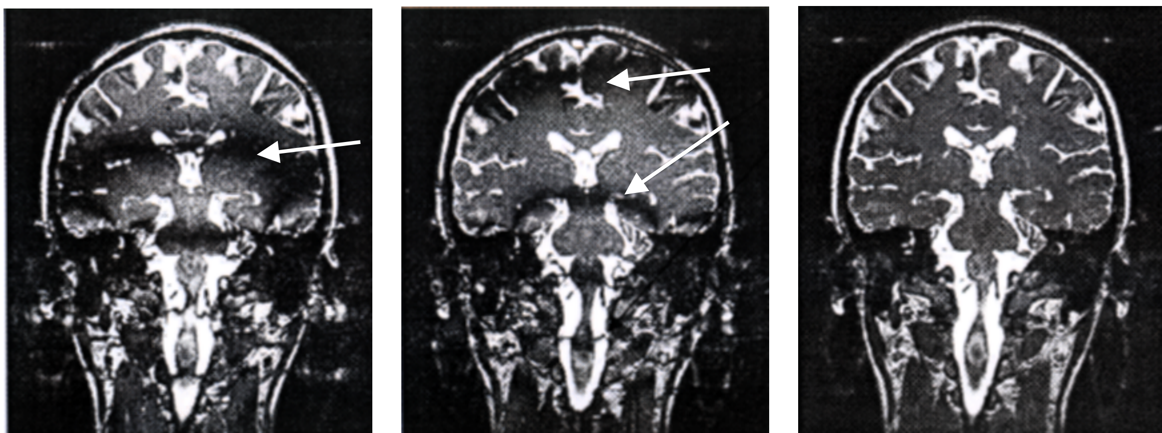


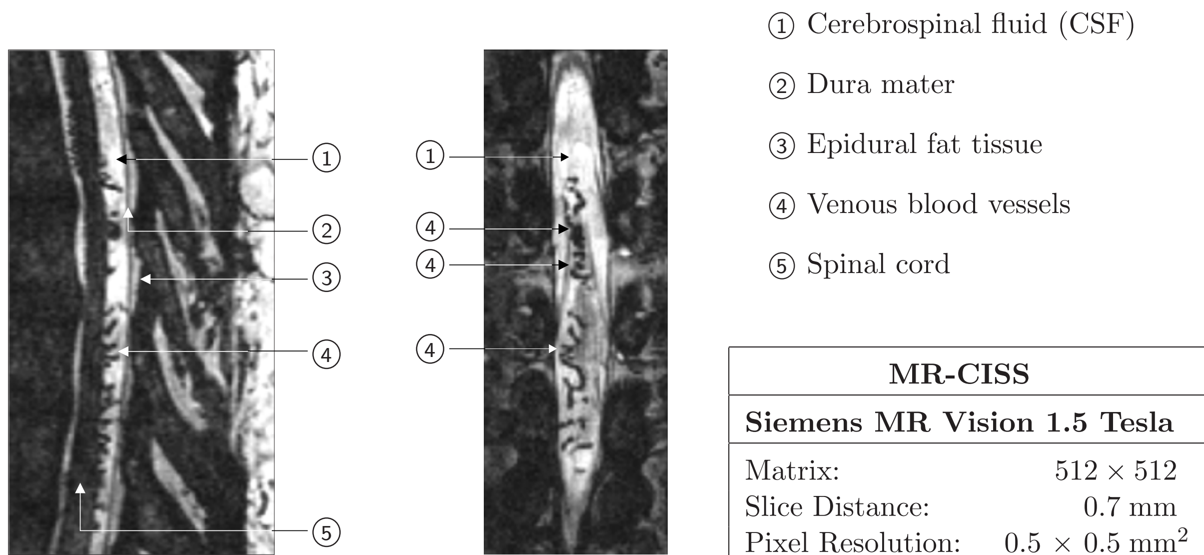
Figure 14.1: The MR-CISS sequence (right) combines two FISP sequences with alternating phase (left and middle) to remove band effects due to field inhomogeneities (arrows).

The MR-CISS[1] sequence is a Siemens proprietary, strongly T_2-weighted sequence, which consists of a pair of true FISP[2] sequences [28]. Although due to local field inhomogeneities a single FISP sequence is affected by dark phase dispersion artifacts, the combination of two FISP sequences with alternating phase of excitation pulses allows a complete reduction of these artifacts (See Figure 14.1). The image combination is automatically performed after data collection, adding some computation time to the reconstruction process. The MR-CISS sequence yields excellent contrast between soft tissue and cerebrospinal fluid at a high spatial resolution.

The slice images displayed in Figure 14.2 show high signal intensities for fluid and fat tissue while other soft tissue types and vascular structures (④) are represented by low signal. The space between the spinal cord (⑤) and the dura (②) is filled with CSF (①) which contains the target vessels. The dura in turn is surrounded by bone structures of low intensity and partly by epidural fat (③) of relatively high signal values. As a result MR-CISS data can replace traditional myelography, since it provides all the necessary information non-invasively [35].

However, to fully understand the spatial relationships of the vascular structures, an appropriate 3D reconstruction of the vessel tree is required. Since the spinal cord as well as the vascular and the bony structures are in the same range of data values, it is neither possible to apply a simple maximum intensity projection *(MIP)* nor to extract the target structures with a global transfer function. Explicit segmentation in general is a highly complex and time-consuming procedure. Due to partial volume effects it turns out to be almost impossible to obtain a valid segmentation in case of the vascular structures with

[1]CISS = constructive interference in steady state
[2]FISP = fast imaging with steady precision



MR-CISS	
Siemens MR Vision 1.5 Tesla	
Matrix:	512×512
Slice Distance:	0.7 mm
Pixel Resolution:	0.5×0.5 mm^2

Figure 14.2: MR-CISS images of the spinal column. Sagittal *(left)* and coronal *(right)* slice image.

their very limited spatial extent. As a solution to this problem we have developed a fast sequence of pre-processing operations. The aim of this procedure is not to explicitly extract the vessel structures, but to coarsely separate different regions which have a similar range of data values.

14.3 Pre-Processing

As mentioned above, we want to follow the strategy to isolate the region of CSF *including* the vascular structures by separating the whole area from the surrounding tissue. If we have obtained such a coarse segmentation, we can visualize the vessel structures by a local transfer function as outlined in Section 5.5. This approach is faster and much more convenient than an explicit segmentation of the vessels. The pre-processing pipeline we have developed consists of a sequence of 3D image processing operations described as follows. In some cases noise reduction has been necessary as an initial step to optimize the data for further pre-processing. Regions of higher homogeneity are obtained by anisotropic diffusion [52], while the exact object boundaries are preserved. Figure 14.3 outlines the sequence of further pre-processing operations.

1. In terms of image processing, the tiny vascular structures contained in the relatively large region of CSF can be interpreted as a high frequency detail. Low-pass filtering could remove these structures, however at the expense of blurring the object boundaries. To avoid this, we apply a morphologic 3D gray-value closing operation with a

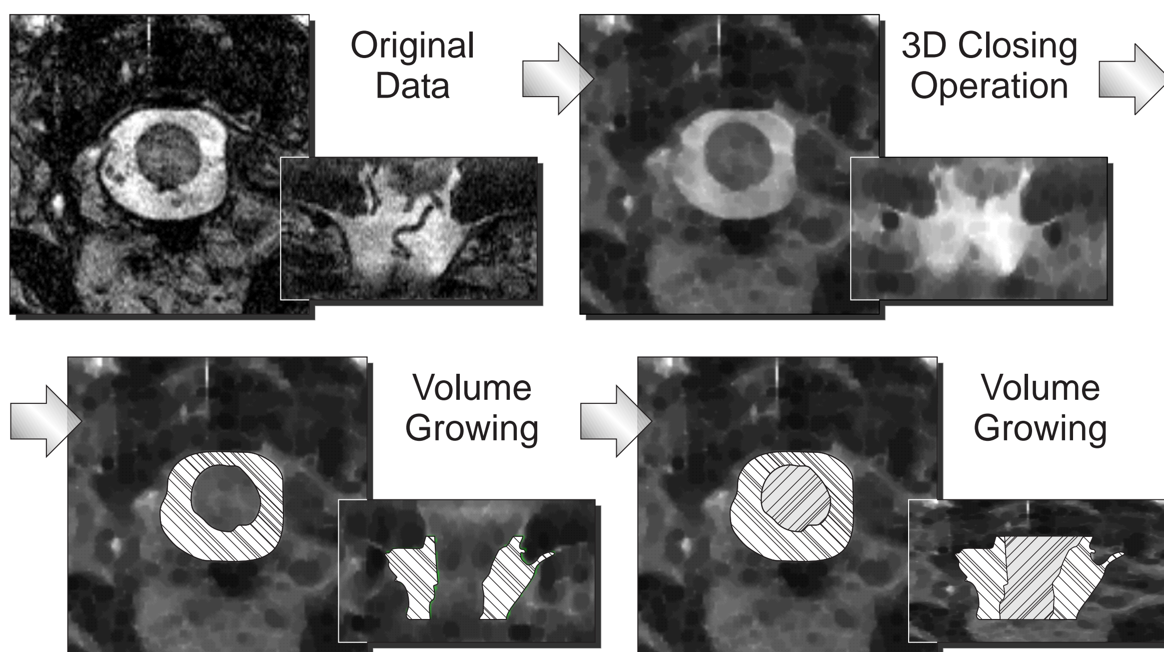


Figure 14.3: The sequence of image processing operations for the coarse segmentation of MR-CISS data.

spherical filter kernel. This operation removes the dark vascular structures within the region of CSF of high intensity, such that the whole region can be easily extracted by a simple threshold operation. For optimal results the size of the spherical filter kernel must be greater than the largest vessel diameter, but smaller than the diameter of the spinal cord.

2. Successively, the closed region of CSF is extracted by volume growing. The segmentation is computed stepwise starting at the top of the vertebral column. Bounding boxes are used to prevent volume offshoots.

3. Although this first segmentation already contains all the interesting structures, it has proven useful to additionally obtain a segmentation of the spinal cord. Again, this is easily achieved with volume growing of the closed image data, using the previous segmentation as a boundary.

Due the low computational cost, the presented user-guided image processing sequence leads to fast and robust segmentation results. Manual correction of segmentation errors were necessary only in very few cases. Based on the segmentation results, the original image data is attributed using unique tag numbers for the CSF, the spinal cord and the surrounding dark tissue. Subsequently local transfer functions as described in Chapter 5.5 are assigned. In order to avoid the visual artifacts that result from interpolation across tag boundaries, we have chosen to use a pre-classification approach based on pixel transfer. To speed up the assignment in clinical routine transfer function templates as described in Chapter 6 were used in combination with manual adjustment.

14.4 Results

Throughout the clinical study the described visualization technique for dural arteriovenous malformations was applied to the image data of several patients. Figure 14.4 shows the visualization results for a patient with a dural arteriovenous fistula in the area of the brain stem. Information about the complex vascular structures and their spatial relations is extremely difficult to obtain from the traditional DSA projection images (A). The coarse segmentation generated by the presented sequence of pre-processing steps and the resulting visualization provides 3D information about the vessel structures (B – D) and their relation to the surrounding anatomy such as the medulla oblongata (green).

As another example from the same clinical study, Figure 14.5 shows data of a patient with a dural arteriovenous fistula in the area of the thoracic spine. The distention of the venous vessel is clearly visible. In this case the closing operation has lead to small visual artifacts caused by epidural fat tissue close to the dura mater. In some cases these structures can be removed by clipping planes. Despite of these segmentation artifacts the vascular structures are represented accurately.

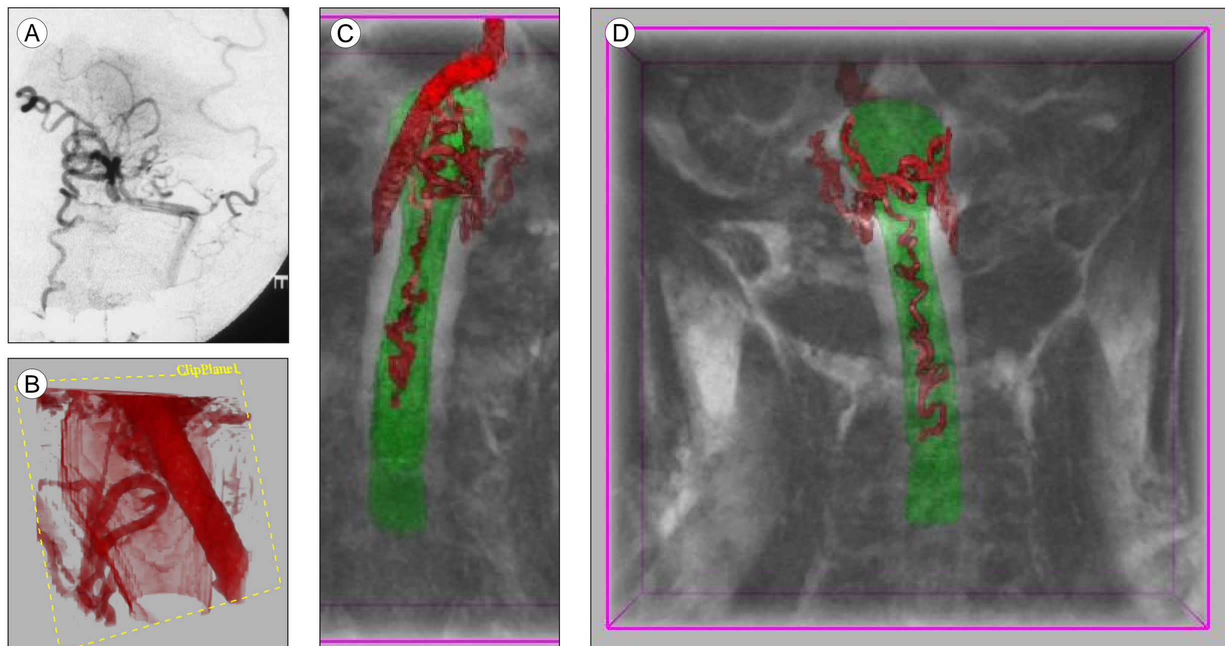


Figure 14.4: Intricate vascular structure in the area of the brain stem of a patient with a dural arteriovenous fistula. The spatial relations of the vessels are extremely difficult to obtain from standard DSA projection images (A). MR-CISS data provides 3D information of the vasculature (B) as well as additional anatomical structures (C and D).

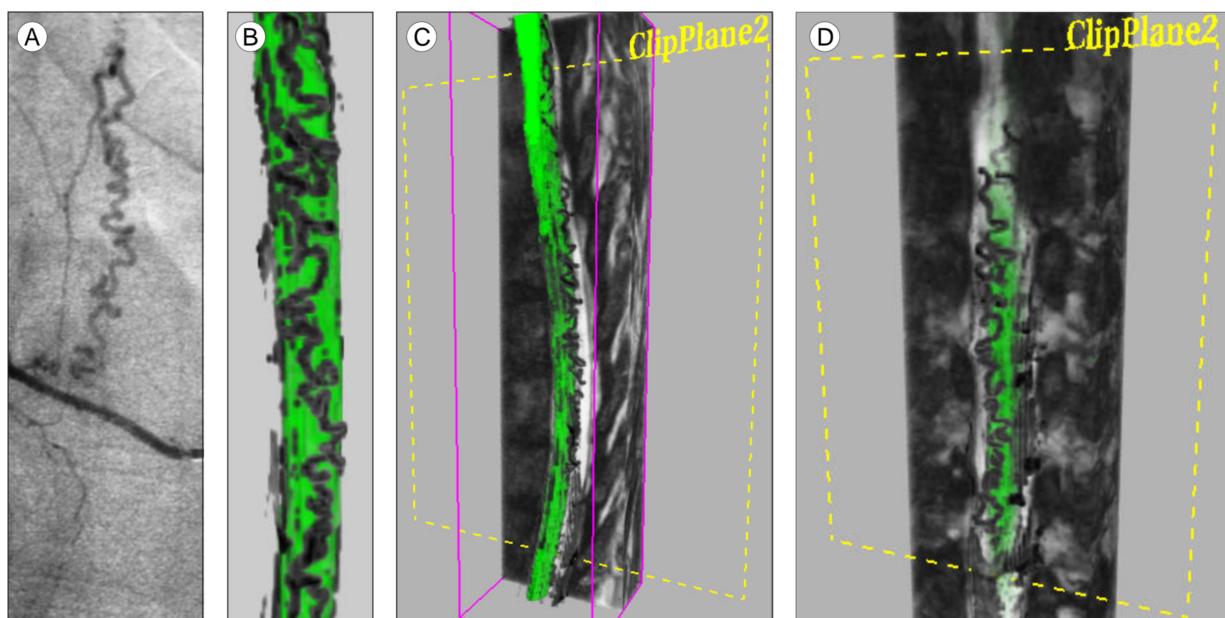


Figure 14.5: Dural arteriovenous fistula in the area of the thoracic spine. The distention of the venous blood vessel caused by the fistula is clearly visible in the DSA image (A). The same structures are visualized with the non-invasive technique based on MR-CISS data (B – D). Segmentation artifacts are partly visible in (B).

In several cases manual post-processing was necessary to obtain images free of segmentation artifacts. For an application in clinical practise however, the presented semi-automatic pre-processing sequence delivers valuable information to facilitate the DSA examination. Throughout the clinical study the number of dye injections for DSA could be significantly reduced.

In an interesting case of a dural arteriovenous fistula in the area of the lumbar spine, the feeding artery was located outside the area of CSF. After multiple DSA examinations the existence of the fistula still could not be verified. The acquisition of MR-CISS data however immediately displayed the distension of the venous blood vessels and a fistula coming from the right internal iliac artery could be identified. With this information obtained from MR-CISS data the nidus of the fistula was finally verified using DSA. This example clearly demonstrates the benefit of the combination of DSA with the presented visualization technique in clinical practise.

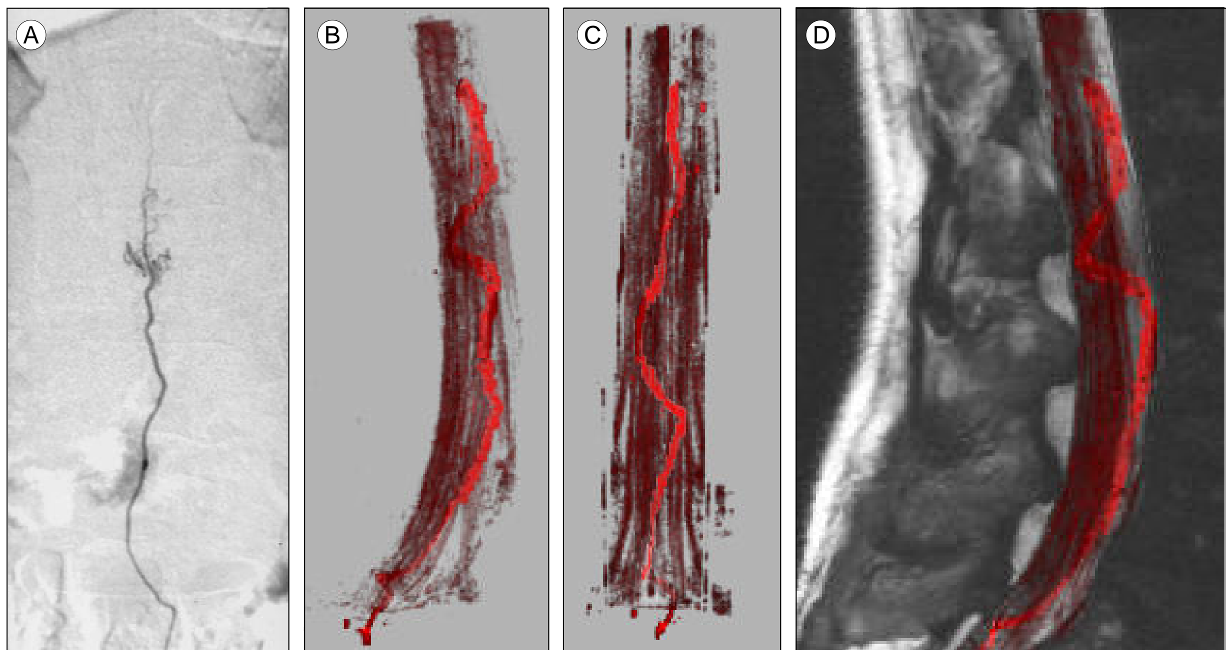


Figure 14.6: Dural arteriovenous fistula in the area of the lumbar spine coming from the right internal iliac artery. MR-CISS (B – D) data displays the distension of a venous blood vessel surrounded by the roots of the spinal nerves. With this information obtained from MR-CISS data the nidus of the fistula was finally verified using DSA (A).

Chapter 15

The Vertebral Column

This chapter reports on a clinical study on the analysis of the vertebral column and related diseases. The aim of this study was to improve the diagnosis and therapy planning of discogenic diseases in orthopaedics.

From the image processing point of view, the procedure is similar to the visualization of dural arteriovenous fistulae described in the previous chapter. However, this study demonstrates the benefit of an optimized adaptation of the imaging sequence to the specific visualization problem.

15.1 Background

The spinal column forms the supporting axis of the human body. It consists of an articulated series of vertebrae connected by ligaments and separated by more or less elastic intervertebral fibrocartilages (Figure 15.1). A single vertebra consists of a cylindrical body, various spinous and articular processes and a dorsal arch providing a protected passage for the spinal cord. Within this spinal canal, the spinal cord serves as a pathway for nervous impulses and gives off several pairs of spinal nerves which lead through the vertebral foramina to the various parts of the limbs.

Degenerative discogenic diseases of the vertebral column are mainly caused by malformation or dislocation of the intervertebral discs and deformations of the spinal cord. Slipped or ruptured disks can cause severe spinal stenosis, narrowing of the spinal canal and a resulting constriction of the nerves. In the same context the term spondylolisthesis refers to a forward displacement of a lumbar vertebra on the sacrum which again produces a compression of nerve roots. For the diagnosis and the therapy planning of intervertebral disk diseases detailed information of the relevant structures and their spatial relations is required.

In clinical practise, the traditional method to examine complex cases of extreme spinal stenosis and herniated intervertebral disks requires the injection of contrast dye into the spinal subarachnoid space (x-ray myelography). This is an invasive and time-consuming procedure that comes with a considerable risk for the patient.

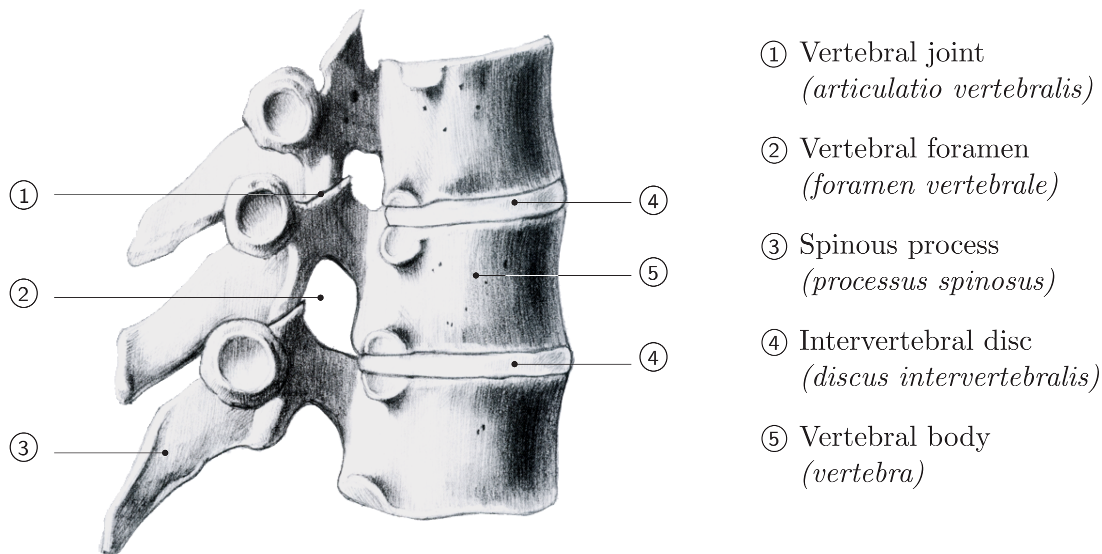


Figure 15.1: Anatomical structures related to the vertebral column.

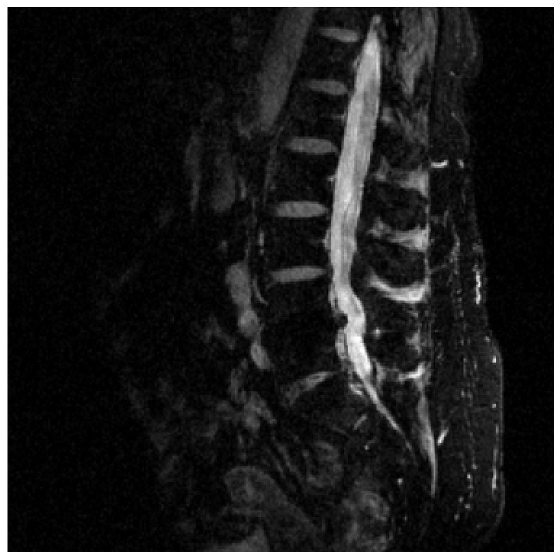
To these ends an alternative examination method based on a highly optimized MRI sequence was developed at the Division of Neuroradiology of the University Erlangen-Nuremberg. For the visualization of the data we again propose a fast sequence of simple image-processing operations for a coarse separation of anatomical structures similar to the visualization of dural arteriovenous fistulae described in the previous chapter. Preliminary results of this study have been published in [61].

15.2 Data Acquisition

The basis of the completely non-invasive examination procedure is the MR-MEDIC[1] sequence. In the resulting slice images the spinal canal filled with cerebrospinal fluid (CSF) is depicted with high signal intensity. The sequence is optimized to provide high contrast between the CSF and the nerve roots. Additionally, it allows the differentiation of intervertebral disks from bony vertebrae. Compared to a first approach based on the MR-FISP[2] sequence [34], the MR-MEDIC sequence delivers an improved in-plane resolution for more accurate measurement. Compared to the MR-CISS sequence described in the previous chapter, the MR-MEDIC sequence also removes the problems caused by epidural fat tissue.

In order to investigate the change in shape and relative position of the elements that form the spine, functional analysis is performed. For each patient three MR-MEDIC data sets are recorded, one for the normal resting position, one for the forward bending (inclination) and one for the backward bending of the spine (reclination). As in some cases

[1]MEDIC = Multi Echo Data Image Combination
[2]FISP = Fast Imaging with Steady State



MR-MEDIC	
Siemens MR Magnetom Symphony 1.5 Tesla	
Matrix:	256×256
Slice Distance:	1.17 mm
Pixel Resolution:	1.17×1.17 mm^2

Figure 15.2: MR-MEDIC slice image of the spinal column provides high contrast between CSF and nervous structures. The separation of intervertebral disks from bony structures is also facilitated significantly.

it is very difficult for the patient to hold the functional position for the duration of the scan, an MR-compatible device [33] supports lifting and lowering of certain parts of the spinal column. Especially for the examination of elderly patients and in case of herniated intervertebral disks, this device has turned out to be an indispensable aid.

15.3 Pre-Processing

For the visualization of the different structures contained in the MR-MEDIC data set we follow the same strategy as described in the previous chapter. A coarse separation is obtained by a fast sequence of image processing operations and the detail structures are extracted subsequently by local transfer functions.

Analogous to the visualization of dural arteriovenous fistulae a coarse separation of the area of CSF including the roots of the spinal nerves is achieved by a morphological closing operation, followed by a semi-automatic threshold operation. For the segmentation of bone structures a slightly different procedure is required as illustrated in Figure 15.3. Starting with the original volume data, an anisotropic diffusion filter is applied for noise reduction. In the MR-MEDIC sequence bony structures are represented by intensity values of about zero. The only structures that lie within the same range of data values is the surrounding air and the main arterial trunk, the Aorta. All of these structures are spatially separated from each other such that the extraction of the bone structure can be performed by a simple volume growing operation with a seed point manually positioned inside a vertebral body. Finally the extracted region is inverted as displayed in Figure 15.3*(bottom right)*. Alternatively, this inversion can also be achieved by means of a local transfer function.

An additional segmentation of the intervertebral disks is also achieved by semi-automatic threshold operations with user-specified seed points. Based on the extracted regions, the original volume data set is attributed with unique tag numbers. During volume rendering the local transfer functions are assigned for each tagged region separately.

In addition to the application of local transfer functions, the voxel-based segmentation of the CSF is used for quantitative measurement of the constriction of the CSF in dependance of the functional position. This is achieved by manual positioning of a measuring widget on the same level with the intervertebral disks. The volume of CSF is determined by counting the voxels contained inside a bounding box of fixed size that is attached to the widget. The comparison of volume measurements for different functional positions allows the quantitative validation of the narrowing of the CSF at multiple levels.

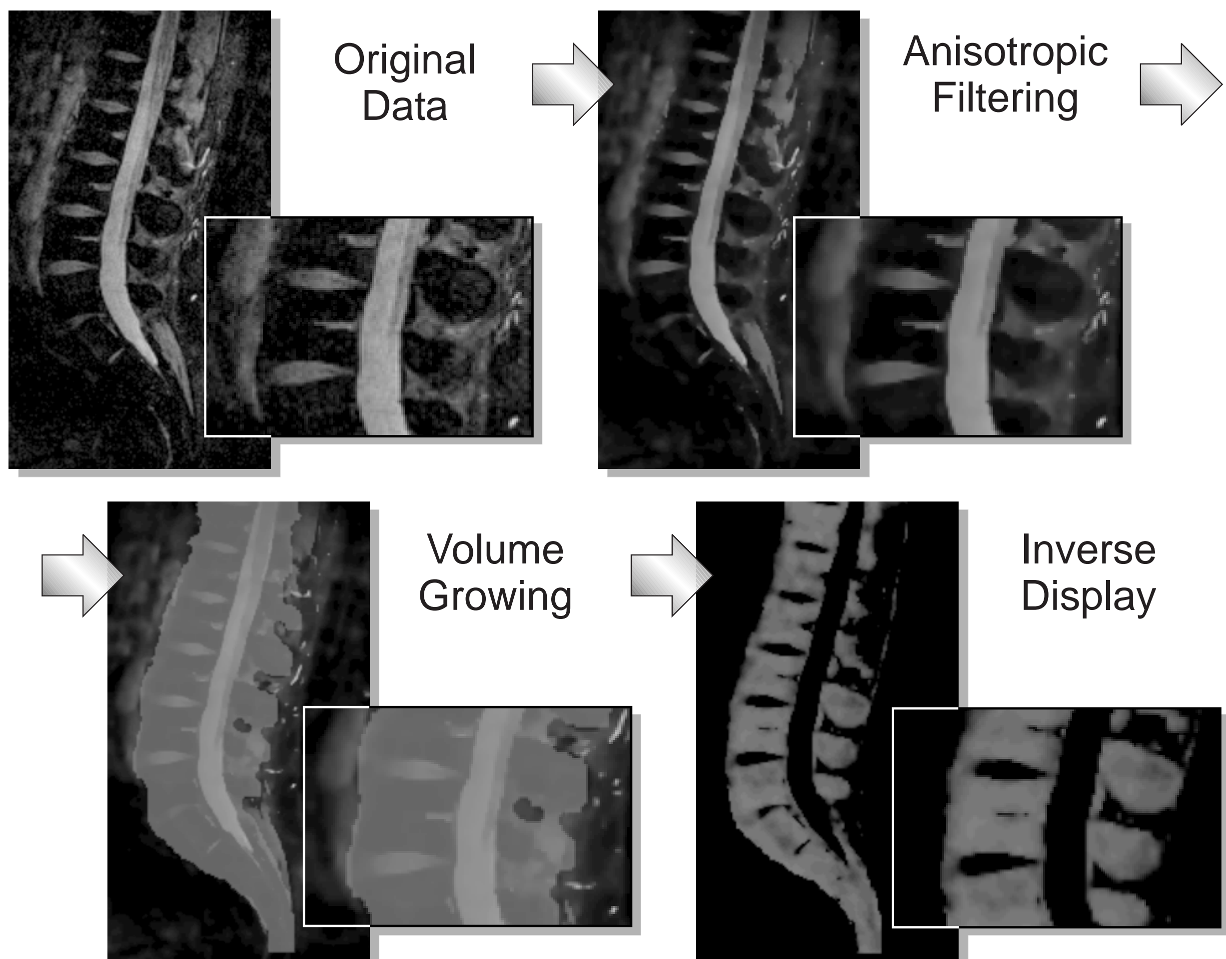


Figure 15.3: The segmentation of the MR-MEDIC data is performed with a fast sequence of 3D image processing operations.

15.4 Results

Within the scope of a clinical study, the presented approach was applied to a high number of patients with severe spinal stenosis and spondylolisthesis as well as to several healthy volunteers. In addition to traditional x-ray myelography, MR-MEDIC data was recorded in normal position, inclination and reclination. In some cases post-myelographic CT images were acquired. For pathologic cases post-operative data was used to compare the anatomical structures before and after the intervention.

As an example, Figures 15.4, 15.5 and 15.6 show the examination results for a 85 year old patient with a multi-segment spinal stenosis at the levels L2/3, L3/4 and L4/5. Due to the speed of the contrast medium, the assessment of the spinal stenosis is difficult in conventional x-ray myelography (Figure 15.4A, 15.5A and 15.6C). Compared to these projection images, the coarse segmentation of the spinal subarachnoid space filled with CSF leads to appropriate 3D visualization and an exact quantification of the spinal stenosis as displayed in the Figures 15.4B and 15.5B. Based on the segmentation of CSF, volumetric measurement at the levels of the stenotic segments shows a significant decrease of CSF in reclined position. The Figures 15.6A and B display the segmented region of CSF in

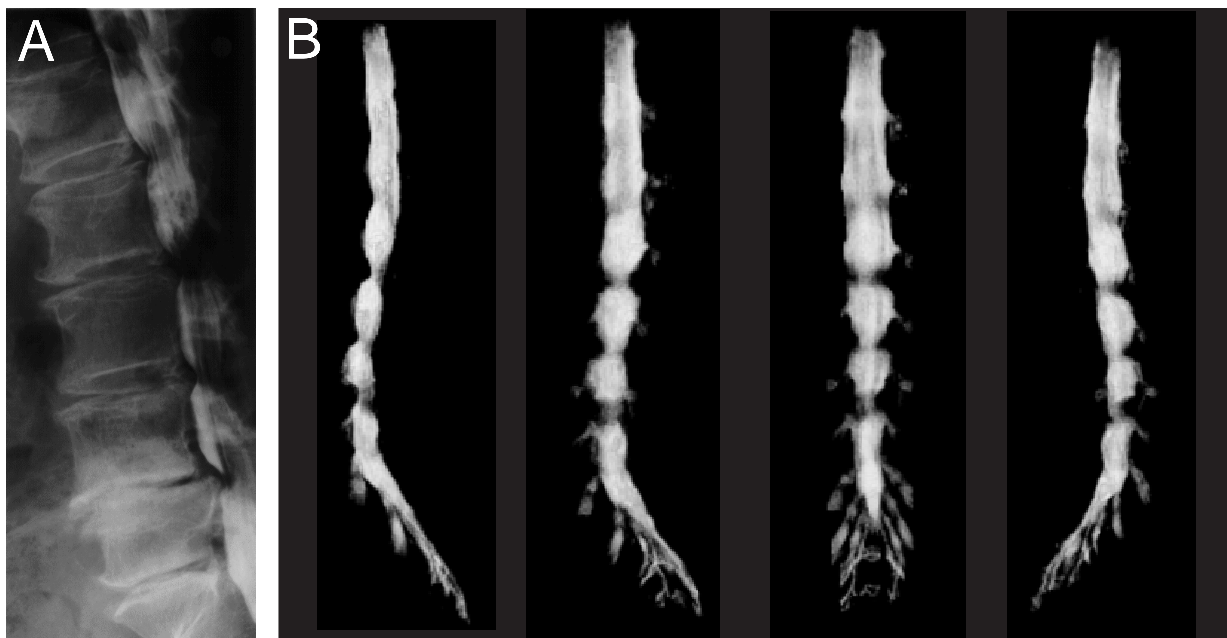


Figure 15.4: Comparison of traditional x-ray myelography (A) with direct volume rendering (B) MR-MEDIC data in inclination for a 85 year old patient with multi-segment spinal stenosis at the levels L2/3, L3/4 and L4/5. The quatification of the spinal stenosis in conventional x-ray myelography projection images is very difficult. 3D visualization of the segmented spinal subarachnoid space allows efficient examination of the pathology. The corresponding images in case of reclination are displayed in Figure 15.5

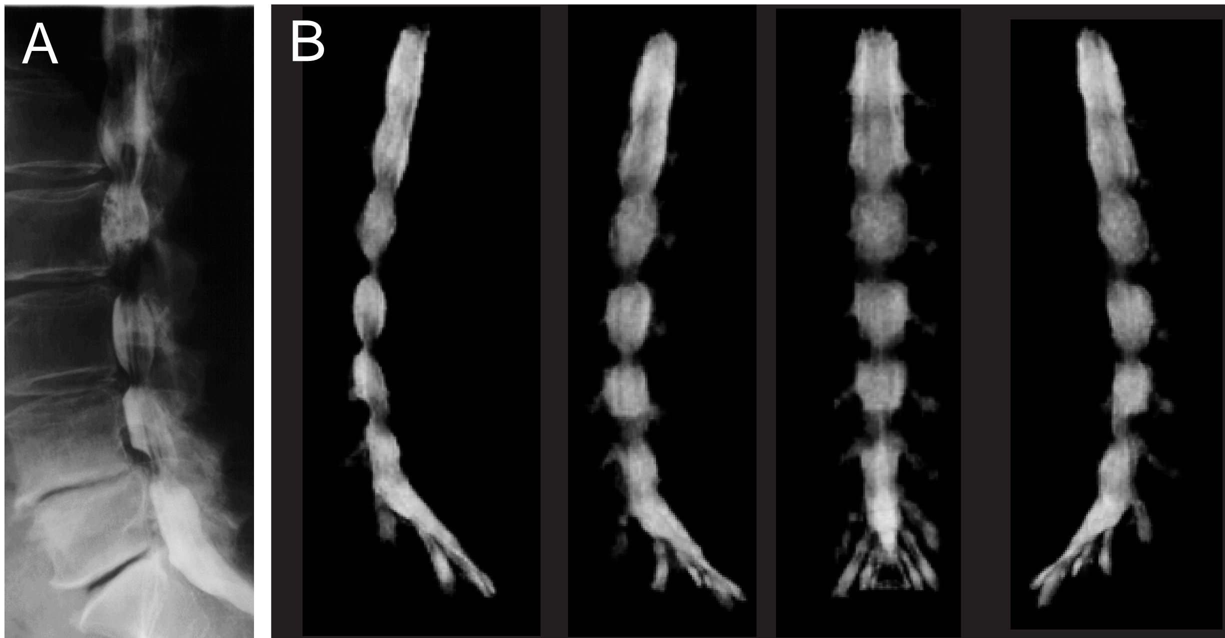


Figure 15.5: Comparison of traditional x-ray myelography (A) with direct volume rendering (B) MR-MEDIC data in reclination for a 85 year old patient with multi-segment spinal stenosis. The corresponding images in case of inclination are displayed in Figure 15.4.

relation to the vertebra and the intervertebral disks using local transfer functions for the extracted regions. The comparison of the inclination to the reclined position clearly shows the extremely limited range of movement. As a result of osteochondrosis the intervertebral disk at level L5/S1 is not visible at all. An axial slice image from post-myelographic CT data (Figure 15.6 (D) shows a calcification of the flaval ligaments (arrow), which finally caused the constriction of the CSF.

Examination results of a different patient evaluated within the scope of the described clinical study are shown in Figure 15.7. The images show volume visualization of the spinal subarachnoid space including the roots of the spinal nerves in combination with surrounding anatomical structures in different functional positions. In this case of a 60 year old patient with minor spinal stenosis and spondylolisthesis at the level L4/5, a significant narrowing of the CSF is visible in the pre-operative data (arrows). After the diagnosis of spondylolisthesis, the displacement of the lumbar vertebra which was the cause for the spinal stenosis has been removed in a surgical procedure. The results of this intervention have been validated by post-operative MR-MEDIC data. The visualization results clearly show that the spinal stenosis was successfully removed.

Throughout the clinical study, the combination of the highly specialized MR-MEDIC sequence, a fast and convenient pre-processing sequence and high-quality volume visualization with local transfer functions has proven its value for the diagnosis of discogenic diseases. In comparison to traditional x-ray myelography, the developed approach represents

a completely non-invasive visualization technique, which allows the interactive analysis of the spatial structures in 3D.

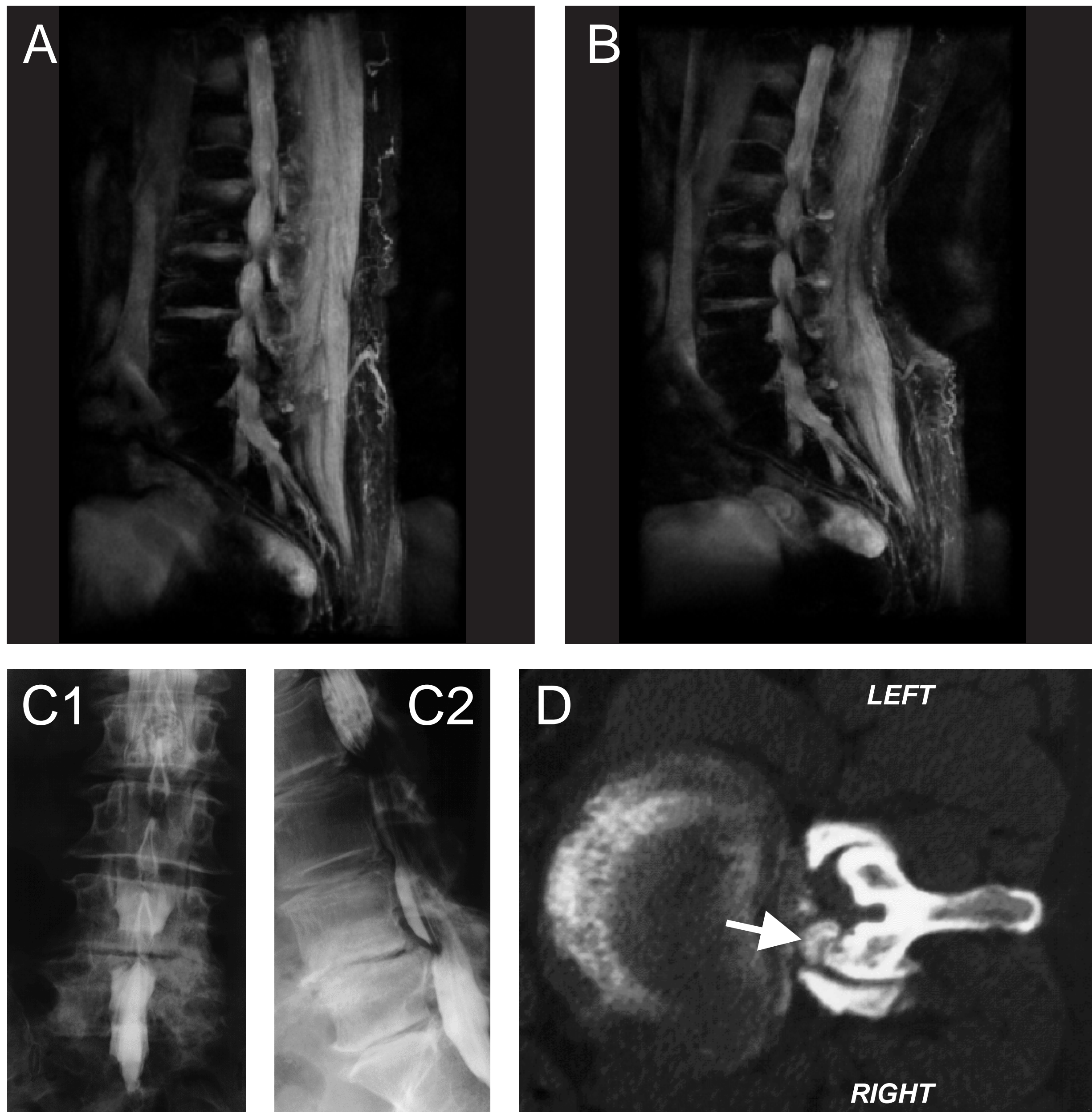


Figure 15.6: Direct volume rendering of functional MR MEDIC data with local transfer functions in inclination (A) and reclination (B). Additionally, conventional x-ray myelography (C.1 frontal, C.2 lateral view) and post-myelographic CT data (D) have been recorded. The axial slice image from the post-myelographic CT (D) shows a thickened flaval ligament(arrow) caused by calcification which lead to the spinal stensosis.

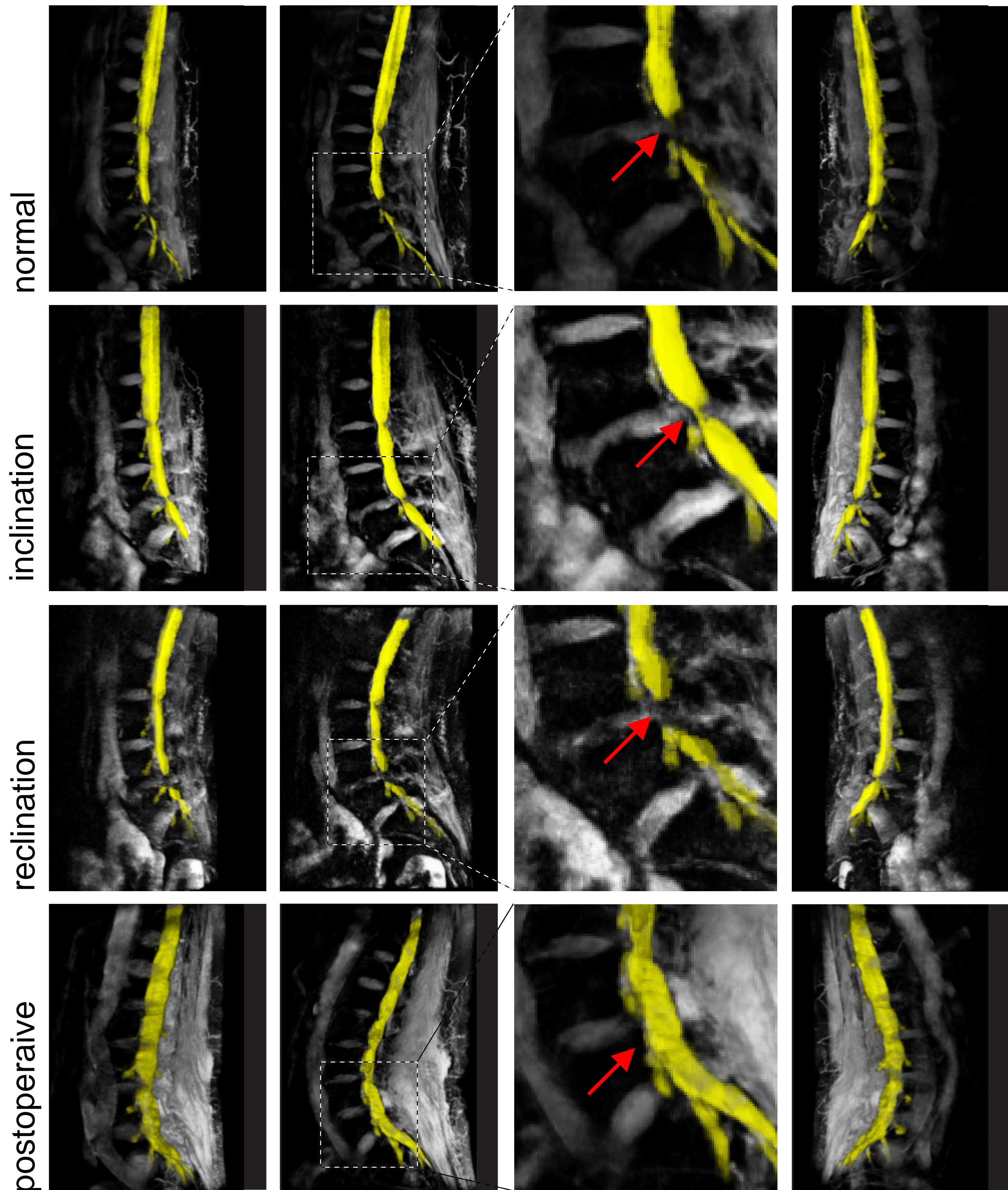


Figure 15.7: MR-MEDIC data of a 60 year old patient with minor spinal stenosis and spondylolisthesis at the level L4/5. Preoperative data recorded in normal position, inclination and reclination shows a significant gap (arrows) at level L5. After the surgical intervention the constriction of CSF has been successfully removed as documented in postoperative data (bottom row) recorded in normal position.

Part IV

Conclusion

Chapter 16

Summary

Efficient volume visualization techniques have been topic of active research over the last decade. Although surface extraction techniques are still important for simulation and rapid prototyping, indirect techniques are more and more replaced by interactive direct approaches, which display the volume object as a semitransparent medium. Motivated by the growing acceptance in medicine, natural science and engineering, in recent years volume rendering techniques have evolved from prototype implementations in research laboratories to commercial desktop products in a variety of application areas.

From experience in scientific environments, both *interactivity* and *image quality* are basic requirements. For intuitive exploration of the data, volume rendering techniques must be interactive in every aspect. This comprises the rotation of the volume in 3D and the changing of viewing parameters as well as the adjustment of shading and classification functions. The accuracy related to the resolution of spacial structures inherent in the data is defined by a physical model of light propagation in combination with the sampling theory for discrete volume data. In addition to the correctness determined by the physical model, image quality is also influenced by visual artifacts that are caused by the specific rendering algorithm.

In this thesis, ray-casting is used as reference in terms of image quality. This physically motivated model is directly derived from the equation of radiative transfer. Neglecting the scattering part allows the integration of the emission and absorption of radiation along linear rays. For discrete volume data, numerical integration requires the reconstruction and the resampling of the discrete data. Due to the high computational expense for accurate ray integration, traditional ray casting approaches do not provide high frame rates. In order to achieve interactive performance, hardware accelerated methods have been developed. The aim of such approaches is to provide image quality which comes as close as possible to the results of ray-casting while sustaining interactive frame rates during interaction. In general there are two different categories of hardware accelerated approaches.

In recent years, several concepts for *dedicated volume rendering hardware* have been presented. Although there are many interesting proposals, only very few implementations have ever left the simulation stage. Up until now the most successful hardware implementation is the VolumePro board. By the time this thesis is printed, the second generation

VolumePro boards will be releases, which is expected to render a 512^3 volume data set at a frame rate of 30Hz. Although this represents a break-through in performance of high-quality volume rendering, the *availability* of the solution is limited by the expensive hardware. In consequence, this thesis investigates approaches that exploit the texturing capabilities of modern general purpose graphics hardware.

The conventional 2D-texture based method represents a hardware accelerated implementation of the shear-warp algorithm. Although this implementation is extremely fast and available on almost all hardware platforms, visual artifacts significantly degrade the image quality. One reason for this is the inconsistent sampling of viewing rays which does not provide adequate results with respect to ray-casting. Increasing the sampling rate is not possible with the conventional 2D-texture based method. This, however, is necessary to account for transfer functions with a high-frequency component. These problems are solved by the use of 3D-textures with hardware accelerated trilinear interpolation. They enable implementations of high quality volume rendering at interactive frame rate. Despite the reduced memory requirements compared to 2D-textures, severe problems arise with the growing size of the data sets. The splitting of large volumes into smaller portions which fit entirely into texture memory turns out to be extremely inefficient with respect to memory bandwidth and load balancing.

In this context 2D-multi-texture based implementations are introduced which represent a hybrid solution of conventional 2D- and 3D-texture based methods. The image quality of these new approaches is equivalent to 3D-textures, while the benefit of the more efficient memory management of the 2D-texture based solution is preserved. Advanced features of modern consumer PC graphics boards, such as multi-stage rasterization, pixel shaders and dependent textures enable the implementation of new efficient volume rendering methods. Local illumination effects are integrated into the rendering approach either as implementation of per-pixel illumination with dynamic light sources or as pre-computed reflection maps which cache the incident light at a single point in space. The performance of all mentioned texture based approaches has been analyzed on different platforms. Several supplements to texture based volume rendering are described for performance enhancement, for efficient clipping and for flow visualization.

The appearance of a semi-transparent volumetric object is determined by a transfer function which classifies the voxels according to their original intensity value and assigns the emission and absorption values required for ray integration. This classification can be performed either before or after the interpolation. The superiority of post-classification is demonstrated by respective image material and the higher accuracy is documented with respect to sampling theory. Several efficient implementations of pre- and post-classification are described. The design of adequate transfer functions is a challenging task which involves both knowledge about the rendering algorithm as well as detailed information about the interesting structures inside the data set. Automatic methods for transfer function design try to derive an optimal transfer function based either on the analysis of resulting images or on some data-driven mechanism. A novel approach to efficiently adapting an established transfer functions to a series of new data sets is analyzed.

Real-time deformation of volumetric objects is a scientific problem that has not yet

been handled sufficiently. To these ends this thesis has introduced two different approaches, which have been especially designed for intuitive modeling and for automatic registration, respectively. The first method is based on a strict separation of shape from appearance. It leverages 3D-textures in combination with an efficient algorithm for the intersection calculation between slice planes and deformed polygonal surfaces. The second approach was especially designed with regard to a hardware-accelerated implementation by substituting traditional tetrahedra deformation by an adaptive decomposition into an octree of hexahedra.

The application of the described methods in medicine is documented by several case studies. The comparison of direct volume rendering and surface reconstruction techniques is exemplified for the visualization of tiny structures related to the inner ear. A medical case study on intracranial aneurysms demonstrates the benefit of direct volume rendering in clinical practise. Hybrid approaches that combine explicit segmentation with local transfer functions have been developed for the diagnosis of spinal vessel malformations and for the analysis of discogenic diseases of the vertebral column. These highly specialized case studies document the benefit of a careful analysis of the clinical problem and an adequate adaptation of both the data acquisition and the visualization procedure.

16.1 Future Challenges

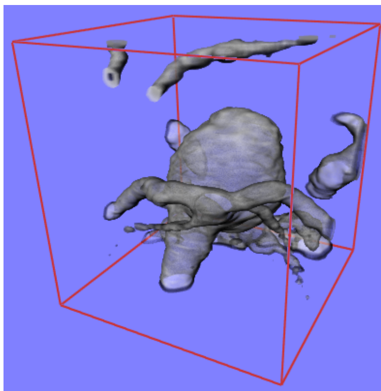
Increasing the availability of interactive high quality volume rendering is a first step towards an improvement of the acceptance of such techniques in clinical and scientific environments. However, further steps are required.

The user handling of volume rendering applications must be improved. The operation of such an application software must be decoupled from detailed knowledge of the underlying algorithm. This is especially true for data classification. From experience, manual transfer function design is not widely accepted among non-expert users. Image-driven techniques such as thumbnail selection might be intuitive, but they are usually non-deterministic and extremely time-consuming. Novel data driven approaches are very promising, but still far from being mature.

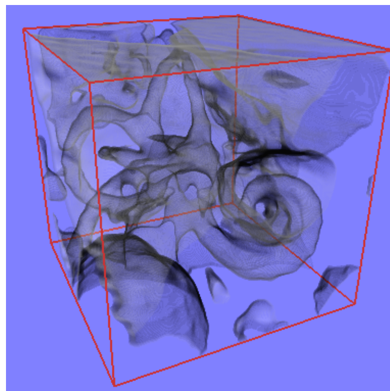
Especially in clinical environments, reproducibility is extremely important for the documentation, the verification and the comparison of examination results. The algorithm for the adaptation of existing classification functions to different data sets represents a first step towards an improved reproducibility. Further improvements are required for the implementation of fully automatic analysis tools to be applied in visualization services via world wide web. Apart from this, existing volume rendering applications should be expanded for intuitive interaction with the volume data such as improved navigation, picking, measurement of spatial relations, collision detection and interactive segmentation in 3D.

Appendix A

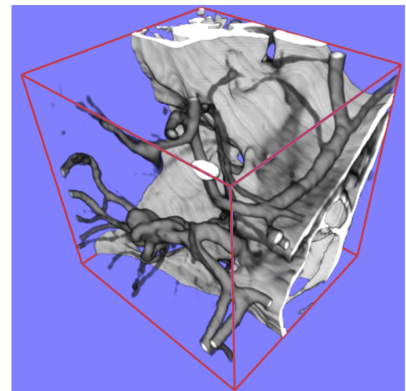
Data Sets



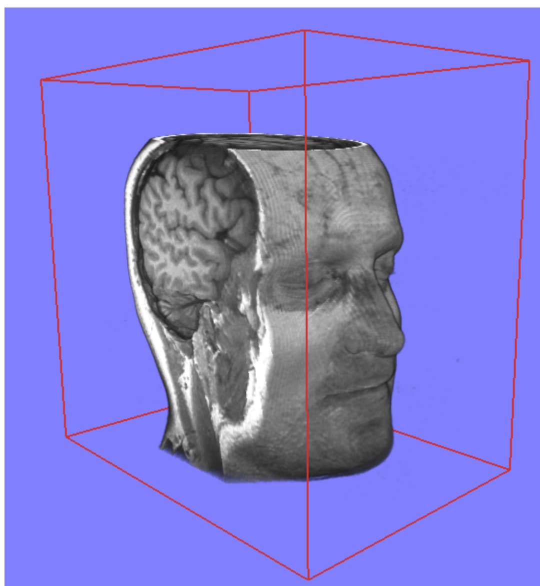
CTA Aneurysma Detail
256 kB (64 x 64 x 64)



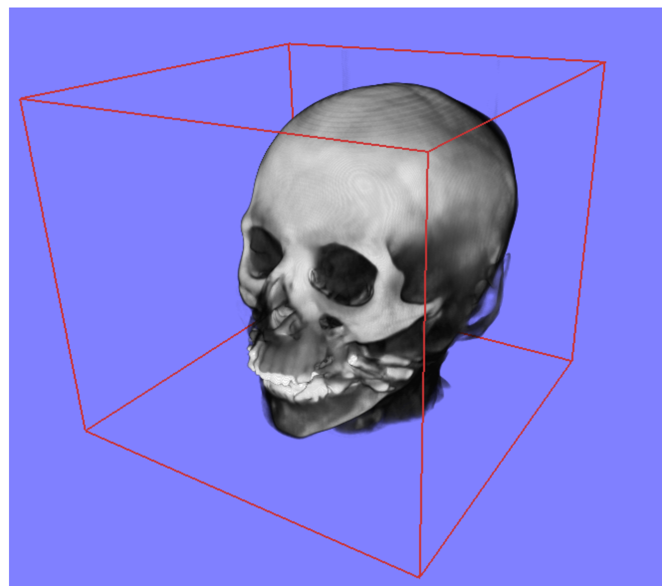
CT Inner Ear Detail
1 MB (128 x 128 x 64)



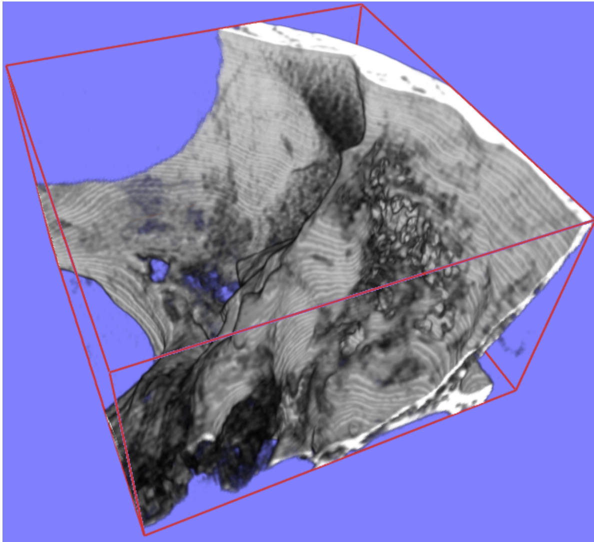
CTA Aneurysma Detail
2 MB (128 x 128 x 128)



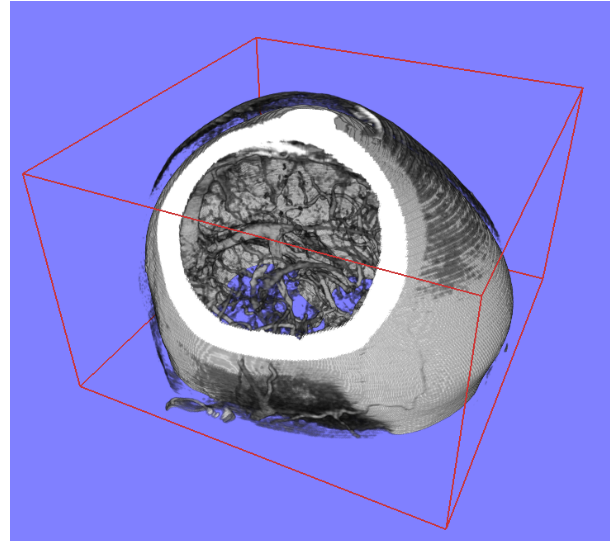
MRI Head
8 MB (256 x 256 x 128)



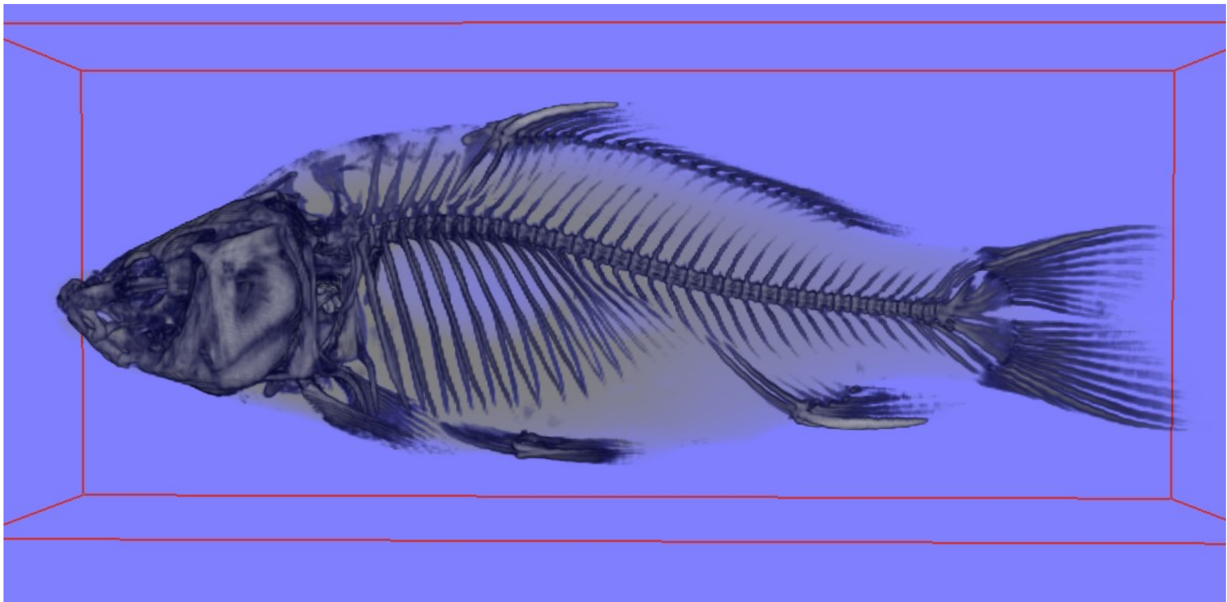
CT Head
16 MB (256 x 256 x 256)



CT Temporal Bone
32 MB (512 x 512 x 128)



CTA Intracranial Vessels
64 MB (512 x 512 x 256)



CT Carp
128 MB (512 x 512 x 512)

Bibliography

[1] M. Abidi and R. Gonzales. *Data Fusion in Robotics and Machine Intelligence*. Academic Press Inc., Boston, 1992. 5.4.2

[2] T. Arbel and F. Ferrie. Viewpoint Selection by Navigation Through Entropy Maps. In *Proc. IEEE Int. Conf on Computer Vision (ICCV)*, 1999. 6.2

[3] C. Bajaj, V. Pascucci, and D. Schikore. The Contour Spectrum. In *Proc. IEEE Visualization*, 1997. 6.3

[4] R. Bajcsy and S.Kovacic. Multiresolution Elastic Matching. *Computer Vision, Graphics and Image Processing*, 46:1–21, 1989. 10.5

[5] B. Bargen and P. Donelly. *Inside DirectX*. Microsoft Press, 1998v. 2.2.2

[6] D. Bartz and M. Meißner. Voxels versus Polygons: A Comparative Approach for Volume Graphics. In *Volume Graphics*, 1999. 1.3.2.1

[7] D. Bechmann. Space Deformation Models Survey. In *Computers & Graphics*, pages 571–586, 1994. 10

[8] J. Becker and M. Rumpf. Visualization of Time-Dependent Velocity Fields by Texture Transport. In *Proc. Eurographics Workshop on Visualization in Scientific Computing '98*, pages 91–101, 1998. 9.3.1

[9] L. Bergmann, B. Rogowitz, and L. Treinish. A Rule-Based Tool for Assisting Colormap Selection. In *Proc. IEEE Visualization*, 1995. 6

[10] J. Blinn and M. Newell. Texture and Reflection in Computer Generated Images. *Communcations of the ACM*, 19(10):362–367, 1976. 7.4

[11] I. Bloch. Information Combination Operators for Data Fusion: A Review with Classification. In *IEEE Transactions on Systems Man Cybernet. Part A: Systems and Humans*, volume 26, 1996. 5.4.2

[12] M. Brady, K. Jung, H. Nguyen, and T. Nguyen. Two-Phase Perspective Ray Casting for Interactive Volume Navigation. In *Proc. IEEE Visualization*, 1997. 3.2.3

[13] M. Bro-Nielsen and C. Gramkow. Fast Fluid Registration of Medical Images. In *Proc. Visualization in Biomedical Comp. (VBC'96)*, 1996. 10.5

[14] L. Brown. A Survey of Image Registration Techniques. *ACM Computing Surveys*, 24(4):325–376, Dez 1992. 14

[15] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *ACM Symp. on Vol. Vis.*, 1994. 3.2

[16] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. In *Proc. SIGGRAPH*, 1993. 9.3, 9.3.1

[17] S. Campagna, L. Kobbelt, and H.-P. Seidel. Efficient Decimation of Complex Triangle Meshes. Technical Report 3, Computer Graphics Group, University of Erlangen–Nuremberg, 1998. 9.3.2, 12.2

[18] G.E. Christensen, M.I. Miller, and M. Vannier. A 3D Deformable Magnetic Resonance Textbook Based on Elasticity. In *Applications of Computer Vision in Medical Image Processing*, AAAI Spring Symposium Series, pages 153–156. Stanford University, March 1994. 10.5

[19] C. Chua and U. Neumann. Hardware-Accelerated Free-Form Deformations. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000. 10

[20] J. Clark and A. Yuille. *Data Fusion for Sensory Information Processing Systems*. Kluwer Academic Publishers, Boston, 1990. 5.4.2

[21] D. Cohen-Or and A. Kaufman. Fundamentals of Surface Voxelization. *CVGIP: Graphics Models and Image Processing*, 56(6):453–461, 1995. 12

[22] A. Collignon, D. Vandermeulen, P. Suetens, and G. Marchal. Automated Multi-Modality Image Registration Based on Information Theory. *Kluwen Acad. Publ's: Computational Imaging and Vision*, 3:263–274, 1995. 10.5

[23] S. Coquillart. Extended Free-Form Deformations. In *Proc. SIGGRAPH*, 1990. 10

[24] F. Dachille and A. Kaufman. GI-Cube: An Architecture for Volumetric Global Illumination and Rendering. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–128, 2000. 1

[25] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-Quality Volume Rendering Using Texture Mapping Hardware. In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, 1998. 7

[26] B.M. Dawant, S.L. Hartmann, and S. Gadamsetty. Brain Atlas Deformation in the Presence of Large Space-occupying Tumors. In *Proc. MICCAI '99*, pages 589–596, 1999. 1.3.1.2

[27] W. de Leeuw and R. van Liere. Comparing LIC and Spot Noise. In *Proc. IEEE Visualization*, 1998. 9.3.1

[28] M. Deimling and G. Laub. *Book of Abstracts*, chapter Constructive Interference in Steady State for Motion Sensitivity Reduction, page 842. Society of Magnetic Resonance in Medicine, 1989. 17

[29] M. Doggett. An Array Based Design for Real-Time Volume Rendering. In *Proc. 10th Eurographics Workshop on Graphics Hardware*, pages 93–101, 1995. 2.3

[30] M. Doggett and G. Hellestrand. A Hardware Architecture for Video Rate Smooth Shading of Volume Data. *Computers and Graphics*, 19(5):695–704, 1995. 2.3

[31] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Proc. SIGGRAPH*, 1988. 1.3.2.1

[32] S.E. Dreyfuss and A.M. Law. *The Art and Theory of Dynamic Programming*. Academic Press, New York, 1962. 6.3.1

[33] K. Eberhardt. Device for Functional Analysis within MR. Technical report, Division of Neuroradiology, Department of Neurosurgery, University of Erlangen–Nuremberg, Germany, 1999. 19

[34] K. Eberhardt, H. Hollenbach, B. Tomandl, and W. Huk. Three-Dimensional MR myelography of the lumbar spine: comparative case study to X-ray myelography. In *European Radiology*, volume 7, 1997. 19

[35] K. Eberhardt, I. Schäfer, M. Deimling, H.-P. Hollenbach, and F. Fellner. Diagnosis of Spinal Dural Arteriovenous Malformations Using a 3D–CISS Sequence. In *Proc. of Soc. of Magn. Res. in Med.*, volume 2, 1997. 17

[36] G. Eckel. *OpenGL Volumizer Programmer's Guide*. SGI Developer Bookshelf, 1998. 4.3, 10.1

[37] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications. In *Proc. IEEE Visualization*, 2000. 1.3

[38] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proc. Graphics Hardware*, 2001. 9

[39] K. Engel, Ove Sommer, and T.Ertl. A Framework for Interactive Hardware-Accelerated Remote 3D-Visualization. In *Data Visualization*, pages 67–177. Springer Computer Science, 2000. 1.3

[40] K. Engel, R. Westermann, and T. Ertl. Isosurface Extraction Techniques for Web-based Volume Visualization. In *Proc. IEEE Visualization*, 1999. 1.3.1.1

[41] K.-H. Höhne et al. *Voxel-Man 3D Navigator Inner Organs*. Springer, electronic media edition, 2000. 5.8, 5.5

[42] A.C. Evans, W. Dai, L. Collins, P. Neelin, and S. Marret. Warping of a Computerized 3-D Atlas to Match Brain Image Volumes for Quantitative Neuroanatomical and Functional Analysis. In *Proc. SPIE Medical Imaging*, pages 236–246, 1991. 1.3.1.2

[43] S. Fang, T. Biddlecome, and M. Tuceryan. Image-Based Transfer Function Design for Data Exploration in Volume Visualization. In *Proc. IEEE Visualization*, 1998. 6.3

[44] S. Fang, S. Rajagopalan, S. Huang, and R. Raghavan. Deformable Volume Rendering by 3D-texture Mapping and Octree Encoding. In *Proc. IEEE Visualization*, 1996. 10.1

[45] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principle And Practice*. Addison-Weseley, 1993. 2.1, 2.1.3, 7, 7.1

[46] L. Forssell. Visualizing Flow Over Curvilinear Grid Surfaces Using Line Integral Convolution. In *Proc. IEEE Visualization*, 1994. 9.3.1

[47] L. Forssell and S. Cohen. Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation and Unsteady Flows. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):133–141, 1995. 9.3.1

[48] J.C. Gee, M. Reivich, and R. Bajcsy. Elastically Deforming 3D Atlas to Match Anatomical Brain Images. *Comp. Assist. Tomogr.*, 17:225–236, 1993. 1.3.1.2

[49] B. Geiger. Reconstruction Software *Nuages*. available online at http://www-sop.inria.fr/prisme/logiciel/nuages.html.en. 12.2

[50] B. Geiger. Three-dimensional Modeling of Human Organs and its Application to Diagnosis and Surgical Planning. Technical Report 2105, INRIA, 1993. 12.2

[51] B. Geiger and J.-D. Boissonnat. Three Dimensional Reconstruction of Complex Shapes Based on the Delaunay Triangulation. Technical Report 1697, INRIA, 1992. 12.2, 12.3

[52] G. Gerig, O. Kübler, R. Kikinis, and F. Jolesz. Nonlinear Anisotropic Filtering of MRI Data. *IEEE Transactions on Medical Imaging*, 11(2):221–232, 1992. 14.3

[53] B. Girod, G. Greiner, and H.Niemann. *Principles of 3D Image Analysis and Synthesis*. Kluwer Academic Publishers, 2000. 1.3

[54] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. 6.2

[55] H. Gouraud. Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers*, 20(6), June 1971. 4.1.2

[56] N. Greene. Environment Mapping and Other Applications of World Projection. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986. 7.4

[57] T. Günther, C. Poliwoda, C. Reinhard, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. VIRIM - A Massively Parallel Processor for Real-Time Volume Visualization in Medicine. In *Proc. 9th Eurographics Workshop on Graphics Hardware*, pages 103–108, 1994. 2.3

[58] M. Hamilton, J. Anson, and R. Spetzler. *The Practice of Neurosurgery*, chapter Spinal Vascular Malformations, pages 2272–2292. Williams & Wilkins, 1996. 14.1

[59] P. Hastreiter. *Registrierung und Visualisierung medizinischer Bilddaten unterschiedlicher Modalitäten.* PhD Thesis, University of Erlangen-Nuremberg, 1999. 1.3.1.2, 2

[60] P. Hastreiter and T. Ertl. Integrated Registration and Visualization of Medical Image Data. In *Proc. CGI*, pages 78–85, 1998. 12.2

[61] P. Hastreiter, C. Rezk-Salama, K. Eberhardt, B. Tomandl, and T. Ertl. Functional Analysis of the Vertebral Column based on MR and Direct Volume Rendering. In *Proc. Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2000. 15.1

[62] P. Hastreiter, C. Rezk-Salama, G. Greiner, and T. Ertl. Efficient representation of cortical convolutions for the analysis of brain surface topology. In *Bildverarbeitung in der Medizin: Algorithmen, Systeme, Anwendungen.* Springer, 1998.

[63] P. Hastreiter, C. Rezk-Salama, G. Greiner, and T. Ertl. Interactive Direct Volume Rendering of the Inner Ear for the Planning of Neurosurgery. In *Bildverarbeitung in der Medizin: Algorithmen, Systeme, Anwendungen.* Springer, 1999. 12

[64] P. Hastreiter, C. Rezk-Salama, C. Nimsky, C. Lürig, and G. Greiner. Registration Techniques for the Analysis of the Brain Shift in Neurosurgery. *Computers & Graphics*, 24(3), 2000. 10.5

[65] P. Hastreiter, C. Rezk-Salama, B. Tomandl, K. Eberhardt, and T. Ertl. Fast Analysis of Intracranial Aneurysms based on Interactive Direct Volume Rendering and CT-Angiography. In *Proc. Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 1998. 15, 13.2, 13.4

[66] P. Hastreiter, C. Rezk-Salama, B. Tomandl, K. Eberhardt, and T. Ertl. Comparing the Quality of Interactive Volume Rendering Methods in Neuroradiology. In *Proc. Rapid Prototyping in Medicine and Computer-Assisted Surgery (CAS)*, 1999.

[67] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of Transfer Functions with Stochastic Search Techniques. In *Proc. IEEE Visualization*, 1996. 6.2

[68] H. Hege, T. Höllerer, and D. Stalling. Volume Rendering, Mathematical Foundations and Algorithmic Aspects. Technical Report TR93-7, Konrad-Zuse-Zentrum für Informationstech., Berlin, 1993. 1.1

[69] J. Hladůvka, A. König, and E. Gröller. Curvature Based Transfer Functions for Direct Volume Rendering. In *Proc. Spring Conference on Computer Graphics*, 2000. 5.4

[70] V. Hlavac, A. Leonardis, and T. Werner. Automatic Selection of Reference Views for Image-Based Scene Reprentations. In *Proc. European Conference on Computer Vision (ECCV)*, 1996. 6.2

[71] J. Holland. *Adaption in Natural and Artifical Systems.* University of Michigan Press, 1995. 6.2

[72] J. Hultquist. Interactive numerical flow visualization using stream surfaces. In *Computing Systems in Engineering*, pages 349–353, 1990. 9.3

[73] M. Hurn, K. Mardia, T. Hainsworth, J. Kirkbridge, and E. Berry. Bayesian Fused Classification of Medical Images. Technical Report STAT/95/20/C, Department of Statistics, University of Leeds, 1995. 5.4.2

[74] V. Interrante and C. Grosch. Visualizing 3D Flow. *IEEE Computer Graphics and Applications*, 18(4):47–53, 1998. 9.3.1, 9.3.2

[75] S. Iserhardt-Bauer, P. Hastreiter, and T. Ertl. Medical Web Service for the Automatic 3D Documentation for Neuroradiological Diagnosis. In *Proc. IEEE Visualization*, 2001. 6.4

[76] S. Iserhardt-Bauer, C. Rezk-Salama, T. Ertl, P. Hastreiter, B. Tomandl, and K. Eberhardt. Automated 3D Video Documentation for the Analysis of Medical Data. In *Bildverarbeitung in der Medizin: Algorithmen, Systeme, Anwendungen.* Springer, 2001. 6.4

[77] J Kajiya and B. Von Herzen. Ray Tracing Volume Densities. In *Proc. SIGGRAPH*, 1984. 1.3.2.1

[78] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models. In *Proc 1st Int. Conference on Computer Vision*, pages 259–268, 1987. 1.3.1.2

[79] A. Kaufman. Voxels as a Computational Representation of Geometry. In *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, 1994. 1.2

[80] J. Kawai, J. Painter, and M. Cohen. Rapidoptimization – Goal-Based Rendering. In *Proc. SIGGRAPH*, 1993. 6.2

[81] G. Kindlmann and J. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *IEEE Symposium on Volume Visualization*, 1998. 6.3

[82] S. Kirkpatrick, C. Gerlatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220, 1993. 6.2

[83] J. Kniss, G. Kindlmann, and C. Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proc. IEEE Visualization*, 2001. 5.4, 6.1

[84] L. Kobbelt. Discrete Fairing. In *Proc. Seventh IMA Conference on the Mathematics of Surfaces*, pages 101–131, 1997. 12.2

[85] L. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature-Sensitive Surface Extraction From Volume Data. In *Proc. SIGGRAPH*, 2001. 1.3.1.1, 12

[86] S. Kochhar. A Prototype System for Design Automation via the Browsing Paradigm. In *Proc. Graphics Interface*, 1990. 6.2

[87] A. König and E. Gröller. Mastering Transfer Function Specification by Using VolumePro Technology. In *Proc. Spring Conference on Computer Graphics*, 2001. 6.2

[88] P. Kovach. *Inside Direct3D*. Microsoft Press, 2000. 2.2.2

[89] Yair Kurzion and Roni Yagel. Space Deformation using Ray Deflectors. In *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pages 21–30, New York, 1995. Springer-Verlag. 10.1

[90] Yair Kurzion and Roni Yagel. Interactive Space Deformation with Hardware-Assisted Rendering. *IEEE Computer Graphics & Applications*, 17(5), 1997. 10.1, 10.4

[91] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform . *Comp. Graphics*, 28(4), 1994. 1.3.2.2

[92] E. LaMar, B. Hamann, and K. Joy. Multiresolution Techniques for Interactive Texture-based Volume Visualization. In *Proc. IEEE Visualization*, 1999. 3.2.3

[93] M. Levoy. Display of Surfaces form Volume Data. *IEEE Comp. Graph. & Appl.*, 8(5):29–37, 1988. 1.3.2.1, 5.4

[94] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Comp. Graph.*, 9(3):245–261, 1990. 1.3.2.1

[95] H. Li, B. Manjunath, and S. Mitra. Multi-sensor Image Fusion Using the Wavelet Transform. *GMIP: Graphical Model Image Process*, 57(3):235–245, 1995. 5.4.2

[96] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Comp. Graphics*, 21(4):163–169, 1987. 1.3.1.1

[97] E. Lum, K. Ma, and J. Clyne. Texture Hardware Assited Rendering of Time-Varying Volume Data. In *Proc. IEEE Visualization*, 2001. 9

[98] R. Luo, M. Lin, and R. Scherp. Multi-Sensor Integration and Fusion in Intelligent Systems. In *IEEE Transactions on Systems Man Cybernet*, volume 19, 1989. 5.4.2

[99] R. MacCracken and K. Roy. Free-Form Deformations with Lattices of Arbitrary Topology. In *Proc. SIGGRAPH*, 1996. 10

[100] J. Maintz and M. Viergever. A Survey of Medical Image Registration. *Medical Image Analysis*, 2(1), 1998. 14

[101] T. Malzbender. Fourier Volume Rendering. *ACM Transactions on Graphics*, 12(3):233–250, 1993. 1.3.2

[102] W. Mark and K. Proudfoot. Compiling to a VLIW Fragment Pipeline. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001. 4.1.1

[103] J. Marks, B. Andalman, P. Beardsley, and H. Pfister. Design Galleries: A General Approach for Setting Parameters for Computer Graphics and Animation. In *Proc. SIGGRAPH*, 1997. 6.2

[104] G. Matsopoulos, S. Marshall, and J. Brunt. Multi-Resolution Morphological Fusion of MR and CT Images of the Human Brain. In *Proc. IEEE Visual Image Signal Processing*, volume 141, 1994. 5.4.2

[105] C. Maurer and J. Fitzpatrick. A Review of Medical Image Registration. In R.J. Maciunas, editor, *Interactive Image-Guided Surgery*, pages 17–44, Park Ridge, IL, American Association of Neurological Surgeons, 1993. 14

[106] N. Max, B. Becker, and R. Crawfis. Flow Volumes for Interactive Vector Field Visualization. In *Proc. IEEE Visualization*, 1993. 9.3

[107] 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. 7

[108] M. Meißner, U. Kanus, and W. Straßer. VIZARD II: A PCI-Card for Real-Time Volume Rendering. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 61–67, 1998. 2.3

[109] Microsoft DirectX 8.0 Specification. available online at http://www.microsoft.com/directX/. 5

[110] D. Moore and J. Warren. Mesh Displacement: An Improved Contouring Method for Trivariate Data. Technical Report TR-91-166, Rice University. Dept. of Computer Science, 1991. 1.3.1.1

[111] E. Mortensen, B. Morse, and W. Barrett. Adaptive Boundary Detection Using "Live-Wire, Two Dimensional Dynamic Programming. In *IEEE Computers in Cardiology*, 1992. 1.3.1.2

[112] E. N. Mortensen and W. A. Barrett. Intelligent Scissors for Image Composition. In *Proc. SIGGRAPH*, 1995. 1.3.1.2

[113] E. N. Mortensen and W. A. Barrett. Interactive Segmentation With Intelligent Scissors. *Graphical Models and Image Processing*, 60(5):349–384, 1998. 1.3.1.2

[114] K. Mueller, T. Möller, and R. Crawfis. Splatting Without the Blur. In *Proc. IEEE Visualization*, 1999. 1.3.2

[115] K. Mueller and R. Yagel. Fast Perspective Volume Rendering with Splatting by Using a Ray-Driven Approach. In *Proc. IEEE Visualization*, 1996. 1.3.2

[116] D. Mukherjee, P. Dutta, and D. Dutta Majumdar. Entropy Theoretic Fusion of Multimodal Medical Images. Technical Report ECSU/2/98, Electronics and Communication Science Unit, Indian Statistical Institute, 1998. 5.4.2

[117] S. Mukhopadhyuy and B. Chanda. Fusion of 2D Grayscale Images using Multiscale Morphology. *Pattern Recognition*, 34(10):1939–1949, 2001. 5.4.2

[118] NetMedicine. Computed tomography library. available online at http://www.netmedicine.com/xray/ctscan/ct.htm. 11.1.1

[119] K. Novins. *Towards Accurate and Efficient Volume Rendering*. PhD thesis, Cornell University, 1994. 1.2

[120] NVidia Corporation. NVidia Developer Relations Site. available online at http://www.nvidia.com/developer. 4.1.2.1, 2

[121] NVidia Corporation. OpenGL Extension Specification. available online at http://www.nvidia.com/developer. 5.2.1.2

[122] "National Library of Medicine". The Visible Human Project. available online at `http://www.nlm.nih.gov/research/visible/`, 1996. 5.5

[123] The Official OpenGL Website. http://www.opengl.org. 2.2.1

[124] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An architecture for low-cost real-time volume rendering. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 131–138, 1997. 1

[125] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L.Seiler. The VolumePro Real-time Ray-Casting System. In *Proc. SIGGRAPH*, 1999. 1

[126] H. Pfister and A. Kaufman. Cube-4 – A Scalable Architecture for Real-Time Volume Rendering. In *IEEE Symposium on Volume Visualization*, 1996. 2.3

[127] B.T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975. 2.1.1, 3, 7.1

[128] Microsoft Press. *Computer Lexicon mit Fachwörterbuch*. Microsoft Press Deutschland, 2001. 8.1.1

[129] K. Proudfoot, W. Mark, P. Hanrahan, and S. Tzvetkov. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proc. SIGGRAPH*, 2001. 4.1.1

[130] Quicktime VR Authoring. available online at `http://www.apple.com/quicktime/qtvr/`. 5.5

[131] H. Ray, H. Pfister, D. Silver, and T. Cook. Ray Casting Architectures for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), March 1999. 2.3

[132] H. Ray and D. Silver. The RACE II Engine fore Real-Time Volume Rendering. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 129–136, 2000. 1

[133] The Real Time Visualization Group. Volume Library Interface User Guide - A Guide to Programming with VolumePro. http://www.merl.com. 2.2.2

[134] 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. 7, 9.1, 11

[135] C. Rezk-Salama, P. Hastreiter, G. Greiner, and T. Ertl. Non-linear registration of pre- and intraoperative volume data based on piecewise linear transformations. In *Proc. Vision, Modelling, and Visualization (VMV)*, 1999. 10.5

[136] C. Rezk-Salama, P. Hastreiter, J. Scherer, and G.Greiner. Automatic Adjustment of Transfer Functions for 3D Volume Visualization. In *Proc. Vision, Modeling and Visualization (VMV)*, 2000. 6.3.1

[137] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. Interactive exploration of volume line integral convolution based on 3d–texture mapping. In *Proc. IEEE Visualization*, 1999. 9.3.1

[138] C. Rezk-Salama and M. Scheuering. Multitexturbasierte Volumenvisualisierung in der Medizin. In *Bildverarbeitung in der Medizin: Algorithmen, Systeme, Anwendungen*. Springer, 2001. 4.4

[139] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast Volumetric Deformation on General Purpose Hardware. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001. 10.3.4

[140] P. Rheingans. Task-Based Color Scale Design. In *Proc. SPIE Applied Image and Pattern Recognition*, 1999. 6

[141] C. Risquet. Visualizing 2D Flows: Integrate and Draw. In *Proc. Eurographics Workshop on Visualization in Scientific Computing*, pages 132–142, 1998. 9.3.1

[142] D. Roberts and A. Marshall. Viewpoint Selection for Complete Surface Coverage of Three-Dimensional Objects. In *Proc. British Machine Vision Conference*, 1998. 6.2

[143] D. Rogerson. *Inside COM – Microsoft's Component Object Model*. Microsoft Press, 1997. 2.2.2

[144] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *Proc. SIGGRAPH*, 1988. 1.1, 1.3.2.1

[145] Y. Sato, C.-F. Westin, and A. Bhalerao. Tissue Classification Based On 3D Local Intensity Structures for Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 6, April 2000. 5.4, 6.3

[146] M. Scheuering, C. Eckstein, C. Rezk-Salama, K. Hormann, and G.Greiner. Interactive Repositioning of Bone Fracture Segments . In *Proc. Vision, Modeling and Visualization (VMV)*, 2001.

[147] M. Scheuering, C. Rezk-Salama, H. Barfuß, K. Barth, A. Schneider, G. Greiner, G. Wessels, and H. Feussner. Intra-operative Augmented Reality (AR) With Magnetic Navigation And Multi-texture Based Volume Rendering For Minimally Invasive Surgery. In *Rechner u. Sensorgestützte Chirurgie Heidelberg*, 2001. 4.4

[148] H. Schumann and W. Müller. *Visualisierung. Grundlagen und allgemeine Methoden*. Springer, 2000. 1.3

[149] T. Sederberg and S. Parry. Free-Form Deformation of Solid Geometric Models. In *Proc. SIGGRAPH*, 1986. 10

[150] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. http://www.opengl.org. 2.1.3, 5.2.2.2

[151] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press London, 1982. 1.3.1.2

[152] H.-W. Shen, C. Johnson, and K. Ma. Visualizing Vector Fields Using Line Integral Convolution and Dye Advection. In *ACM SIGGRAPH Symposium on Volume Visualization*, pages 63–70, 1994. 9.3.1

[153] H.-W. Shen and D. Kao. UFLIC Line Integral Convolution Algorithm for Visualizing Unsteady Flows. In *Proc. IEEE Visualization*, 1997. 9.3.1

[154] Silicon Graphics Inc. http://www.sgi.com. 5.2.2.1

[155] K. Sims. Artificial Evolution in Computer Graphics. In *Proc. SIGGRAPH*, 1991. 6.2

[156] K. Sims. Evolving Virtual Creatures. In *Proc. SIGGRAPH*, 1994. 6.2

[157] G. Sivewright and P. Elliot. Interactive Region and Volume Growing in MR and CT. *Medical Informatics*, 19:71–80, 1994. 1.3.1.2

[158] D. Stalling and H.-C. Hege. Fast and Resolution Independent Line Integral Convolution. In *Proc. SIGGRAPH*, 1995. 9.3.1

[159] V. Takeushi and N. Ohnishi. Active Vision Systems Based on Information Theory. *Systems and Computers in Japan*, 29(11), 1998. 6.2

[160] J.-P. Thirion. Non-Rigid Matching using Demons. In *Computer Vision and Pattern Recognition*, pages 245–251. IEEE Computer Society Press, Los Alamitos, CA, 1996. 10.5

[161] N. Thompson. *3D Graphics Programming for Windows 95*. Microsoft Press, 1996. 2.2.2

[162] U. Tiede, T. Schiemann, and K.-H. Höhne. High-Quality Rendering of Attributed Volume Data. In *IEEE Visualization, Late Breaking Hot Topics*, 1998. 5.5

[163] S. Todd and W. Latham. *Evolutionary Art and Computer Graphics*. Academic Press, 1992. 6.2

[164] B. Tomandl, P. Hastreiter, K. Eberhardt, H. Greess, U. Nissen, C. Rezk-Salama, and W. Huk. Virtual labyrinthoscopy: Visualization of the inner ear with interactive direct volume rendering. *Radiographics*, 20(2), March 2000. 12.4

[165] C. Upson and M. Keeler. V-BUFFER: Visible Volume Rendering. In *Proc. SIGGRAPH*, 1988. 1.3.2

[166] M. van de Panne and E. Fiume. Sensor-Actuator Networks. In *Proc. SIGGRAPH*, 1993. 6.2

[167] J. van Wijk. Spot Noise – Texture Synthesis for Data Visualization. In *Proc. SIGGRAPH*, 1991. 9.3.1

[168] P. Viola. *Alignment by Maximization of Mutual Information*. PhD thesis, M.I.T., 1995. 10.5

[169] S. Wang and A. Kaufman. Antialiased Voxelization. In *Proc. IEEE Visualization*, 1993. 12

[170] C. Ware. Color Sequences for Univariate Maps: Theory, Experiments, and Principles. *IEEE Computer Graphics and Applications*, 8, 1988 1998. 6

[171] A. Watt and F. Policarpo. *3D Games: Real-time Rendering and Software Technology*, volume one. Addison-Wesley, 1st edition, 2001. 4.1.1

[172] R. Wegenkittl and E. Gröller. Fast Oriented Line Integral Convolution for Vector Field Visualization via the Internet. In *Proc. IEEE Visualization*, 1997. 9.3.1

[173] R. Westermann. *A Multiresolution Framework for Volume Rendering*. PhD Thesis, University of Dortmund, Germany, 1997. 1.3.2

[174] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. SIGGRAPH*, 1998. 7.2, 9.2

[175] R. Westermann and C. Rezk-Salama. Real-Time Volume Deformation. In *Computer Graphics Forum (Eurographics)*, 2001. 10.2

[176] R. Westermann, O. Sommer, and T. Ertl. Decoupling Polygon Rendering from Geometry using Rasterization Hardware. In *Proc. 10th Eurographics Workshop on Rendering*, 1999. 10.2.2

[177] L. Westover. Interactive Volume Rendering. In *Chapel Hill Volume Visualization Workshop*, 1989. 1.3.2

[178] L. Westover. Footprint Evaluation for Volume Rendering. In *Proc. SIGGRAPH*, 1990. 1.3.2

[179] L. Westover. *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*. PhD thesis, UNC Chapel Hill, 1991. 1.3.2

[180] J. Wilhelms and Van Gelder A. A Coherent Projection Approach for Direct Volume Rendering. In *Proc. SIGGRAPH*, 1991. 1.3.2

[181] O. Wilson, A. Van Gelder, and J. Wilhelms. Direct Volume Rendering via 3D-textures. Technical Report UCSC-CRL-94-19, Univ. of California, Santa Cruz, 1994. 3.2

[182] A. Witkin and M. Kaas. Spacetime Constraints. In *Proc. SIGGRAPH*, 1988. 6.2

[183] S.W. Zucker. Region Growing: Childhood and Adolescence. *Computer Graphics and Image Processing*, 5:382–399, 1976. 1.3.1.2

Index

2D-multi-texture
  performance, 95
  volume rendering, 43
2D-texture
  performance, 93
  volume rendering, 25
3D-texture, 32
  brick, 35
  performance, 94
  volume rendering, 32

absorption, 2
  coefficient, 3
accelerated graphics port, 89
active edge data structure, 116
AGP, *see* accelerated graphics port
aliasing, 4, 58, 147
alpha blending, 103
  maximum intensity projection, 30
  per-fragment operations, 18
  radiative transfer, 29
alpha buffer, 17
alpha test
  for isosurfaces, 78
  per-fragment operations, 17
ambient light, *see* illumination
aneurysm, 144
angiography, 30
  CTA, 144
anisotropic diffusion, 153
API, 18
  Direct3D, 19
  DirectX, 19
  OpenGL, 19
  requirements, 18
appearance, 115
application programming interface, *see* API
artery
  cerebral, 144
artifacts
  sampling, 4, 31
  slicing, 31
  switching effects, 32
  transfer function, 58, 147
ATI
  Radeon, *see* hardware
attributed volume, 64
availability, 7

boundary emphasis function, 71
box filter, 5
brick, 35
bricking, 35
  performance, 95
bus
  crossbar, 92
  SGI EBus, 92
  unified memory, 91, 92

central differences, 59
cerebrospinal fluid, 150, 157
classification, 10, 48
clipping, 116
  arbitrary geometry, 105
  rendering pipeline, 16
  stencil buffer, 105
closing, *see* morphologic operations
cochlea, *see* inner ear
color table
  paletted texture, 51
Compact Cubes, *see* isosurface

compatibility
 of an API, 19
compositing, 28–30, 103
computed tomography, *see* volume data
computed tomography angiography, *see* angiography
contour lines, *see* surface reconstructions
contrast dye
 CTA, 144
 DSA, 150
conversion
 surface to voxel, 109
 voxel to surface, *see* surface reconstruction
CSF, *see* cerebrospinal fluid
CT, *see* volume data
CTA, *see* angiography
cube map, *see* reflection map
Cube-4, *see* hardware
curvature
 classification, 60

dAVF, *see* fistula
dAVM, *see* malformation
deformation
 free-form, 115
 of appearance, 119
 of shape, 118
 piecewise linear patches, 122
 ray deflectors, 114
 real-time tessellation, 114
 tetrahedra, 115
 volumetric, 114
density-emitter model, 3, 11
 alpha blending, 28
 approximation, 31
dependent texture, *see* texture
depth buffer, 18
depth test
 per-fragment operations, 18
derivatives
 partial, 70
diffuse reflection, *see* illumination
digital subtraction angiography, 150
Direct3D, *see* API
DirectX, *see* API
discogenic disease, 157
display traversal, *see* graphics pipeline
DSA, *see* digital subtraction angiography

edge flag, 35
efficiency
 of an API, 18
EMCube, *see* hardware
emission, 2
 coefficient, 3
emission-absorption model, 3
environment map, *see* texture
extensibility
 of an API, 19

fairing, *see* surface reconstruction
fill rate, *see* pixel fill rate
filter
 box, 5
 sinc, 4
 tent, 5
filtering
 visualization pipeline, 6
fistula
 dural arteriovenous, 150
flow visualization, 109
 LIC, *see* line integral convolution
 stream surfaces, 109
foundation layer
 DirectX, 20
fourier-space technique, *see* volume rendering
fragment
 rasterization, 16, 38
frame buffer
 alpha buffer, 17
 depth buffer, 18
 stencil buffer, 18
frequency domain, *see* volume rendering
fusion, 63

GeForce, *see* hardware
genetic algorithm, 69
 genotype, 69
 phenotype, 69
genotype, *see* genetic algorithm
geometry limit, 88
geometry processing, *see* graphics pipeline
 clipping, 16
 lighting, 16
 primitive assembly, 16
 transformation, 16
GICube, *see* hardware
GPU, *see* hardware
gradient estimation
 central differences, 59
gradient vector, 59, 70
graphics pipeline, 14
 geometry processing, 15
 per-vertex operations, 15
 transform & light unit, 15

hardware, 89
 array-based ray casting, 21
 ATI Radeon, 37, 90
 ATI Radeon 8500, 90
 Cube-4, 22
 EMCube, 22
 GICube, 22
 GPU, 90
 NVidia GeForce 2 Go, 90
 NVidia GeForce 256, 90
 NVidia GeForce 2 GTS, 90
 NVidia GeForce 2 Ultra, 90
 NVidia GeForce 3, 37, 90
 RACE2, 22
 SGI O2, 54, 91
 SGI Octane, 54
 SGI Octane 2, 92
 SGI Onyx, 54, 92
 special purpose, 21–23
 VIRIM, 21
 VIZARD-II, 22
 volume rendering, 21–23
 VolumePro, 22
Hounsfield scale, 136

illumination
 ambient, 76
 Blinn-Phong model, 76
 diffuse reflection, 76
 directional derivative, 120
 forward differences, 120
 Lambertian reflection, 76
 local, 75
 per-pixel, 79
 Phong model, 76
 specular reflection, 76
image order approach, *see* volume rendering
image quality, 7
immediate mode
 Direct3D, 20
inner ear, 138
interactivity, 6
interpolation
 arbitrary slices, 44
 bilinear, 25
 multi-planar reformatting, 44
 multi-textures, 44
 trilinear, 32, 44
intersection
 plane–mesh, 116
 viewport-aligned slices, 35
intervertebral disk, 157
isosurface, 8
 Compact Cubes, 8
 curvature, 60
 Marching Cubes, 8
 non-polygonal, 77, 146
 semi-transparent, 146
 thick, 146

labyrinth, *see* middle ear
Lambertian reflection, *see* illumination
LIC, *see* line integral convolution
lighting

Blinn-Phong model, 76
definition, 75
Phong model, 76
rendering pipeline, 16
line integral convolution, 110
local intensity structures, 60

magnet resonance tomography, *see* volume data
malformation
dural arteriovenous, 150
mapping
visualization pipeline, 6
Marching Cubes, *see* isosurface
matrix
Hessian, 60
modeling, 16
modelview, 16
viewing, 16
maximum intensity projection, 30, 67, 145
media layer
DirectX, 20
memory
DDR SDRAM, 90
SDRAM, 90
unified, 91
memory bandwidth, 88
mesh
decimation, *see* surface reconstruction
reduction, *see* surface reconstruction
refinement, *see* surface reconstruction
middle ear, 138
MIP, *see* maximum intensity projection
modeling
paradigm, 115
transformation, 16
modelview matrix, 16
morphologic operations
closing, 153
MPR, *see* multi-planar reformatting
MR, *see* volume data
MRI, *see* volume data
MR-CISS, 151, 158
MR-FISP, 151, 158
MR-MEDIC, 158
multi-planar reformatting, 44
multi-scale analysis, 60
multi-texture, *see* texture
OpenGL standard, 39
rendering speedup, 101
trilinear interpolation, 39, 44
multivariate data, 63

nerves
cerebral, 144
noise-reduction, 153
non-polygonal isosurfac, *see* isosurface
non-polygonal isosurface
example, 146
normal map, *see* texture
Nuages, *see* surface reconstruction
NVidia
GeForce, *see* hardware
register combiners, *see* rasterization
Nyquist rate, 4

OpenGL, *see* API
GL_BLEND, 28, 29
GL_TEXTURE_2D, 55
glBindTexture, 55
glBlendFunc, 28, 29
glColorTableEXT, 52
glGet, 27
glPixelMap, 51
glPixelTransfer, 51
glTexEnvi, 55
OpenGL extension, 19
EXT_blend_minmax, 30
EXT_paletted_texture, 51
EXT_shared_paletted_texture, 51
EXT_texture_env_dot3, 79
glActiveTextureARB, 55
glBlendEquationEXT, 29
glColorTableSGI, 53
NV_register_combiner, 42
NV_texture_shader, 54

SGI_color_matrix, 78
SGI_pixel_texture, 54
SGI_texture_color_table, 53
ossicles, *see* middle ear

partial derivatives, 59
partial volume effect, *see* volume data
PC, *see* personal computer
PCI, *see* peripheral component interconnect
per-fragment operations
rendering pipeline, 17
per-vertex operations, *see* graphics pipeline
performance
2D-multi-textures, 95
2D-textures, 93
3D-textures, 94
frame rate, 6
illumination, 99
supersampling, 96
transfer functions, 97
peripheral component interconnect, 89
personal computer, 89
Intel x86, 89
phenotype, *see* genetic algorithm
pixel fill rate, 88
pixel shader, *see* rasterization
illumination, 79
transfer function, 54
pixel transfer, 51
platform independence
of an API, 19
position function, 71
post-classification, *see* transfer function
pre-classification, *see* transfer function
primitive
assembly, 16
geometric, 16
primitives
proxy geometry, 25
projection
rendering pipeline, 16
proxy geometry, 25

RACE2, *see* hardware
Radeon, *see* hardware
radiance, 2
alpha blending, 28
analytic solution, 3
numerical solution, 3
radiation field, *see* radiance
radiative transfer, 2
radiology, 135
rasterization
pixel shader, 40
polygon, 17
register combiner, 42
register combiners, 44, 45
rendering pipeline, 16, 38
ray casting, *see* volume rendering
array-based, *see* hardware
reconstruction, 4
reconstruction filter, 4
reflection map
cube map, 84
longitude-latitude, 83
register combiners, *see* rasterization
registration, 130–133
rendering
primitive, 16
visualization pipeline, 6
rendering pipeline, *see* graphics pipeline
retained mode
Direct3D, 20

sampling
theory, 4
transfer function, 56
viewport-aligned slices, 35
scattering, 2
segmental arteries, 150, 157
segmentation, 8
example, 139
explicit, 139
semicircular canals, *see* inner ear

SGI
  O2, *see* hardware
  Octane, *see* hardware
  Onyx, *see* hardware
shading
  definition, 75
shape, 115
shear-warp-algorithm, 12, 26
Silicon Graphics, *see* SGI
sinc, *see* filter
slicing
  stack selection, 27
smoothing, *see* surface reconstruction
specular reflection, *see* illumination
spinal column, 150, 157
spinal cord, 150, 157
spine, *see* vertebral column
splatting, *see* volume rendering
spondylolisthesis, 157
stencil buffer, 18, 105, 146
stencil test, 105, 146
  per-fragment operations, 18
stenosis
  spinal, 157
subdivision
  discrete fairing, 140
  hexahedra, 124
surface reconstruction, 8
  contour lines, 9, 139
  example, 138
  fairing, 140
  mesh decimation, 140
  mesh reduction, 140
  mesh refinement, 140
  Nuages, 139
  smoothing, 140
  subdivision, 140
  triangulation, 139

tagged volume, 64
temporal bone, 138
tent filter, 5
tessellation, 116
texel, 17
texture, 17, 39
  application, 17, 39
  color table, 53
  cube map, 84
  dependent, 54
  environment map, 83
  generation, 17
  mapping, 17, 39
  multi-texture, 39
  normal map, 83
  paletted, *see* color table
  pixel transfer, 51
time surface, 112
time volume, 112
transfer function, 10, 48
  attributed volume, 64
  automatic adaptation, 72
  contour spectrum, 70
  curvature-based, 60
  data-driven, 70
  dependent texture, 54
  design, 67–74
  design galleries, 69
  dynamic programming, 72
  editor, 67
  genetic algorithm, 69
  gradient weighted, 59
  hill climbing, 69
  image processing, 70
  image-driven, 69
  implementation, 50–56, 61–63, 65–66
  interactive evolution, 69
  inverse design, 69
  local, 64
  local intensity structures, 60
  multi-dimensional, 58
  pixel shader, 54
  pixel transfer, 51
  post-classification, 50, 53
  post-interpolative, 50, 53
  pre-classification, 49, 50
  pre-interpolative, 49, 50

principles, 48
sampling theory, 56
semi-automatic, 70
simulated annealing, 69
tagged volume, 64
texture color table, 53
texture palette, 51
thumbnail selection, 69
transform & light unit, *see* graphics pipeline
transformation
affine, 115
modeling, 16
perspective, 16
piecewise linear, 115
viewing, 16
triangulation, *see* surface reconstruction

UMA, *see* unified memory
unified memory, 91

vertebra, 157
vertebral column, 150, 157
vessel malformation, 144
viewing transformation, 16
VIRIM, *see* hardware
visualization pipeline, 6
for indirect volume rendering, 7
for ray casting, 10
for surface reconstruction, 9
VIZARD-II, *see* hardware
volume data
computed tomography, 136
grid structure, 5
magnet resonance tomography, 136
partial volume effect, 136
sampling theory, 4–5
volume growing, 153
volume rendering
2D-multi-textures, 43
2D-textures, 25
3D-textures, 32
compositing, 28–30
computational cost, 25
density-emitter model, 3
direct methods, 10
emission-absorption model, 3
fourier-domain, 10
frequency-domain, 10
image order, 10
indirect methods, 7
isosurface, 8
physical, 2
ray casting, 11
shear-warp, 12
splatting, 10
surface reconstruction, 8
texture-based, 25
VolumePro, *see* hardware
Volumizer, 115
voxel model, 5
voxel tag, *see* tagged volume
voxelization, 109