Bridging the Gap between Volume Visualization and Medical Applications

Von der Fakultät Informatik, Elektrotechnik und Informations-technik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Friedemann Andreas Rößler

aus Stuttgart

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. B. Preim
Tag der mündlichen Prüfung:	18.12.2009

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2009

Für Paul

Acknowledgments

I am most grateful to my advisor Thomas Ertl, who supported and guided my work. Thank you for giving me the chance to gain insight into the exciting research fields of medical visualization and computer graphics. Special thanks go to my co-advisor Bernhard Preim from the University of Magdeburg, who tracked the progress of my work on many conferences and who is a great supporter of medical visualization research in germany and allover the world. I want to thank Sabine Iserhardt-Bauer and Peter Hastreiter from the University Erlangen-Nuremberg for the fruitful cooperation on the field of automated medical visualization. Especially Sabine laid important foundations with her preceding work on the medical visualization service. Many thanks to Markus Knauff from the University of Gießen and Thomas Fangmeier from the University of Freiburg, who introduced me into the very interesting world of cognitive neuro sciences and functional brain imaging and who gave the input for my start-up project at VIS.

I want to thank Tobias Schafhitzel, who had always time for another cup of coffee and with whom I had many scientific and non-scientific discussions inside and outside university. Thank you for your patience in repeatedly explaining me the mystery of curvature and LIC computation. Furthermore, I thank Ralf Botchen, who was also a perfect coffee mate, for being my fellow sufferer in neverending paper sessions on multi-volume rendering. Many thanks go to Eduardo Tejada, with whom I corporated for my first project on fMRI visualization. I would like to thank my long-term room mate Magnus Strengert for having always the right answer to my countless questions on computer graphics, visualization and graphics hardware. Many thanks go to my short-term room mate Shymaa Elleithy for giving me interesting insights into Egyptian and Muslim culture. Moreover, I want to thank all my colleagues for the nice time we had together at VIS. Special thanks go thereby to Harald Bosch, Martin Falk, Benjamin Höferlin and Markus Üffinger for proof reading my thesis.

I express my gratitude to all the students who supported me in my scientific work. First, I want to thank my auxiliary scientists Michael Krone and Steffen Frey, who did great jobs and who later became colleagues of mine. Special thanks go to Matey Nenov and Torsten Wolff, whose study and diploma theses provided important contributions to my work. Furthermore, I would like to thank Senad Hrnjadovic, Meike Rudolph, Simone Tonte and Qing Yang.

Finally, I want to thank my parents for their neverending support during all phases of my education and for guiding me through all the ups and downs I lived through during this time. Last but not least, I thank my son Paul, who gave me the power to never give up.

Friedemann Rößler

CONTENTS

List of Abbreviations and Acronyms

1D	one-dimensional	HTTP	hypertext transfer protocol
2D	two-dimensional	HLSL	high level shading language
3D	three-dimensional	HU	hounsfield unit
API	application programming interface	ID	identifier
ASM	active shape model	i.e.	id est (that is)
BOLD	blood oxygen level dependent	JPEG	joint photographic experts group
Cg	C for graphics	JSP	Java server page
CPU	central processing unit	LIC	line integral convolution
CT	computed tomography	MB	megabyte
CTA	CT angiography	MIP	maximum intensity projection
CUDA	compute device framework architecture	MPEG	moving picture experts group
DTI	diffusion tensor imaging	MPI	message passing interface
DVR	direct volume rendering	MRI	magnetic resonance imaging
e.g.	exempli gratia (for example)	OpenGL	open graphics library
et al.	et alii, et aliae, et alia (and others)	OpenCL	open compute language
etc.	et cetera	PC	personal computer
EPI	gradient-recalled echo-planar imaging	PET	positron emission tomography
FEM	finite element method	PNG	portable network graphics
FID	free induction decay	SPECT	single-photon emission computed tomography
fps	frames per second	pixel	picture element
fMRI	functional MRI	RAM	random access memory
GB	gigabyte	RF	radio frequency
G-Buffer	geometry buffer	RGB	red, green, and blue
GHz	gigahertz	RGBA	red, green, blue, and alpha
GIF	graphics interchange format	ROI	region of interest
GIS	geographic information system	SIMD	single instruction, multiple data
GLSL	OpenGL shading language	SPM	statistical parameteric map
GPU	graphics processing unit	TCP/IP	transmission control protocol/ internet protocol
GPGPU	general purpose computation on GPUs	texel	texture element
GUI	graphical user interface	URL	unified resource locator
HTML	hypertext markup language	US	ultrasound
		voxel	volume element
		XML	extensible markup language

Abstract

Direct volume visualization has been established as a common visualization technique for tomographic volume datasets in many medical application fields. In particular, the introduction of volume visualization techniques that exploit the computing power of modern graphics hardware has expanded the application capabilities enormously. However, the employment of programmable graphics processing units (GPUs) usually requires an individual adaption of the algorithms for each different medical visualization task. Thus, only few sophisticated volume visualization algorithms have yet found the way into daily medical practice. In this thesis several new techniques for medical volume visualization are presented that aid to bridge this gap between volume visualization and medical applications. Thereby, the problem of medical volume visualization is addressed on three different levels of abstraction, which build upon each other.

On the lowest level a flexible framework for the simultaneous rendering of multiple volume datasets is introduced. This is needed when multiple volumes, which may be acquired with different imaging modalities or at different points in time, should be combined into a single image. Therefore, a render graph was developed that allows the definition of complex visualization rules for arbitrary multi-volume scenes. From this graph GPU programs for optimized rendering are generated automatically.

The second level comprises interactive volume visualization applications for different medical tasks. Several tools and techniques are presented that demonstrate the flexibility of the multi-volume rendering framework. Specifically, a visualization tool was developed that permits the direct configuration of the render graph via a graphical user interface. Another application focuses on the simultaneous visualization of functional and anatomical brain images, as they are acquired in studies for cognitive neuroscience. Moreover, an algorithm for direct volume deformation is presented, which can be applied for surgical simulation.

On the third level the automation of visualization processes is considered. This can be applied for standard visualization taks to support medical doctors in their daily work. First, 3D object movies are proposed for the representation of automatically generated visualizations. These allow intuitive navigation along precomputed views of an object. Then, a visualization service is presented that delegates the costly computation of video sequences and object movies of a volume dataset to a GPU-cluster.

In conclusion, a processing model for the development of medical volume visualization solutions is proposed. Beginning from the initial request for the application of volume-visualization techniques for a certain medical task, this covers the whole life cycle of such a solution from a prototype to an automated service. Thereby, it is shown how the techniques that where developed for this thesis support the generation of the visualization solutions on the different stages.

KURZFASSUNG UND KAPITELÜBERBLICK

Die direkte Volumenvisualisierung hat sich in vielen medizinischen Anwendungsgebieten als allgemeine Visualisierungstechnik für tomographische Volumendatensätze etabliert. Insbesondere die Einführung von Volumenvisualisierungstechniken, die die Rechenleistung moderner Graphikhardware ausnutzen, hat die Anwendungsmöglichkeiten stark erweitert. Allerdings erfordert der Einsatz von programmierbaren Graphikprozessoren (GPUs) normalerweise für jede unterschiedliche Visualisierungsaufgabe die individuelle Anpassung der angewendeten Algorithmen. Deshalb haben bisher nur wenige technisch anspruchsvolle Volumenvisualisierungsalgorithmen den Weg in die tägliche medizinische Praxis gefunden. In dieser Arbeit werden mehrere neue Visualisierungstechniken vorgestellt, die dabei helfen, diese Lücke zwischen Volumenvisualisierung auf der einen Seite und medizinischen Anwendungen auf der andern Seite zu schließen. Dabei wird das Problem der medizinischen Volumenvisualisierung auf drei unterschiedlichen, aufeinander aufbauenden Ebenen behandelt.

Auf der untersten Ebene wird ein flexibles Framework für die simultane Visualisierung mehrerer Volumendatensätze eingeführt. Dieses wird benötigt, wenn mehrere Volumendatensätze, die beispielsweise zu unterschiedlichen Zeitpunkten oder mit unterschiedlichen Aufnahmetechniken erfasst wurden, in einer einzigen Darstellung kombiniert werden sollen. Hierfür wurde der sogenannte Rendergraph entwickelt, der die Festlegung komplexer Visualisierungsvorschriften für beliebige, aus mehreren Volumina bestehende Szenen ermöglicht. Aus diesem Graphen werden automatisch GPU-Programme für das optimierte Rendering generiert.

Die zweite Hierarchieebene umfasst interaktive Volumenvisualisierungsanwendungen für unterschiedliche medizinische Zwecke. Hier werden mehrere Werkzeuge und Techniken präsentiert, die die Flexibilität des Multivolumenrenderingframeworks veranschaulichen. Im Speziellen wird ein Visualisierungswerkzeug vorgestellt, das es ermöglicht, den Rendergraphen direkt über die graphische Benutzerschnittstelle zu konfigurieren. Eine andere Anwendung fokussiert auf die gleichzeitige Darstellung von funktionellen und anatomischen Aufnahmen des Gehirns, die beispielsweise im Rahmen von kognitiven Studien erfasst werden. Darüber hinaus wurde ein Algorithmus für die direkte Deformation von Volumendatensätzen entwickelt, der in der Chirurgiesimulation eingesetzt werden kann.

Auf der dritten Hierarchiebene wird die Automatisierung von medizinischen Visualisierungsprozessen betrachtet. Diese kann für Standardvisualsierungsaufgaben zur Unterstützung von Medizinern in ihrer täglichen Arbeit eingesetzt werden. Für die Darstellung der Visualisierungsergebnisse werden 3D-Objectmovies vorgeschlagen, die eine interaktive Navigation entlang vorberechneter Ansichten eines Objektes ermöglichen. Außerdem wird ein Visualsierungsservice vorgestellt, der die teure Berechnung von Videos und 3D-Objectmovies auf einen GPU-Cluster auslagert.

Abschließend wird ein Vorgehensmodell für die Entwicklung von Volumenvisualisierungslösungen vorgeschlagen. Ausgehend von der initialen Anforderung, Volumensvisualisierungtechniken für eine bestimmte medizinische Aufgabe einzusetzen, umfasst dieses Vorgehensmodell den kompletten Lebenszyklus einer Visualisierungslösung, von einem Prototypen bis hin zu einem automatisierten Visualisierungsservice. Hierbei wird gezeigt, wie die im Rahmen der vorliegenden Arbeit entwickelten Techniken die Erstellung der Visualisierungslösungen auf den unterschiedlichen Stufen unterstützen.

Kapitelüberblick

Kapitel 1: Einleitung

Das Einleitungskapitel führt in die Themenstellung dieser Arbeit – Volumenvisualisierung für medizinische Anwendungen – ein. Es wird dargestellt, dass Volumenvisualisierung heutzutage zwar in viele unterschiedlichen medizinischen Anwendungsgebieten eingesetzt wird, dass aber fortschrittliche Visualisierungstechniken, die moderne Graphikhardware einsetzen, noch selten zur Anwendung kommen. Als Grund hierfür wird die komplizierte und aufwendige Programmierung von Graphikkarten angeführt. Aus dieser Problematik wird die Zielsetzung dieser Arbeit, die Lücke zwischen modernen Visualisierungstechniken und deren Anwendung in der Medizin zu schließen, abgeleitet.

Zur Einordnung der in dieser Arbeit vorgestellten Techniken wird gezeigt, dass das Problem der medizinischen Volumenvisualisierung auf drei unterschiedlichen Abstraktionsebenen, die aufeinander aufbauen, betrachtet werden kann. Die unterste Ebene stellen die Algorithmen für das direkte Volumenrendering dar, auf der zweiten Ebene sind interaktive Visualisierungsapplikationen für unterschiedliche medizininische Einsatzgebiete angesiedelt und die dritte Ebene umfasst Techniken zur Automatisierung des Visualisierungsprozesses. Abschließend wird ein kurzer Überblick über die Arbeit gegeben und der Bezug zwischen den verschiedenen Themen und den drei Hierarchiestufen der medizinischen Volumenvisualisierung hergestellt.

Kapitel 2: Grundlagen der medizinischen Bildgebung und Visualisierung

Im zweiten Kapitel werden die Grundlagen der medizinischen Bildgebung und der medizinischen Visualisierung, die für das weitere Verständnis der Arbeit nötig sind, erläutert.

Zu Beginn werden die wichtigsten tomographischen Bildgebungstechniken eingeführt, mit deren Hilfe volumetrische Aufnahmen des Körperinneren eines Patienten gemacht werden können. Nach einer kurzen Erläuterung der Röntgentechnik, deren Erfindung den Beginn der modernen medizinischen Bildgebung darstellt, werden die

Computertomographie (CT) und die Magnetresonanztomographie (MRT) näher beschrieben.

Im folgenden Abschnitt wird die medizinische Visualisierungspipeline vorgestellt, die auf der allgemeinen Visualisierungspipeline aufbaut. Sie besteht aus den vier Stufen Bilderfassung, Vorverarbeitung und Bildanalyse, Visualisierung und visuelle Analyse. Diese vier Stufen, die teilweise aus mehreren Piplineschritten bestehen, werden näher erläutert und es werden beispielhafte Algorithmen und Verfahren besprochen. Dabei werden die letzten beiden Pipelinestufen hervorgehoben, da diese Arbeit sich auf diese fokussiert.

Da die in dieser Arbeit vorgestellten Visualisierungstechniken auf dem Verfahren des direkten Volumenrenderings aufbauen, wird im letzten Abschnitt des zweiten Kapitels diese Technik ausführlicher erläutert. Zunächst wird das Volumenrenderingintegral und seine numerische Auswertung beschrieben. Dann wird die praktische Umsetzung des Verfahrens anhand der Volumenrenderingpipeline erklärt und schließlich auf zwei GPU-basierte Renderingalgorithmen, das scheibenbasierte (slice-based) Rendering und das Raycasting, eingegangen.

Kapitel 3: Flexibles Multivolumenrendering

Im dritten Kapitel wird ein Framework für das flexible Rendering von mehreren Volumen (Multivolumenrendering) vorgestellt. Zu Beginn wird allgemein in das Thema Multivolumenrendering eingeführt und gezeigt, dass es drei Wege gibt, die Darstellung mehrerer Volumendatensätze in einem Bild zu kombinieren. Dabei wird herausgestellt, dass die Technik, die Volumendatensätze auf Akkumulationsebene zu kombinieren, d.h. bevor die Farben entlang eines Sichtstrahls aufakkumuliert werden, am besten für medizinische Anwendungsfälle geeignet ist. Bei Anwendung dieser Technik zerfällt die Problemstellung des Multivolumerenderings in zwei Hauptpunkte: das simultane Abtasten mehrere Volumendatensätze (Sampling) und die kombinierte Abbildung der Datenwerte an den Abtastpunkten auf Farben (Shading). Für beide Problemstellungen werden in den weiteren Abschnitten des zweiten Kapitels Lösungen vorgestellt und es wird gezeigt, wie diese in einem einheitlichen Framework zusammengefasst werden können.

Zunächst werden zwei GPU-basierte Multivolumenrenderingtechniken vorgestellt, die auf eine optimierte Abtastung der Volumendatensätze abzielen: scheibenbasiertes Multivolumenrendering und Multivolumenraycasting. Beide zerlegen die Volumendatensätze in Bereiche sich überlappender Volumen und wenden auf diese Bereiche speziell angepasste GPU-Programme an.

Im darauffolgenden Abschnitt wird ein Verfahren präsentiert, mit dem diese GPU-Programme automatisch aus einem individuell zusammengestellten Graphen, dem sogenannten Rendergraphen, generiert werden können. Zunächst wird das Konzept des Rendergraphen erläutert und die verschiedenen Typen von Renderknoten eingeführt. Dann wird die Verwendung des Rendergraphen an einem Beispiel veranschaulicht und schließlich wird beschrieben, wie aus einem Rendergraphen die zugehörigen GPU-

Programme erzeugt werden. Abschließend wird erläutert, wie die Generierung der GPU-Programme dynamisch in den Multivolumerenderingprozess integriert werden kann.

Kapitel 4: Interaktive medizinische Volumenvisualisierung

In Kapitel 4 werden drei verschiedene interaktive Visualisierungsanwendungen vorgestellt, die aufbauend auf dem Multivolumenrenderingframework an bestimmte medizinische Anwendungsgebiete angepasste Visualisierungsmethoden und Interaktionsmechanismen zur Verfügung stellen. Im ersten Teil wird ein Visualisierungswerkzeug präsentiert, das eine direkte Manipulation des Rendergraphen über die graphische Benutzungsschnittstelle ermöglicht. Hierdurch kann der Anwender individuelle Visualisierungen erzeugen und die zur Verfügung gestellten Visualisierungstechniken auch bei neuartigen Fragestellungen einsetzen. Zunächst wird der Aufbau der Benutzungsoberfläche des Visualisierungswerkzeugs erläutert. Dann wird gezeigt, wie die Funktionalität durch neue Renderknoten erweitert werden kann. Neben der Implementierung des eigentlichen Renderknotens muss hierfür ein geeignetes Eingabeelement zur Einstellung der Parameter und Funktionalität zur Serialisierung und Deserialisierung des Knotenzustands zur Verfügung gestellt werden. Im darauffolgenden Abschnitt werden exemplarisch einige Renderknoten beschrieben, die bereits in das Visualisierungswerkzeug integriert sind. Abschließend werden einige medizinische Anwendungsbeispiele präsentiert und die Leistungsfähigkeit der unterschiedlichen Renderingtechniken diskutiert.

Im zweiten Teil des vierten Kapitels wird eine Anwendung zur Visualisierung von funktionellen Aufnahmen des Gehirns vorgestellt. Zu Beginn wird in die funktionelle Bildgebung und deren Einsatz auf dem Gebiet der kognitiven Neurowissenschaften eingeführt. Hierbei wird die Technik der funktionellen Magnetresonanztomographie (fMRT) erklärt, der Ablauf einer kognitiven Studie beschrieben und auf die statistische Auswertung der erfassten Daten eingegangen. Darauffolgend wird beschrieben, wie die funktionellen Aktivierungen in Kombination mit anatomischen Referenzaufnahmen dreidimensional dargestellt werden können. Es wird gezeigt, dass mit Hilfe der Multivolumenrenderingtechnik aussagekräftige Darstellungen erzeugt werden können, die es erlauben die funktionellen Aufnahmen dreidimensional zu analysieren.

Darüberhinaus wird eine erweiterte Renderingtechnik vorgestellt, bei der die Struktur der Gehirnoberfläche durch Line-Integral-Convolution (LIC), ein Verfahren aus der Strömungsvisualisierung, hervorgehoben wird. Ziel ist es, die Gehirnstruktur klar darzustellen, ohne dabei die innenliegenden Aktivierungsregionen zu verdecken. Zunächst werden die mathematischen Grundlagen der LIC-Berechnung eingeführt. Dann wird gezeigt, wie die Beschleunigungstechnik des verzögerten (deferred) Shadings in den Multivolumenrenderingframework integriert werden kann, welche eine performante Visualisierung der Oberflächenkrümmung des Gehirns mittels LIC ermöglicht.

Im dritten Teil des vierten Kapitels wird ein Verfahren zu direkten Deformation von Volumendatensätzen präsentiert, das beispielsweise in der Chriurgiesmulation eingesetzt werden kann. Ziel dieses Verfahrens ist es, beliebige Volumendatensätze ohne Vorverarbeitung direkt deformieren zu können. Dabei wird die Deformation komplett auf der GPU durchgeführt und kann so nahtlos in den Visualisierungsprozess integriert werden. Als Basis des Verfahrens dient der sogenannte 3D-ChainMail-Algorithmus. Nach der Einführung der Grundlagen des 3D-ChainMail-Algorithmus, wird gezeigt, wie dieser auf die GPU portiert werden kann. Hier wird erläutert, wie das vom 3D-ChainMail-Algorithmus verwendete Deformationsgitter durch eine 3D-Textur repräsentiert wird, welche Anpassungen des Deformationsverfahrens für die GPU-Implementierung vorgenommmen werden müssen und wie es möglich ist, das invertierte Defomationsgitter für die direkte Visualisierung des verformten Volumes einzusetzen. Daran anschließend wird eine vollständig GPU-basierte Deformationspipeline vorgestellt, die alle Schritte von der interaktiven Manipulation durch den Anwender, über die Durchführung des eigentlichen Deformationsverfahrens, bis hin zur Visualisierung umfasst.

Kapitel 5: Automatisierte medizinische Volumenvisualisierung

Im fünften Kapitel wird die Automatisierung des Volumenvisualisierungsprozesses behandelt. Diese hat zum Ziel, die Mediziner in ihrer täglichen Arbeit zu unterstützen und standardisierte Vorgehensweisen für die Visualisierung bei bestimmten medizinischen Fragestellungen einzuführen. Als Anwendungsbeispiel dient die Analyse von intrakraniellen Aneurysmen. Hierzu werden zunächst die medizinischen Grundlagen dieser Aufgabenstellung erläutert und es wird gezeigt, welche Schritte für einen standardisierten Analyse- und Visualisierungsprozesses durchgeführt werden müssen. Ergebnis dieses Prozesses sind mehrere vordefinierte Videosequenzen, die die Gefäßstruktur im Gehirn entweder als Gesamtes oder Teilbereiche davon darstellen. Mit Hilfe dieser Videosequenzen kann ein untersuchender Mediziner auf einfache Weise überprüfen, ob ein oder mehrere Aneurysmen vorliegen.

Im anschließenden Abschnitt wird eine alternative Methode für die Darstellung von vorberechneten 3D-Visualisierungen des Untersuchungsgebietes, die sogenannten 3D-Objectmovies, vorgeschlagen. Bei diesen werden an festgelegten Kamerapositionen auf einer sphärischen Hülle um das dargestellte Objekt Visualisierungen vorberechnet. Mit einem speziellen Betrachtungsprogramm ist es dann möglich, entlang dieser vorberechneten Ansichten interaktiv zu navigieren. Für die automatisierte medizinische Volumenvisualisierung wird ein neu entwickeltes Objectmovieformat vorgestellt, das speziell an die Anforderungen der Analyse von medizinischen Volumendaten angepasst ist. Zunächst wird der Aufbau des Formats erläutert, welches ermöglicht, unterschiedliche Darstellungen eines Objekts in einem Objetmovie zusammenzufassen. Dann wird ein speziell entwickeltes Betrachtungsprogramm beschrieben, das auf Java basiert und als Applet in eine Webseite eingebunden werden kann. So ist es auf einfach Weise möglich, ein Objectmovie über das Internet verfügbar zu machen. Schließlich wird gezeigt, wie das neue Objectmovieformat für die standardisierte Analyse von intrakraniellen Aneurysmen eingesetzt werden kann.

Im dritten Abschnitt des fünften Kapitels wird ein Webservicesystem präsentiert, mit dem die standardisierte Analyse und Visualisierung von medizinischen Volumendaten automatisiert auf einem speziellen Server ausgeführt werden kann. Über ein dynamisches Webinterface kann ein Anwender Volumendaten auf den Server hochladen, dort werden automatisch Videosequenzen oder Objectmovies generiert und schließlich können die Visualisierungsergebnisse über das Webinterface abgerufen werden. Zunächst wird die Architektur des Systems erläutert, dann wird näher auf den sogenannten Renderserver eingegangen. Hierbei handelt es sich um einen Clustercomputer, bei dem die einzelnen Knoten mit leistungsfähigen Grafikkarten ausgestattet sind. Zur Beschleunigung des Visualisierungsprozesses werden die Videosequenzen und Objectmovies verteilt auf den Clusterknoten berechnet und am Ende zusammengefasst. Abschließend wird eine für die standardisierte Untersuchung intrakranieller Aneurysmen entwickelte Webanwendung vorgestellt und es werden einige Leistungsmessungen des Systems diskutiert.

Kapitel 6: Iterative Entwicklung von medizinischen Volumenvisualisierungslösungen

Im sechsten Kapitel wird ein allgemeines Vorgehensmodell für die iterative Entwicklung von medizinischen Volumenvisualisierungslösungen vorgeschlagen. Dieses besteht aus vier Entwicklungsstufen, die auf den im Rahmen der Arbeit entwickelten Visualisierungskonzepten basieren. Zunächst werden die vier Entwicklungsstufen detailliert und es wird gezeigt, welche der vorgestellten Visualisierungsverfahren und -techniken jeweils zum Einsatz kommen können. Die erste Entwicklungsstufe beschreibt den Zustand der klassischen 2D-Analyse medizinischer Aufnahmen. Hierauf aufbauend wird für die zweite Entwicklungsstufe ein 3D-Visualisierungsprototyp entwickelt und evaluiert. In der dritten Stufe wird dann mit Hilfe der zuvor gewonnenen Erkenntnisse eine speziell an den medizinischen Anwendungsfall angepasste interaktive Visualisierungapplikation entwickelt. Schließlich werden in der vierten Entwicklungsstufe die Visualisierungsprozesse automatisiert und als Service einer breiten Gruppe von Anwendern zur Verfügung gestellt. In einer abschließenden Diskussion werden die unterschiedlichen Entwicklungsstufen bzgl. unterschiedlicher Fragestellungen miteinander verglichen, z.B. wer ist die jeweilige Anwendergruppe und für welche Anwendungsfälle sind sie geeignet, und anhand konkreter Beispiele aus der Arbeit veranschaulicht.

Kapitel 7: Schlussfolgerung

Das letzte Kapitel fast die in dieser Arbeit vorgestellten Visualisierungsverfahren zusammen, hebt noch einmal den Zusammenhang der unterschiedlichen Themen hervor und gibt einen kurzen Ausblick auf zukünftige Arbeiten.

CHAPTER 1 INTRODUCTION

Modern tomographic medical imaging techniques provide a detailed volumetric insight into a patient's body to a physician. They are applied for different purposes, such as diagnosis, treatment planning, intraoperative guidance, and postoperative control. There is a wide range of imaging modalities, which are suitable for the display of different anatomical or functional aspects. Frequently, a patient is examined with different imaging modalities to get an integrated view about his or her disease.

A tomographic scan consists of a number of two-dimensional (2D) images, also referred to as slices, which represent cross-sections through the observed part of the body. Usually, the 2D slices are taken at equidistant intervals. In conventional diagnosis a medical doctor analyzes a scan slice by slice and he has to reconstruct the three-dimensional (3D) structures in his mind, which requires a good spatial sense. The task gets even harder when the information of scans from different modalities has to be fused.

To overcome these drawbacks, 3D volume rendering techniques have been introduced, which support the visual analysis of the volumetric data. In the beginning, mainly surface-based rendering methods for the visualization of explicitly extracted surface models of anatomical or pathological structures dominated the field of medical visualization. But with the ongoing performance increase of computer hardware, particularly of programmable graphics processing units (GPUs), direct volume-rendering techniques, which directly visualize a volume data set without the need of an intermediate surface representation, gained the abilty to generate high quality visualizations with interactive frame rates. Thus, direct volume rendering has been established in many medical application fields and is nowadays integrated into most medical visualization environments.

Nevertheless, there is still a wide gap between modern volume-visualization techniques that were recently developed in visualization research and those that are regularly employed in medical practice. Especially in the context of GPU-based volume rendering many improvements have been achieved. But for the application of these techniques to specific medical tasks one needs to have sophisiticated GPU-programming skills and deep medical knowledge simultaneously. Furthermore, when a new volume rendering technique is utilized for a special medical purpose, usually a specialized solution is implemented that can not be ported easily to other application fields. To bridge this gap between volume visualization on the one hand and medical applications on the other hand is the aim of this thesis. Therefore, several general visualization techniques have been developed, which can be easily adapted for specific medical purposes.

1.1 Medical Volume Visualization

The field of medical volume visualization involves a wide variety of different aspects. Basically, the problem domain can be subdivided into three layers of abstraction (see Figure 1.1).

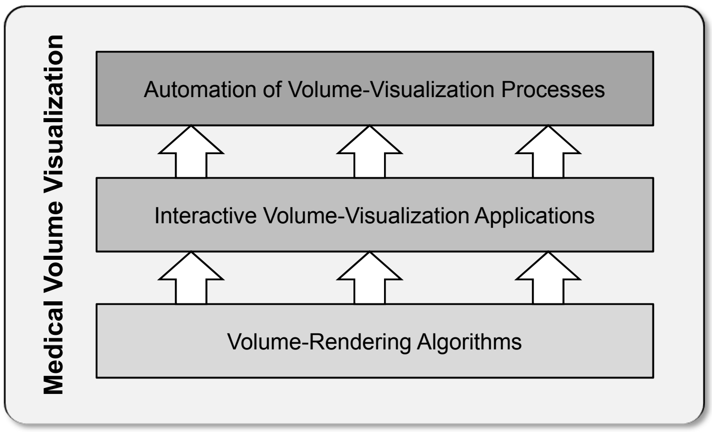

Medical Volume Visualization
Automation of Volume-Visualization Processes
Interactive Volume-Visualization Applications
Volume-Rendering Algorithms

Figure 1.1: Hierarchical ordered layers of the medical volume visualization domain.

The lowest layer represents the basic volume rendering techniques. Here, algorithms should be designed that are appropriate for the visualization of medical image data and that allow the interactive investigation of a dataset. On this level one has to decide if an algorithm should be specialized in the rendering of certain kinds of datasets and visualize them in a determined way, or if it should provide general functionality which can be applied to a wide range of medical data. Further, different algorithms for the visualization of single volumes (*single-volume rendering*) and for the combined visualization of multiple volumes (*multi-volume rendering*) can be used. Often, there is a trade-off between the generality of an algorithm and its rendering performance.

The second abstraction layer is built by interactive medical visualization applications that are built on top of the rendering algorithms. They should suit the requirements of medical practice and should support the interactive analysis of medical volume datasets in an intuitive way. Thereby, two strategies can be pursued. Either a visualization application provides generic functionality, which can be adopted for several different medical tasks, or it is designed for a specific medical purpose and restricts interaction to dedicated functions. While applications of the first type involve high flexibility for experienced users, applications of the latter type can support the daily work of medical doctors who are not visualization experts.

The visualization approaches of the third layer go one step further by the automation of medical volume visualization processes. In clinical routine medical doctors regularly have to investigate similar medical problems, for example the diagnosis of a certain disease. Therefore, often informal workflows for the visual analysis of the acquired volume datasets have been established. By automation of these workflows the visual analysis can be supported and improved. On the one hand, other tasks can be carried out while a computer is performing the automated visualization; on the other hand, the visualization results can be provided in a standardized way. This allows, for example, the easy comparison of different cases and supports the collaboration of medical experts. Techniques for automated visualization should incorporate two objectives. First, it should be easy to initiate an automated visualization, and, second, it should be possible to observe the visualization results in an intuitive way.

1.2 Thesis Overview

This thesis addresses the problem of medical volume visualization on all three abstractions layers shown in Figure 1.1. For each layer general and/or problem specific visualization approaches are presented. Furthermore, it is illustrated how the approaches on the different levels complement one another, and how they can be combined to an integrated solution for visualization in medical application.

After an introduction to the fundamentals of medical imaging and visualization in Chapter 2, Chapter 3 regards the lowest abstraction layer of medical volume visualization. A flexible GPU-based rendering technique for multi-volume scenes is described, which allows the design of visualizations on the abstract level of a so-called render graph. From this graph GPU shaders for optimized rendering are dynamically generated. The strength of the approach is its generality. It can be applied to arbitrary combinations of volume datasets, which, e.g., are taken with different imaging modalities or at different points in time. Furthermore, the modular concept can even simplify the generation of single-volume visualizations.

In Chapter 4 three interactive visualization applications the second abstraction layer of medical volume visualization are presented. Each of them is based on the before introduced multi-volume rendering technique. First, there is a visualization tool that passes the flexibility of the render graph directly to the user. Intended users of this tool are medical visualization experts who want to create meaningful visualizations for new medical problems. The second application targets the field of cognitive neuro science. Specialized visualization techniques are presented that support the simultaneous visualization of functional images, which are gathered in cognitive studies, and anatomical reference volumes. At last, a GPU-based algorithm for deformation of medical volume datasets is introduced, which can for example be used in surgery simulation. The focus lies here on the direct integration of interactive volume deformation into the visualization procedure

Chapter 5 discusses the automation of medical volume visualization processes – the third layer of the visualization hierarchy – using the example of standardized analysis of intracranial aneurysms. First, a technique for the interactive presentation of pre-computed visualization results, so-called medical object movies, is introduced. Then, a web-service system is presented which offers automated analysis and visualization of medical image data. After the upload of a medical volume dataset via a webinterface, the data is processed in parallel on a GPU-cluster and the generated visualizations are provided for download.

Finally, in Chapter 6 it is shown how the techniques presented in this thesis can support the development of visualization solutions for specific medical tasks. An iterative processing model is introduced that starts with the analysis of the requirements of an expert in the respective medical domain and ends with automated visualization processes that can be distributed to a wide group of clinical users. Following this processing model advanced volume visualization techniques can be easly adopted for many medical applications.

Chapter 2 Fundamentals of Medical Imaging and Visualization

Medical volume visualization generates 2D images of volumetric datasets that were acquired from a patient with a tomographic imaging device. There exist several tomographic imaging techniques, which are appropriate for different medical purposes. For the analysis and visualization of tomographic datasets a large variety of algorithms and techniques has been developed, which are often specialized for certain tasks. Nevertheless, a common pipeline of processing steps has been established, which is applied in a similar way for most problems of medical volume visualization. In this chapter the fundamentals of this pipeline of medical imaging and visualization are introduced. Thereby, the focus is laid on those techniques and algorithms that build the basis of the visualization solutions presented in this thesis. Section 2.1 starts with a brief introduction to the most important tomographic imaging techniques. Then, in Section 2.2 the stages of the common medical visualization pipeline are introduced. Since most visualization techniques that were developed for this thesis utilize GPUs, Section 2.3 gives an introduction to the basics of hardware accelerated rendering. Finally, in Section 2.4 the technique of direct volume rendering, which builds the fundament of this thesis, is explained in detail.

2.1 Tomographic Medical Imaging Techniques

Medical imaging technologies aim to give insight into a patients body without the need of invasive interventions. They can assist physicians and surgeons e.g. in the analysis of pathological structures, in the diagnosis of diseases, or in the planning of surgical operations. The discipline of diagnostic medical imaging (*radiology*) has its origins in the discovery of *X-rays* by Wilhelm Conrad Röntgen in 1895 [105]. X-radiation is electromagnetic radiation with wavelengths in the range of 10 to 0.01 nm, which is differently absorbed in different materials.

In medical diagnosis X-rays are generated by a X-ray tube, sent through a patients body and then recorded on a film. Basically, the film measures the attenuation of the X-rays by the tissue that they pass while traveling through the human body. The

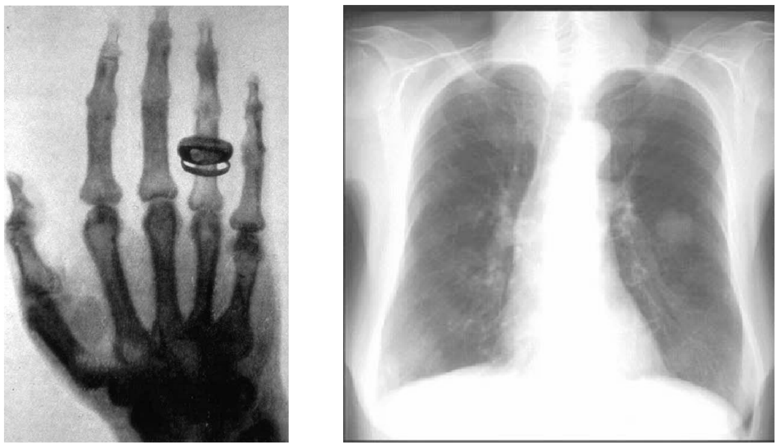


Figure 2.1: Two X-ray images: (a) A historical image of a human hand which was taken by Röntgen in 1896. (b) A recent X-ray image of the human thorax (Image courtesy Peter Hastreiter, University Hospital Erlangen). Please note that the image (a) has inverted intensities due to a different recording technique.

attenuation is caused by two processes: X-rays are absorbed by the structures they hit, and X-rays are scattered by the so-called Compton effect. Bony structures have the highest absorption rates and, thus, appear bright in an X-Ray image. Other tissue types show less absorption, which results in darker areas. Figure 2.1 shows two exemplary X-ray images. Image (a) presents one of the first X-ray photographs taken by Röntgen himself. Here the image intensities are inverted relative to modern X-ray scans, like the one shown in image (b).

The measured X-ray intensity I depends on the initial intensity I_0 and a material dependent X-ray attenuation coefficient μ. It decreases exponentially with increasing thickness d of the passed tissue [98]:

$$I = I_0 \cdot e^{-\mu \cdot d} \tag{2.1}$$

The product $\mu \cdot d$ depicts the attenuation S of the passed material. Usually, the traversed tissue does not exhibit a homogenous attenuation behavior. Thus, in general the total attenuation S is determined by integration of the attenuation coefficients $\mu(l)$ along the ray. Then the resulting intensity is

$$I = I_0 \cdot e^{-\int \mu(l) dl} \tag{2.2}$$

An X-ray image represents a 2D projection of the observed object. Thus, it is difficult to perceive the spatial relationships between the imaged structures. Furthermore, dense material like bone may occlude other details. For this reason in medical diagnosis often two images from different viewing directions are taken. Tomographic imaging methods overcome these drawbacks. Here, an image represents a cross-sectional 2D slice of the scanned object. By taking several 2D slices at distinct positions a 3D image of the inner structures of a patient's body can be acquired. Those 3D images build the basis of medical volume visualization.

There are four major types of tomographic medical imaging techniques, which exploit different physical effects for image acquisition:

- *Computed tomography* (*CT*) uses X-rays, which are sent from different directions through the patient's body.
- *Magnetic resonance imaging* (*MRI*) exploits the magnetic behavior of the hydrogen nuclei in the patient's body when brought to a strong magnetic field.
- *Ultrasound* (*US*) scanning is based on high frequency sound waves, which are emitted into a body part and reflected at interfaces between two tissue types.
- *Nuclear imaging* methods measure the signals of radiopharmaceutical substances when they are processed in the body. The radiopharmaceutical substances are either injected into a vein – *positron emission tomography* (*PET*) – or administered orally – *single-photon emission computed tomography* (*SPECT*).

In the following computed tomography and magnetic resonance imaging, which play a major role in medical volume visualization, are detailed. The explanations are based on the books of Dössel [26], Lehmann et al. [73], and Preim and Bartz [98]. There the interested reader can find more details about CT and MRI as well as about the other imaging techniques.

2.1.1 Computed Tomography

Computed tomography (CT) was introduced by Godfrey Hounsfield in 1967 [52; 53]. It uses the X-ray technology to generate tomographic images. The basic idea is to measure the X-ray attenuation along single rays from many different directions around the examined object. From these measurements a tomographic slice can be reconstructed.

Mathematically CT reconstruction is based on the *Radon transform*. This transform describes a two-dimensional function $f(x, y)$ by all integrals along straight lines over the domain of $f(x, y)$. Consider a straight line $g_{\Phi,s}(l)$ that is parametically defined and determined by the angle Φ from the x-axis and the distance s to the origin. Then the Radon transform of $f(x, y)$ can be defined in dependence of Φ and s:

$$\mathcal{R}(\Phi, s) = \int_{-\infty}^{\infty} f(g_{\Phi,s}(l))dl \tag{2.3}$$

A line of $\mathcal{R}$ with $\Phi = const.$ is called projection $p_\Phi(s)$.

In combination with the *Fourier transform* one can reconstruct the original function f from its Radon transform $\mathcal{R}$. The *Fourier slice theorem* (see Figure 2.2) says that the 1D Fourier transform $P_\Phi(w)$ of the projection $p_\Phi(s)$ describes the values of the 2D Fourier transform $F(u, v)$ of the function $f(x, y)$ along a radial line with the angle Φ.

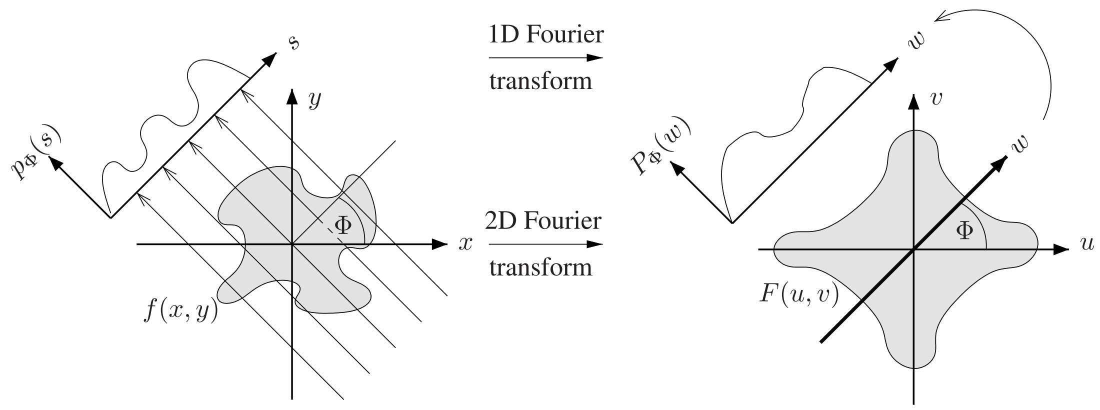


Figure 2.2: Fourier slice theorem: The 1D Fourier transform $P_\Phi(w)$ of the projection $p_\Phi(s)$ is equal to a slice under the angle Φ in the 2D fourier transform $F(u,v)$ of the original function $f(x,y)$.

Consequently, the Fourier transform $F(u,v)$ can be obtained from the Radon transform $\mathcal{R}(\Phi, s)$ and then the original function $f(x,y)$ can be determined by an inverse 2D Fourier transformation.

In computed tomography the Radon transform of the position-dependend X-ray attenuation coefficient $\mu(x,y)$ is measured (see Equation 2.2). Hence, the previously described reconstruction scheme allows the determination of $\mu(x,y)$ on a cross-sectional 2D slice through the observed object. However, today mostly *filtered back projection* is applied [101], which has less computational cost.

First CT scanners used an image acquisition scheme similar to Figure 2.2 left. A single X-ray emitter/detector pair was translated along the object and then rotated to gain the next series of measurements. In contrast to that, modern CT devices use fan beam emitters and multiple detectors. Thus, a number of projections that cover the whole object can be obtained at the same time and the translating movement is no longer needed. Moreover, state-of-the art scanners can acquire up to 64 slices in a single rotation around the object by the application of multiple layers of emitters and detectors.

In medical imaging the computed X-ray attenuation values (μ) are normalized into so-called *Hounsfield units* (HUs). This normalization maps the attenuation of water

tissue type	air	fat tissue	water	liver	heart	kidney	bones
HU Interval	-1000	-900...-170	0	20...60	20...50	30...50	45...3000

Table 2.1: Intervals of Hounsfield values for selected tissue types [98]

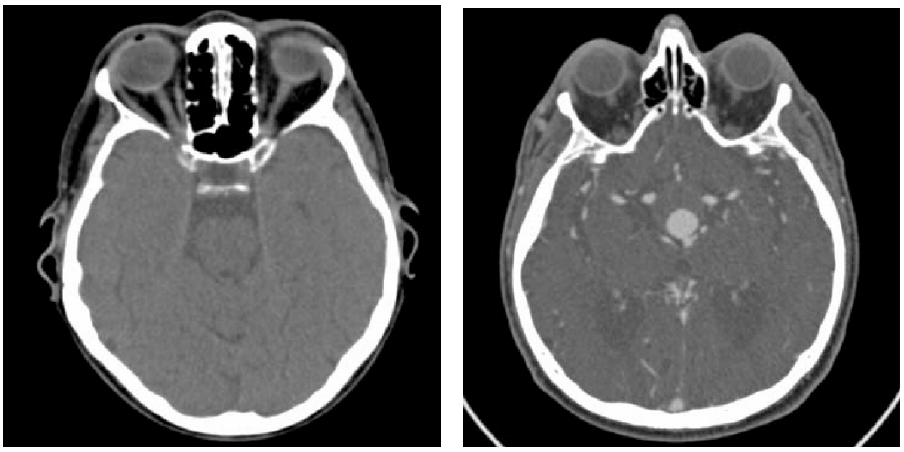


Figure 2.3: Two CT slices through the human head: (a) A conventional CT scan in which the bone structure of the skull can be clearly distinguished; (b) A CTA scan in which blood vessels are emphasized by a previously injected contrast agent. (Image courtesy Peter Hastreiter, University Hospital Erlangen [49])

(μ_{H_2O}) to zero and the attenuation of air to -1000:

$$\mu_{HU} = \frac{\mu - \mu_{H_2O}}{\mu_{H_2O}} \cdot 1000 \tag{2.4}$$

Different organs and tissue types are mapped to typical ranges of the Hounsfield scale (see Table 2.1). It can be seen that the intervals of soft-tissue organs are similar or even overlap. Thus, it is difficult to differentiate those tissue types. In contrast, bony structures are mapped to high Hounsfield units, which can be clearly distinguished. For the examination of vascular structures often *CT angiography* (CTA) is applied. Here, a contrast agent is injected to the venous system, which emphasizes the blood vessels in the CT scan. Figure 2.3 shows two CT slices through the human head. The left image (a) presents a standard CT, the right image (b) shows a CTA with contrast-enhanced vessels.

2.1.2 Magnetic Resonance Imaging

Magnetic resonance imaging (MRI) is based on the effect that atomic nuclei emit an electro-magnetic signal when stimulated by magnetic fields. This signal can be measured and used for reconstruction of tomographic slices through the observed object. For medical purposes usually the magnetic resonance of hydrogen nuclei is examined. Therefore, the patient is brought into a strong external magnetic field. This leads to an alignment of the hydrogen nuclei, which can be considered as small dipole magnets, either parallel or anti-parallel along the magnetic field. Furthermore, the nuclei rotate around themselves (*spin*) and precess (rotate) around the z-axis of the magnetic field B

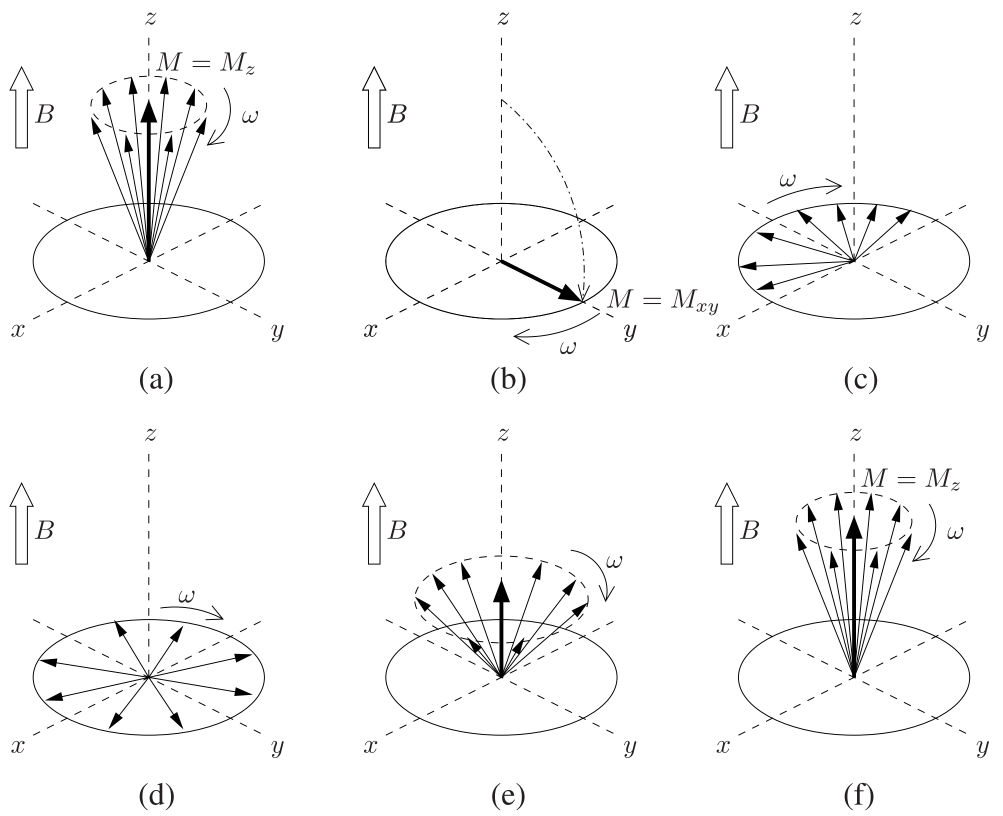


Figure 2.4: Relaxation of hydrogen nuclei after stimulation with a 90° RF pulse: (a) In a static magnetic field B the nuclei rotate with the Larmor frequency ω around the axis of the magentic field. This leads to a longitudinal magnetization $M = M_z$. (b) A 90° impulse alignes the nuclei perpendicular to B. This leads to a transversal magnetization $M = M_{xy}$. (c)-(d) After the impulse the nuclei dephase in the time $T2$ (spin-spin relaxation). (e)-(f) The nuclei slowly realign with the magnetic field B in the time $T1$ (spin-lattice relaxation). (Inspired by [26])

with a specific *Larmor frequency* ω. This frequency depends on the type of the nuclei and the strength of the magnetic field.

When a perpendicular (90°) radio-frequency (RF) pulse signal with the Larmor frequency is additionally activated the nuclei are moved perpendicular to the magnetic field and are forced to preceed in phase. After deactivation of the RF pulse the nuclei slowly release the received energy. The resonance signal after the pulse is called *free induction decay* (FID). It is originated in two different relaxation effects (see Figure 2.4). First, there is the the transversal relaxation (also called *spin-spin relaxation*), which describes the dephasing of the precession in the xy-direction. The time required for this relaxation is called $T2$ and is in the order of a few milliseconds. The second relaxation process is the longitudinal relaxation (*spin-lattice relaxation*). It describes the realignment of the nuclei with the static magnetic field. The time needed for this process is called T1 and is in the order of a second. The times $T1$ and $T2$ differ for

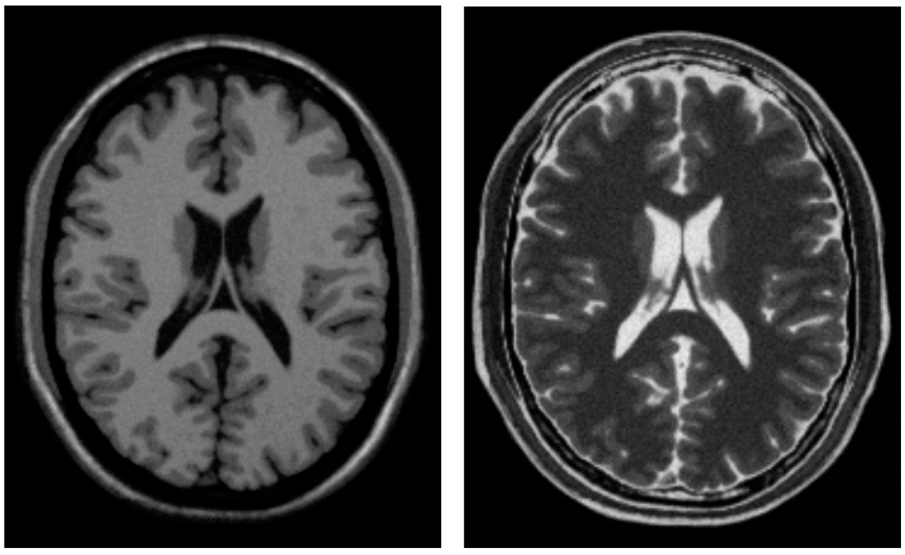

(a) (b)

Figure 2.5: Two MRI slices through the human head: (a) A T1-weighted scan which accentuates the brain tissue. (b) A T2-weighted image in which the brain fluid is emphasized. (Data courtesy *BrainWeb* [77])

different tissue types.

For image acquisition the spatial position of a voxel is encoded by applying three additional gradient magnetic fields. A gradient in z-direction allows the selection of a certain xy-slice, because only those nuclei are stimulated for which the frequency of the RF pulse is equal to the Larmor frequency, which depends on the strength of the static magnetic field. A second gradient in y-direction is activated right after the activation of the RF pulse. This encodes the y-coordinate of a nucleus in the phase of the emitted signal. Finally, the x-coordinate is encoded in the frequency by applying a x-gradient during the measurement of the signal. More precisely, in MRI the 2D fourier transform of the current slice is acquired. By activating the y-gradient for different time spans and measuring the accumulated signal of all activated nuclei at different points in time during the activation of the x-gradient the fourier space, usually called *k-space* in MRI, is sequentially sampled. The final image is then obtained by an inverse fourier transformation.

For MRI imaging the FID signal is not measured directly but so-called echos that are generated by additional $180°$ RF pulses. Different sequences of RF pulses allow the accentuation of different anatomical or functional aspects. E.g., $T1$-weighted images emphasize the difference between the $T1$-relaxation times of different materials; $T2$-weighted images accentuate differences in the time $T2$. Figure 2.5 shows that in $T1$-weighted images different tissue types can be distinguished, while in $T2$-weighted images fluid signals are emphasized. Furthermore, several specialized MRI sequences have been developed, which support the diagnosis and examination of certain diseases.

So-called functional MRI (fMRI) is a MRI technique which allows the measurement of the activations of brain regions while a patient is performing certain cognitive or behavorial tasks. It is, on the one hand, frequently applied in brain research. On the other hand, it can support neuro surgeons in the detection of diseased brain regions. In Chapter 4.2 the fMRI technique and its application in cognitive neuroscience is further detailed.

Another application field of MRI is diffusion tensor imaging (*DTI*). Here, the diffusion of water, described by a diffusion tensor, is measured by a sequence of six MRI images. DTI allows the extraction of the fibers of neural pathways and heart muscles, because water diffusion shows an anisotropic behavior along the fibers. The DTI technique is, e.g., applied in neuro surgery to locate neural pathways that should not be harmed in an intervention.

2.1.3 Discussion

CT and MRI are both tomographic imaging techniques that allow the acquisition of three-dimensional information about a patients body. Due to the different physical effects on which the two techniques are based, they are usually applied for different diagnostic purposes. CT produces a good quality signal for skeletal structures and for contrast-enhanced blood vessels, but is less suited for the differentiation of soft tissue. The latter effect is caused by the similar X-ray attenuation of different soft tissue types. In contrast to that, MRI is not appropriate for the observation of skeletal structures due to the lack of water, which is basically needed for MRI measurements. On the other hand, MRI can provide a high soft tissue contrast and supports the examination of different anatomical and functional aspects by the application of different measurement sequences. Summarizing, CT and MRI are complementary techniques, which are often applied in combination to get an integrated view of a patient's disease. However, it has to be pointed out that CT is based on ionizing X-rays, which can cause cancer, while the magnetic fields applied in MRI are not known to have harmful effects. On the other hand, MRI can not be applied for patients with metallic implants due to the strong magnetic field. Futhermore, the contrast agents that may be used for both techniques can cause allergique reactions in rare cases.

2.2 Medical Visualization Pipeline

The tomographic imaging techniques described in the previous section produce detailed 3D image data of a patient's body. The interpretation and analysis of this data is a sophisticated task and requires a lot of experience. Medical visualization aims to support the analysis of medical image data and can provide new insights. Thereby, medical visualization methods follow the general visualization pipeline [139] (see Figure 2.6), which describes the way from initial data to the final image. The first step of the pipeline is the *acquisition* of input data. This can originate from an arbitrary

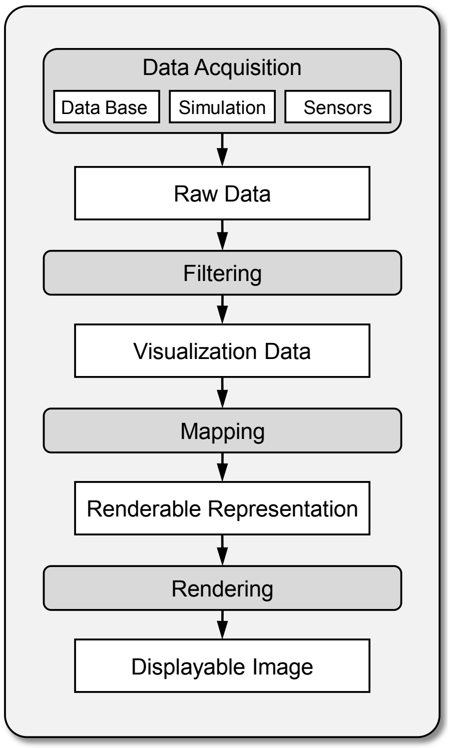


Figure 2.6: General visualization pipeline: Gray boxes represent pipeline steps, white boxes represent (intermediate) results.

data source, e.g. a numerical simulation, a real-world measurement, or a data base. In the filtering step the *raw data*, which is usually not suited for direct visualization, is transformed into abstract *visualization data*. Typical filtering operations are denoising of the data or the elimination of uninteresting samples. The next step of the pipeline, the visualization *mapping*, transforms the visualization data into a geometric (or renderable) representation and applies additional attributes like size, color, or textures. Finally, the *rendering* step generates a *displayable image* by appropriate computer graphics methods.

For medical visualization the general visualization pipeline can be extended to a *medical visualization pipeline* like it is shown in Figure 2.7. It incorporates additional steps for preprocessing of medical image data and takes the interaction with the user into account. The medical visualization pipeline consists of four major stages, *data acquisition*, *preprocessing*, *visualization*, and *visual analysis*, which are detailed in the following sections.

2.2.1 Data Acquisition

The data used for 3D medical visualization is acquired by tomographic medical imaging (see Section 2.1). Which imaging technique is applied, depends on the medical

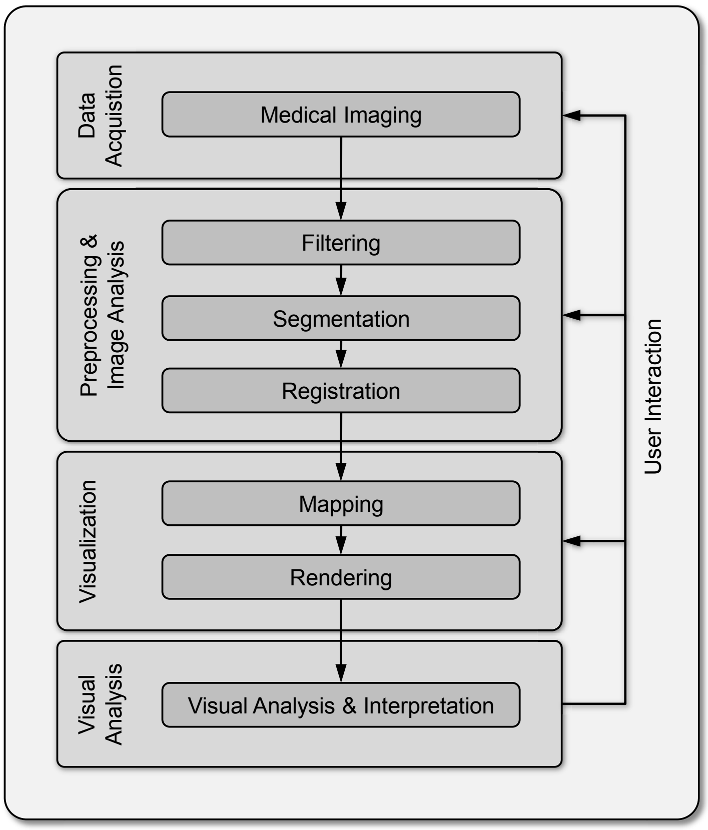


Figure 2.7: Medical analysis and visualization pipeline. It is build of seven pipeline steps (light gray) which can be arranged into four major pipeline stages. This pipeline follows approximately the definition of Hastreiter [49].

task for which the data is taken. Usually, radiologic protocols describe the way how images for specific purposes, e.g. the analysis of a certain disease, should be taken. Sometimes several 3D images with different modalities or different MRI sequences are taken to get a more detailed view of a patient.

Tomographic imaging techniques generate series of cross-sectional 2D slices of the scanned body part. The 2D images can be considered as 2D uniform grids (see Figure 2.8 left), in which the pixels represent the data values at the grid points. The distance between two grid points depends on the imaging modality and is typically constant in both directions. Moreover, the grid point distances in x- and y- direction are usually identical. The geometrical position of a grid point can be determined by multiplication of the grid point distances in x- and y-direction with the two-dimensional index (i, j) of the grid point. Thus, it has not to be stored explicitly.

For 3D visualization the 2D slices are assembled into a *3D volume*. Here, the data elements are called *voxels* (volume elements) and build a 3D uniform grid (see Figure 2.8 right). To each voxel a unique three-dimensional index (i, j, k) is assigned. Like for 2D grids, the position of a grid point can be computed from its index and

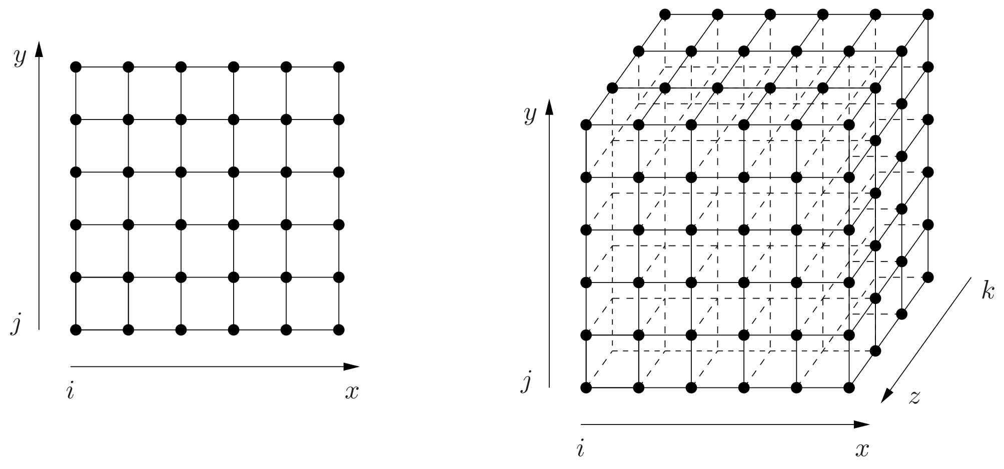


Figure 2.8: 2D grid (left) and 3D grid (right): The data values at the grid points (black dots) are either called pixels (2D) or voxels (3D). Four (2D) respectively eight (3D) neighboring grid points build a grid cell.

the voxel distances in the three directions. If the slice distance (z-direction) is equal to the pixel distances in x- and y-direction the grid is called *cartesian*. When the pixel distance differs in at least one direction it is more generally called *uniform*. In a uniform 3D grid each grid point is connected to six direct neighbors (thus, it has regular topology). Eight neighboring grid points build a cuboid cell. The data values at the corners of a grid cell are given by the corresponding voxels. In-between data values can be computed by interpolation. Details about interpolation in 3D uniform grids are given in Section 2.4.2.

While medical volume datasets are mainly arranged on uniform grids, there are other grid types which provide a greater flexibility by varying spacing, varying topology or varying cell types. Those grids can often be found in numerical simulations. *Rectilinear* grids and *curvilinear* grids are like uniform grids *structured grids* with a regular topology but allow a better adaption to the underlying data. In rectilinear grids the spacing in a certain direction can differ throughout the volume but is constant between cells with similar indices. In contrast to that, curvilinear grids do not restrict the positions of their grid points.

Unstructured grids have neither predefined geometry nor predefined topology. Thus, vertex coordinates and vertex connectivity have to be stored explicitly. This increases the storage size and complicates data access and interpolation but provides, on the other, hand a high degree of flexibility. Most popular are *tetrahedral grids*, which are solely assembled of tetrahedral cells.

2.2.2 Preprocessing and Image Analysis

The preprocessing and image analysis stage aims to enhance and analyze the raw image data for improved visualization. It can consist of up to three pipeline steps. In the *filtering* step typical filtering operations, like denoising or contrast enhancement, are performed. In the *segmentation* step anatomical or pathological structures are extracted. When several datasets should be visualized simultaneously, they have to be aligned in advance in the *registration* step. While filtering operations are regularly conducted, the application of segmentation and registration depends on the specific visualization task. Further on, the order of segmentation and registration can be reversed.

Filtering

The filtering step comprises all operations that restrict the amount of data and enhance it for further processing. As a first step often a *region of interest* (ROI) is selected that covers all structures that are relevant for further analysis. Usually, a cuboid sub-volume is chosen. The selection of a ROI can, on the one hand, accelerate subsequent computations and, on the other hand, improve visualization, because irrelevant structures are excluded.

Many filtering operations take the *histogram* of a data set into account. A histogram gives for a discrete data value the frequency of its occurrence in a discrete dataset. If a volume contains N voxels and the data values are in the set $G = \{0, 1, .., G_{max}\}$, the occurrence probability of a certain data value $g \in G$ is

$$p(g) = \frac{n_g}{N}, \tag{2.5}$$

where n_g is the total number of occurrences of g.

The peaks (local maxima) in a histogram are often related to a certain tissue type. This information can be used to enhance important structures and to suppress the display of tissue that is not relevant. To enhance the overall image contrast *histogram equalization* can be applied, which transforms the data values such that the occurrence of data values is equally distributed in the resulting histogram. The transformation T is defined as follows:

$$T(g) = \left(\sum_{i=0}^{g} p(i)\right) \cdot G_{max}. \tag{2.6}$$

For noise reduction a filter function F is applied to the volume data set. The general concept of filtering of a function g is the convolution of g with the filter function F:

$$g'(x) = (g * F)(x) = \int_{-\infty}^{\infty} g(\xi) \cdot F(x - \xi) d\xi. \tag{2.7}$$

For discrete images the convolution is expressed as weighted sum of the discrete signal g over the discrete filter kernel F with $2N + 1$ elements:

$$g'(u) = \sum_{i=-N}^{N} g(u - i) \cdot F(i) \tag{2.8}$$

The 1D filtering concept can be easily adapted to 2D images or 3D volumes by applying a 2D filter kernel or a 3D filter kernel.

Since it can be assumed that noise occurs in high frequencies, typical noise reduction filters are low pass filters which suppress high frequencies. Frequently a *Gaussian* filter kernel is applied, which represents a discrete version of the symmetric Gaussian distribution function with the standard deviation σ:

$$G(x,\sigma) = \frac{1}{\sigma \cdot \sqrt{2\pi}} e^{\frac{x}{\sigma^2}}. \tag{2.9}$$

There are numerous filters for other image-processing purposes. E.g., gradient filters emphasize edges in an image, which can be useful for subsequent edge detection. Specialized filters can enhance certain structures like bone or vessels. More details about filtering can be found in [73], [98], and [145].

Segmentation

Segmentation decomposes a medical volume dataset into anatomical and/or pathological structures that are relevant for a specific visualization task. Thereby, segmentation comprises two aspects. On the one hand, relevant structures have to be reliably identified. On the other hand, the shape of a segmented structure should be precisely specified.

Technically, segmentation assigns to each voxel a unique *tag* (label) that indicates its membership to a specific structure. These tags are stored in an additional volume, a so-called *tagged volume*, which has equal extent as the original dataset. Most segmentation methods work on 2D images. They can be applied to 3D volumes by processing them slice by slice. However, for some techniques 3D counterparts have been developed.

Segmentation can be performed manually, semi-automatic, or fully automatic. In manual segmentation a user has to mark the voxels that belong to a certain structure manually on each slice of a volume dataset. This method is time consuming and usually not practical for regular application. In contrast, fully-automatic methods perform segmentation without any user interaction. Since this is a sophisticated challenge, most segmentation approaches are semi-automatic. They take into account that the detection of relevant structures is a high-level task, which is best performed by a human, while the delineation of the precise shape of a structure can be better carried out by a computer. (Semi-)automatic segmentation approaches can be grouped into four classes:

Pixel-based methods Pixel-oriented segmentation methods only take the intensity of a voxel into account. A pixel is applied to a certain structure if its intensity lies within an interval of a lower and upper intensity threshold. For the selection of a threshold usually the histogram of an image or volume is observed. A local minimum often represents a threshold that optimally separates two tissue types.

Region-based methods For region-oriented segmentation not only the intensity of a pixel but also its neighborhood is examined. E.g., *region growing* starts with one or more user selected seed points and adds neighboring voxels until the intensity of a voxel exceeds a user-defined threshold. The threshold can again be chosen via the histogram.

Watershed segmentation considers a volume or image as a topographic landscape with ridges and valleys. The height of a voxel is typically defined by its intensity value or its gradient magnitude. This landscape is stepwise "flooded", which leads to a large number of separated regions, so-called *catchment bassins*. When the water level exceeds a watershed between two neighboring catchment bassins, they are merged. The algorithm has to be stopped when an appropriate segmentation of the dataset is achieved.

Edge-based methods Edge-based segmentation techniques try to find continuous edges that enclose the searched structure. An important representative of this class is *livewire* segmentation [88]. This method uses Dijkstra's graph search algorithm to find a path with minimal cost between two user defined control points in a 2D slice image. The cost function between two neighboring pixels depends on their intensities and on the gradient magnitude and gradient direction. While one of the control points is fixed, the user can replace the other until the generated edge best fits the target structure. By repeatedly adding new control points the complete boundary of the target structure can be determined.

Model-based methods Model-based segmentation methods use an initial model that makes assumptions about size, shape, gray level distribution etc. of the target structure. This model is iteratively fitted to the examined dataset.

An *active contour model* or *snake* [60] is a two-dimensional parametric curve, which is deformed towards the boundary of the target structure by internal and external forces. Therefore, an energy function is defined that is composed of an inner energy, which represents the smoothness of the curve, and an external energy, which is derived from the image's intensity values and gradient magnitudes. By minimizing this energy function a smooth boundary of the target structure can be found. *Balloon segmentation* [128] extends the snake concept to 3D.

Level-set segmentation techniques [121] also use internal (smoothness) and external (image) constraints to determine the boundary of a target structure. But in contrast to active contour models, the boundary is implicitly defined by a so-called *level-set function*, which is evolved under control of a partial differential equation. Level-set methods can be applied for 2D images and 3D volumes.

Active shape models (ASMs) [23] are parameterized descriptions of the shape of anatomical structures. They are generated from a number of reference datasets by statistical analysis and describe the main modes of shape variation. For segmentation the mean shape of the searched structure is initially placed in the target image. Then the model parameters (weights) are iteratively adapted until an optimal fit is achieved, for example in dependence of the image's gradient magnitude.

In [73], [98] and [145] more details about segmentation in general, about the introduced segmentation methods and about further specialized techniques can be found. Further more, *direct volume rendering* provides a kind of implicit segmentation via *transfer functions*. Here segmentation information is not explicitly generated, but is applied on-the-fly during rendering. This technique is detailed in Section 2.4.2.

Registration

Registration is the process of finding a spatial transformation that aligns one medical image or volume with another medical image or volume. After registration it is easier to compare the two datasets with each other, and it is possible to generate combined visualizations. There are three major application scenarios for medical image registration:

- **Different Points in Time** For many medical purposes a patient is tomographically imaged at different points in time, e.g. to monitor the course of a disease or the effect of a treatment. Registration allows here a better comparison of the acquired image datasets, for example with respect to tumor growth. Furthermore, registration can be applied for the matching of pre-, intra- and postoperatively taken images. This can ease, for example, the take-over of preoperative analysis and planning results into the operation room. With coaligned pre- and postoperative tomographic scans the success of a surgical intervention can be verified.

- **Multimodal Imaging** To get a better view of a disease, a patient is often examined with several different imaging modalities. Before a direct comparison the scans have to be registered because usually the patient is differently positioned in the different imaging devices. Furthermore, the different modalities may show differing imaging errors.

- **Atlas Matching** Often a patient specific dataset is registered with an atlas dataset that represents the anatomical average of a certain body part. This allows, e.g., the comparison of the patient with the average or the easy identification of certain structures that are already labeled in the atlas.

Usually, registration is an iterative optimization process. In each step first the transformation is slightly adapted, then one of the two datasets is accordingly transformed, and finally the quality of the registration with respect to a certain similarity measure is evaluated. If the similarity is not sufficient, the optimization process is continued.

Registration techniques can be classified either by the employed transformation type or by the applied similarity measurement. Concerning the transformation type, we distinguish between methods that apply global transformations and those that utilize local transformations. *Global transformations* are defined by a small set of parameters and are applied to the whole dataset in a similar way. Rotations and translations are typical global transformations. These *rigid transformations* do not change the geometry of an object and are usually applied to compensate differences in the location and orientation of the imaged structures. More general, any kind of affine transformation can be utilized for global registration. In contrast, *local transformations* apply individual deformations to local areas in a dataset. This permits a better compensation of individual differences in the datasets, which may originate from deformations of the imaged structures. For local transformations B-splines or elastic models can be applied.

Regarding the applied similarity measure, one can distinguish between geometry-based measures and intensity-based measures. For *geometry-based similarity* often the correspondence of explicitly specified control points, so-called landmarks, is examined [104]. Alternatively, the results of a preceding segmentation can be used to match corresponding structures. *Intensity-based similarity measures* compare the intensities of voxels in the untransformed dataset with the corresponding voxels in the transformed dataset. If the two datasets are taken with the same modality, the intensities can be directly compared. This does not work for images taken with different modalities, for example with CT and MR. Here, often *mutual information*, a measure from information theory, is used, which is based on the 2D histogram of the two datasets [136].

For further details about medical image registration and about specific registration techniques the reader is referred to [98] and [145].

2.2.3 Visualization

In the visualization stage the preprocessed 3D volume data is converted into a 2D image. This process is called volume visualization and comprises the mapping of the volume data into a renderable representation and the generation of a 2D projection (*rendering*) due to this representation. There are three different ways of volume visualization (see Figure 2.9), which fundamentally differ in the rendering step and also in the mapping step:

Plane-based Volume Visualization (Figure 2.9 (a)) In plane-based volume visualization a 2D cross-sectional slice of the volume is presented as a 2D image. The slice is often oriented perpendicular to a coordinate axis, but modern computer graphics methods also permit the display of arbitrarily oriented slices. In the mapping step a polygon is computed that represents the 2D cut through the bounding box of the 3D volume. In the rendering step this polygon is rendered parallely to the screen and the corresponding intensity values are mapped to it with 3D texturing methods. The intensity values are usually mapped linearly into gray values, but theoretically any kind of

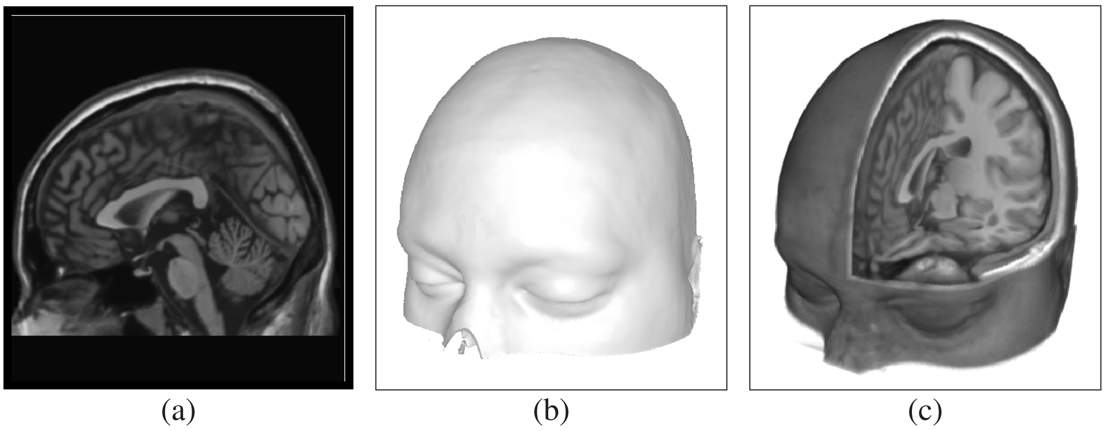


Figure 2.9: Three ways of volume visualization of a MRI dataset of a human head: (a) slice-based visualization of a cross-sectional cut perpendicular to the y-axis; (b) surface-based visualization of the skin; (c) direct volume visualization with a part cut out to get insight to inner structures.

image processing, like contrast enhancement, could be applied.

The advantage of plane-based volume visualization is its similarity to the traditional slice-by-slice examination of a tomographic scan. Thus, medical doctors are already familiar with this way of volume examination. Furthermore, since the displayed information is just two-dimensional, there is no problem with occlusion, which regularly occurs when 3D data is projected to a 2D image. On the other hand, the 3D relationship of the visualized structures is still not presented directly and has to be reconstructed in mind. To partly overcome this drawback, often three perpendicular slices are displayed simultaneously and the user can interactively change the selected slices.

Surface-based Volume Visualization (Figure 2.9 (b)) For surface-based volume visualization primarily a polygonal 3D surface model – usually a triangle mesh – of a certain anatomical or pathological structure is extracted (mapping), which is then rendered with standard 3D rendering methods. The surface model can either be generated along the boundary of a pre-segmented structure or along a so-called *isosurface*. An isosurface is a surface in a 3D dataset on which the corresponding intensity has everywhere the same *isovalue*. It is implicitly defined by the set I of all points $\mathbf{x}$ in the 3D dataset, regarded as a continuous 3D function $V(\mathbf{x})$, for which the difference between the data value $V(\mathbf{x})$ and the isovalue iso is zero:

$$I(iso) = \{\mathbf{x} \in \mathbb{R}^3 | V(\mathbf{x}) - iso = 0\} \tag{2.10}$$

An isosurface effectively separates a volume dataset into a part outside the isosurface and a part inside the isosurface.

The most famous method for the extraction of an isosurfaces from a volume dataset is the *marching cubes algorithm* which was introduced by Lorensen and Cline [75].

This algorithm processes the cuboid cells of the volume grid (see Section 2.2.2) in sequential order. For each of the eight vertex values of a grid cell it is checked if it lies inside or outside the isosurface. There are 256 different configurations of vertex states which can be summarized to 15 different cases. For each of these cases exists a corresponding triangle configuration that approximates the isosurface inside the cell. These configurations are stored in a look-up table and are added to the generated surface mesh when a corresponding vertex configuration occurs. The position of the triangle vertices along the cell edges is computed by linear interpolation. The marching cubes algorithm can also be applied to segmented volume data but then so-called stair-case artifacts occur due to the steep transition between voxels inside and outside a segmented region. To suppress these artifacts the segmented volume dataset and/or the generated mesh can be smoothed.

A generated triangle mesh can be directly rendered with classical graphics hardware support. To achieve a three-dimensional impression of the structure the surface has to be additionally illuminated and shaded. For this reason the surface-based volume rendering technique is often referred to as *surface shaded display*.

The advantage of surface-based rendering is its direct support by the 3D graphics hardware. But in return the generation of a surface mesh is an expensive pre-processing step. Furthermore, a surface model presents only one certain structure of a dataset and not the dataset as a whole. When different structures of a dataset should be analyzed simultaneously, several surface meshes have to be generated and rendered in combination.

Direct Volume Visualization (Figure 2.9 (c)) Direct volume visualization does not use any intermediate representation for rendering but directly accesses the original volume dataset. Basically, for each pixel a *viewing ray* is sent through the volume that accumulates color contributions at equidistant sample points. The color contribution of a sample point is determined by a *transfer function* that maps the density value at the sample point into a color and an opacity value. By variation of the transfer function different structures of a dataset can be emphasized or suppressed. In direct volume visualization mapping and rendering builds a unit that can not be separated into two independent steps.

The advantage of direct volume visualization is the ability to highlight different structures of a dataset without the need of costly preprocessing. Furthermore, there are several rendering and shading techniques by which different aspects of a dataset can be emphasized. Clipping techniques can give insight into inner structures, while the outer shape is provided simultaneously as context. However, direct volume visualization is more expensive than surface-based volume rendering. But there exist several GPU-based rendering algorithms that can achieve interactive frame rates for average-sized medical volume datasets. Thus, direct volume visualization is nowadays applicable for medical practice.

Since direct volume visualization builds the base of the visualization algorithms and techniques presented in this thesis, this topic is further detailed in Section 2.4. In literature frequently the alternative term *direct volume rendering* can be found, which is often used exchangeable. In this thesis it is distinguished between direct volume rendering (DVR) as the technique of computing a 2D image directly from a 3D volume and direct volume visualization as the process of composing a meaningful visual representation of the information contained in a volume dataset by means of DVR. Often, direct volume rendering and direct volume visualization is shortly denoted as volume rendering and volume visualization (in contrast to surface-based rendering and surface-based visualization).

2.2.4 Visual Analysis

The last step of the medical visualization pipeline is the visual analysis and interpretation of the displayed data, for example to gain information about a disease or for the planning of a treatment. This task is mainly performed by the user, but it should be supported by the visualization application by means of intuitive interaction functionality. Ideally, the parameters of any pipeline step can be manipulated and the impact to the visualization result can be directly analyzed.

Concerning the rendering step, a medical visualization tool should allow the interactive manipulation of the camera position and orientation. By manipulation of the mapping step parameters the visual representation of a dataset can be influenced. Since many preprocessing and image analysis algorithms are semi-automatic, the requirement of interaction is immanent for this pipeline stage. Finally, the visualization result could give feedback for the adjustment of the parameters of the applied image acquisition techniques either for a repeated observation of the same patient or for future observations of other patients with similar indications.

This thesis lays its focus on the last two stages of the medical visualization pipeline – visualization and visual analysis – and the interaction loop between these two stages. The preceding steps of the pipeline build the basis of the presented visualization techniques but are not in the scope of this work.

2.3 Hardware-Accelerated Rendering

Modern computers are equipped with graphics cards, which are specialized hardware extensions for the generation and output of images to a display. While first generations were solely responsible for the mapping of text and graphics to the video output, a contemporary graphics card usually possesses a specialized processor for hardware-accelerated 3D rendering – a so called *graphics processing unit* (*GPU*) – and special memory for the storage of graphics data. In general, a GPU is a data-parallel streaming processor that is optimized for the parallel synthesization of a 2D image from a 3D scene. Originally, GPUs were designed for surface-based rendering, but they do now provide a flexible programming model that allows their application for other purposes. It is obvious to exploit these capabilities for interactive 3D visualization. For example, GPUs can be employed efficiently for direct volume rendering.

2.3.1 Rendering Pipeline

Starting point of the generation of a 2D raster image from a 3D scene by a GPU is the decomposition of complex 3D objects into geometrical primitives, namely points, lines, triangles, quads, and planar polygons. The primitives are defined by a set of vertices. These are sent to the GPU as an ordered input stream. Besides its position in 3D space a vertex can hold additional information, like the direction of the surface normal or a color. Additional data can be stored in so-called *textures* (data arrays), which are held in the GPU memory. The data elements of a texture are called *texels*. There exist 1D-, 2D-, and 3D-textures. A vertex is mapped to a data value in a texture via additionally attached *texture coordinates*. They are defined in a normalized texture space where a texture has edge length one in each direction. Sample values at texture coordinates between *texels* are computed by interpolation from neighboring data values.

The processing of the vertex input stream on the GPU is specified by the so-called *rendering pipeline* (see Figure 2.10). On an abstract level this pipeline is build of three major processing stages:

- **Geometry Processing** In the geometry processing stage the incoming vertices are first transformed from their local object space into the common world space (modeling matrix), then into the view space of the camera (viewing matrix) and finally into the normalized screen space (projection matrix). In the *primitive assembly step* the vertices are joined together to the geometric primitives that they originally formed. Then, in the *primitive processing step* the primitives are clipped against the view frustum, are transformed into the two-dimensional device space and are finally mapped to the viewport.

- **Fragment Processing** In the fragment processing stage the primitives are first rasterized into *fragments*, which correspond to pixels in the *frame buffer* (storage of the output image). Furthermore, the vertex attributes, e.g. texture coordinates, are interpolated. Then, per-fragment operations are performed which compute the final color from the interpolated vertex attributes and the corresponding texture samples.

- **Compositing** In the final compositing stage the fragments are written to the frame buffer. But first, several tests are performed that check if a fragment must be discarded, for example because of occlusion. If the fragment is not discarded, its color is combined with the color that is already stored at the corresponding raster position in the frame buffer. For this purpose, different *blending rules* can be applied.

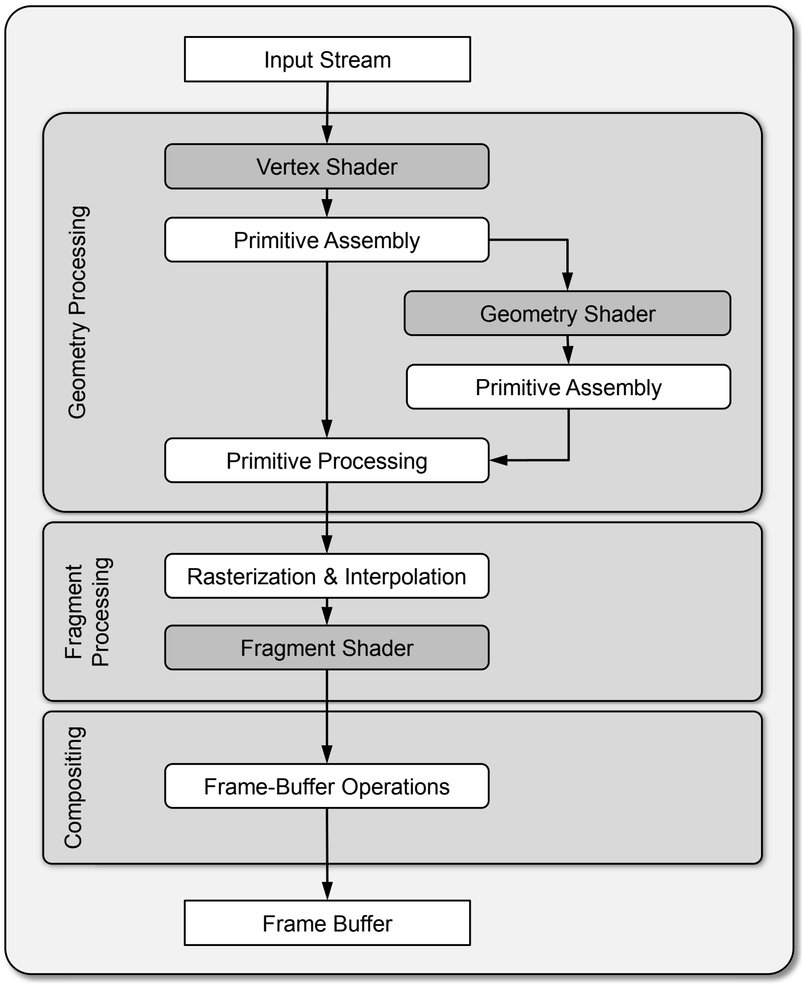


Figure 2.10: The programmable rendering pipeline. White boxes depict fixed functionality, the steps in the grey boxes are programmable by shader programs.

The first generation of GPUs implemented fixed-function rendering pipelines, which provided only small adaptability. However, modern GPUs provide the possibility to replace some of the pipeline steps by user written *shader programs*. Figure 2.10 shows the programmable graphics pipeline as it is approximately realized on current GPUs. There are three programmable shader units. A *vertex shader* replaces the vertex-processing step in the fixed function pipeline. Arbitrary per-vertex operations, like transformations, per-vertex lighting or the computation of texture coordinates, can be performed. A *geometry shader* introduces an additional (and optional) pipeline step after the primitive assembly. Here, a single incoming primitive can be replaced by several outgoing primitives, which for example refine the structure of a surface. Since the geometry shader outputs unconnected vertices, the primitive assembly step has to be performed again. Finally, a *fragment shader* provides the possibility of performing arbitrary operations on the rasterized fragments, like texturing or per-fragment lighting. Since current GPUs support loops and conditional jumps in a fragment shader, sophisticated shading operations can be realized.

There are two major *application programming interfaces* (APIs) for GPUs with associated shader programming languages: *OpenGL* [93] (open graphics library) with *GLSL* [113] (OpenGL shading language) and *DirectX* [85] with *HLSL* (high level shader language). While OpenGL is an open standard with implementations on several platforms, DirectX is developed by Microsoft and only available for Windows environments. For the implementation of the visualization techniques presented in this thesis solely OpenGL and GLSL was applied. Thus, the reader may sometimes find terminology which is specific for this technology. Nevertheless, the proposed solutions can be realized with any graphics API and shader programming language.

2.3.2 General Purpose Computation on GPUs

The enormous parallel computing capabilities of modern GPUs suggest to be exploited for non-graphics computations. In fact, an independent research discipline dealing with general purpose computation on GPUs (*GPGPU*) has been established [44]. Basically, any algorithm that fits to the SIMD (single instruction, multiple data) stream programming model can be adapted for computation by a GPU. Typical application fields of GPGPU are grid-based computational problems like fluid dynamics [48], image processing [58], or computer vision [39]. But GPUs have also been employed for solving systems of linear equations [69], the simulation of large astronomical n-body systems [97], and many other tasks. In Section 4.3 of this thesis a GPGPU algorithm is presented that exploits the GPU for interactive volume deformation.

To use a GPU for general purpose, the general programming model for stream processors has to be mapped to the concepts of GPU-based graphics programming. In general, a stream processor applies a series of operations (*kernel*) to all elements in a set of data (*stream*). On a GPU a stream corresponds to a texture (typically a 2D texture) and a *kernel* can be realized by a shader (typically a fragment shader). To initiate the processing of a kernel, the viewport is set to the size of the stream texture and a screen-filling quad is rendered. Then the elements of the stream are automatically processed in parallel.

For reading operations the stream texture can be bound like a normal texture. For writing operations it has to be bound as a render target. Inside a shader any texel of an input texture can be accessed, but the output element (the currently processed fragment) is determined in advance. Thus, data can be *gathered* from other data elements in a stream, but it is not possible to *scatter* computing results to other elements like the one currently processed. If intermediate computing results should be communicated to other elements in a stream, the data has to be written to the output element and then gathered by the other elements in a further render pass. Hence, most GPGPU algorithms employ several render passes in which they apply the same or different kernel shaders. Furthermore, it is not possible to simultaneously read from and write to a texture in a single render pass. For this reason, a processing scheme called *ping-pong* *rendering* has been established. Here, the stream texture is duplicated and one texture is bound as input and the other as output. After each rendering pass the two textures are interchanged to provide the output from the previous pass as input to the next pass.

Due to the success of GPGPU applications, several APIs have been developed that provide a more general access to the computing capabilites of a GPU than the classical Graphic APIs. E.g., the *Brook* library [16] realizes a general stream processing layer that covers the underlying graphics API. The *CUDA* [91] (compute device framework architecture) framework by NVIDIA provides a direct stream programming interface for GPUs, which is integrated into the C language. Similar ideas are followed by *OpenCL* [61] (open compute language) and Microsoft's DirectX 11 *Compute Shader* [9]. Finally, it can be said that GPUs more and more turn from specialized processors for graphics purposes to general coprocessors for parallel computation tasks.

2.4 Direct Volume Rendering

In Section 2.2.3 direct volume rendering was introduced as one of three common techniques for the visualization of a medical volume dataset. In contrast to indirect surface-based rendering, it does produce a 2D projection directly from the 3D volume without employing an intermediate geometrical representation. Direct volume rendering is nowadays regularly applied in medical volume visualization and builds the basic algorithm for the visualization techniques that are presented in this thesis. This section gives details about the concepts of direct volume rendering as far as they support the comprehension of the following chapters. It starts with the theoretical background, then covers practical aspects of the volume rendering pipeline, and finally describes the two most popular algorithms for GPU-based volume rendering, slice-based rendering and ray casting. For an in-depth overview of modern techniques for volume graphics and volume visualization the reader is referred to the book by Engel *et al.* [28].

2.4.1 Theoretical Background

In general, direct volume rendering generates 2D images from arbitrary 3D scalar fields. A 3D scalar field can be written as function that maps positions in 3D space to 1D scalar values:

$$\phi : \ \mathbb{R}^3 \to \mathbb{R}, \ \mathbf{x} \mapsto \phi(\mathbf{x}) \tag{2.11}$$

In the context of medical images ϕ is defined by a discret regular grid. At positions outside the boundaries of the volume data set (the *bounding box*) ϕ is mapped to zero; scalar values at positions between grid points are computed by interpolation from the scalar values at the surrounding grid points (see Section 2.4.2).

Direct volume rendering algorithms usually regard a volume as a distribution of gaseous particles. For image synthesis the transport of light along a *viewing ray* that passes the volume and then reaches the camera is modelled (see Figure 2.11). Thereby, three different types of interaction between the light and the participating media can be taken into account. First, the particles can partly absorb the incoming light; second, the passed particles can actively emit light; third, the light can be scattered at the participating media. While scattering may be incorporated for photo-realistic rendering, volume rendering for scientific purposes usually applies a simplified *emission-absorption model*.

Volume Rendering Integral

When the emission-absorption model is employed, the differential change of the light intensity I at a position s along a ray S is described by the following differential equation:

$$\frac{dI}{ds} = -\tau(s) \cdot I(s) + q(s). \tag{2.12}$$

$\tau(s)$ is the so-called *extinction coefficient*. It attenuates the incoming intensity $I(s)$ at position s. $q(s)$ is the so-called source term that gives the amount of light emitted at position s. Usually, $q(s)$ is substituted by $\tau(s) \cdot \tilde{q}(s)$. This takes into account that $q(s)$ depends on a normalized intensity $\tilde{q}(s)$ and the density of particles at position s, which is indirectly given by $\tau(s)$.

Equation 2.12 is the so-called *volume rendering equation* in its differential form. The formal solution of the differential equation yields the *volume rendering integral* [76]

$$I(s) = I_0 \cdot e^{-\int_{s_{start}}^{s} \tau(t)dt} + \int_{s_{start}}^{s} q(s') \cdot e^{-\int_{s'}^{s} \tau(t)dt} ds', \tag{2.13}$$

which gives the light intensity at an arbitrary position s along the ray. Here I_0 is the initial light intensity at the entry point s_{start} to the volume. The attenuation factors

$$t(x, s) = e^{-\int_{x}^{s} \tau(t)dt} \tag{2.14}$$

in equation 2.13 lie in the interval $[0 \ldots 1]$ and describe the *transparency* of the passed material between x and s.

Since a scalar field does not directly represent the properties $\tau(s)$ and $\tilde{q}(s)$, they have to be computed from the scalar value $\phi(s)$ at position s. Therefore, transfer functions $T_\tau(\cdot)$ and $T_{\tilde{q}}(\cdot)$ are applied that map scalar values to the respective optical properties. Thus, the extinction coefficient $\tau(s)$ and the source term $q(s)$ are obtained in the following way:

$$\begin{aligned} \tau(s) &= T_\tau(\phi(s)), \\ q(s) &= \tau(s) \cdot \tilde{q}(s) = T_\tau(\phi(s)) \cdot T_{\tilde{q}}(\phi(s)). \end{aligned} \tag{2.15}$$

Numerical Evaluation

To compute the light intensity I that reaches the camera from a ray S, the volume rendering integral has to be evaluated from the ray's entry point to the volume at position s_{start} to the exit point at position s_{end} (see Figure 2.11). In general, there is no

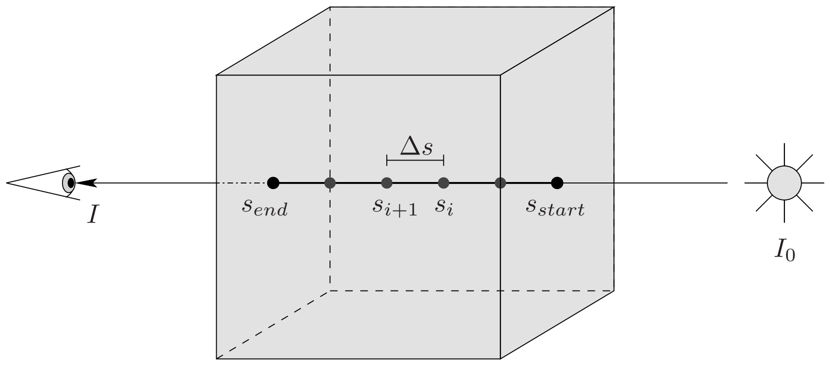


Figure 2.11: Principle of direct volume rendering: A viewing ray is traced on its way to the camera. While the ray passes the volume, the incoming light intensity I_0 is altered by emission and absorption, resulting in the final intensity I that reaches the camera. For numerical evaluation of the volume rendering integral the ray is sampled at equidistant positions between the entry point s_{start} and the exit point s_{end}.

analytical solution. Thus, the integral has to be approximated numerically. Therefore, the interval $[s_{start} \dots s_{end}]$ is divided into n segments of equal length $\Delta s = \frac{s_{start}-s_{end}}{n}$. Then the transparency $t(s_i, s_{end})$ of the material between the position $s_i = s_{start}+i\cdot\Delta s$ and the end position s_{end} can be approximated by means of the Riemann sum:

$$\begin{aligned} t(s_i, s_{end}) &= e^{-\int_{s_i}^{s_{end}} \tau(t)dt} \\ &\approx e^{-\sum_{k=i}^{n-1} \tau(s_k)\Delta s} = \prod_{k=i}^{n-1} e^{-\tau(s_k)\Delta s}. \end{aligned} \tag{2.16}$$

When approximating the volume rendering integral (see Equation 2.13) in a similar way, we get the approximated light intensity

$$\begin{aligned} I = I(s_{end}) &= I_0 t(s_{start}, s_{end}) + \int_{s_{start}}^{s_{end}} q(s')t(s', s_{end})ds' \\ &\approx I_0 \prod_{k=0}^{n-1} e^{-\tau(s_k)\Delta s} + \sum_{k=0}^{n-1} q(s_i)\Delta s \prod_{j=k+1}^{n-1} e^{-\tau(s_j)\Delta s}. \end{aligned} \tag{2.17}$$

Defining

$$t_i = e^{-\tau(s_i)\Delta s} \tag{2.18}$$

as the approximated transparency and

$$c_i = q(s_i)\Delta s = \tau(s_i)\tilde{q}(s_i)\Delta s \tag{2.19}$$

as the approximated light intensity of the i-th ray segment, Equation 2.17 can be rewritten to

$$\begin{aligned} I &\approx I_0 \prod_{k=0}^{n-1} t_k + \sum_{k=0}^{n-1} c_i \prod_{j=k+1}^{n-1} t_j \\ &= c_{n-1} + t_{n-1}\left(c_{n-2} + t_{n-2}\left(c_{n-3} + \cdots t_1\left(c_0 + t_0 I_0\right)\cdots\right)\right) \end{aligned} \qquad (2.20)$$

This leads to an iterative computation scheme of the final intensity $I = I_n$ starting with the initial intensity I_0:

$$I_i = c_{i-1} + t_{i-1} I_{i-1}. \qquad (2.21)$$

2.4.2 Volume-Rendering Pipeline

There are several different volume-rendering algorithms, which perform the numerical evaluation of the volume rendering integral in different ways. However, they all apply similar processing steps, which can be arranged in a general *volume rendering pipeline* (see Figure 2.12). Starting from the volume dataset, in the first *sampling stage* the volume is sampled at discrete positions and the corresponding sample values are determined. In the second *shading stage* the sample values are mapped to colors and further shading operations like illumination are applied. Finally, in the *compositing stage* the sample colors are accumulated along viewing rays to obtain the pixel colors of the output image. Usually, the three pipeline stages are not strictly separated but are repeatedly applied in a loop at different sampling positions, and the final pixel colors are iteratively accumulated. Since the sampling stage can be further subdivided into *data traversal* and *interpolation* and the shading stage comprises *classification* and *illumination*, the volume-rendering pipeline consists all in all of five steps, which are detailed in the following.

Data Traversal

Volume rendering algorithms primarily vary in the way they gather the volume samples. Basically, we distinguish between *image-order* algorithms and *object-order* algorithms. Image-order techniques iterate over the pixels of the final image and evaluate for each pixel the volume rendering integral by traversing the corresponding viewing ray, which starts from the camera, runs through the pixel and then passes the volume. Image-order techniques are usually referred to as *ray casting* [59; 74].

Object-order algorithms sample the volume cell-by-cell, compute for each cell the color contribution and project this color to the image plane. The most prominent object-order algorithms for regular grids are *slice-based volume rendering* [24; 144] and *volume splatting* [143].

In Section 2.4.3 the two most important techniques for GPU-based volume rendering, slice-based volume rendering and ray casting, are described in detail.

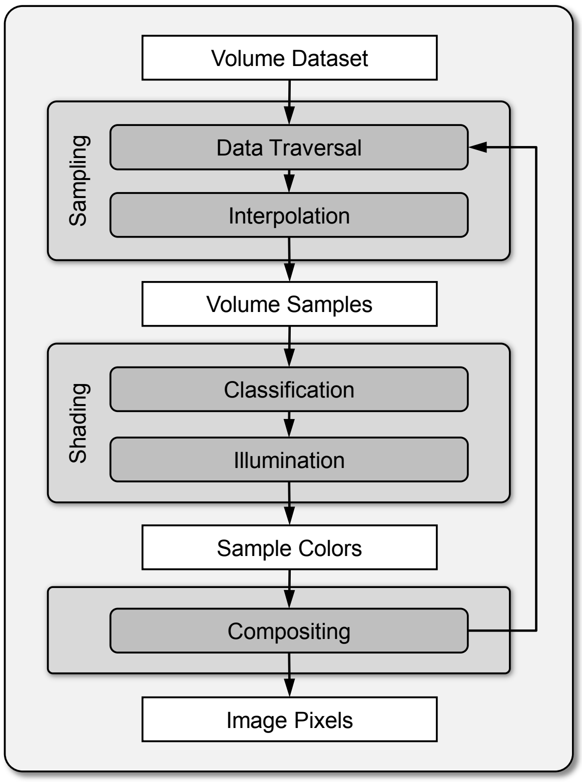


Figure 2.12: The volume rendering pipeline. It consists of three major stages (light gray) that convert a 3D volume dataset via several intermediate representations (white boxes) into a 2D image. The three stages are further subdivided into a total number of five pipeline steps (dark gray) that are repeatedly applied in a loop.

Interpolation

The traversed sample positions usually do not lie directly on a grid point of the sampled volume grid. Thus, the sample values have to be interpolated from the data values at the surrounding grid points. The simplest interpolation scheme is *nearest-neighbor interpolation*. Here, the grid point that is nearest to the current sampling point is evaluated, and the data value of this grid point is used as sample value. Nearest-neighbor interpolation produces discontinuities at the transitions between the influence areas of neighboring grid points. Since this leads to visible artifacts in the final image, normally interpolation methods are applied that take more than one grid point into account.

Trilinear interpolation computes a sample value from the eight grid points that form the border of the grid cell in which the sampling point lies. It is based on one-dimensional *linear interpolation* (see Figure 2.13 (a)) of scalar values along a straight line between two points $\mathbf{X_0}$ and $\mathbf{X_1}$. Here, the scalar value $\tilde{\phi}(\mathbf{X})$ for a point $\mathbf{X}$ between $\mathbf{X_0}$ and $\mathbf{X_1}$ is computed from the scalar values $\phi(\mathbf{X_0})$ and $\phi(\mathbf{X_1})$ at $\mathbf{X_0}$ and $\mathbf{X_1}$ by the

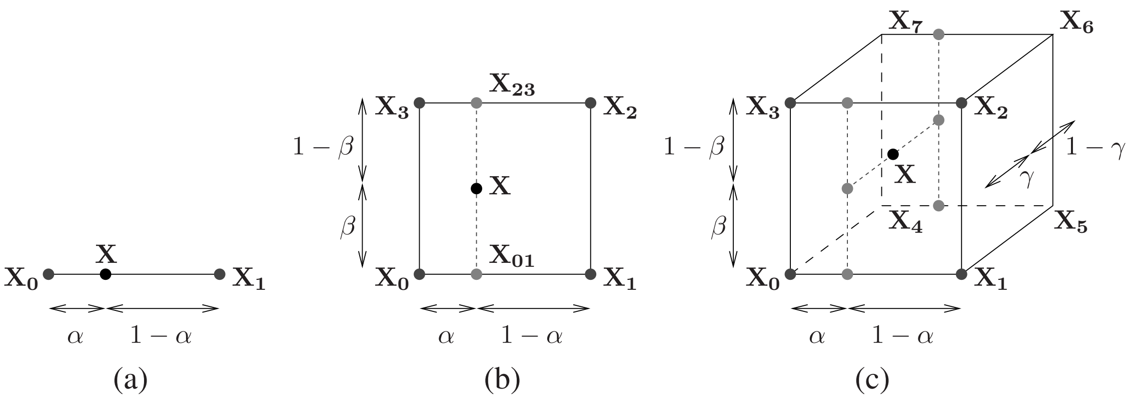


Figure 2.13: First-order interpolation: (a) 1D linear interpolation; (b) 2D bilinear interpolation; (c) 3D trilinear interpolation

following rule:

$$\tilde{\phi}(\mathbf{X}) = (1-\alpha)\phi(\mathbf{X_0}) + \alpha\phi(\mathbf{X_1}), \qquad \text{with } \alpha = \frac{|\mathbf{X}-\mathbf{X_0}|}{|\mathbf{X_1}-\mathbf{X_0}|}. \tag{2.22}$$

This means that the influence of the point $\mathbf{X_0}$ linearly decreases with its distance from $\mathbf{X}$. The same holds for $\mathbf{X_1}$.

One dimensional linear interpolation can be easily extended to two dimensional *bilinear interpolation* in a rectangle (see Figure 2.13 (b)). First, linear interpolation is applied for one dimension along two opposing edges and then the resulting scalar values are again linearly interpolated in the second direction. The extension to *trilinear interpolation* in cuboid cells is obvious (see Figure 2.13 (c)). After bilinear interpolation on two opposing cell faces the final scalar value is obtained by another linear interpolation in the third direction. Note that bilinear and trilinear interpolation are not linear with respect to there local coordinates any more.

There are higher order interpolation schemes that incorporate further grid points. Even though these techniques deliver a more accurate approximation of the underlying continuous scalar field, trilinear interpolation is the most frequently applied interpolation method. This is because of the fact that trilinear interpolation is natively supported by modern GPUs and that the costs for other methods are much higher.

Classification

In the classification step the scalar values at the sampling positions are mapped to the optical properties of the volume rendering integral (see Section 2.4.1). This can be used to distinguish different objects or materials in a volume dataset. Hence, classification can be exploited for implicit segmentation of interesting structures.

Transfer Functions Classification is usually based on transfer functions that give a global mapping of scalar values to optical properties. In Section 2.4.1 transfer functions $T_\tau(\cdot)$ and $T_{\tilde{q}}(\cdot)$ for the extinction coefficient τ and the normalized source term $\tilde{q}$ have been introduced. However, in practice the sample values are directly mapped to the optical properties that are applied in the discrete version of the volume rendering integral (see Equation 2.20), i.e. the approximated transparency t and the approximated light intensity c of a ray segment between two sample points (a so-called *slab*). More precisely, transfer functions $T_\alpha(\cdot)$ for the opacity

$$\alpha = 1 - t \tag{2.23}$$

and $T_c(\cdot)$ for the color intensity

$$\mathbf{c} = (r, g, b), \tag{2.24}$$

(a RGB color triple) of a ray segment are used. Please note that $\mathbf{c}$ represents a so-called *associated* color that already takes the extinction coefficient τ into account (see Equation 2.19). When a transfer function defines a mapping to a non-associated color $\tilde{\mathbf{c}}$, the corresponding associated color $\mathbf{c}$ can be approximated by multiplying $\tilde{\mathbf{c}}$ with the related opacity value α.

For many medical volume datasets the mapping of scalar values to optical properties does not permit a proper discrimination of different structures. For example, in CT datasets different soft tissue organs are displayed in similar ranges of Hounsfield units (see Table 2.1). To improve the discrimination of the different structures that are contained in a dataset, multi-dimensionsional transfer functions can be applied. Here, the dimension of the transfer function domain is extended by taking additional information like the gradient magnitude, second order derivatives and the curvature measures into account. E.g., Levoy [74] proposed to use the gradient magnitude for the distinction of object boundaries. Kniss *et al.* [63] take a second-order derivative measure as third dimension to further improve the boundary detection. Kindlmann *et al.* [62] use curvature information to emphasize the contours of volumetric stuctures.

Pre-integration The numerical evaluation scheme of the volume rendering integral presented in Section 2.4.1 approximates the underlying scalar field ϕ along a viewing ray S by a piecewise constant function (see Figure 2.14 (a)). To achieve a good approximation the sampling distance has to be chosen with respect to the maximum frequency (*Nyquist frequency*) that occurs in ϕ. Furthermore, the nonlinearity of the applied transfer function has to be taken into account since this additionally influences the effective *nyquist frequency* of the reconstructed signal. Thus, the sampling rate that is required to avoid undersampling of the volume rendering integral is usually much higher than the sampling rate that is needed for an adequate reconstruction of the volume data itself.

To overcome these drawbacks, Engel *et al.* [29] proposed *pre-integrated transfer functions* for which the scalar field ϕ is approximated by a piecewise linear function (see Figure 2.14 (b)). More precisely, the change of ϕ between two consecutive sampling points s_i and s_{i+1} along a ray is linearly reconstructed by

$$\bar{\phi}(l) = (1 - l)\phi(s_i) + l\phi(s_{i+1}). \tag{2.25}$$

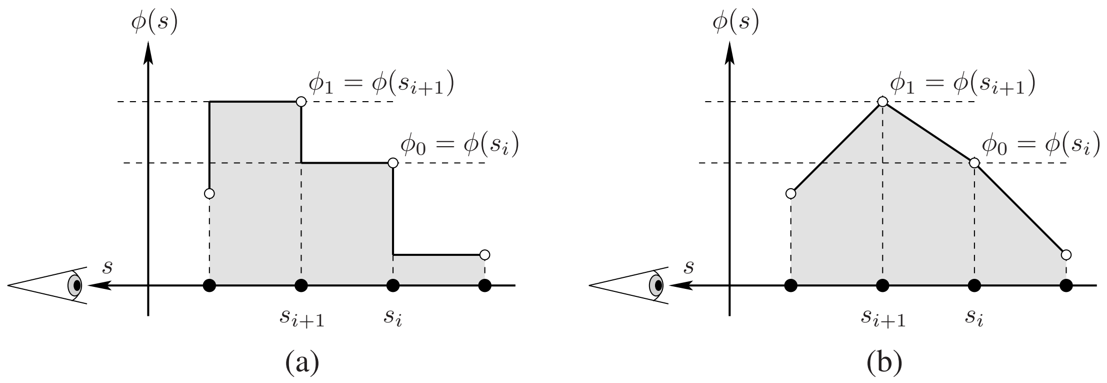


Figure 2.14: Approximation of the scalar field ϕ along a viewing ray S: (a) Piecewise-constant approximation; (b) Piecewise-linear approximation.

Hence, the opacity α_i of the slab between s_i and s_{i+1} can be approximated by

$$\begin{aligned} \alpha_i &= 1 - e^{-\int_{s_i}^{s_{i+1}} T_\tau(\phi(t))\mathrm{d}t} \\ &\approx 1 - e^{-\int_0^1 T_\tau(\bar{\phi}(l))d\mathrm{d}l} \end{aligned} \qquad (2.26)$$

and the color $\mathbf{c_i}$ by

$$\begin{aligned} \mathbf{c_i} &= \int_{s_i}^{s_{i+1}} T_{\mathbf{c}}(\phi(s'))e^{-\int_{s'}^{s_{i+1}} T_\tau(\phi(t))\mathrm{d}t}\mathrm{d}s' \\ &\approx \int_0^1 T_\tau(\bar{\phi}(l))T_{\tilde{\mathbf{c}}}(\bar{\phi}(l))e^{-\int_l^1 T_\tau(\phi(l'))d\mathrm{d}l'}d\mathrm{d}l. \end{aligned} \qquad (2.27)$$

α_i and $\mathbf{c_i}$ only depend on the scalar values $\phi_0 = \phi(s_i)$ and $\phi_1 = \phi(s_{i+1})$ at the start and the end point of a slab and the distance $d = s_{i+1} - s_i$ between these two points. Consequently, α_i and $\mathbf{c_i}$ can be precomputed from the transfer functions T_τ and $T_{\tilde{\mathbf{c}}}$ for discrete values of ϕ_0, ϕ_1, and d and stored in a three dimensional lookup table, which represents the pre-integrated transfer function. When a fixed sampling distance d along all viewing rays is chosen, the third parameter d can be eliminated, which leads to a much smaller two-dimensional lookup table.

In fact, pre-integration separates the sampling of the volume's scalar field and the sampling of the transfer function, which allows much lower sampling rates than needed for classical transfer functions.

Implicit Isosurfaces The classification step can be alternatively employed for the extraction of an implicit isosurface. Here, it is simply checked if a viewing ray is passing a specific isovalue between two consecutive sampling points s_i and s_{i+1}. In the positive case a predefined opacity α and a predefined color $\mathbf{c}$ is set as output. Otherwise the opacity is set to 0. This approach can be easily extended to the simultaneous extraction of multiple isosurfaces by performing the check repeatedly for several isovalues. More details about implicit isosurface rendering can be found in Section 4.1.3.

Illumination

In the illumination step local illumination effects are applied to the sample colors to emphasize the three-dimensional appearance of the visualized structures. Thereby, it is assumed that the light of a global light source impinges on a sample point without scattering or absorption along its way through the volume. The most popular local illumination model is the *Phong model* [95], which aims to create a realistic illumination lighting effects. However, there exist many illustrative shading techniques, like *tone shading* [43] or *cartoon shading* [70], that enhance the perception of an object by abstraction. An extensive overview about illustrative volume shading techniques can be found in [27] and [28]. In section Section 4.1.3 some illumination approaches for volume rendering are presented in detail.

Most local illumination models incorporate the normal direction of the illuminated surface. In volume shading the gradient of the underlying scalar field can serve as the normal vector because it is identical to the normal vector on the isosurface that passes the respective sampling point. The gradient at a point $\mathbf{x} = (x, y, z)^T$ in a (discrete) scalar field ϕ can be approximated by *finite differences*. One can, e.g., employ second order *central differences*

$$\nabla\phi(\mathbf{x}) = \begin{pmatrix} \partial\phi/\partial x \\ \partial\phi/\partial y \\ \partial\phi/\partial z \end{pmatrix} \approx \begin{pmatrix} \frac{\phi((x+d_x,y,z)^T)-\phi((x-d_x,y,z)^T)}{2d_x} \\ \frac{\phi((x,y+d_y,z)^T)-\phi((x,y-d_y,z)^T)}{2d_y} \\ \frac{\phi((x,y,z+d_z)^T)-\phi((x,y,z-d_z)^T)}{2d_z} \end{pmatrix}. \tag{2.28}$$

In regular grids d_x, d_y and d_z are usually set to the distances between the grid points in x-, y-, and z-direction. The gradients can either be precomputed at the grid points or computed on-the-fly at the sampling points. The first approach requires additional storage for the discrete gradient field, the second employs costly gradient approximation during rendering.

Compositing

The final step of the volume rendering pipeline is the compositing of the sample colors along a ray to a single pixel color. Basically, the iterative computation scheme proposed in Equation 2.21 can be applied. Mapping this scheme to an output color $\mathbf{C}$ and incorporating the opacity α_i of an array segment instead of the transparency t_i yields the iteration rule

$$\mathbf{C}_i = \mathbf{c}_{i-1} + (1 - \alpha_{i-1})\mathbf{C}_{i-1} \tag{2.29}$$

with $\mathbf{C}_0 = \mathbf{c}_{back}$ ($\mathbf{c}_{back}$ is the initial background color) and $\mathbf{C} = \mathbf{C}_n$. This scheme corresponds to a *back-to-front* traversal of the volume samples.

When the samples are traversed in *front-to-back* order starting from the camera, an adapted front-to-back iteration scheme

$$\begin{aligned} \mathbf{C}_{i-1} &= (1 - a_i)\mathbf{c}_{i-1} + \mathbf{C}_i, \\ a_{i-1} &= (1 - a_i)\alpha_{i-1} + a_i \end{aligned} \tag{2.30}$$

has to be applied, with $\mathbf{C}_n = 0$ and the accumulated opacity a initialized to a_n. The output color is $\mathbf{C} = (1 - a_0)\mathbf{c}_{back} + \mathbf{C}_0$.

The above compositing schemes create images in terms of the discrete volume rendering integral (see Equation 2.20). In those images important structures can be occluded by other structures that lie in front. Since this may obstruct medical diagnosis, alternative *order-independent* compositing rules have been developed. E.g., *pseudo X-ray* rendering builds a weighted sum (weighted with the opacities) of the sample colors along a viewing ray. When only gray values are used, this results in X-ray like images with which medical doctors are familiar. In *maximum intensity projection* (MIP) the color of the sample point with the highest intensity along a viewing ray is chosen as output color of the respective pixel. A typical application of MIP is the visualization of vessel structures (virtual angiography). The drawback of order independent compositing methods is the loss of depth information in the output images.

2.4.3 GPU-based Volume Rendering

Since the volume rendering pipeline can be processed independently for each viewing ray, it is obvious to perform this task in parallel and to accelerate it by the application of a GPU. However, current graphics APIs are designed for the rendering of surface meshes and do not directly support volume rendering operations. For this reason, GPU-based volume rendering techniques initialize the volume sampling by the rendering of some proxy geometry and employ GPU shaders for pipeline steps like classification and illumination.

A volume dataset is generally stored in a 3D texture for which modern GPUs natively support trilinear interpolation. Classical transfer functions are stored as look-up table in a 1D texture, which maps discrete sample values to four-component color vectors (three for the RGB color plus one for the opacity α). For pre-integrated transfer functions either 2D textures (fixed sampling distance) or 3D textures (variable sampling distance) are applied.

A single volume sample is processed by a fragment shader as follows. First, the sampling value at the current sampling position is looked up in the 3D volume texture (*interpolation*). Then, the related color and opacity is read from the transfer-function look-up table (*classification*) and optionally lighting or other shading operations are applied (*illumination*). The sample gradient is either looked up in another 3D texture that holds precomputed gradients or is computed on the fly. In the latter case, additional sample values at surrounding neighbor positions have to be read from the volume texture. When pre-integration is applied, two sample values are looked up, one at the current sampling position and one at the following sampling position along the viewing ray. The final *compositing* step is either realized in the fragment shader, or the GPU's compositing unit is utilized for this task.

There are two popular GPU-based volume rendering approaches that mainly differ in the way the volume data is traversed:

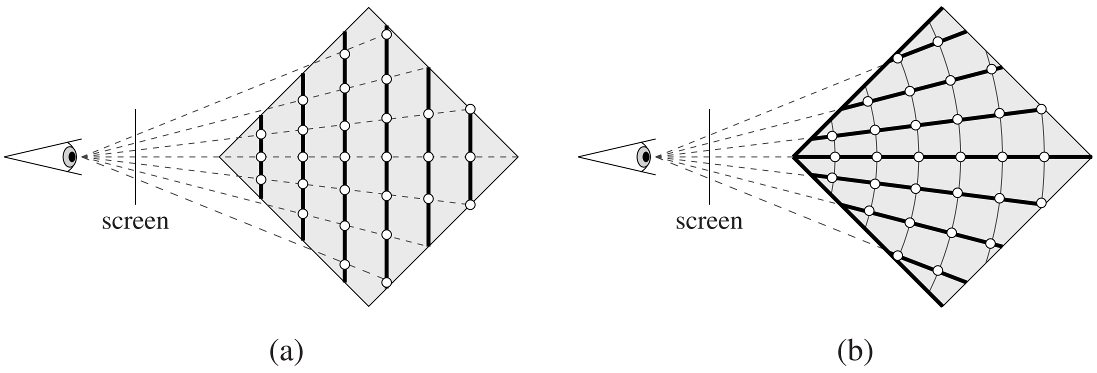


Figure 2.15: Two technique for GPU-based volume rendering: (a) Slice-based rendering and (b) GPU-based ray casting. The doted lines show the (imaginary) viewing rays, which traverse the gray volume. The thick black lines represent the (proxy) geometry, along which the volume is sampled. The white points depict the sampling points.

Slice-based Volume Rendering (Figure 2.15(a)) Slice-based volume rendering is an object-order volume rendering technique that was especially designed for the adoption of a GPU. Primarily, the cuboid bounding box of the volume dataset is cut into equidistant proxy slices (flat polygons) perpendicular to the current viewing direction (*view aligned*). Then, the volume is implicitly sampled by sequentially rendering these slices either in back-to-front or in front-to-back order. The proxy slices have to be recomputed each time the camera's position or orientation is changed or when the sampling rate is adapted. The related 3D texture coordinates of the volume texture are attached to the vertices of the proxy slices. These are automatically interpolated by the rasterization step of the rendering pipeline (see Section 2.3.1) and can then directly be used in the fragment shader for the lookup of the related sample value.

The compositing of the sample values is implicitly performed by the compositing unit of the GPU, which blends the colors of new fragments with the colors already written to the frame buffer. Therefore, the blending rule has to be adequately set depending on the rendering order of the proxy slices (see Equation 2.29 for back-to-front rendering and Equation 2.30 for front-to-back rendering).

GPU-based Ray Casting (Figure 2.15(b)) The image-order ray-casting algorithm is the most straight-forward way to numerical evaluate the volume rendering integral. Starting from the camera, for each image pixel a viewing ray is cast through the volume. Therefore, the entry point of the ray to and the exit point from the volume is computed and then the volume is sampled at equidistant positions between these two points. Since the ray traversal is started from the camera, the iterative front-to-back compositing scheme is applied (see Equation 2.30).

While ray casting was initially restricted to CPU-based implementations or special purpose hardware [80; 94], the flexible programmability of modern graphics hardware nowadays also permits interactive GPU-based implementations of this algorithm.

Here, the ray traversal is initialized by rendering the front faces of the volume's bounding box. This creates for each viewing ray a fragment at the related viewing rays entry point to the volume. Starting from these fragments, the corresponding ray can be cast through the volume. First GPU-implementations realized this ray traversal in multiple passes [68]. But current generations of GPUs support loops and dynamic branching, which allows the realization of the ray traversal within a single shader [123]. Therefore, a loop is implemented that samples the ray starting from the implicitly given entry point at equidistant sampling positions until the ray leaves the volume. Inside the loop the pipeline steps interpolation, classification, illumination, and compositing are explicitly applied to each sampling point. To avoid the expensive computation of texture coordinates at each sampling point, the ray can be cast directly in texture space.

Comparing the two rendering approaches, one can say that slice-based rendering used to show better performance results because of its better adaption to a GPUs classical rendering pipeline. However, ray casting in general creates better rendering results. The reason for this can be clearly seen in Figure 2.15. While ray casting uses a constant sampling distance for all viewing rays, the sampling distance slightly differs from ray to ray when slice-based rendering is applied. This leads to two problems. On the one hand, the volume may be undersampled along viewing rays that correspond to pixels at the image border. On the other hand, the transfer function defines the mapping of sample values to colors and opacity for a fixed sampling distance. The occurring rendering errors can be reduced by correcting the colors and opacities for each viewing ray or by using 3D look-up tables when pre-integration is applied, but this is often neglected due to the impact on the rendering performance and memory consumption.

Furthermore, ray casting easily allows the implementation of optimization strategies. E.g., the ray traversal can be stopped in advance (*early ray termination*) when the accumulated opacity of a pixel is nearly one since the color contributions of following samples will not be visible. If there are large connected regions that are visualized completely transparent, the ray traversal can jump over those regions without sampling (*empty space skipping*). Finally, the sampling rate could be reduced in areas where the volume signal shows only small changes (*adaptive sampling*). These optimization methods can be applied to partly compensate the performance drawback of the ray casting approach. Furthermore, it can be stated that the performance differences more and more vanish with the ongoing evolution of GPUs.

Chapter 3 Flexible Multi-Volume Rendering

In medical practice frequently a couple of different tomographic datasets of an antomical or pathological structure are observed simultaneously to achieve a better understanding of the addressed medical problem. In Section 2.2.2 three typical scenarios for the combined analysis of multiple volumes have been described. The datasets are either gathered at different points in time to control the progress of a disease or of a treatment; or they are taken with different imaging modalities to get a comprehensive view of a patient; or a patient specific dataset is compared with an atlas dataset, which represents the anatomical average of the investigated structures.

In the registration step of the medical visualization pipeline (see Section 2.2) the different datasets are mapped into a common coordinate system, which eases their direct comparison. Consequently, combined visualizations could be generated, in which the information of the different datasets is displayed together in a single image. However, the direct volume rendering techniques presented in Section 2.4 are restricted to the visualization of single volume datasets. Thus, to support the simultaneous rendering of multiple volumes (*multi-volume rendering*) the algorithms for single-volume rendering have to be adapted. The major difference is the fact that at a single sampling position the contributions of several volume datasets have to be taken into account.

The simultaneous rendering of multiple medical volume datasets is an ongoing research topic. E.g., Hastreiter and Ertl [50] presented a system for the combined visualization of two pre-registered CT and MR datasets of the human head for medical diagnosis; Beyer *et al.* [7] combined several sampling, shading and accumulation techniques for a specialized planning system for neurosurgical interventions; Jainek *et al.* [57] applied different illustrative volume shading techniques for the simultaneous visualization of anatomical and functional MRI data. Other work addresses the multi volume rendering problem in a more general way. For example, Cai and Sakas [18] and Ferre *et al.* [32] examined how the information of multiple volumetric datasets can be fused. Chen and Tucker [21], Nadeau [90], Grimm *et al.* [45], Lehmann *et al.* [72] and Plate *et. al* [96] proposed different concepts for the combined rendering of arbitrary volume datasets.

When several shading and illustration techniques shall be combined for the visualization of a single volume, similar algorithmic problems like for classical multi-volume rendering occur. The volume is virtually separated into multiple volumetric objects, which are shaded independently and then combined in a single image. A typical application is *focus+context rendering*. Here, a focus object is emphasized and surrounding structures are provided as context information. For example, Viola *et al.* [135], Krüger *et al.* [67], and Bruckner *et al.* [14] employ this concept for volume visualization. In [13] Bruckner and Gröller cut a volume with multiple clip planes into several parts and create so-called exploded views, like they are known from medical illustrations.

Alternatively, an explicitly generated segmentation mask (see Section 2.2.2) can be used to separate a volume dataset into several independent objects. For example, Tiede *et al.* [129] use an additional tagged volume to apply different colors to different anatomical structures. Vega *et al.* [134] employ a similar concept for the visualization of nerve fibers and vascular structures in the brain. Hauser *et al.* [51] and Hadwiger *et al.* [46] proposed a system where different transfer functions and shading styles can be applied to the different segmented objects.

Summarizing, it can be stated that there are many concepts and algorithms that target the combined rendering of multiple volumetric objects. However, the proposed techniques are usually designed for a certain (medical) application case or cover only a sub-domain of multi-volume rendering, like rendering of multi-modality image data, illustrative rendering or rendering of pre-segmented volumes.

In this chapter a flexible GPU-based multi volume rendering framework that abstracts from these differences and that can be employed for many application areas is presented. Thereby, the two major challenges of multi-volume rendering, the simultaneous sampling of multiple volumetric objects and the simultaneous shading of multiple volume samples, are addressed. In Section 3.2 it is shown how slice-based volume rendering and GPU-based ray casting can be extended for efficient rendering of multiple intersecting volumes, and in Section 3.3 a flexible shader-generation technique is introduced that dynamically generates optimized shaders for arbitrary multi-volume scenes. But first, in Section 3.1 a detailed introduction to the problem domain of multi-volume rendering is given. The presented concepts where first published in [107] and [108] in collaboration with Ralf P. Botchen from the Universität Stuttgart.

3.1 Introduction to Multi-Volume Rendering

When several intersecting volumetric objects shall be visualized in a single image, it has to be decided how the different objects contribute to the final image. For this multi-volume rendering problem Cai and Sakas [18] determined three levels of volume intermixing, which are involved into different stages of the volume rendering pipeline (see Figure 3.1). According to the original publication these intermixing strategies are called *image-level intermixing*, *accumulation-level intermixing*, and *illumination-level intermixing*.

For *image-level intermixing* (Figure 3.1 (a)) the volumes are rendered independently and the resulting images are intermixed on a per-pixel basis. The compositing of the pixels can either take only the color values into account or can be based additionally on opacities and/or depth values. For *accumulation-level intermixing* (Figure 3.1 (b)), the visual contributions of the volumes are intermixed per sample. At each sampling position along a viewing ray the sample values of the different volumes are first mapped independently to colors and opacities (*shading*). Then, these values are intermixed to a single sample color, which is finally accumulated along the viewing ray (*compositing*). In contrast to that, *illumination-level intermixing* (Figure 3.1 (c)) performs the intermixing before shading. Instead of accumulating independent sample colors and opacities the different volume samples are fused to a single merged sample and then a specialized multi-volume classification and illumination model is applied.

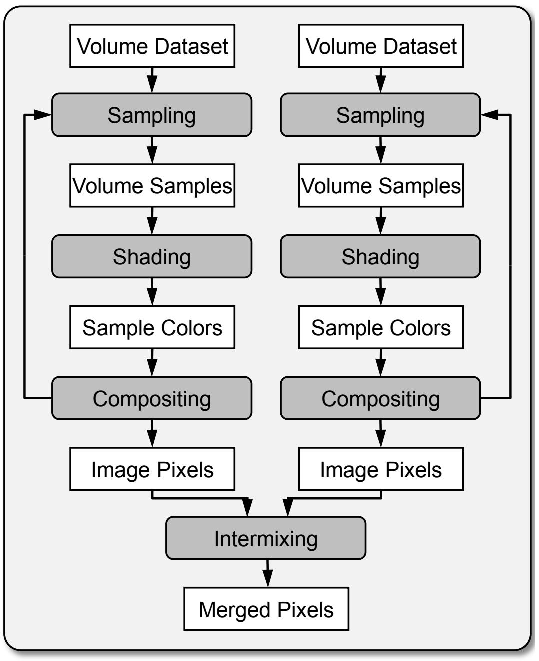
The advantage of the image level intermixing approach is its simplicity of implementation. The basic volume rendering algorithm does not have to be changed; just an additional image intermixing step has to be appended to the pipeline. However, it does not support correct depth cueing of the volumes, which may lead to confusing visual results.

Both, accumulation-level and image-level intermixing, overcome this drawback by performing volume intermixing on a per-sample base. Illumination-level intermixing allows the generation of physically inspired results. E.g., X-Ray like images can be generated by first accumulating the densities of the different volume samples and then mapping the fused densities to gray values. However, if the intermixed volumes originate from different imaging modalities, like CT and MRI, the scalar values of the volumes may have different physical sources and different ranges. Usually, they can not be intermixed directly but have to be weighted before. Thus, for each multi-modality configuration another multi-volume illumination model has to be applied. In contrast, accumulation level intermixing provides the possibility to apply independent transfer functions and shading styles to the different volumes. Thereby, different volumes of different imaging modalities can be handeled independently. Further, different illustrative and non-illustrative shading styles can be combined to generate comprehensive multi-volume visualizations. For these reasons, accumulation level intermixing is the most widely used intermixing approach for (medical) multi-volume rendering.

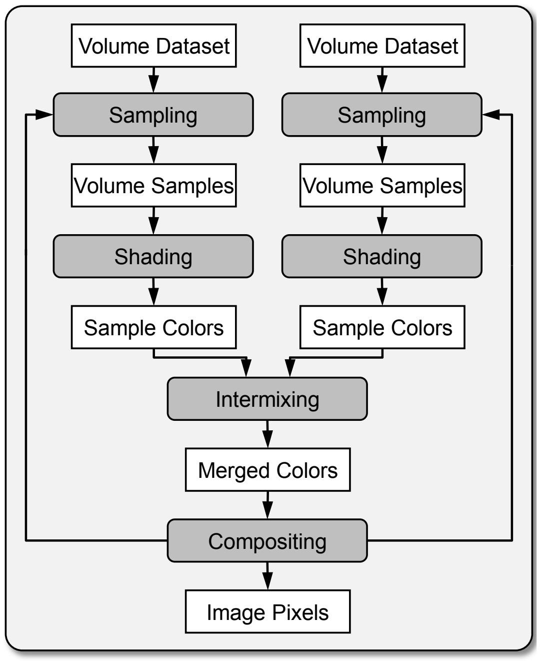
When accumulation-level intermixing is used, the multi-volume rendering problem can be divided into three sub-problems. First, a volume rendering algorithm has to be applied that can sample several intersecting volumes simultaneously. Second, for each combination of transfer functions and shading styles a specialized multi-volume shading procedure has to be provided. Finally, an intermixing rule has to be implemented that creates adequate visualization results for a certain application domain.

The intermixing of the different volume sample colors is often performed by standard alpha blending using the recursive *over operator* [45; 50]:

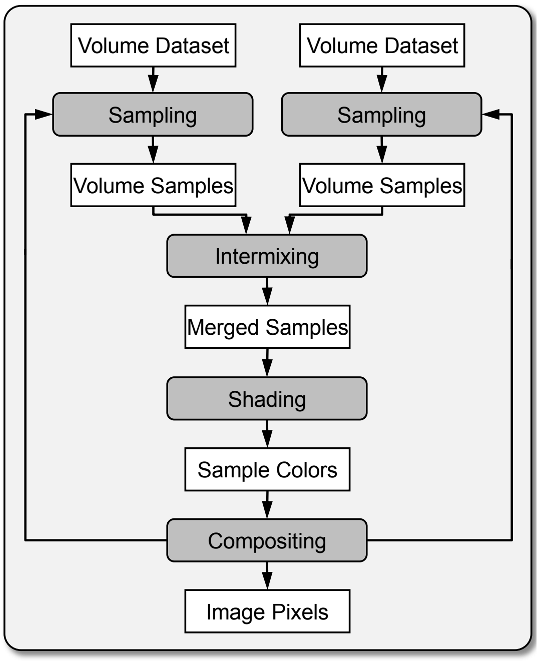
$$\begin{aligned} \alpha_{out} &= (1-\alpha_i)\alpha_{out} + \alpha_i \ , \\ \mathbf{c_{out}} &= (1-\alpha_i)\mathbf{c_{out}} + \mathbf{c_i} \ , \quad i = 1, \cdots, n \end{aligned} \tag{3.1}$$



(a)



(b)



(c)

Figure 3.1: Three multi volume rendering pipelines with different levels of intermixing: (a) Image-level intermixing; (b) Accumulation-level intermixing; (c) Illumination-level intermixing

$\mathbf{c_i}$ and α_i are the pre-multiplied color contribution and opacity of the i-th volume sample, and n is the total number of volumes. A problem of this operator is the fact that the resulting color and opacity depend on the order in which the volumes are applied. For this reason, other intermixing operators have been proposed that calculate a weighted sum of the single contributions. An example is the *inclusive opacity* operator of Cai and Sakas [18]. Here, the opacity α_{out} is computed similar like above, but the color values are weighted with their normalized opacity:

$$\mathbf{c_{out}} = \sum_{i=1}^{n} \frac{\alpha_i}{\alpha_{sum}} \mathbf{c_i} \ , \qquad \text{with } \alpha_{sum} = \sum_{i=1}^{n} \alpha_i . \tag{3.2}$$

Many other multi-volume intermixing schemes have been developed for specific application cases. E.g., Bruckner *et al.* [14] used fuzzy operators to control the contribution of different volumetric objects for illustrative rendering.

A special case is the rendering of segmented volumes. Here, a single volume is visualized in combination with a segmentation mask that gives for each voxel a unique ID. This ID determines the affiliation of the voxel to a specific region of the dataset, e.g. an anatomical or pathological structure. When the segmentation mask is used to apply different colors or illumination styles to the different objects [46; 51; 129; 134], the implicit intermixing rule is as follows:

$$\mathbf{c_{out}} = \sum_{i=1}^{n} \delta_{i,ID} \mathbf{c_i} \ , \qquad \text{with } \delta_{i,ID} = \begin{cases} 1, & ID = i \\ 0, & else \end{cases} \tag{3.3}$$

For multi-volume sampling and multi-volume shading often specialized techniques have been developed that are adapted to a specific medical scenario. Examples are the systems of Hastreiter and Ertl [50] or Beyer *et al.* [7]. Those systems usually provide a selected number of shading options for a fixed set of volumes. In contrast, general multi-volume rendering systems provide shading and sampling concepts that make them applicable for a wider range of applications. E.g., Nadeau [90] and Chen and Tucker [21] proposed different graph based techniques that both aim to compose complex scenes from several volumetric objects. Grimm *et al.* [45] presented a CPU-based ray casting approach that allows to combine different shading operations for a given scene of multiple volumes. Lehmann *et al.* [72] focused on the efficient rendering of multiple volumes on a GPU. Plate *et al.* [96] implemented a graphical GPU-shader composer for the easy manipulation of the display of a multi-volume scene.

The multi-volume rendering framework presented in this chapter follows a similar idea like Plate *et al.*. The visual representation of an arbitrary multi-volume scene is defined by an abstract render graph, which is used to generate optimized multi-volume shaders for different multi-volume rendering algorithms. In contrast to the shader composer of Plate *et al.* the render graph hides details about the underlying shading operations. This allows the creation of complex multi-volume visualizations without the need of deeper knowledge about rendering and GPUs.

3.2 GPU-based Rendering Techniques

GPU-based rendering of a single volume is based on the idea to render some proxy geometry for the initialization of sampling, shading and compositing (see Section 2.4.3). The shape of the proxy geometry depends on the applied rendering technique. For slice-based rendering the volume is cut into equidistant slices and the slices are rendered in back-to-front order. Here, sampling is implicitly realized by rasterization, shading is performed inside a shader and compositing is implicitly performed by the blending unit of the GPU. For GPU-based ray casting the front faces of the volumes are rendered to implicitly initialize the start positions of the viewing rays. Sampling, shading and compositing are completely performed inside a shader.

For multi-volume rendering at each sampling position not only a single sample value has to be looked up but one for each participating volume dataset. In case that all participating volumes have equal extent, position, and orientation, the same proxy geometry as for single volume rendering can be applied to all volumes. In more complex multi-volume scenes, in which the extent, position and/or orientation of the volumes differ, the single-volume proxy geometries do not match. Here, the proxy geometries have to be fused first to a combined multi-volume proxy geometry. Further, it should be taken into account that the bounding boxes of the volumes do not intersect at each sampling position. So, for efficiency reasons, it should be ensured that a volume is only accessed at sampling positions that lie inside the volume's bounding box. Two techniques for the fusion of proxy geometries, one for slice-based rendering and one for ray casting, are detailed in the following.

3.2.1 Slice-based Multi-Volume Rendering

Basically, slice-based multi-volume rendering is realized by slicing the volumes independently and merging the proxy slices to complex polygons that cover all volumes in the scene. Listing 3.1 describes this process schematically in pseudocode. To avoid inconsistencies, the volumes in the multi-volume scene are primarily transformed into camera space. Then, all volume bounding boxes are sliced equidistantly along the viewing direction in fixed distances from the camera position (see Figure 3.2). This leads to *multi-volume slices*, each containing coplanar proxy slices of the different volumes in the scene. The sampling distance between the multi-volume slices is chosen with respect to the volume with the smallest voxel dimensions. The multi-volume slices are rendered in back-to-front order with a specialized shader for multi-volume shading (see Listing 3.2). Inside this shader the fragment position, which is given in camera space, is transformed for each involved volume into its specific object space and then each volume is sampled and shaded independently. Finally, the different sample colors are intermixed to a single output color. This color is blended outside the shader with the content of the frame buffer. The fusion of the proxy slices that are contained in a multi-volume slice to combined proxy polygons can be realized in three different ways (see Figure 3.3):

```
void sliceBasedRenderingCPU() {
  list slices, multiVolSlices;
  for each volume in scene {                   // slice volumes independently
    slices.clear();                            //   and combine slices to a
    sliceVolume(volume, slices);               //   single list of
    combineWith(multiVolSlices, slices);       //   multi-volume slices
  }

  for each multiVolSlice in multiVolSlices       // render multi-volume slices
    switch intermixingMode                       // choose intermixing strategy
      case MERGE :                                 // MERGE
        slice mrgSlice = merge(multiVolSlice);
        activateShader(shadeMultiVolSliceGPU, mrgSlice.vols);
        renderSlice(mrgSlice);
      case SEPARATE :                              // SEPARATE
        for each slice in multiVolSlice {
          activateShader(shadeMultiVolSliceGPU, slice.vol);
          renderSlice(slice);
        }
      case INTERSECT :                             // INTERSECT
        list interSlices = intersect(multiVolSlice);
        for each slice in interSlices {
          activateShader(shadeMultiVolSliceGPU, slice.vols);
          renderSlice(slice);
        }
}
```

Listing 3.1: Pseudocode for slice-based multi-volume rendering on the CPU. According to the chosen multi-volume accumulation method different rendering strategies are applied. The *shadeMultiVolSliceGPU* shader is given in listing 3.2.

```
in:  vec4 fragPos; list vols;
out: vec4 outColor;

void shadeMultiVolSliceGPU() {
  vec4 volSamplePos; float volSample;
  vec4 volColors[vol.numOfVols];

  for each vol in vols {
// loop over multi slice volumes
    volSamplePos = vol.trafo * fragPos;                 // lookup samples and compute
    volSample = lookup(vol, volSamplePos);              //   colors for each volume
    volColors[vol.num] = shadeSample(vol, volSample);   //   independently
  }

  outColor = intermixColors(volColors);             // intermix sample colors
}
```

Listing 3.2: Pseudocode for shading of a multi-volume slice on the GPU. The samples of the different volumes are individually mapped to colors by *shadeSample* and finally accumulated by *intermixColors*.

Merge (Figure 3.3 (a)) This method merges the geometry of all proxy slices in a multi-volume slice into a single hull, which is not necessarily convex. This hull is tessellated and all triangles are rendered with a single shader that accumulates the contributions of all n volumes in a multi-volume scene, requiring no shader switches at all. In return, it suffers from the need to always collect the color

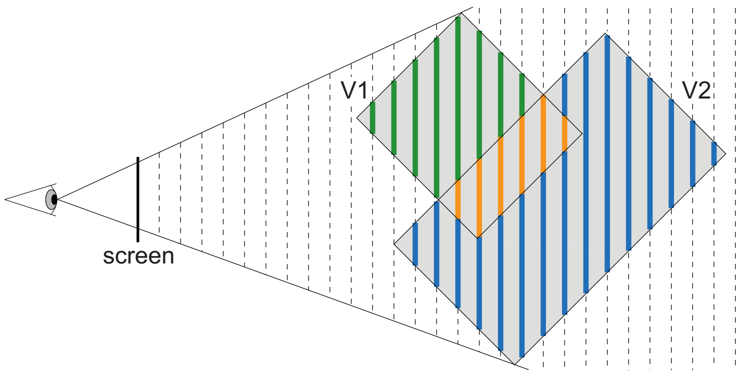


Figure 3.2: Slicing of a two-volume scene consisting of volume V1 and volume V2. The volumes are sliced at fixed distances from the camera postion. Corresponding slices of the different volumes are combined to multi-volume slices.

contributions of all n volumes, even for those fragments that are just covered by a subset of volumes. However, the merge technique provides full flexibility in choosing any kind of multi-volume intermixing function.

Separate (Figure 3.3 (b)) This technique is based on the strategy to handle each proxy slice separately. Here, no expensive merge of the volume slices is necessary. For each of the n volumes an individual shader is assembled, resulting in up to n shader switches per multi-volume slice. In contrast to the other techniques, the separate method directly blends the color contribution of a single volume sample into the framebuffer and exploits the fast hardware-supported blending operations. In return, it is limited to standard accumulation with the over operator (see Equation 3.1). The separate-shader-per-volume concept avoids the expensive branching inside a shader, but in exchange it leads to the necessity to evaluate multiple shaders in regions where volumes overlap.

Intersect (Figure 3.3 (c)) The third technique goes one step further by handling each possible combination of intersecting volumes with a different shader. This method divides the regions of intersecting volumes into single bounding polygons and tessellates them. Regarding the shader switches, the upper limit is 2^n due to the number of possible combinations of n overlapping volumes. Inside a shader up to n volumes are observed. Even though the complexity looks rather bad, the advantage of this method is, that each fragment is evaluated only once. Moreover, the multiple volumes are only processed when effectively needed and, thus, no unnecessary operations are performed. Similar to the merging technique, intersecting benefits from the possibility of using any intermixing function to merge the color contributions of the different volumes.

A further discussion of the advantages and disadvantages of the three accumulation techniques and comparative performance results for different multi-volume scenarios can be found in the next chapter in Section 4.1.

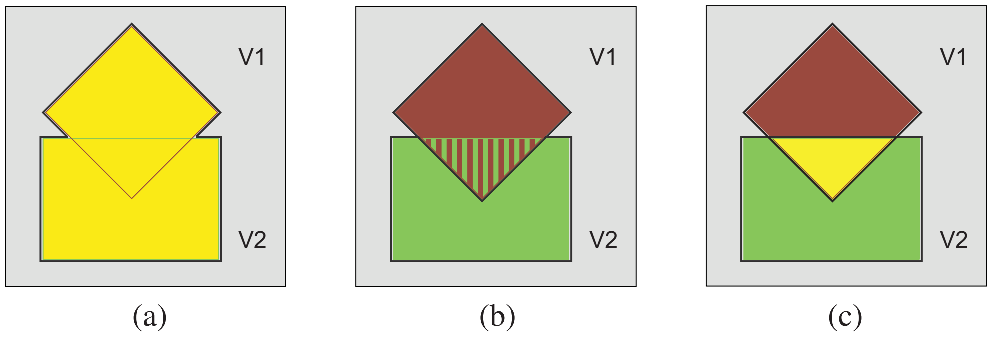


Figure 3.3: Three types of multi-volume slice fusion on the example of two overlapping proxy slices: (a) Merge; (b) Separate; (c) Intersect. Each differently colored region is processed by a separate shader. The gray square illustrates the intersection layer of the multi-volume slice.

3.2.2 Multi-Volume Ray Casting

The basic idea of GPU-based ray casting of a single volume (see Section 2.4.3) is to render the front faces of the volume's bounding box and then to cast the viewing ray for each rasterized fragment. Since modern GPUs provide loops and dynamic branching, the whole ray traversal can be realized in a single shader.

The obvious approach for raycasting of multiple volumes on the GPU extends the single pass method for single volumes to a three-pass rendering method that takes all volumes in a scene simultaneously into account (see Figure 3.4). First, the front faces of all volumes' bounding boxes are rendered with activated depth test and the depth function set to *"less than the current depth value"*. For each viewing ray, this yields the first entry point to a volume in the scene. The coordinates of these entry points are stored in a texture. In the second pass, the back faces of the bounding boxes are rendered with the depth function set to *"greater than the current depth value"*. This generates the exit points from the furthermost volumes along the rays. Given the entry and exit points for the union of all volumes, the whole multi-volume scene can now be rendered in a third pass by drawing a screen-filling quad that initializes the rays. The ray traversal is performed by a single fragment shader that reads for each pixel the pre-computed entry and exit points and traverses the viewing ray between these points in front-to-back order. At each sampling position along the ray the shader evaluates the color contributions of all volumes in the scene and intermixes them to a single sample color, which is blended to the ray's output color.

The major disadvantage of this approach is the fact that at each sampling position all volumes are evaluated, even if the sampling position lies outside a volume's bounding box. Alternatively, it can be tested if a sampling point lies inside a volume before the evaluation is done, but this introduces a huge number of expensive branching operations, especially for complex scenes.

An alternative multi-volume ray casting technique overcomes these drawbacks by first dividing the multi-volume scene into depth-ordered segments of intersecting volumes and then applying to each of these segments an optimized shader that only takes

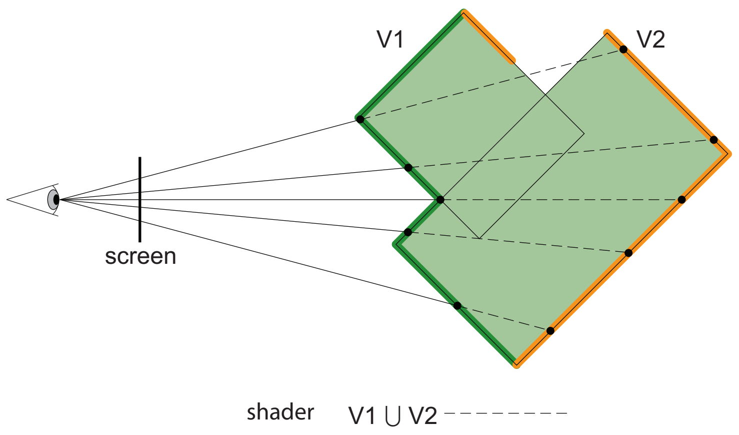


Figure 3.4: Multi-volume raycasting of a two-volume scene (*V1, V2*) with the obvious three-pass rendering approach: In the first and second pass the nearest (*dark green*) and the farthest (*orange*) intersection points of the viewing rays with *V1* and *V2* are computed. In the third pass the ray segments between foremost and furthermost intersection points are traversed by a single multi-volume shader responsible for both volumes.

the currently involved volumes into account. The segmentation of the scene is realized by depth peeling. Figure 3.5 and the pseudocode in Listings 3.3, 3.4 and 3.5 illustrate the concept of this approach.

The depth peeling starts (Listing 3.3, lines 2-6) with the computation of the rays' entry points similar to the above described three-pass method. Then, the volume bounding boxes are rendered once again (Listing 3.3, lines 12-16) with depth test set to *"less than the current depth value"* and a special shader applied (Listing 3.4, `computeNextLayerGPU`) that "peels away" the first entry points by comparing their stored z-values with the z-values of the currently rendered fragments. This generates for each viewing ray the second intersection point with a volume bounding box. The following intersection points are computed similarily by taking the previous intersection points as new start points and again applying the `computeNextLayerGPU` shader. The highest possible number of intersections per ray for n volumes is $2n$, since there can be at most n entry points and n exit points.

The depth peeling algorithm generates layers of ray segments that are banded by two consecutive intersection points, so each segment traverses a constant set of overlapping volumes. However, these volume sets can differ across a ray segment layer; Figure 3.5 shows this where the green layer consists of two disjoint regions of volume V1 and volume V2. This means that no single shader can render both volumes. Instead, the ray segment layer has to be further divided in screen space into regions of equal ray segments. To easily determine which volumes are intersected by a ray segment the integer arithmetic capabilities of NVIDIA's current GPUs (G8 series and newer) can be exploited. In the depth peeling step a bit vector is applied to each ray segment, which encodes the intersected volumes – i.e. the current permutation of affected volumes. Since 32-bit integer values are used, the total number of volumes in

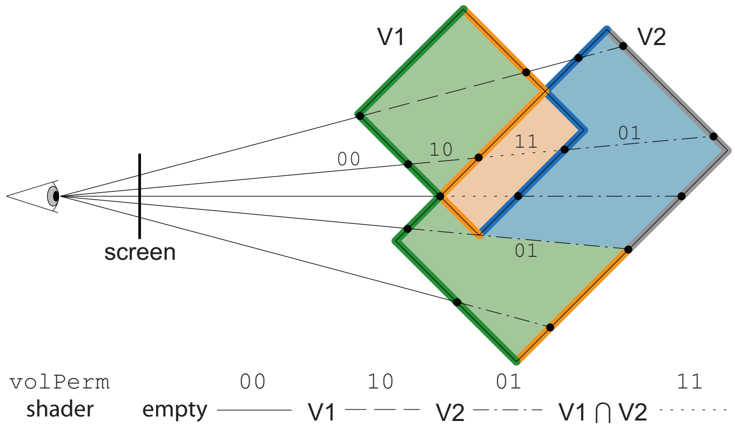


Figure 3.5: Multi-volume raycasting of a two-volume scene (*V1, V2*) with the optimized depth-peeling approach: The scene is iteratively segmented into three layers by depth peeling. Each layer is rendered with several optimized shaders that take only the currently covered volumes (*volPerm*) into account. E.g., the first layer (*light green*) consists of two disjoint regions of *V1* and *V2* which are handeled by two different shaders.

a scene is limited to 32, which is sufficient for common scenes. The volume permutations are stored in an integer texture which is initialized with 0. The permutation of the current ray segment layer is computed by incrementally changing the previous layer's permutation. For each bounding box the ID of the corresponding volume – encoded as bit vector `volBit` – is given as input to the depth peeling shader (Listing 3.3, lines 14-15).

If the currently rendered bounding box face is a front face a rendered fragment represents a point where the volume is entered, so the new volume permutation `curPerm` is computed from the previous permutation `prevPerm` by appending `volBit` with bitwise OR "|", (Listing 3.4, line 15). At back faces the viewing ray is leaving the volume. Here, the new volume ID is subtracted by merging the previous permutation with the bitwise complement "$\sim$" of `volBit` by bitwise AND "&", (Listing 3.4, line 16).

Basically, a ray segment layer is rendered several times by specialized shaders for each possible permutation of volumes (Listing 3.5). To avoid unnecessary computations for not affected ray segments, each shader first reads the ray segment's volume permutation and tests it against the permutation for which the shader was written. If they are not equal, the execution of the shader is discarded. Since all ray segments that pass the same overlapping volumes usually cover connected regions, and since dynamic branching is efficiently performed for coherent fragments on current GPUs, the overhead of these tests is relatively low. However, the number of shaders that have to be executed per layer is $2^n - 1$, which is the total number of permutations minus the zero permutation that covers no volume. This number is getting quite large even for small numbers of volumes. It can be reduced remarkably by exploiting the dependence of the volume permutations covered by the current ray segment layer on the permutations covered by the preceding layer.

```
void raycastingCPU() {
  activateShader(computeFirstLayerGPU);          // generate first depth layer
  for each volume in scene {
    volBit = 1 << volNum;
    renderFrontFaces(volBit);
  }

  list prevPermList, curPermList;                // init lists of permutations
  curPermList.add(0);

  while (layerNum < maxNumOfLayers) {            // loop over depth layers
    activateShader(computeNextLayerGPU);         // generate next depth layer
    for each volume scene {
      volBit = 1 << volNum;
      renderBoundingBox(volBit);
    }

    prevPermList = curPermList;                    // check permutations of
    curPermList.clear();                           //    previous layer
    for each prevPerm in prevPermList
      for each volPerm in singleBitFlip(prevPerm) {
        activateShader(raycastingGPU, volPerm);       // perform raycasting
        renderScreenFillingQuad();                    //    for volPerm
        if (anyFragmentWritten())
          curPermList.add(volPerm);
      }
  }
}
```

Listing 3.3: Pseudocode for the raycasting procedure on the CPU. The shaders *computeFirstLayerGPU* and *computeNextLayerGPU* are given in Listing 3.4. The shader *raycastingGPU* is presented in listing 3.5

```
in:  vec4 fragPos; uint volBit; bool frontFace;
out: vec4 pos; uint volPerm;

void computeFirstLayerGPU() {
  pos = fragPos; volPerm = volBit;             // write position and volume
}                                              //    bit vector

void computeNextLayerGPU() {
  vec4 prevPos; uint prevPerm, curPerm;        // read previous layer values
  readPreviousValues(prevPos, prevPerm);       //     from textures

  if (fragPos.z < prevPos.z)                   // discard fragments in front
    discard;                                   //    of previous layer

  if (frontFace) curPerm = prevPerm | volBit;  // compute current
  else curPerm = prevPerm & ~volBit;           //    permutation

  pos  = fragPos; volPerm = curPerm;
}
```

Listing 3.4: Pseudocode for the computation of ray segment layers on the GPU.

At the segment border of a single viewing ray the corresponding volume permutation changes only in a single bit because either a new volume is entered or an old one is left. For the whole segment layer this means that only those volume permutations have

```
in:  uint shaderPerm, float sampleDist;
out: vec4 outColor;

void raycastingGPU() {
  if (shaderPerm != getSegmentPerm())        // discard if permutation of shader
    discard;                                 //    and ray segment do not fit

  vec4 startPos, endPos;                     // compute sampling step along
  readStartAndEnd(startPos, endPos);         //     the ray
  vec4 step = norm(endPos - startPos) * sampleDist;

  vec4 pos = startPos;                       // start ray traversal
  vec4 sampleColor; outColor = vec4(0,0,0,0);
  while (pos.z < endPos.z) {
    vec4 volSamplePos; float volSample;
    vec4 volColors[vol.numOfVols];
    for each vol in shaderPerm {               // loop over volumes of shaderPerm
      volSamplePos = vol.trafo * fragPos;         // lookup samples and compute
      volSample = lookup(vol, volSamplePos);      //   colors for each volume
      volColors[vol.num] = shadeSample(vol, volSample); // independently
    }

    sampleColor = intermixColors(volColors);// accumulate colors
    outColor += (1.0 - outColor.a) * sampleColor; // blend them to outputColor

    pos += step;                                  // go to next sampling position
  }
}
```

Listing 3.5: Pseudocode for raycasting of a ray segment layer on the GPU. The `shadeSample` and `intermixColors` functions are similar to the functions that are used for shading and intermixing in slice-based multi-volume rendering (see Listing 3.2).

to be tested that can be generated from the permutations covered by the previous layer by single bit flips (Listing 3.3, lines 18-21). To determine which permutations have been covered by a ray segment layer, hardware supported occlusion queries are used. For each tested shader a query is started that returns whether any fragment was written to the frame buffer, which indicates if any ray segment has covered the corresponding permutation.

3.3 Dynamic Generation of Multi-Volume Shaders

In the previous section two algorithms for GPU-based multi-volume rendering have been introduced. Both techniques divide the applied proxy geometry into sections that cover a fixed subset of the volumes in the scene. To achieve the best possible performance, each of the proxy-geometry sections is rendered with an optimized shader that takes only the currently covered volumes into account. These shaders can all be written and optimized by hand, but this brings several problems and drawbacks with it. First of all, the implementation of the multi-volume shaders is a complex and time-consuming task. Furthermore, it has to be ensured that the shading results for a single volume does not differ between shaders for different volume subsets. This demands the development of many redundant code, which is prone to errors. Finally, for each combination of volumes and shading styles new shaders have to be implemented. This limits the generality of the rendering system and turns the design of new visualization solutions into a task for GPU experts only. The listed drawbacks can be overcome by the introduction of an abstraction layer that allows the definition of arbitrary multi-volume shading configurations on a level that is independent of the underlying rendering technique and graphics API. From this abstract definition optimized shaders for any volume subset can be automatically generated.

The encapsulation of low-level graphics APIs by abstraction layers that permit the access to GPU functionality in a more intuitive and task specific way is a common concept. In particular, for GPGPU computations (see Section 2.3.2) several approaches have been presented. E.g., Krüger and Westermann [69] implemented several linear algebra operators, which are evaluated on the GPU. The *Glift* library of Lefohn *et al.* [71] provides high-level data structures for GPUs. Buck *et al.* [16] developed *Brook*, a high-level API for general purpose stream computations on GPUs. Many other systems aim to ease the use of GPUs for classical graphics purposes. For example, Cook [22] developed a visual tree-based shader language called *Shade Trees*, which was primarily implemented by means of a graphical editor by Abram and Witted [1]. Goetz *et al.* [42] followed a similar idea and introduced a XML-based visual shading language. McCool *et al.* [78] presented a shader algebra with predefined operators to manipulate shader programs and, thus, to facilitate deferred shading. Trapp and Döllner [132] and Boyer [10] developed techniques that merge predefined shaders to a combined shader for complex shading operations. A flexible method to combine single functions into a composed GPU shader program was proposed by Folkegård *et al.* [33]. This technique dynamically creates shader programs by combining user-defined sections of code snippets to various shader algorithms. McGuire et al. [79] follow a similar approach but propose *abstract shade trees* for combining the shaders on a graphical level. The presented idea is based on the shade trees of Cook but automates the concatenation of input and output values of the tree elements.

In the following a dynamic shader generation concept is presented that was especially designed for the flexible configuration of multi-volume shaders. It is based on a similar idea like the *abstract shade trees* and uses also a graph representation, the so-called *render graph*, for shader modeling. The nodes of the render graph encapsulate – in contrast to other systems – high-level shading functionality, like direct volume rendering, illumination, or clipping. This allows the concentration on the visual output during shader design instead of being occupied with the combination of low-level shading operations. Based on the render graph individual shaders for different multi-volume rendering algorithms and different volume subsets are automatically generated. Currently, the system is designed for OpenGL and its shader language GLSL, but it can be easily ported to other graphics APIs.

3.3.1 The Render Graph Concept

The render graph allows the description of a complex multi-volume shading algorithm by the combination of several *render nodes*. Each render node describes a certain part of the shading process and the final GPU shaders are automatically generated based on the current graph configuration. Unlike a classical scene graph, which permits the creation and manipulation of complex scenes, the render graph describes the visualization of a given multi-volume scene on the level of the shading of a single multi-volume sample. The description is thereby independent of the finally applied volume rendering technique. There are three basic types of render nodes, which represent different stages of the shading process.

The Scene Node

The root of the entire render graph is always defined by a single *scene node*, which serves as interface between the external description of the scene objects (e.g. camera, light sources and volumes) and the graph itself. Therefore, the scene node collects the required information from these objects and passes it on to its children. To provide flexible access to arbitrary kinds of scene objects, the scene node does not perform this task by itself but delegates it to several sub-nodes. Each sub-node is responsible for a certain scene object. This allows, on the one hand, the easy integration of new scene objects by appending new sub-nodes. On the other hand, standard access strategies, e.g. to the volume data, can be replaced by more sophisticated algorithms for specific scenarios without affecting the handling of other scene objects.

Structural Nodes

Starting at the scene node, all volumes are initially treated equivalently regarding the shading process. To allow a separate handling of different volumes as a whole or just parts of them, *structural nodes* are introduced. These nodes do not directly contribute to the shading result, rather they provide capabilities to dynamically control the evaluation of the render graph by branching and manipulation. Three kinds of structural nodes are supported:

- **Splitter Node:** The *splitter node* is used to divide the handling of the volumes into several branches. Therefore, an arbitrary number of groups can be created. Each group contains one or more volumes. Moreover, a volume can be placed into several groups simultaneously. Every group results in a new branch of the render graph. Thus, it is possible to define different rendering styles for different volumes or to combine several rendering styles for a single volume. This can be considered as a branching on object level.

- **Conditional Node:** In contrast to the splitter node, a *conditional node* performs a subdivision of the volume objects themselves. This means that during the shading process only the branch is chosen for which the condition is true. These conditions are normally evaluated on the basis of the current sample position and can for example describe the selection of a segmented structure or the clipping against an implicitly given geometry, e.g. a plane.

- **Transformation Node:** To spatially separate whole volumes or previously subdivided parts of them, it is possible to insert a *transformation node* into the render graph. This node implements an affine transformation, which is applied to all volumes that are assigned to the current branch. Thereby, volume displacement can be realized.

Shader Nodes

The third kind of nodes are the *shader nodes*, which exclusively implement low-level shading operations to compute the resulting image. The shader nodes can be placed anywhere in the render graph, and several shader nodes can be cascaded on a path from the root down to a single leaf of the graph. In this case, a successor either overwrites or manipulates the result of a preceding shader node.

3.3.2 A Render Graph Example

The abstract functionality of the render graph and its nodes can be best clarified by detailed investigation of a structural example. Figure 3.6 (left) presents an exemplary render graph which is applied to a dual-volume scene. Render nodes are represented by grey boxes; the colored lines describe the paths of the volumes; the black arrows indicate the parent-child relationship of the render nodes. With volume *V1* as the hand dataset and volume *V2* as the bucky ball, the graph results in the image shown in Figure 3.6 (right).

The mapping of the render graph to its corresponding graphical output is started at the scene node. Here, the two volumes are attached and then their path leads through the graph in top-down manner. The first splitter node *Split V1/V2* divides the paths of the volumes into two branches. The hand volume takes the left branch and the bucky ball takes the right one. Volume *V1* passes another splitter node *Split V1/V1* which virtually splits the single path of the volume into two independent branches. Both branches work on the same hand volume but lead to different shader nodes, indicated by the continuous lines and the dashed lines. The *Skin Shader* node in the left branch is responsible for the semi-transparent iso-surface rendering of the skin, while the *Bone Shader* node in the right branch performs direct volume rendering of the bone structure. Both nodes are succeeded by illumination nodes, that manipulate the previously calculated colors with lighting computations.

Investigating the right branch of node *Split V1/V2*, volume *V2* encounters a conditional node *Condition V2*. This node splits the bucky ball into two halves using a

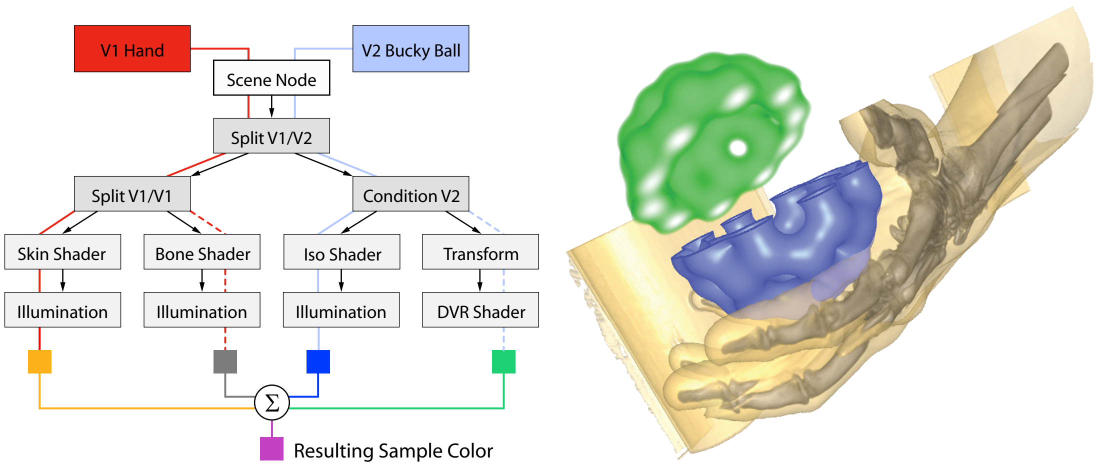


Figure 3.6: The abstract render graph on the left structure represents a scene of two volumes with different rendering styles applied. The resulting image is shown on the right.

clipping plane. The left branch passes an *Iso Shader* node, followed by an illumination node, resulting in a lighted iso-surface of the first half. The right branch runs into transformation node *Transform* that translates and rotates the other half, before the direct volume rendering node *DVR Shader* delivers the unlighted color for this path. Finally, the contributions of the different branches are mixed according to the defined intermixing operation.

3.3.3 Render Node Containers

The goal of dynamic shader generation is to convert the abstract representation of the render graph into a specially adapted GPU-based shader program that can be used for hardware accelerated multi-volume rendering. To support this process, each render node has to provide the information that is needed to perform its desired task. For this purpose, a render node builds a container that stores a set of output variables. These variables either act as input for succeeding render nodes or as final output value of the current volume sample. For each of the output variables the following information has to be provided:

1. *Name and type:* A unique name and a data type to permit correct access by other render nodes.
2. *Shader code part*: Predefined code that implements the computation of the output variables.
3. *Input variables*: Output variables of previous render nodes on which the computation of the output variable is based.
4. *Externals*: External parameters and textures that are needed for the output computation. They are passed to the shaders as uniform variables.

5. *Scope*: A variable can either be valid for the whole scene, for a certain transformation or for a specific volume.

Which output variables a render node provides, highly depends on its type. The scene node delivers all information of the given multi-volume scene to the other nodes of the graph. This is, e.g., the current camera matrix or the position of a light source. Additionally, it provides the current sample position and the volumes' scalar values, gradients and curvatures at this position. To facilitate complex shading algorithms like pre-integration or isosurface shading, these values are also provided for the succeeding sample position along the viewing ray.

A shader node generally computes the sample color for a single volume. There are two major types of shader nodes. Those that are computing the resulting color directly from the current volume sample, e.g. direct volume rendering or isosurface shading, or those that are manipulating the previously computed sample color to apply for example illumination or ghosting effects.

Structural render nodes do not directly contribute to the rendering result, which means that they usually do not provide any output variable that can be used by a succeeding node. Nevertheless, condition nodes have to provide a boolean condition variable for each outgoing branch that indicates if the related branch should be evaluated due to the applied condition.

By means of the render graph example presented in Section 3.3.2 (see Figure 3.6) the definition of output variables and their dependent components can be illustrated. Listing 3.6 gives pseudo code for the render node containers of the graph's branch that is detailed in the following. The *Illumination* node at the end of the left branch of condition node *Condition V2* provides the volume-specific *sample color* as output variable. The associated code part performs standard Phong illumination based on a previously computed *sample color*, the current *sample normal*, and the *light position* (input variables). No external parameters are needed.

The preceeding *Iso Shader* node serves the requested *sample color* and *sample normal* (output variables). The *sample color* depends on the *sample value* at the current and the following sampling position (input variables). If the chosen *iso value* (external) lies between these two values the *sample color* is set to the pre-defined *iso color* (external). The *sample normal* is linearly interpolated from the *sample gradient* at the current and the following sampling position.

Sample value, *sample gradient* and the *light position* are provided by the *scene* node. The *light position* is taken from an external uniform. The *sample value* is looked up in an external *volume texture*. The *sample gradient* is either computed on-the-fly or read from a pre-computed *gradient texture*. Both, *sample value* and *sample gradient* have volume scope and need the volume-specific *texture coordinate* as input. This is provided as an internal input variable. Finally, nearly all computations depend on the *sampling position* which is unique for the whole scene.

```

// *** scene node ***
input:    sampling position
external: volume texture V1
external: volume texture V2
external: light position
code:     compute texture coordinates for V1, V2 and V2 trafo
code:     look up sample values;
code:     compute gradients;
output:   sample value V1
output:   sample value V2
output:   sample value V2 next
output:   sample value V2 trafo
output:   sample gradient V1
output:   sample gradient V2
output:   sample gradient V2 next
output:   sample gradient V2 trafo
output:   light position

// *** left branch of Split V1/V2 ***

  ...

// *** right branch of Split V1/V2 ***

  // *** Condition V2 ***
  input:    sampling position
  external: plane parameters
  code:     evaluate plane equation for
            orginal and transformed sampling position;
  output:   plane condition
  output:   plane condition trafo

  // *** left branch of Conditon V2 (plane conditon is true) ***

    //  *** Iso Shader ***
    input:    sample value V2
    input:    sample value V2 next
    input:    sample gradient V2
    input:    sample gradient V2 next
    external: iso value
    external: iso color
    code:     check if iso surface is hit and set color respectively;
    code:     compute normal by interpolation of gradients;
    output:   sample color V2
    output:   sample normal V2

    //  *** Illumination ***
    input:    sample color V2
    input:    sample normal V2
    input:    light position
    code:     compute Phong illumination;
    output:   sample color V2

  // *** right branch of Conditon V2 (plane conditon trafo is true) ***

    ...
```

Listing 3.6: Pseudocode that partly represent the render node containers that are shown in the render graph example in Figure 3.6.

The boolean condition variables served by condition node *Condition V2* are determined by the standard *plane function*

$$f(x, y, z) = ax + by + cz + d, \tag{3.4}$$

which requires the *sampling position* and externally given *plane parameters* as input. The *plane function* divides the world space into two halves and the condition variables of the two outgoing branches are set accordingly to true and false. If the halves of the clipped volumes are transformed after clipping (like the upper half of the bucky ball in the render graph example), there are in fact two clip planes at different positions. To realize this, the clip plane function is evaluated twice, once for the original *sampling position* and once for the previously transformed position. So *plane function* and *condition variables* have different outcome for different transformations, which means that they have transformation scope.

3.3.4 Two-pass Shader Assembly

Based on the definition of output variables, the related shader code, and the dependencies on input variables, it is possible to generate a specific shader program for the computation of the final color of a multi-volume sample. Therefore, the shader generation process is divided into two passes (see Figure 3.7). The first pass evaluates the graph and determines all output variables that have to be computed for the requested sample color. This information is stored in the so-called *variable state*, which is a structural copy of the render graph that holds only the currently used variables and links to the original render graph nodes. In the second pass the pre-computed variable state is used to combine the associated shader code parts for the final shader program. The division of the shader assembly into two passes is done for three reasons:

1. While the computation of the variable state is independent of the applied rendering algorithm, the generation of shader programs can differ for different rendering techniques.

2. The generation of different shaders that are responsible for different combinations of volumes can be based on the same variable state, which has to be determined only once for a certain state of the render graph.

3. It is easier to optimize the generated shader code, if the output variables that are required by other render nodes are known in advance.

1st Pass

For the computation of the variable state the render graph is traversed recursively in depth-first order and for each render node an associated *variable state node* is created (see Listing 3.7). When a leaf node of the render graph is reached, it is tested if it can

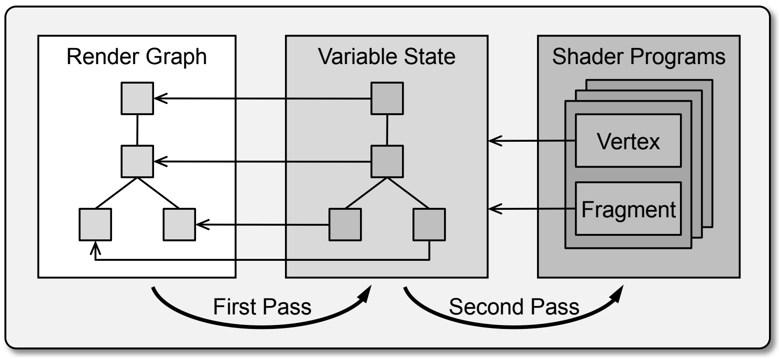


Figure 3.7: The two-pass shader generation algorithm. In the first pass the required variables are determined and saved in the variable state. In the second pass the shader programs are assembled from predefined code parts.

provide the sample color, and, if positive, this variable is stored in the related variable state node. Furthermore, the applied input variables are deposited in a list of required variables. On the way back to the root of the graph, this list is re-investigated for each passed node and variables that can be provided are replaced by their associated input variables. If the render graph configuration defines a valid shader, the list of required variables will be empty in the end.

Branches of the render graph – originating from condition and splitter nodes – are evaluated independently on the way down. At the backward traversal the different lists of required variables are merged to a single one. In addition, at splitter nodes it is determined which volumes are investigated at the different branches. Since the sample colors only have to be computed for still active volumes, this information is additionally stored in the related variable state at the leaf node and propagated to the required input variables. At conditional nodes for each outgoing branch the related conditional variable is added to the list of required variables and then processed just like the others.

A transformation node plays a special role in the variable gathering pass. All variables with volume or transformation scope that are computed on the succeeding branch have to be adjusted due to the defined transformation. The same has to be done for the required variables on the way up to the root. In addition, if the same volume is examined multiple times on different branches with different transformations, it effectively has to be rendered multiple times at different positions. To cope with this fact, all volumes that are currently active on a transformation node's branch are cloned, which means that the clones point to the original volumes and have additional transformation matrixes attached. Furthermore, the active volumes at the outgoing branch of a transformation node are replaced by their related clones. In the subsequent processing steps of shader generation and rendering, all volumes in the scene – originals and clones – are treated equivalently.

```
varList createVarState(renderNode, targetVar, vols) {
  stateNode = new VarStateNode(renderNode); // create a new variable state node

  switch renderNode.type              // handle render node types differently
    case SHADERNODE :                 // shader nodes
      if (renderNode.isLeafNode())
        return stateNode.checkVarForVols(targetVar, vols);
      else {
        reqVars = createVarState(renderNode.child, targetVar, vols);
        return stateNode.checkVars(reqVars);
      }
    case SPLITTER :                   // splitter nodes
      for each branch in renderNode.branches {
        tmpVars = createVarState(branch.child, targetVar, branch.vols);
        reqVars = merge(reqVars, tmpVars);
      }
      return reqVars;
    case CONDITION :                  // condition nodes
      for each cond in renderNode.conditions {
        tmpVars = createVarState(cond.child, targetVar, vols);
        reqVars = merge(reqVars, tmpVars);
        tmpVars = stateNode.addCondVar(cond);
        reqVars = merge(reqVars, tmpVars);
      }
      return reqVars;
    case TRANSFORMATION :             // transformation nodes
      newVols = clone(vols);
      return createVarState(rendeNode.child, targetVar, newVols)
}
```

Listing 3.7: Pseudocode for variable state computation in the first shader assembly pass. The render graph is traversed in depth-first order by recursive invocation of `createVarState`. The processing differs for the differnt types of render nodes.

2nd Pass

The code generation pass produces shader programs – consisting of a vertex shader and a fragment shader – that are responsible for a certain subset of volumes and that are specially adapted to the applied rendering algorithm. For slice-based multi-volume rendering and multi-volume ray casting fragment shaders are generated that basically look like the shaders presented in Listing 3.2 and Listing 3.5 in Section 3.2. They differ significantly because the shader for slice-based rendering is just responsible for the shading of a single multi-volume sample, while the ray-casting shader performs a complete traversal of a ray segment. Nevertheless, the shading and intermixing operations for a single multi-volume sample are similar and can be treated equivalently for both techniques. Both times, the generic shading loop over all participating volumes is replaced by specific code that avoids expensive branching and looping. The same is done for the intermixing of the single sample colors.

To assemble the multi-volume shading code for a given set of volumes, the precomputed variable state is traversed in depth-first order. At each variable state node the shader code parts that are associated with the stored variables are taken and added to the shader. If a variable has either transformation scope or volume scope, its code segment is defined only once by the render node but is appended to the shader several times for each requested transformation and/or volume respectively. To ensure the distinction of the different computations, the variable names are additionally extended by a unique per-volume postfix.

If there are branches in the render graph, the shader code is assembled independently for each branch and finally combined. In order to avoid unnecessary computations, the code parts of variables of previous render nodes are added to the shader as late as possible. If a variable is used in all outgoing branches of a structural node, it is placed before branching, but if it is only needed for a single branch, it is computed inside exclusively. Conditional branches are evaluated, if the associated conditions – represented by boolean condition variables – are satisfied. This is realized by nesting the branches inside *if*-statements. If a transformation node is placed somewhere below a conditional branch and the branching condition depends on the transformation, for each leaf of the outgoing subtree, the condition has to be evaluated as an independent if-branch in the shader code.

After the traversal of the variable state graph, code for the intermixing of the color contributions (see Section 3.1) of the different volumes at the different branches is added to compute the final output color.

Basically, the assembled multi-volume shading code starts the computation of the output color for each multi-volume sample from the current sampling position. This includes, e.g., expensive matrix multiplications that transform the sampling position from camera space into world, object and texture space. But the fact that these computations are affine transformations allows avoiding these costs by linearly interpolating the results from previously computed sample values instead. This optimization is not restricted to affine transformations, but can be exploited for all linear-interpolatable functions, e.g. the plane function presented in Equation 3.4.

For slice-based rendering the automatic linear interpolation of varyings between vertex and fragment shader can be utilized. Therefore, linear interpolatable output variables have to be additionally labeled and their computation is placed inside the vertex shader. If a non-interpolatable output variable from the fragment shader directly depends on an interpolatable variable from the vertex shader, additional code is added to both shaders, which delivers the input variable to the fragment shader by a varying. However, the number of varying components that can be used in a single GPU program is limited and depends on the used graphics hardware. To handle this restriction the number of potential varying components is counted before assembling the shaders. If the hardware limit is exceeded the demand of varyings is reduced to the allowed maximum by placing the computation of some of the interpolatable variables in the fragment shader instead.

The fact that ray casting does perform the whole ray traversal for a ray segment inside a single fragment shader (see Section 3.2.2) avoids the direct exploitation of the vertex shader for optimizations. Nevertheless, ray casting can as well benefit from interpolation. Therefore, an initialization step before the ray traversal loop is introduced. Here, the values of interpolatable variables are pre-computed for the first and the second sampling point along the ray segment and the sample-to-sample step size is calculated by subtracting the two values from each other. Inside the loop the step size is used to generate a new variable value from the previous one by incrementation.

A further potential for optimization is given by the fact that several volume shading algorithms, like pre-integration or implicit iso-surface rendering, are not calculating a color for a single sample position but for the whole slab between two samples. This means that some variables have to be computed for the current sampling position (*front sample*) as well as for the following one (*back sample*). Instead of re-computing both values for each sampling step, it is sufficient just to compute the new value for the current back sample and to copy the value for the front sample from the back sample of the previous step. In addition, the front value has to be initialized before the ray traversal.

3.3.5 Rendering

Since the described two-pass shader assembly technique allows the generation of specific shader programs for any subset of volumes, it can be directly applied to the two multi-volume rendering approaches presented in Section 3.2. The CPU-hosted rendering procedures for slice-based multi-volume rendering (Listing 3.1) and multi-volume ray casting (Listing 3.3) require only light adaptions. The computation of the variable state (1st pass) has to be performed once before rendering and needs recomputation only if the render graph configuration has changed. The algorithm-specific shader programs for certain combinations of volumes are generated on the fly (2nd pass) the first time they are activated during the rendering process. They are stored for later reuse by other proxy slices in case of slice-based rendering or other ray segment layers in case of ray casting. Like the variable state the shader programs have only to be regenerated after changes in the render graph.

In contrast, external parameters, such as the current camera position or transfer function settings, are subject to change between consecutive frames. For this reason, each render node provides an individual preparation method that realizes its specific setups of uniform variables, textures and other hardware resources. These preparation methods are called once per frame for each used shader program.

3.4 Conclusion

In this chapter a flexible framework for the simultaneous rendering of multiple volume datasets has been presented. In relation to the medical visualization pipeline introduced in Section 2.2 this framework covers the visualization stage. Here, the dynamic shader generation technique presented in Section 3.3 represents the mapping step. Via the render graph various visualization rules for a given multi-volume scene can be defined on an abstract level. From this abstract render graph optimized GPU shader programs are generated, that can be adapted for different rendering approaches. In Section 3.2 two techniques for multi-volume rendering have been presented, that can benefit from the shader generation concept: slice-based multi-volume rendering and multi-volume raycasting. Both subdivide the multi-volume scene into areas of overlapping volumes and apply shaders that only take the currently covered volumes into account. This avoids unnecessary sampling and shading of not affected volume datasets.

The multi-volume rendering framework distinguishes between the sampling step and the shading step of the multi-volume rendering pipeline with accumulation-level intermixing (see Figure 3.1 (b)). While the render graph is responsible for multi-volume shading, the rendering techniques described in Section 3.2 are responsible for multi-volume sampling. The two-pass shader assembly (see Section 3.3.4) concatenates these two steps to an integrated multi-volume rendering algorithm. The separation of sampling and shading allows for the application of new sampling approaches without the need to adapt the modules of the render graph. This flexibility makes the framework utilizable for a great variety of visualization tasks. Some medical applications are presented in the next chapters.

CHAPTER 4 INTERACTIVE MEDICAL VOLUME VISUALIZATION

In the previous chapter a flexible multi-volume rendering framework was presented that is applicable for a wide range of medical visualization problems. But while the framework builds the technical basis for the creation of high-quality volume-rendered images, interactive medical visualization applications have to take additional aspects, like the support of visual analysis and interaction through the user interface, into account. In this chapter it is demonstrated how the multi-volume rendering framework can be employed for interactive visualization in different medical application fields. Thereby, the focus is laid on the last two stages of the medical visualization pipeline – visualization and visual analysis – and the interactive control of the visual output by the user, as it is illustrated in Figure 4.1. Primarily (Section 4.1), a generic multi-volume visualization tool is introduced that provides a direct manipulation of the render graph via the user interface. Then (Section 4.2), various visualization concepts for the simultaneous visualization of functional and anatomical MRI datasets are presented. Finally (Section 4.3), an algorithm for the interactive deformation of volume datasets is described that can be integrated seamlessly into the volume-visualization process.

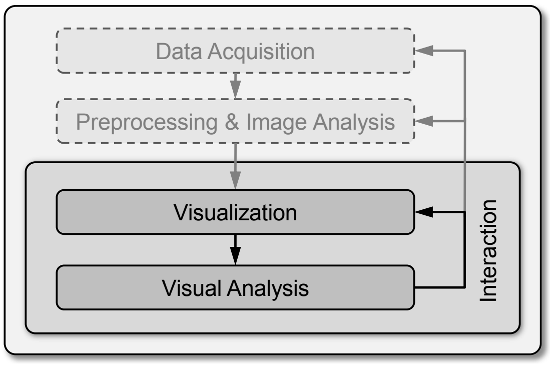


Figure 4.1: This chapter focuses on the visualization and the visual analysis stage of the medical visualization pipeline. The interactive manipulation of the visual output is thereby an integral element of interactive volume visualization applications.

4.1 Generic Multi-Volume Visualization

The multi-volume rendering framework from Chapter 3 provides a high degree of flexibility. The render graph with its freely combinable render nodes allows the generation of visualizations for new application scenarios without the need to care about the underlying rendering and shading techniques. Thus, it is obvious not to restrict this flexibility to an application programmer only, but to directly provide it to an experienced user of a visualization application. In this section a visualization tool is presented that was especially designed for the task of generic multi-volume visualization. A medical visualization expert can directly manipulate the render graph on a graphical level and gets direct feedback about the visualization result. Furthermore, it is easily possible to integrate new render nodes and to combine them with existing ones. This permits the fast implementation of new shading techniques and allows studying their applicability to certain medical visualization problems.

Several other tools and frameworks have been presented that provide application-independent (multi-)volume visualization functionality. Hadwiger *et al.* [46] introduced a framework for flexible visualization of segmented volume data. It allows the combination of different shading techniques, transfer functions and compositing modes for explicitly segmented objects of a single volume. Krüger *et al.* [67] presented *ClearView*, a volume-visualization tool that provides intuitive focus+context visualizations for arbitrary datasets. *VolumeShop*, developed by Bruckner *et al.* [14], is an interactive system for direct volume illustration. A user can interactively extract focus objects from a volume dataset, manipulate their appearances, and attach annotations. Plate *et al.* [96] presented a multi-volume shader framework for arbitrarily intersecting datasets. Multiple volumes can be combined with so-called lenses to complex multi-volume scenes. The multi-volume shaders can be interactively defined by the user via a graphical shader composer.

All these solutions provide a certain visualization metaphor to the user and most are restricted to a fixed configuration of datasets. In contrast, the visualization tool that is presented here can be used for several different application cases, like multi-modality rendering, rendering of segmented datasets and/or illustrative volume rendering, and those techniques can even be combined into a single image. The presented work was partly published in [107] and [108] in collaboration with Ralf P. Botchen from the Universität Stuttgart.

4.1.1 GUI Design and Interaction

The main objective of the generic multi-volume visualization tool is to provide the flexibility of the render graph directly to the user. For this purpose, the graphical user interface (GUI) consists of three views (see Figure 4.2). The main *render view* shows the visualization results due to the current scene and render graph configuration and permits interactive manipulation of the camera position with the mouse. The two other views are placed above each other on the left hand side of the render view. The lower

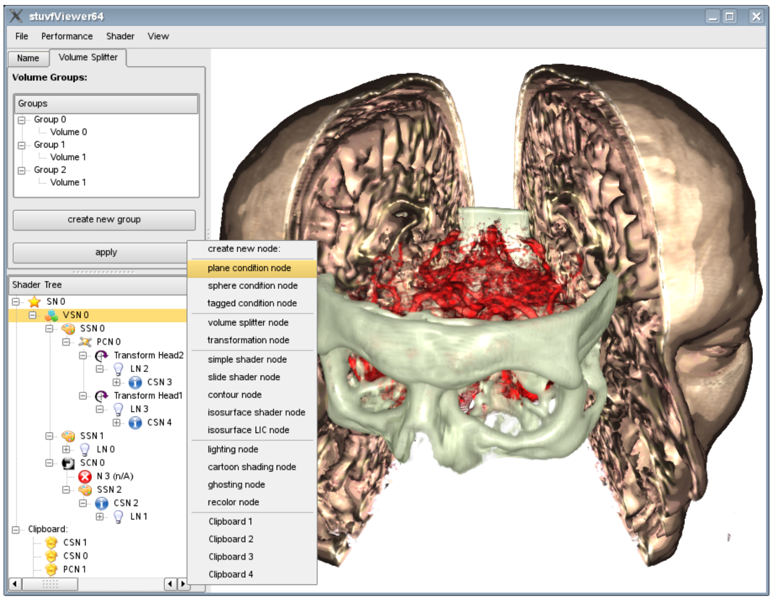


Figure 4.2: A screenshot of the generic multi-volume visualization tool. The main render view on the right shows the visualization result for the render graph configured in the render graph view (left bottom). Via context menu new render nodes can be inserted. The render node view (left top) provides an individual dialog for the manipulation of the parameters of the currently selected render node.

render graph view presents the render graph as a hierarchical tree. The graph in a whole can be manipulated by appending and deleting nodes. Deleted nodes are stored in a clipboard and can be re-inserted later. This facilitates the user to re-configure the render graph at higher levels without loosing already arranged branches.

If a node of the render graph is selected, the above *render node view* shows an individual dialog for the manipulation of the nodes' individual parameters. This may contain a graphical transfer function editor or controls for the adjustment of the position and orientation of a clip plane. Changes in the node view are either directly mapped to the underlying render node or have to be explicitly applied by the user. The second case avoids potentially costly re-rendering during the interactive manipulation of parameters. After the changed parameters have been applied, the effects to the visualization are shown immediately in the render view.

The configuration of the render graph can be stored persistently in a file for later reconstruction. This permits, on the one hand, the easy reproduction of earlier generated visualizations. On the other hand, a once generated render graph can be applied to other datasets from the same application field. This provides for example the possibility to generate comparable visualizations for similar datasets. The storage of the render graph is based on XML serialization, which is described below.

4.1.2 Extensibility

A major strength of the generic multi-volume visualization tool is the possibility to easily extend it by new render nodes. These can be combined with existing ones for the creation of new visualizations. The integration of a new render node to the system requires three steps to be carried out:

1. A new render node class has to be implemented. It has to be inherited from a basic node class that is provided by the system.

2. An individual render node view has to be provided that permits interactive manipulation of the render node's parameters.

3. Functionality for serialization and deserialization of a render node's individual state has to be implemented.

Implementation of New Render Node Classes

As described in Section 3.3.1, there are three basic types of render nodes: the scene node, structural nodes and shader nodes. The class of structural nodes is further subdivided into a splitter node, a transformation node and condition nodes. The scene node, the splitter node and the transformation node have fixed functionality and can be directly integrated into a render graph. For shader nodes and condition nodes the system provides base classes that have to be inherited for the implementation of specific functionality. The basic implementations provide all functions for integration of a render node into the shader generation framework. Thus, the task of a node programmer is restricted to the implementation of some predeclared methods that have to provide a node's individual shading functionality. These methods are automatically called during the process of shader generation.

For shader nodes basically four individual methods have to be implemented: `fill-VariableMap`, `generateShaderParts`, `prepareGL` and `cleanupGL`. In `fillVariableMap` the output variables of a node have to be defined. In addition, their dependencies on input variables, their dependencies on uniforms (externals) and their scope have to be given (see Section 3.3.3). The `fillVariableMap` method is called each time the variable state is recomputed (see Section 3.3.4). Thus, changed node parameters can be taken into account for the variable declaration. The `generateShaderParts` method is called by the shader generator for each generated shader. Here, the shader code for the computation of the output variables has to be filled in. This code depends on the previously declared input variables. To structure the code for complex computations, additional local variables can be introduced. The methods `prepareGL` and `cleanupGL` are called before and after the rendering of a single frame (see Section 3.3.5). They can be used to prepare and undo any OpenGL-specific settings that are needed for the correct evaluation of the node's shader code. In general, they should be used to prepare textures and to bind uniforms to the current shader program.

Condition nodes do not provide output variables for shading, but they need boolean condition variables to decide which conditional output branch should be evaluated (see Section 3.3.3). These output variables depend on input variables from other render nodes and their computation differs for the different types of possible conditions. For this reason the `fillVariable` method is pre-implemented in the condition node base class, and the definition of the dependencies of the condition variables is delegated to a specialized function that is called for each required condition variable.

When a condition node divides a volume into two or more parts, additional clip surfaces are introduced into the volume. The normal orientations along the clip surfaces differ from the gradients of the volume data. To achieve correct results for subsequent computations that are based on the normal direction, e.g. lighting computations, the normal has to be adjusted for sample points nearby a clip surface. For this purpose, condition nodes provide a normal correction, which is based on an idea presented by Weiskopf *et al.* [140]. Depending on the distance d of the current sampling position to the clip surface, on the normal $\mathbf{n_{clip}}$ of the nearest point on the clip surface and on the volume normal $\mathbf{n_{vol}}$ (the gradient) at the sampling position, the normal is adjusted in the following way:

$$\hat{\mathbf{n}}_{\mathbf{vol}} = w\mathbf{n_{vol}} + (1-w)\mathbf{n_{clip}}\ , \qquad \text{with } w = \begin{cases} 1, & d > d_{max} \\ 0, & else \end{cases} \tag{4.1}$$

If the sampling position is further away from the clip surface than the predefined maximum distance d_{max}, the original volume normal is provided to subsequent render nodes; otherwise, the normal of the clip surface is used. This creates a layer of finite thickness along the clip surface and produces good illumiantion results for varying sampling rates. The condition node base class already supports the normal adjustment along clip surfaces but needs as input the clip surface distance and the clip surface normal. These differ for different kinds of conditions and have to be provided by the specialized condition node sub classes.

Individual Render-Node Views

The render node view of the visualization tool shows an individual dialog for the modification of the state of the currently selected render node (see Section 4.1.1). Therefore, a render node view factory provides specific manipulation dialogs for the different node types. When no node-specific view exists, a standard view is selected that does not allow any manipulations. To ensure easy manipulation of parameters, a node programmer should provide an individual render-node view for each new render node. These can be integrated into the system by registration at the render node view factory.

There are no restrictions for the design of a render node view. Thus, it can be implemented in a way that is most intuitive for the manipulation of the changeable parameters. However, all render node views should share some basic functionality that controls the transfer of parameter changes to the underlying render nodes. Namely, each render node view should contain a check box that permits choosing if parameter manipulations should be directly transferred to the render node or if they have to be applied explicitly via an ”Apply Changes”-button.

Serialization of Render Graph Configurations

Serialization is the process to convert the in-memory state of a complex object structure into a format that can be used for transportation of the information via a network or for the storage in a file. The process of reconstructing the object structure from the serialized data is called deserialization.

The generic multi-volume visualization tool uses serialization and deserialization for the storage and reconstruction of the state of the render graph. XML (extensible markup language) is used as format for the serialized data. It was chosen for several reasons. First, XML has an intrinsic hierarchical structure. Thus, the storage of a graph structure is straightforward. Furthermore, a lot of libraries for parsing and processing of XML data do exist. Finally, XML stores the data in human-readable text format. This permits easy manipulation of a stored render graph state and even allows the configuration of a complete render graph on a textual level.

The serialization and deserialization of a render graph is carried out by the so-called *serialization manager*, which uses the document object model (DOM) for the handling of the XML data. For serialization the serialization manager starts at the scene node and traverses the render graph in depth-first order. At each render node a `node` element is added to the XML output. Listing 4.1 shows the structure of such an element. It has two attributes that give the unique `type` of the render node and a user-definable `name`. A node element can have two sub-elements, `parameters` and `childnodes`. The `parameters` element encapsulates a couple of `parameter` elements that store the state of the current render node. Each `parameter` element has a `type` and a `name`. The `type` determines the data type of the attribute, like `float`, `int`, or `string`. The `name` is used for identification of a certain parameter. Possible parameters are the `float` elements of a RGB color value or a `string` that identifies a file in which a transfer function is stored. The `childnodes` element holds the children of the render node. These children are again `node` elements. Hence, a complete render graph is serialized by recursive nesting of `node` elements.

The hierarchical tree structure of the XML output is automatically generated by the serialization manager during the traversal of the render graph. For the storage of a render node’s parameters the respective render node class has to implement an individual `serialize` method. This is called by the serialization manager when a render node is passed. To hide internals of the XML serialization from the render nodes the serialization manager provides methods for parameter adding for each possible parameter type.

When a stored render graph shall be deserialized, the serialization manager first deletes the current graph by deleting the single child of the scene node. Then, the XML render graph structure is traversed and for each node element a specific render

```
<node type="" name="">>
   <parameters>
      <parameter type="" name=""> ... </parameter>
      <parameter type="" name=""> ... </parameter>
      <parameter type="" name=""> ... </parameter>
    </parameters>
    <childnodes>
       <node type="" name="">
         ...
       </node>
       <node type="" name="">
         ...
       </node>
     <childnodes>
   </node>
</rendergraph>
```

Listing 4.1: XML-Structure of a serialized render node.

node object is created and added to its previously generated parent node. The construction of the specific render node objects is done via a factory class. This class takes the stored node type as input and creates an instance of the affiliated render node object. After creation, a node's individual state is reconstructed via a `deserialize` method, which has to be individually implemented for each render node type. Similar to serialization, the serialization manager provides methods for taking parameters of the different types from the XML data.

4.1.3 Exemplary Render Nodes

In the previous section it was shown how new render nodes for new shading techniques can be easily integrated into the generic multi-volume visualization tool. To permit the usage of the tool without the need of implementing render nodes, several nodes for a wide range of volume-visualization tasks have already been implemented. Besides the standard nodes for splitting of volume branches and for transformation, several different shader and condition nodes are provided. The group of shader nodes can be further separated into primary shader nodes that compute the colors from the current sample value and other inputs, and secondary render nodes that manipulate an already computed output color, e.g. for applying lighting effects. In the following the currently available render nodes are presented to give an idea about the capabilities of the system. These nodes should illustrate the flexibility of the visualiszation and should show that it can be applied for many different visualization tasks. For other applications further render node types may have to be integrated, which is easily possible.

Primary Shader Nodes

DVR Node A *Direct Volume Rendering (DVR) Node* performs standard volume shading. It takes the current sample value as input and takes the related color from a look-up table. This look-up table is based on a freely definable transfer function. Per default a standard transfer function is used that allows the manipulation of the color channels independently, but it is also possible to apply task-specific transfer functions. Optionally, pre-integration can be activated. Then, the current and the following sample value is taken to read the precomputed slab color from a 2D pre-integration table.

Isosurface Node An *Isosurface Node* renders a volume's isosurface corresponding to a certain isovalue i. For this purpose, the current sample value ϕ_0 and the next sample value ϕ_1 are retrieved and compared to the isovalue i in the following way:

$$iso = \begin{cases} 1, & ((\phi_0 <= i < \phi_1) \vee (\phi_1 < i <= \phi_0)) \\ 0, & else \end{cases} \tag{4.2}$$

iso indicates if the isosurface is intersected while the viewing ray is passing the slab from the current sampling position to the next one. To get the RGBA output color a user-definable RGBA isosurface color is multiplied with iso. Thus, if the isosurface is not hit, the returned alpha value is zero, and, thereby, the current volume sample will not be visible at all.

The isosurface is displayed at each surface sample with the same color, so its appearance is initially flat. To get a 3D impression of the surface, an *Isosurface Node* should be combined with a render node that applies some illumination effects. Those usually take the surface normal into account. The *Isosurface Node* approximates this surface normal $\mathbf{n}$ by linear interpolation of the volume gradients $\mathbf{g_0}$ and $\mathbf{g_1}$ at the current and the following sampling position, due to the sample values ϕ_0 and ϕ_1 and the isovalue i:

$$\mathbf{n} = a\mathbf{g_0} + (1-a)\mathbf{g_1} \ , \qquad \text{with } a = \frac{|\phi_0 - i|}{|\phi_1 - \phi_0|} \tag{4.3}$$

LIC Node *Line integral* convolution (LIC) is a technique for the dense visualization of vector fields. Those vector fields often present measured or simulated flow or can be derived from scalar fields. Examples for derived vector fields are the volume gradients or the first principal curvature vectors of the implicitly defined isosurfaces. The first principal curvature indicates thereby at each surface point the direction along the highest curvature.

A *LIC Node* uses line integral convolution to visualize the first principle curvatures on a pre-selected isosurface. It can be used for illustrative accentuation of the surface structure of a certain organ or a pathological formation. First, the isosurface is extracted in the same way like for the standard *Isosurface Node* and, then, the LIC computation is applied on the pre-computed curvature field. Details on curvature and LIC computation on isosurfaces can be found in section 4.2.3.

Secondary Shader Nodes

Illumination Node An *Illumination Node* realizes standard illumination according to the *Blinn-Phong model* [8]. With the incoming color $\mathbf{c_{in}}$, the sample normal $\mathbf{n}$ and the light direction $\mathbf{l}$, the output color $\mathbf{c_{out}}$ is computed in the following way:

$$\mathbf{c_{out}} = k_a \mathbf{c_{in}} + k_d(\mathbf{n} \cdot \mathbf{l}) + k_s(\mathbf{n} \cdot \mathbf{h})^p \mathbf{c_{in}}. \tag{4.4}$$

The first term is the *ambient term*, the second the *diffuse term*, and the third is the *specular term*. $\mathbf{h}$ is the so called halfway vector between the viewing direction $\mathbf{v}$ and the light direction $\mathbf{l}$

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{|\mathbf{v} + \mathbf{l}|}. \tag{4.5}$$

k_a, k_d, and k_s are reflection coefficients that specify the influence of the different terms. The width of a specular highlight, the so-called *shininess*, is controlled by the specular-reflection exponent p.

An *Illumination Node* can be combined with any primary shader node or even cascaded with other secondary nodes. The normal direction is usually similar to the gradient direction at the sampling position, but there are render nodes, e.g. condition nodes and isosurface nodes, that modify the normal. For this reason, preceding render nodes have to provide the sample color and the sample normal as input.

Cartoon-Shading Node *Cartoon shading* [70] is an illustrative illumination technique that imitates the shading style of cartoonists. They usually paint areas that point towards the light source in a single constant color, and areas that point away in another constant color. Cartoon shading produces similar results by applying distinct colors depending on the dot product $(\mathbf{n} \cdot \mathbf{l})$ between the surface normal and the light direction.

A *Cartoon-Shading Node* modifies the incoming sample color by multiplying it with the following quantized intensity i:

$$i(\mathbf{n}) = \min\left(\frac{\lfloor |\mathbf{n} \cdot \mathbf{l}| \cdot s \rfloor + 1}{s}, 1\right) \tag{4.6}$$

i takes s distinct values from the set $\{1/s, 2/s, ..., 1\}$. For $s = 2$ the result is equal to classical cartoon shading; for $s > 2$ there are more than two areas of constant color.

Ghosting Node *Ghosting* is an illustrative technique that is used to display internal features, while simultaneously external structures are provided as context information. Therefore, the opacity of the external structures is selectively reduced to give view to the inside. The modification of the opacity can, e.g., depend on the current view direction [67] or on features contained in the visualized data sets [12].

The *Ghosting Node* implements a ghosting model that modifies the incoming opacity α_{in} by multiplication with a weighting factor w due to a user-definable sphere. w depends on the current sampling position $\mathbf{p} = (p_x, p_y, p_z)^T$, and on the sphere's radius r and center $\mathbf{c} = (c_x, c_y, c_z)^T$:

$$w(\mathbf{p}) = \max\left(\frac{(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2}{r^2}, 1\right) \tag{4.7}$$

The left parameter of the max-function is derived from the standard sphere equation, normalized with respect to the sphere radius. At the sphere center this term takes the value 0, at the sphere surface it is 1, outside the sphere it is greater than 1. Thus, w has an quadratic progression from 0 to 1 inside the sphere and is clamped to 1 by the max-function outside the sphere.

Recolor Node The *Recolor Node* modifies the incoming sample color by multiplying it with another user-defined color. In combination with condition nodes, this allows the defintion of the shading style for a whole volume data set and the manipulation of the basic output color after the evaluation of the conditions. Especially for the visualization of segmented data (see below) this is a helpful functionality.

Condition Nodes

Plane Condition Node A *Plane Condition Node* separates the active volumes into two halves along one or more clip planes. The i-th clip plane is defined by its standard clip plane function

$$cp_i(\mathbf{p}) = a_i p_x + b_i p_y + c_i p_z + d_i, \tag{4.8}$$

with $\mathbf{p} = (p_x, p_y, p_z)^T$ as the current sampling position. When $cp_i(\mathbf{p})$ is greater or equal to zero, the current sampling position lies on one side of the plane; if it is less than zero, it lies on the other side. To decide which outgoing branch of a `Plane Condition Node` with n clip planes should be chosen the n plane conditions $cp_i(\mathbf{p}) \geq 0$ are conjuncted by logical ands:

$$branch(\mathbf{p}) = \begin{cases} 0, & \bigwedge_{i=1}^{n} cp_i(\mathbf{p}) \geqq 0 \\ 1, & else \end{cases} \tag{4.9}$$

Sphere Condition Node A *Sphere Condition Node* cuts the volume scene into two parts along the surface of a sphere. One part is the sphere's inside and its surface; the other part is the area outside the sphere. The sphere shape can be implicitly described by the standard sphere equation, which depends on the sphere's center $\mathbf{c} = (c_x, c_y, c_z)^T$ and its radius r. The branch selection rule is as follows:

$$branch(\mathbf{p}) = \begin{cases} 0, & (p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 \leqq 0 \\ 1, & else \end{cases} \tag{4.10}$$

Tag Condition Node A tag volume is a volume dataset in which each voxel holds a unique integer ID (*tag*). This tag indicates the affiliation of the voxel to a certain structure that is contained in the dataset. The tag volume can either represent explicitly segmented anatomical or pathological structures (see Section 2.4.2) or can be derived from a standardized medical atlas.

A *Tag Condition Node* uses an explicitly applied tag volume to separate the incoming volumes into several distinct objects. It can have an arbitrary number of outgoing branches, and to each branch one or more tags can be assigned. Each tag can be assigned solely to a single branch. To decide which branch should be chosen at a certain sampling position $\mathbf{p}$, first, the affiliated tag $t(\mathbf{p})$ is evaluated by a nearest neighbor lookup in the tag volume. Then, it is checked for each outgoing branch i if $t(\mathbf{p})$ is contained in the applied tag set T_i. In the positive case this branch is chosen.

Since nearest neighbor lookup for evaluation of a tag is used, the border of a visualized object may show blocky step artifacts. To avoid this, the object borders can be alternatively smoothed by trilinear interpolation, as it was proposed by Hadwiger *et al.* [46]. Therefore, the tags for the eight neighboring voxels of a sampling position are looked up in the tag volume. Then, it is checked for each outgoing branch if the tags of the eight neighbors are contained in the branch's tag set. In the positive case, the respective voxel gets an intermediate ID of 1, otherwise 0 is applied. From these IDs a floating point membership value for the current sampling position is computed by trilinear interpolation. If this value is greater than 0.5, the current sampling position is belonging to the resepective tag set and the associated branch is chosen. A drawback of this smoothing method are the higher costs for the additional lookups in the tag volume.

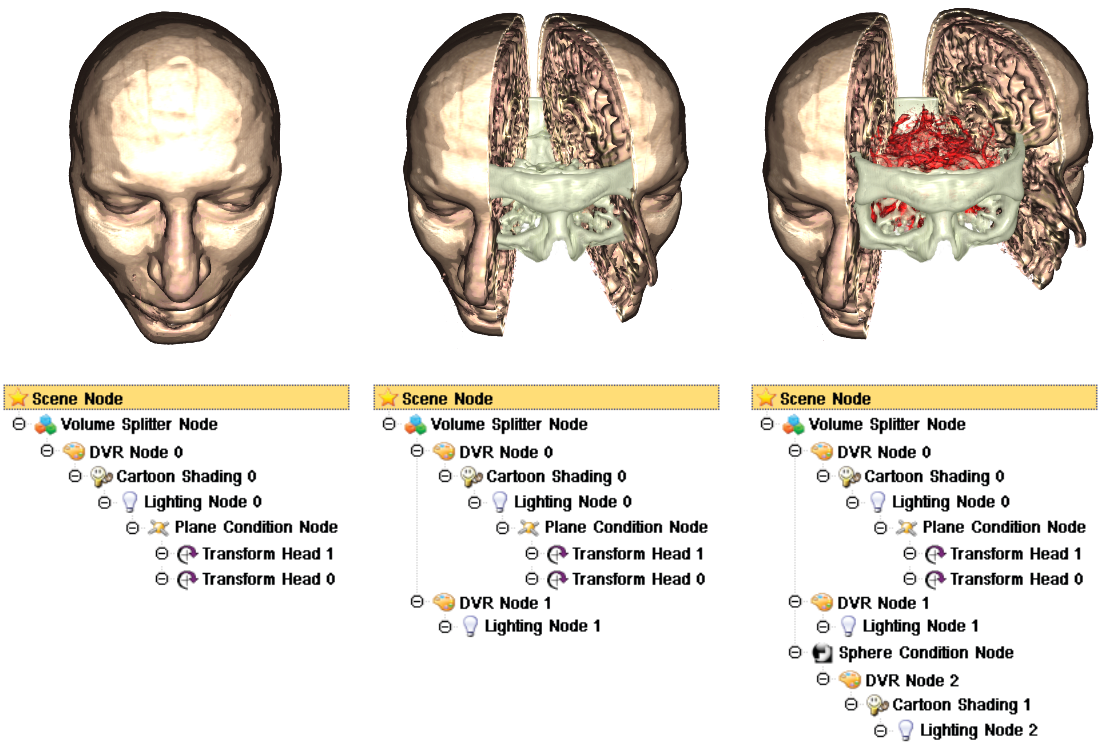
4.1.4 Case Studies

To demonstrate the flexibility of the presented generic multi-volume rendering tool, it was applied to several medical use cases, which are detailed in the following. In addition, several performance measurements of the system for the different setups are presented, and the advantages and drawbacks of the different rendering techniques introduced in Section 3.2 are discussed. The visualization results of the example setups and the corresponding render graphs are shown in Figure 4.3 and Figure 4.4.

Neuroradiological Diagnosis

The first use case is an example from the field of neuroradiological diagnosis for the detection of malformations of cerebral blood vessels. In this case a CTA scan of the patient's head is taken in which the vessel structures are emphasized by a previously injected contrast agent (see Section 2.1.1). In addition, a MRI scan, that accentuates the brain tissue, is acquired to get the patient specific relationship between the blood vessels and the anatomical structure of the brain. The CTA and the MRI scan are co-registered in a preprocessing step before visual diagnosis.

Figure 4.3 shows an example visualization (Setup I) of this two-volume scene. The images (a-c) in the top row illustrate different visualization steps during the composition of the render graph. The images in the bottom row give the corresponding render graph configurations. The goal of the visualization is to present the intracranial brain vessels in relation to the surrounding skull and in the context of the brain



```
Scene Node
⊖ Volume Splitter Node
    ⊖ DVR Node 0
        ⊖ Cartoon Shading 0
            ⊖ Lighting Node 0
                ⊖ Plane Condition Node
                    ⊖ Transform Head 1
                    ⊖ Transform Head 0
```

```
Scene Node
⊖ Volume Splitter Node
    ⊖ DVR Node 0
        ⊖ Cartoon Shading 0
            ⊖ Lighting Node 0
                ⊖ Plane Condition Node
                    ⊖ Transform Head 1
                    ⊖ Transform Head 0
    ⊖ DVR Node 1
        ⊖ Lighting Node 1
```

```
Scene Node
⊖ Volume Splitter Node
    ⊖ DVR Node 0
        ⊖ Cartoon Shading 0
            ⊖ Lighting Node 0
                ⊖ Plane Condition Node
                    ⊖ Transform Head 1
                    ⊖ Transform Head 0
    ⊖ DVR Node 1
        ⊖ Lighting Node 1
    ⊖ Sphere Condition Node
        ⊖ DVR Node 2
            ⊖ Cartoon Shading 1
                ⊖ Lighting Node 2
```

(a) (b) (c)

Figure 4.3: Multi-Volume Setup I: Combination of a CTA dataset and a related MRI dataset of a human head. The MRI dataset provides the skin and brain tissue. It is vertically cut and the two halves are moved away from each other to get insight to the inner structures. The CTA dataset contains the skull and the vessels which are rendered with different transfer functions. The top row shows three stages (a-c) of an interactive multi-volume visualization session. The bottom row shows the corresponding render graph configurations.

structure. Therefore, the visualization path of the two volumes is first split into one branch for the MRI volume and two branches for the CTA volume by a *Splitter Node*. Then, the MRI volume, which contains the skin and the brain tissue, is rendered with a *DVR node*, and the surface structure is emphasized by the combination of a *Cartoon-Shading Node* and an *Illumination Node*. To get insight into the inner structures, the MRI head is divided vertically by a *Plane Condition Node* and the two halves are rotated and moved away from each other by two *Transformation Nodes*. The first branch of the CT volume is responsible for the visualization of the skull. For this purpose, a *DVR Node* with a transfer function that extracts the bone tissue and a standard *Illumination Node* is applied. On the second CT branch the vessel structure inside the skull is extracted. Primarily, a *Sphere Condition Node* is applied, which approximates the brain volume by a sphere and cuts away all vessels outside this sphere. Then, a *DVR Node* with a transfer function that extracts the vessels is attached; finally, the vessels are emphasized by cartoon and Phong shading.

Illustration with Ghosting and LIC

Illustrative volume rendering techniques become more and more important in medical volume visualization because they permit to emphasize significant information in the datasets while nonrelevant information is suppressed. While the major application of illustrative volume rendering is the creation of illustrations for presentation and education, it can also be used for diagnostic and analytic purposes.

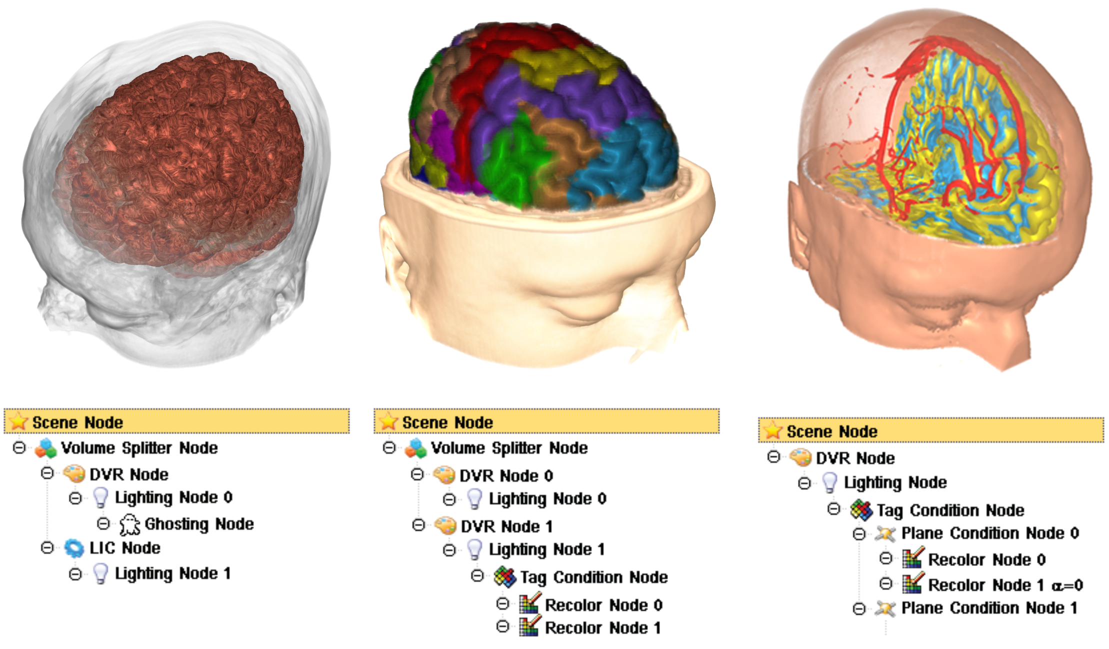
Figure 4.4 (a) (Setup II) shows an illustrative medical multi-volume visualization that was generated with the generic multi-volume visualization tool. It presents a two-volume scene that consists of a MRI dataset of a human head and a second dataset that contains the explicitly segmented brain from the first dataset. The whole MRI volume is shaded with DVR and illuminated with Blinn-Phong shading. Additionally, a *Ghosting Node* is appended, which subsequently increases the transparency of a sample with respect to the center and radius of the predefined sphere. By this means, the inside brain becomes visible, which is rendered as an illuminated isosurface with an additional 3D LIC computation applied (*LIC Node*) to emphasize the surface curvature.

Functional Brodmann Areas

In the next use case (Figure 4.4 (b), Setup III), the brain data of the previous example is subdivided into several functional regions due to a given Brodmann brain atlas [11]. Therefore, the MRI head is again shaded with DVR and illuminated with Blinn-Phong but with another transfer function as in the previous setup. The upper half of the head is cut away by a *Plane Condition Node*, which is placed in front of the *DVR node*. The brain is initially shaded with DVR, with a gray value transfer function applied, and also illuminated. Then, a *Tag Condition Node* is attached, which takes the brain atlas as tag volume. Several tag groups are defined, and to each outgoing branch of the *Tag Condition Node* a *Recolor Node* is attached, which multiplies the incoming gray values with a pre-defined color.

Pre-segmented Anatomical Structures

For the last example (Figure 4.4 (c), Setup IV) similar visualization concepts like in the previous setup are applied to another combination of datasets. Here, a simulated MRI dataset of the BrainWeb database [4; 77] is visualized in combination with a corresponding anatomical segmentation volume. The segmentation volume assigns to each voxel a unique ID of the tissue type to which the voxel belongs. First, the whole MRI dataset is shaded and illuminated with a gray value transfer function applied. Then, a *Tag Condition Node* with the anatomical segmentation volume as tag volume is attached and conditional output branches for skin, skull, grey matter, white matter, and vessels are defined. To each of these branches a *Recolor Node* is attached to give the different tissues individual colors. In addition, skin, skull, grey matter, and white matter are partly clipped away by several *Plane Condition Nodes*, each consisting of two orthogonal clip planes. While skull and brain are completely removed by setting



(a) (b) (c)

Figure 4.4: Multi-Volume Setups II-IV and the corresponding render graphs: (a) Setup II shows the combination of an illuminated DVR shaded MRI head with a *Ghosting Node* applied to show the inside. The interior brain is rendered as illuminated isosurface with 3D LIC applied, to emphasize the curvature. (b) In Setup III the upper half of the head is cut away, to lay open the brain, which is segmented and colored due to a functional Brodmann brain atlas. (c) Setup IV shows an MRI data set of a head segmented into different anatomical regions, such as skin, brain tissue, and vessels. The regions are differently colored and partly cut away by two clip planes.

the alpha value to zero, the clipped skin is still rendered semi-transparent to give a feeling of the whole head's anatomy.

Performance Measurements and Discussion

The rendering performance of the system was measured for each of the four example setups. The CTA and the MRI dataset used for Setup I have both a resolution of $256 \times 256 \times 120$ voxels, the head and the brain dataset for Setup II and III have a resolution of $181 \times 217 \times 181$ voxels, and the dataset of Setup IV has a resolution of $256 \times 256 \times 181$ voxels. Table 4.1 shows the achieved frame rates for the three multi-volume slicing techniques presented in Section 3.2.1 and the multi-volume ray casting technique presented in Section 3.2.2.

Regarding the three slicing techniques, it can be seen that, depending on the complexity of the applied render graph and the total number of volumes in the scene, the advantages of the different techniques are accentuated. For most cases the separation

	Slice-based			Ray casting
	Merge	Separate	Intersect	
Setup I (a)	57	67	57	85
Setup I (b)	50	60	40	30
Setup I (c)	37	57	22	18
Setup II	19	21	21	25
Setup III	55	70	63	45
Setup IV	53	56	56	52

Table 4.1: Performance of the three multi-volume slicing techniques and of multi-volume ray casting on a 512^2 viewport given in frames per second (fps). Measurements have been performed on a NVIDIA GeForce GTX280 graphics board with 1024 MB memory.

method dominates in terms of performance, but with the significant drawback that the intermixing functionality is restricted to standard GPU blending operations. If more sophisticated intermixing functions are required, the two other slicing techniques are the only choice, which have the disadvantage of high cost for the additionally required tessellation. For Setup II, Setup III and Setup IV merge is slower than intersect since the merge method has to test for each sample whether it belongs to a volume or not, even if the volumes do completely overlap. In Setup I (b) and Setup I (c) the advantage turns over to merge because of the exponentially raising effort for tesselating the overlapping proxy geometries that is needed by the intersect approach. Summarizing, the choice of the slicing technique highly depends on the graph configuration and the desired quality of the visualization result.

For most of the test cases raycasting shows slower rendering performance compared to the best performing slicing method. However, in Setup I (a) ray casting clearly wins. Here, only one volume dataset is contained in the scene and, thus, no costly depth peeling has to be applied. This case shows that single-volume ray casting becomes favourable over slice-based rendering of single volumes on current GPU generations. For the LIC computation in Setup II, which is very expensive due to the filter kernel, the early-ray termination technique for raycasting takes effect. In this case, raycasting is slightly faster. Summarizing, it can be concluded that with increasing complexity of the multi-volume visualization the performance drawback of the raycasting approach levels out, while the evaluation of the volume integral is more appropriate and leads generally to better visual results. Thus, it can not be said which approach is favorable over the other as the suitability highly depends on the scene.

Regarding the system's complexity, the effort for shader generation has also to be taken into account. It is linear with respect to the number of volumes and the number of render nodes because each node has to be processed two times for each volume, once in each pass of the two-pass shader assembly (see Section 3.3.4). Since the total number of volumes and render nodes is rather small, the generation time is minimal in contrast to the rendering performance. Another aspect is the complexity of the generated shader programs, which is also linearly increasing with the number of volumes and render nodes. Additionally, it depends on the complexity of the applied shading algorithms, e.g. the LIC computation in setup II is very expensive and, thus, highly effecting the frame rates. Nevertheless, for both rendering techniques the performance tests have shown that the system provides interactive framerates even for complex scenarios. So, it fits well to a wide range of medical problems and supports the creation of meaningful and comprehensive visualizations in an intuitive way.

4.2 Visualization of Functional Brain Images

The multi-volume visualization tool that was introduced in the previous section enables an expert user to apply it to varying medical visualization tasks. However, the daily work of medical doctors and medical researchers is often restricted to a dedicated application area. To best support their work, an optimal visualization tool should adapt and restrict the provided visualization and interaction features to those that are required for the specific task for which it was designed. In this section a visualization solution for the analysis of functional brain images is presented that focuses on the needs of cognitive neuroscience.

The field of cognitive neuroscience seeks to understand the links between human thoughts, feelings and actions, and the functions of our brains. Its main belief is that all facets of our psychic life have a neuronal basis. Early research in this field primarily explored which psychic functions are distorted if parts of the brain have been damaged by accidents, tumors or strokes. Today, however, the *via regia* to explore the neural basis of mental activities are the so called neuroimaging methods of which the main goal is to make visible the activities of the brain, for example *functional Magnetic Resonance Imaging* (fMRI).

The three-dimensional visualization of these functional brain images in relation to their anatomical context would help cognitive scientists in gaining a deeper insight into the data. But a major problem of 3D visualization of fMRI data is the fact that the anatomical brain tissue is surrounding the activation data and, thus, will occlude it when standard 3D visualization techniques are applied. Many systems try to solve this problem by projecting the functional data onto the brain surface. This projection is either done along the surface normal [122; 124] or along the viewing direction [106]. Both approaches, however, produce an incorrect depth perception. If projection along the surface normal is applied, deep objects will appear greatly magnified; if projection along the viewing vector is used, the functional data would appear to move when the viewpoint changes. For this reason, other approaches use direct volume rendering for the fused visualization of anatomical and functional data. E.g., König *et al.* [65] introduced transfer function volumes to distinguish between activated and not activated voxels of the brain. Beyer *et al.* [7] combined anatomical and functional MRI datasets for neurosurgical planning and enable viewing of the functional data by clipping away occluding anatomical structures. Jainek *et al.* [57] mixed surface-based rendering for the anatomical brain structure with direct volume rendering for the fMRI activation.

The visualization approaches presented in this section aim to solve the occlusion problem by combining several different illustrative and non-illustrative direct volume rendering techniques on the basis of the multi-volume rendering framework introduced in Chapter 3. After giving a short introduction to functional imaging methods, i.e. fMRI (Section 4.2.1), a visualization tool for fMRI data and its application to measured data sets of a cognitive study is presented in Section 4.2.2. In Section 4.2.3 this tool is extended by line integral convolution to emphasize the brain structure. The described work was first published in [110] (Section 4.2.2) and in [116] (Section 4.2.3). The visualization tool presented in Section 4.2.2 was developed in cooperation with Eduardo Tejada, Universität Stuttgart. The work described in Section 4.2.3 was carried out in collaboration with Tobias Schafhitzel, Universität Stuttgart, who contributed the LIC computation. The cognitive neuroscientists Markus Knauff and Thomas Fangmeier from the University of Gießen and the University of Freiburg gave the background information about fMRI, provided the applied data sets and tested the software.

4.2.1 Neuroimaging in Cognitive Neuroscience

Contemporary research in the field of cognitive neuroscience is to a great extent performed by using fMRI. This method takes advantage of the fact that cognitive processes lead to a local increase in oxygen delivery in the activated cerebral tissue [34]. Physically, the fMRI technique relies on the understanding that deoxy-hemoglobin is paramagnetic and oxy-hemoglobin diamagnetic. Increased presence of oxy-hemoglobin leads to changes in the local magnetic field homogeneity, which is commonly referred to as the *Blood-Oxygen-Level-Dependent* (*BOLD*) *effect* [92; 100]. A local increase in oxygen delivery is thought to be correlated with brain activation.

To measure these changes in blood flow, a number of people are placed one after the other in a magnetic resonance tomograph. They typically lie on their back and their head position is fixed in a head coil. A mirror system is placed on the coil so that they can see a projection screen mounted on the rear of the scanner. The cognitive tasks are either presented on this screen or via headphones, and the participants respond to them by pressing buttons of a MRI-compatible response box. Typically, functional images are collected in a *gradient-recalled echo-planar imaging* (*EPI*) sequence, allowing the sampling of up to 32 parallel slices that cover parts of the brain or the whole brain. The principle of fMRI experiments is to measure brain activation of quickly repeated intervals and to explore differences among them. In the classical paradigm the baseline activity is measured when the volunteer is at rest, and other measurements are taken when the participant performs certain cognitive tasks. Then, the activity in the baseline condition is subtracted from the activity measured during the performance of the cognitive tasks, and the resulting data is statistically analyzed. In more modern experiments combinations of experimental conditions are compared to other combined conditions.

A great majority of cognitive scientists use the *SPM* (Statistical Parametric Mapping) software [38; 142] to statistically analyze the brain activations. It has been developed by members of the *Wellcome Trust Center for Neuroimaging* in London and allows the analysis of whole sequences of brain imaging data. The sequences can be series of images from different groups of people or time series from the same subject. Basically, the statistical analysis tests the measurements against some previously determined model hypotheses. The statistical results are then transfered into so-called statistical parametric maps (SPMs), which can be used for analysis and visualization. Broadly speaking, an SPM gives the degree of activation during a certain cognitive task for each voxel in a dataset.

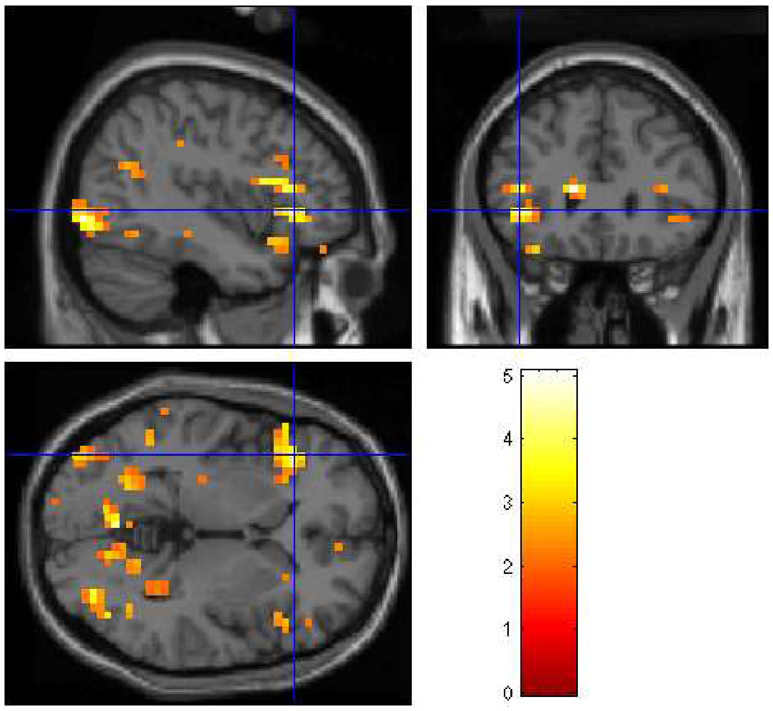


Figure 4.5: 2D visualization of a statistical parametric map (SPM) provided by the SPM software. Three axis aligned slices of an anatomical template dataset are overlaid by the color coded brain activation that is stored in the SPM. The color bar on the left bottom shows the mapping of the color to the degree of activation. (Image courtesy Thomas Fangmeier, University of Freiburg)

4.2.2 Visualization of Statistical Parametric Maps

Before statistical analysis the fMRI scans of a measured series are co-aligned to each other and then mapped into a standard anatomical brain space, a process which is called *spatial normalization*. This allows, e.g., for easy inter-subject comparisons. The standard brain space is either defined by the brain atlas of Talairach and Tournox [125] or the newer *MNI brain* generated by Evans *et al.* [30]. Since the fMRI measurements are spatially normalized, the resulting SPMs are as well spatially normalized. For visualization a SPM is usually rendered in combination with a corresponding anatomical template dataset of a standardized brain that corresponds to the applied brain space. The SPM software provides a simple 2D visualization technique that overlays some previously selected slices of the anatomical template dataset with the corresponding activation slices of the SPM (see Figure 4.5). In the following it is illustrated how the same template dataset can be used for 3D visualization.

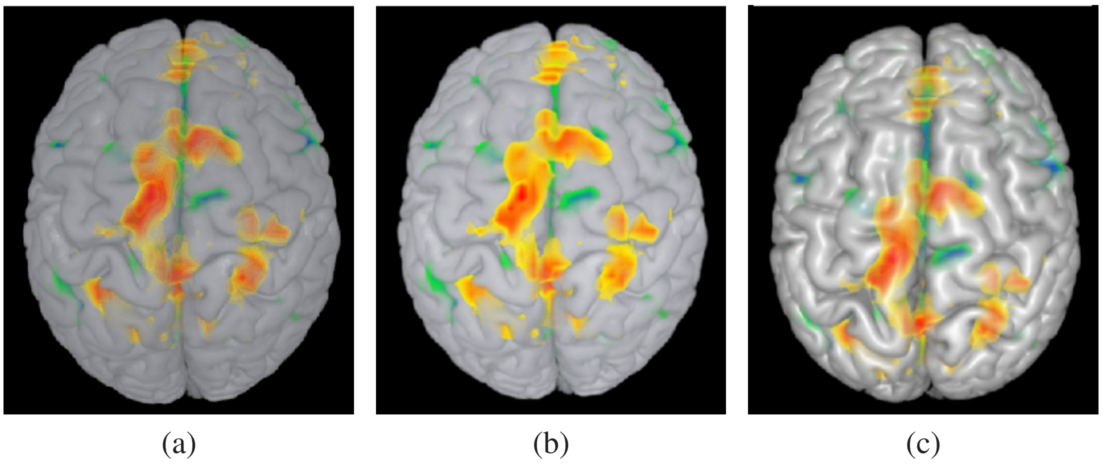
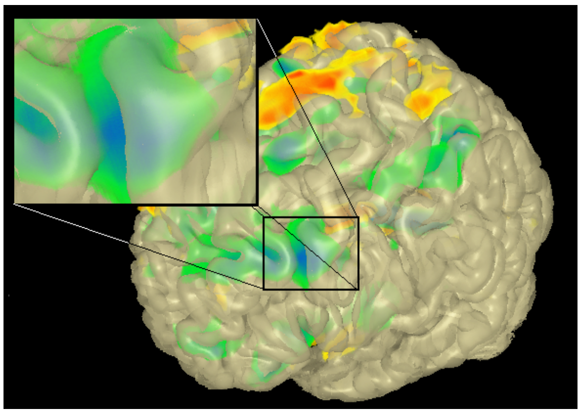


Figure 4.6: Simultaneous rendering of a SPM and a related anatomical template dataset of a human brain. Both datasets are rendered with DVR. A gray value transfer function is applied to the anatomical brain. Positive values of the SPM activation are mapped to colors between yellow and red, negative values to colors between green and blue. In (a) standard DVR is applied for both datasets; in (b) the brain is rendered with pre-integration; in (c) additional illumination is applied to the brain.

3D Visualization Methods

3D visualization of a SPM in combination with the anatomical template dataset can be easily done with the multi-volume rendering framework from Chapter 3. Since the two datasets origin from different imaging modalities (fMRI and MRI), it does not make sense to treat them similarly for visualization. Thus, it is obvious to first split up the visualization paths of the two volumes by a *Splitter Node* and then to apply different combinations of shader nodes. Figure 4.6 shows three visualizations where a standard *DVR Node* is applied to both volumes. For the template brain a standard transfer function is used, that maps the volume's scalars independently to grey values. For the SPM a specialized transfer function is employed that better fits to the nature of the contained data.

A SPM contains floating-point values that give the degree of activation. There is no fixed scale, and there is theoretically no upper bound. Often activation differences between two different cognitive tasks are studied. The resulting datasets, which are called contrasts, can contain positive and negative values. The transfer function for the SPM data is inspired by the color coding that is used for 2D visualization by the SPM software. A user can define a lower threshold and an upper bound for the accepted positive values and an upper threshold and lower bound for negative values. To each of these thresholds and bounds a color has to be assigned. Positive data values between the positive bound and the positive threshold are mapped to colors that are linearly interpolated from the two predefined boundary colors. Values above the positive bound are constantly mapped to the color that is applied to this bound. Negative data values are treated similarly. Data values between the positive and the negative threshold are


(a)

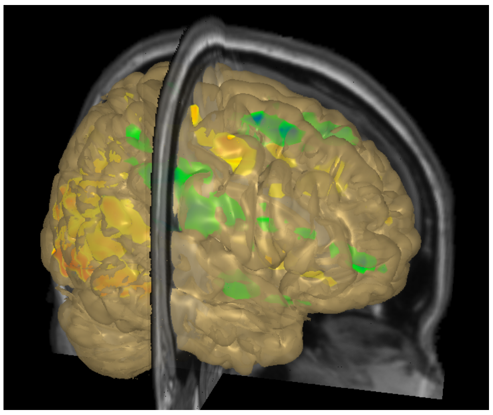

(b)

Figure 4.7: Application of isosurface rendering to make activation in deeper brain regions visible. (a) The surface of the template brain is rendered as a semi-transparent illuminated isosurface. The detail gives a closer look to a selected area. (b) An additional visualization branch renders two axis-aligned 2D slices of the unsegmented template MRI dataset to give further anatomical context

skipped. This allows the culling of noise and of small activation values that are not important for the analysis.

For the visualizations in Figure 4.6 not the whole MRI template dataset is used for rendering but only the explicitly pre-segmented brain. This allows visualizing the SPM activation in conjunction with the brain surface. In Figure 4.6 (a) standard DVR is applied to both datasets. Positive activation values are tranformed into colors from yellow to red; negative values are mapped into a range from green to blue. In Figure 4.6 (b) the anatomical brain is rendered with pre-integration, which visibly improves the appearance of the surface. The brain surface is further emphasized in Figure 4.6 (c) by applying additional illumination.

When DVR shading is used for the visualization of the brain, one has to choose a relatively high opacity to get a well distinguishable presentation of the brain surface. As a consequence, only activation near by the brain surface will be visible. To show activation of deeper brain areas as well, isosurface rendering can be used instead for the representation of the brain. In Figure 4.7 (a) the brain surface is rendered as a semi-transparent, illuminated isosurface. This gives insight to the underlying brain activation, while the anatomical brain structure is simultaneously provided as reference. In Figure 4.7 (b) an additional visualization branch is applied in which two orthogonal 2D slices of the unsegmented MRI template dataset are rendered for supplemental orientation. For this kind of visualization a new render node was implemented that allows the rendering of up to three axis-aligned 2D slices of a volume dataset. The three slices can be freely positioned along their associated coordinate axis. This is implemented by checking at each sampling position if the viewing ray passes one of the slices on its way from the current to the following sampling point. In the positive case,

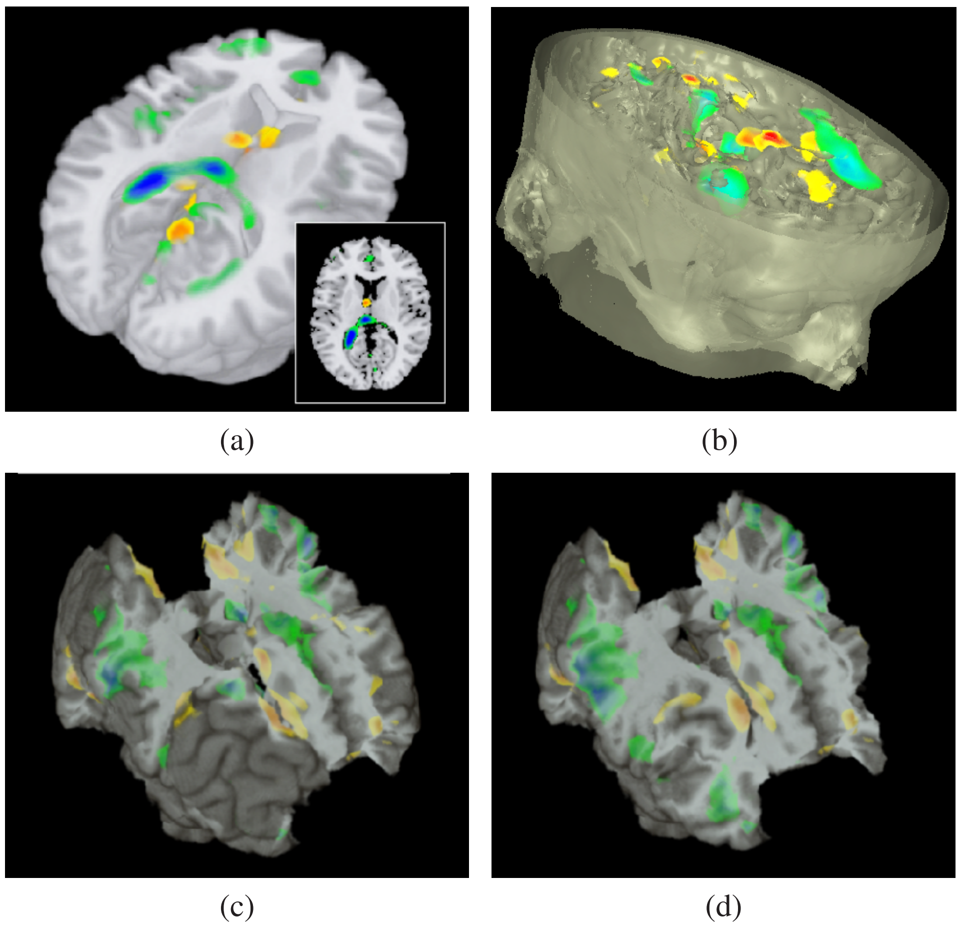
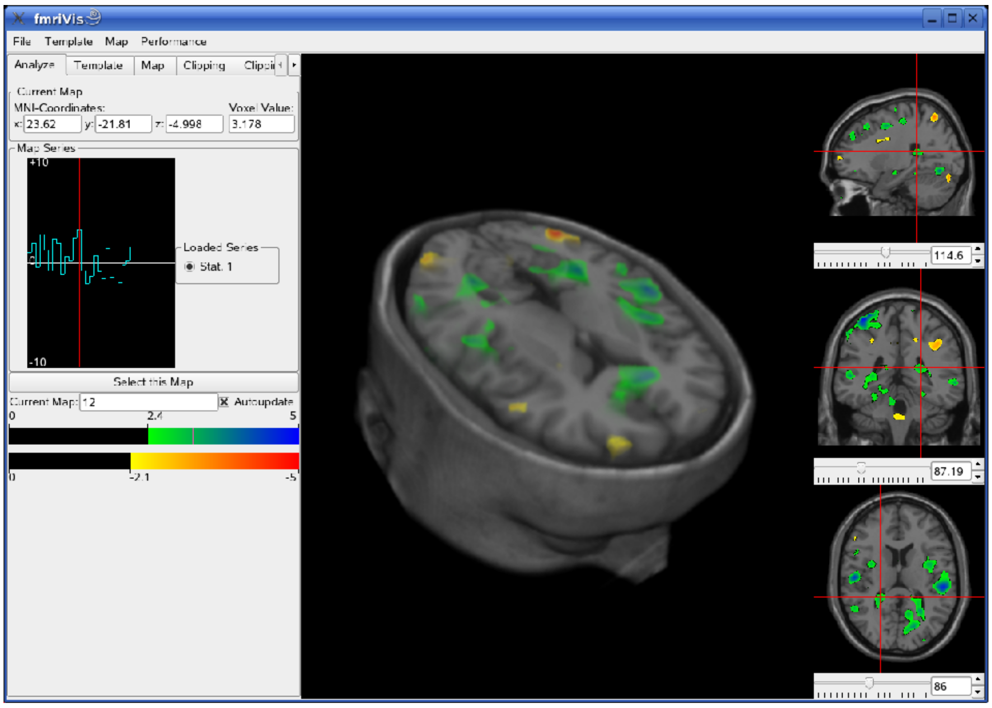


Figure 4.8: Several cut-away views that give insight into the brain. (a) Standard DVR rendering of the template brain with the upper half clipped away along a clip plane. The small image shows the 2D slice that corresponds to the clip surface. (b) Semi-transparent isosurface rendering of the whole anatomical head with the upper half cut away. (c) 24 out of 48 functional Brodmann areas are cut out with a *Tagged Condition Node*. (d) Similar to the configuration in (c) but only 19 Brodmann areas are shown.

the underlying sample value is mapped to a color due to an arbitrary transfer function, otherwise the sample is discarded.

Another way to get insight to the brain is cutting away some parts of it by a conditional render node. In Figure 4.8 (a) the upper parts of the brain volume and the SPM datset are cut away by a single clip plane. The same is done in Figure 4.8 (b), but this time the whole template dataset is visualized by a semi-transparent isosurface. This provides the brain activation in the context of the whole head's anatomy. In Figure 4.8 (c) and Figure 4.8 (d) some parts of the brain and the activation are cut away with a *Tag condition node*. The functional Brodmann Atlas [11] is used as tag volume. Thus, it is possible to investigate the activation of certain functional brain areas.



MNI-Coordinates: x: 23.62 y: -21.81 z: -4.998 Voxel Value: 3.178

Current Map: 12 [✓ Autoupdate]

Figure 4.9: A screenshot of the visualization tool for functional SPM data. The large main view in the center shows the combined 3D volume visualization of an SPM activation dataset and a related anatomical template dataset. The three views on the right side give 2D slices of the same datasets. On the left side there are several controls for the manipulation of the visualization.

Application

To optimally support cognitive scientists during their work, a specialized visualization tool was developed (see Figure 4.9). In its main view it provides the 3D-visualization capabilities described above. Different rendering styles for the anatomical template and the functional activation can be chosen, the transfer functions can be manipulated individually, and clipping can be activated when needed. In addition, there are three 2D-slice views that show three axis-aligned slices of the template dataset overlaid by the associated SPM activation. In each of these slice views the user can navigate through the whole template datasets along the respective axis. The change of the position is immediately shown in the other 2D views by a cross hair.

Besides the template dataset, one can load any SPM dataset of each step of the analysis pipeline, from pre-processing to statistical activation maps. To allow the easy comparison of different co-related datasets, a whole series of SPMs can be loaded simultaneously and the user can interactively switch between the different functional datasets. Furthermore, a certain voxel can be selected via positioning of the cross hair in the 2D-slice views. The change of the activation in this voxel over the whole functional series is plotted in an extra view.

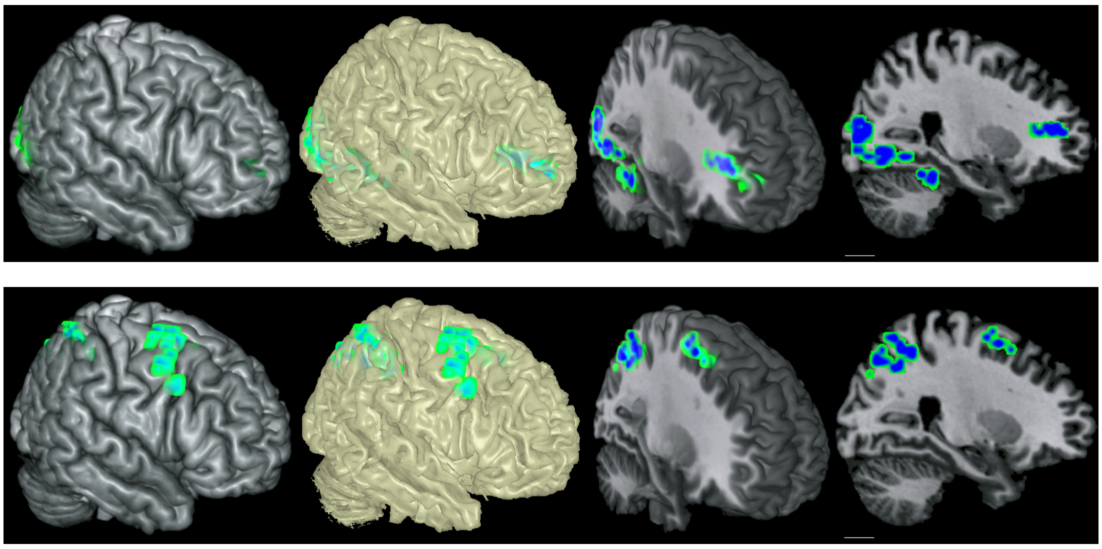


Figure 4.10: Two different stages in the reasoning process. From left to right: pre-integration with illumination, semi-transparent isosurfaces, pre-integration with one clipping plane and a corresponding two-dimensional slice.

The SPM visualization tool has been successfully used for the visualization of data gathered from a study that has been conducted at the University of Freiburg [31]. In these experiments the participants performed logical reasoning problems while the brain activity was measured. During the logical reasoning problem, the participants were asked to draw conclusions from given premises, and later their responses were evaluated for logical validity. For instance, they saw two premises:

Premise 1 : *V X* *(V is on the left of X).*
Premise 2 : *X Z* *(X is on the left of Z).*

and they had to decide afterwards and indicate by a key press whether the following statement logically follows from the premise:

Conclusion: *V Z* *(V is on the left of Z)?.*

Figure 4.10 shows two different stages in the reasoning process (event-related design, 12 participants) for different rendering modes. In the top row it is possible to see activations in the occipito-parietal cortex and the anterior prefrontal cortex, whilst the bottom row depicts the activation during the validation in the parietal and the prefrontal cortex (more details of the study can be found in [31]).

4.2.3 Enhanced Surface Perception by Flow Visualization

The two presented strategies for the simultaneous visualization of inside brain activation and the surrounding anatomical brain – clipping and semi-transparent rendering – bring some disadvantages for the analysis of the fMRI data. The first strategy of clipping encounters the problem that important parts of the data may be clipped away and that the user can not get an overview of the functional activation in its entire anatomical context. When semi-transparency is employed, one has to cope with the trade-off between the visibility of the brain activation and the perceptibility of the brain surface.

In this section an alternative rendering strategy is presented that aims to overcome these drawbacks. The idea is to reduce the occlusion effects of a semi-transparent isosurface by replacing its surface representation by a sparser line representation. The lines are chosen along the principal curvature directions of the isosurface and are rendered by a flow visualization method that is called *line integral convolution* (*LIC*). The application of the LIC algorithm results in fine line structures that improve the perception of the isosurface's shape in a way that permits the rendering with small opacity values. To achieve high performance, the curvature vectors of the brain dataset are precomputed and stored in a separate 3D texture, and the LIC computation is performed only on the current visible surface.

The proposed technique is inspired by the work of Interrante [54] who precomputes the complete LIC integral at the voxel positions of the volume dataset. Hadwiger *et al.* [47] also apply LIC for curvature visualization on an isosurface but compute the curvature on the fly for the previously projected isosurface. In the following, first the mathematical and algorithmic basics for curvature and LIC computation are introduced. Then, it is shown how deferred LIC computation can be integrated into the multi-volume rendering framework, and, finally, it is presented how these techniques are applied to the visualization of SPMs.

Mathematical and Algorithmic Basics

Surface Curvature The *first principal (curvature) direction* is defined as the direction along the highest curvature on a surface. The corresponding curvature strength is called *first principal curvature*. The second principal direction indicates the direction to the flattest area. By construction, both principal directions are perpendicular to the surface normal. The two principal directions and the corresponding curvatures can be obtained by computing the eigenvalues of the second fundamental form and the corresponding eigenvectors [64; 87].

In the following, it is assumed that the surface is defined as an implicit isosurface of a 3D scalar field. First, an orthogonal frame $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ is constructed (see Figure 4.11). One basis vector is defined as $\mathbf{e}_3 = \nabla s(\mathbf{p})$, where s is the scalar value at the position $\mathbf{p}$. The vector $\mathbf{e}_3$ is the normal vector on the isosurface. The basis vector $\mathbf{e}_1$ is chosen within the plane that is perpendicular to $\mathbf{e}_3$; the direction within that plane can be chosen arbitrarily. The remaining basis vector is computed as $\mathbf{e}_2 = \mathbf{e}_1 \times \mathbf{e}_3$ and also lies in the plane that is perpendicular to $\mathbf{e}_3$. Then, the second fundamental form can be formulated as the matrix

$$A = \begin{bmatrix} \widetilde{\omega}_1^{13} & \widetilde{\omega}_1^{23} \\ \widetilde{\omega}_2^{13} & \widetilde{\omega}_2^{23} \end{bmatrix}, \tag{4.11}$$

where $\widetilde{\omega}_j^{i3}$ represents the deflection in the direction of $\mathbf{e}_i$ when one moves along $\mathbf{e}_j$.

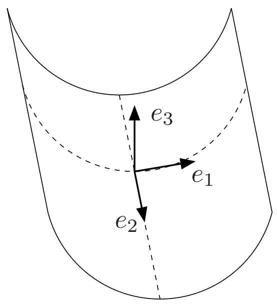


Figure 4.11: Orthogonal frame for the computation of the surface curvature. Here, e_1 stands for the first principal direction, e_2 for the second principal direction, and e_3 for the surface normal.

These values are formally known as *twists* if $i \neq j$ and can be obtained by the dot product of $\mathbf{e}_i$ and the derivative of the gradient in $\mathbf{e}_j$ direction:

$$\widetilde{\omega}_j^{i3} = \mathbf{e}_i \cdot \frac{\partial \mathbf{e}_3}{\partial \mathbf{e}_j} \tag{4.12}$$

Note that the twist terms $\widetilde{\omega}_2^{13}$ and $\widetilde{\omega}_1^{23}$ need to be equal. In order to compute the eigenvalues of A, the matrix is first rotated arround $\mathbf{e}_3$ until the twist terms disappear. This is done by diagonalizing A to obtain D_A:

$$A = SD_AS^{-1} = \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}^{-1}, \tag{4.13}$$

where $(u_i, v_i)^T$ denote the eigenvectors and κ_i with $i \in [1, 2]$ stands for the first and the second eigenvalues of A. Finally, the principal directions are given by

$$\mathbf{x_i} = u_i\mathbf{e_1} + v_i\mathbf{e_2} \qquad \text{with } |\mathbf{e_1}| = |\mathbf{e_2}| = 1. \tag{4.14}$$

Line Integral Convolution The line-integral-convolution (LIC) algorithm [17] is the basis for one of the most common texture-based techniques in flow visualization. This method uses an integration along curves defined by a vector field. The curves are constructed by tracing the motion of particles along the vector field, leading to streamlines in the case of a steady (i.e., stationary) vector field. Only steady vector fields are considered because the principal curvature directions will not change after the isosurface has been extracted. First, LIC on 2D planes is discussed and, later, it is extended to curved surfaces.

The particle tracing integrates the position of a particle n positive and n negative steps along the vector field, starting at the seed position $\mathbf{p} = (x_0, y_0)^T$. The resulting streamline is denoted by $\phi_0(t)$ and describes the position of a particle at a varying curve parameter t. Note that $\phi_0(0) = \mathbf{p}$ represents the seed point. $T(x, y)$ stands for the input noise texture and $k(t)$ denotes the convolution filter. Then, the intensity I at $\mathbf{p}$ is defined as the convolution along the streamline:

$$I(\mathbf{p}) = \int_{-L}^{L} k(t)\ T(\phi_0(t))\, dt\ . \tag{4.15}$$

For the filter $k(t)$, usually a symmetric function is used, e.g. a box or a tent function. The noise texture $T(x, y)$ contains (filtered) white noise. Applying Equation (4.15), the high spatial frequencies along the streamlines are reduced or completely removed, while maintaining the high frequencies perpendicular to those lines. This leads to line-like visual patterns.

The work presented here applies an adaption of the LIC algorithm for the visualization of flow on curved surfaces, which was developed by Weiskopf and Ertl [141]. One of the most important advantages of this method is that it is independent of the surface parameterization. Actually, no parameterization is necessary, what makes the approach appropriate for the application on implicit isosurfaces. The algorithm consists of two stages: (1) the projection of the surface and its corresponding vector field to the image plane and (2) the LIC computation on the image plane. This method computes each component in its appropriate domain and facilitates the evaluation of the LIC integral, which is computed on a planar 2D domain only for the visible parts of the object. Furthermore, this algorithm is well suited for an efficient GPU implementation.

In the following technical discussion, image-space coordinates are used as a representation of positions on the image plane. In addition, the original 3D object space of the isosurface is considered. An object can be transformed from object space to image space by applying a projection onto the image plane.

The idea of this algorithm is to evaluate the LIC integral (Equation 4.15) on a per-pixel basis with respect to the image space. In addition, the particle traces are also represented in 3D object space in order to achieve temporal coherence under camera rotations. To exploit the advantage of a combined image-space and object-space representation, the particle paths are simultaneously computed in both domains. The particle path in object space is obtained by solving the particle tracing equation,

$$\frac{\mathrm{d}\mathbf{p}_{\text{obj}}(t)}{\mathrm{d}t} = \mathbf{v}(\mathbf{p}_{\text{obj}}(t))\ , \tag{4.16}$$

where $\mathbf{v}(\mathbf{p}_{\text{obj}})$ denotes the vector field on the surface, and $\mathbf{p}_{\text{obj}}(t)$ denotes the object-space position of the particle at integration time t. This equation is solved by applying an explicit numerical integration, such as a first-order Euler scheme. After each integration step, $\mathbf{p}_{\text{obj}}$ is projected to image-space coordinates $\mathbf{p}_{\text{img}}$. The 3D object-space position $\mathbf{p}_{\text{obj}}$ is used to access the noise field T, which is also defined in 3D object space. The noise contributions are accumulated according to Equation (4.15) in order to obtain the final LIC result.

To simplify the representation of, and access to, the vector field $\mathbf{v}$, the 2D image-space position $\mathbf{p}_{\text{img}}$ is used to access the vector field. In fact, the vector field is not stored with respect to 3D object space, but with respect to 2D image space. Therefore, the image-space representation of the vector field needs to be initialized before the LIC integral is computed; namely, the vector field is projected from its original object-space representation onto the image plane. Then, particle tracing is based on the slightly modified equation

$$\frac{\mathrm{d}\mathbf{p}_{\text{obj}}(t)}{\mathrm{d}t} = \mathbf{v}(\mathbf{p}_{\text{img}}(t)) , \tag{4.17}$$

where $\mathbf{p}_{\text{img}}$ is computed from $\mathbf{p}_{\text{obj}}$ by projection onto the image plane.

Deferred Multi-Volume Shading

Deferred shading is a rendering technique that decouples the determination of the visibility of a fragment from performing shading operations for actually visible fragments. In a first rendering pass the 3D positions of the fragments and additional information, like gradients, curvature vectors, etc., are written to a so-called *G-buffer* (geometry buffer) [114]. The G-buffer is usually realized by one or several offscreen render targets. In a second render pass a screen-filling quad is rendered with the G-buffer render targets bound as input textures. At each fragment the related geometry information is read from the G-buffer and then shading operations, for example illumination, are performed. The advantage of this technique is that expensive shading operations are only performed for visible surface fragments. On the other hand, deferred shading techniques can only be applied to opaque surfaces where the position of a pixel in 3D object space is unambiguously defined.

The above introduced LIC algorithm for flow visualization on curved surfaces utilizes the deferred shading approach to minimize the cost for the expensive evaluation of the LIC integral. In the first rendering pass the currently selected isosurface is rendered opaquely. At each visible surface fragment the 3D position and additionally the associated vector from the 3D vector field is stored in the G-buffer. In the second pass the LIC computation is only performed for the visible surface fragments.

For the application case of visualizing functional activation maps in combination with the curvature-emphasized brain surface the deferred LIC computation has to be combined with standard multi-volume rendering. Therefore, deferred shading can be integrated into the multi-volume rendering framework from Chapter 3. The basic idea is to separate the rendering process into three passes (see Figure 4.12). In the first pass G-buffers for each applied volume are generated, in the second pass intermediate rendering results for the different volumes are computed, and in the third pass the intermediate results are merged to a single image. To achieve valid visualization results, one has to ensure that the rendered fragments of the different volumes either all lie on the same surface, or that they can be easily sorted in correct depth order. When no deferred shading is needed, the intermediate image of a volume can be alternatively rendered directly without the generation of a G-buffer.

For the first and second pass two different render graphs for the same multi-volume scene are applied. The first render graph describes the generation of the G-buffers,

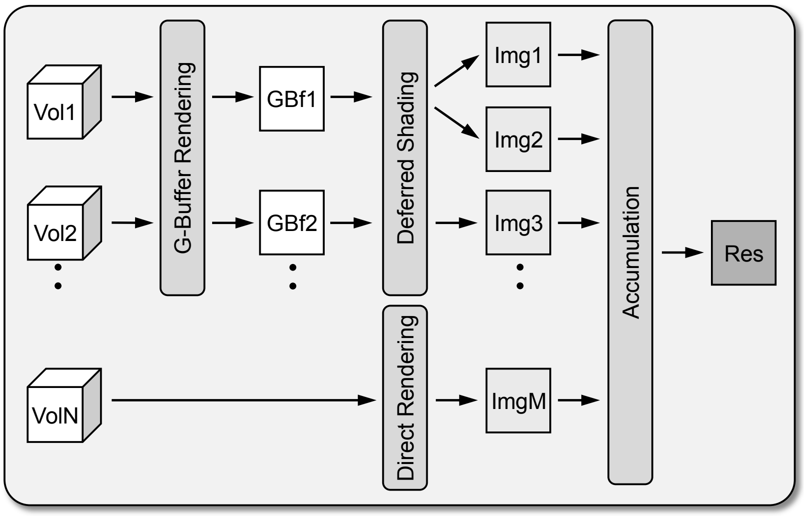


Figure 4.12: Pipeline of deferred multi-volume shading. First the G-buffer – 3D object-space position, gradient, curvature, etc. – is generated independently for each volume. Then, deferred shading operations are performed on the G-buffers. Finally, the intermediate images for single volumes are accumulated to a fused result image. The visualization rules for both, G-buffer rendering and deferred shading, are determined by a render graph. When no G-buffer is needed, the intermediate images can be generated directly.

e.g. along an isosurface. In contrast to the standard rendering process, the generated shaders do not write a single output color but have several output values, like the current 3D position or the current surface normal, which are written to the related G-buffers. For this purpose, the standard multi-volume rendering module has to be extended in a way that the results are written to several offscreen render targets. Further on, the shader generation module has to support the generation of shaders with several output variables.

The second render graph is responsible for the evaluation of complex shading operations on the previously computed G-buffers. Here, any shader node that takes one or several pre-computed G-buffer values into account can be applied. Instead of computing these values from the underlying volume dataset, they are read from the G-buffer. Therefore, the *Scene Node*, namely the sub-node which is responsible for the volumes (see Section 3.3.1), is adapted in a way that it hides the G-buffer from subsequent shader nodes. Thus, the same shader nodes like for standard multi-volume rendering can be applied. The deferred shading process can be performed in one or several render passes. Basically, a screen filling quad is rendered with the related G-buffers as input. Either a combined shader writes several output colors to several render targets in a single pass, or for each intermediate image a separate shader is used. Alternatively, if only one shader is used, the accumulation can be directly integrated into this shader. Then no additional accumulation pass is needed.

Applying Curvature LIC to SPM visualization

In the context of SPM visualization the LIC algorithm for curved surfaces is used for the accentuation of the brain surface structure along the first principle curvature direction, while simultaneously the occlusion of the activation data is minimized. The applied vector field gives the first principle (curvature) direction for each point in the 3D volume. This vector field is precomputed and stored in a 3D texture. Curvature vectors at intermediate positions are trilinearly interpolated from the curvature vectors at the surrounding voxels.

In the final image three different visualization styles are combined. The SPM activation map is rendered with standard DVR, and the anatomical brain surface is rendered once as illuminated opaque isosurface and once as isosurface with LIC along the first principal direction. For both isosurfaces the same isovalue is applied. Activation data outside the visualized brain surface is cut away. Thus, it is possible to render the SPM activation separately and to store it in an intermediate image for accumulation. For the rendering of the illuminated isosurface and of the LIC isosurface deferred shading is applied. A single G-buffer for the anatomical brain dataset is generated. There the 3D object-space positions of the foremost fragments of the isosurface, the associated gradient vectors of the pre-computed gradient texture, and the associated first-principle-direction vectors of the pre-computed curvature texture are stored. Two different shading operations are performed on the G-buffer. On the one hand, an intermediate image is generated by applying standard Phong illumination to the surface. On the other hand, the LIC algorithm for curved surfaces, as described above, is evaluated.

The three intermediate images are intermixed with a special blending function that takes the lines obtained from the LIC computation, the illuminated isosurface, and the DVR rendered brain activation into account. The blending function is defined as

$$\mathbf{C}'_{\text{out}} = \alpha \mathbf{C}_{\text{iso}} + (1-\alpha)(1-I_{\text{LIC}})\mathbf{C}_{\text{act}} \ , \tag{4.18}$$

and describes the final output color $\mathbf{C}'_{\text{out}}$. The alpha blending between the functional data $\mathbf{C}_{\text{act}}$ and the anatomical data $\mathbf{C}_{\text{iso}}$ is governed by the adaptable opacity α of the isosurface, i.e. the opacity of the isosurface determines the visibility of the brain activation behind it. Figure 4.13 illustrates this blending process.

The intensity of the curvature lines, I_{LIC}, further modifies the image compositing. According to the multiplication by the factor $(1 - I_{\text{LIC}})$, the LIC intensity provides different weights for the brain activation. Since dark lines should be obtained, it is necessary to negate the intensity. Furthermore, empty areas inside the anatomical hull have to be considered. Black empty regions would lead to a multiplication by zero, which means that in these areas no curvature lines would be visible. In order to apply the curvature lines also in these regions, the brain activation data is rendered using a white background.

The brain structure can be further emphasized by applying illumination based on the curvature LIC represention, as proposed by Schafhitzel *et al.* [117]. Then the data

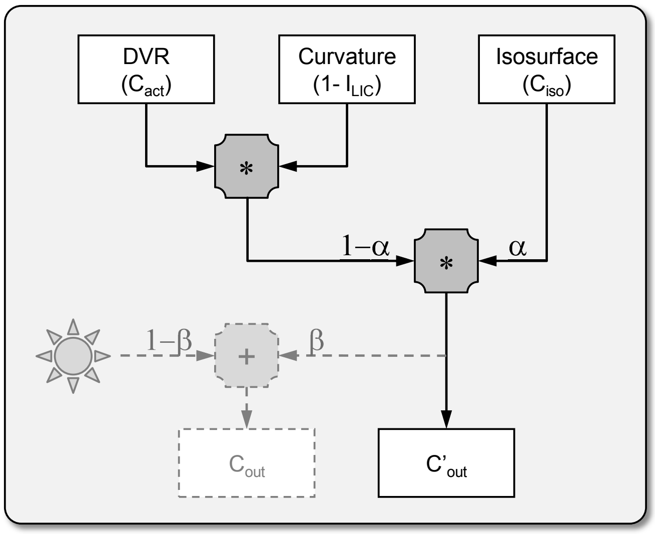


Figure 4.13: Blending of the functional data rendered by DVR and the surrounding anatomical data using isosurface shading. The curvature lines resulting from the LIC computation affect the DVR rendering by defining the intensity of the functional data as well as the isosurface shading of the anatomical data due to their tangential behavior. If curvature line illumination is enabled, a second blending is applied which adds the light contribution to the emissive representation.

flow changes slightly (see Figure 4.13). Lighting is based on normal vectors on the surface geometry, which can be related to the gradients of a texture-based representation of geometry. For curve illumination a real-time gradient computation for the curvature lines is employed. Since the curvature lines are computed on the image plane in a view-dependent way, a pre-computation of the gradients is not possible. Furthermore, it has to be considered that the first derivative requires neighborhood information. The gradient computation is implemented as an additional deferred shading pass directly after the LIC evaluation, which delivers the intensity values for each pixel in image space. This image can be considered as a 2D scalar field that serves as input for the gradient computation. Central differences are used to compute the 2D image-space gradient, which is combined with the surface normal to obtain the 3D normal vector in world space. In the final rendering step, this vector field is employed to apply diffuse illumination of the curvature lines. This illumination component extends Equation (4.18) to

$$\mathbf{C}_{\text{out}} = \beta(\mathbf{N} \cdot \mathbf{L}) + (1 - \beta)\mathbf{C}'_{\text{out}} , \tag{4.19}$$

where $\mathbf{N}$ stands for the normal vector and $\mathbf{L}$ denotes the position of the light source. Both vectors are given in world space. Usually, only a small light contribution is sufficient for emphasizing the line structures on the isosurface. From experience, β should be chosen between 0.1 and 0.2.

Discussion

In the following, the application of the presented method to SPM data is discussed. We compare the visualizations using different parameter settings. The parameters are: curvature masking, noise density, and curve illumination. Figure 4.14 (a) shows an example of the combination of an illuminated surface and DVR. In this image, the volume-rendered brain activation serves as focus while the surrounding brain tissue is represented by an isosurface, which builds the corresponding context. Obviously, the main goal of this visualization is to facilitate the spatial perception of the activation areas inside the human brain. Therefore, it is necessary to have a clear visualization of both objects. In particular, the shape of both objects should be perceivable at a glance. Actually, the visualization quality depends on the materials of the surrounding context and the focus object. These materials are usually defined by transfer functions, and so it depends on the user to find an appropriate setting for an optimal visualization. Obviously, the visualization quality is strongly influenced by the rendering of the context. For example, if the isosurface is rendered opaque, the structure of the anatomy is rendered in high quality, but it completely occludes the brain activation. On the other hand, if the isosurface is chosen too transparent, the shape of the covered brain activation is clearly identifiable, whereas the quality of the isosurface suffers.

In Figure 4.14 (b), a naive mapping of the computed curvature lines onto a selected isosurface is applied. Indeed, the structure of the isosurface is clearly perceptible, supported by the curvature lines. Nevertheless, the high number of lines drawn on the surface makes it hard to identify the shape of the brain activation behind. Furthermore, areas of low curvature, like the depressions on the cortex surface (sulcus), lead to short lines, which negatively influence the visualization. In Figure 4.14 (c), the issue of occlusion is addressed by changing the noise intensity that influences the number and the size of the drawn LIC lines. By decreasing the number of lines, the isosurface becomes more transparent and the quality of the brain activation shape increases. The problem of noise in flat areas is solved by curvature masking, which is based on a threshold that masks lines in areas of low curvature. As a consequence of this masking, the lines in the *sulci* disappear while the shape of ridges (*gyri*) are emphasized by line drawing.

Figure 4.14 (d) shows the result if the density of the lines on the isosurface is further decreased. It is necessary to consider the disadvantages that might appear if the density is chosen too small: due to the behavior of our LIC algorithm, the resulting lines are a weighted intensity of the brain activation. Actually, the weights of the streamlines are chosen with a positive offset to avoid black lines, which might affect the visualization. If the number of lines is decreased too much, either the intensity offset must be reduced or the perception of the line structures has to be improved. Indeed, the first option would lead to a higher contrast, but it would also imply a higher degree of occlusion caused by opaque lines. An alternative option is to improve the perception of line structures by illumination (Figure 4.14 (d)). Please note that already a small contribution from illumination is sufficient for an improved perception

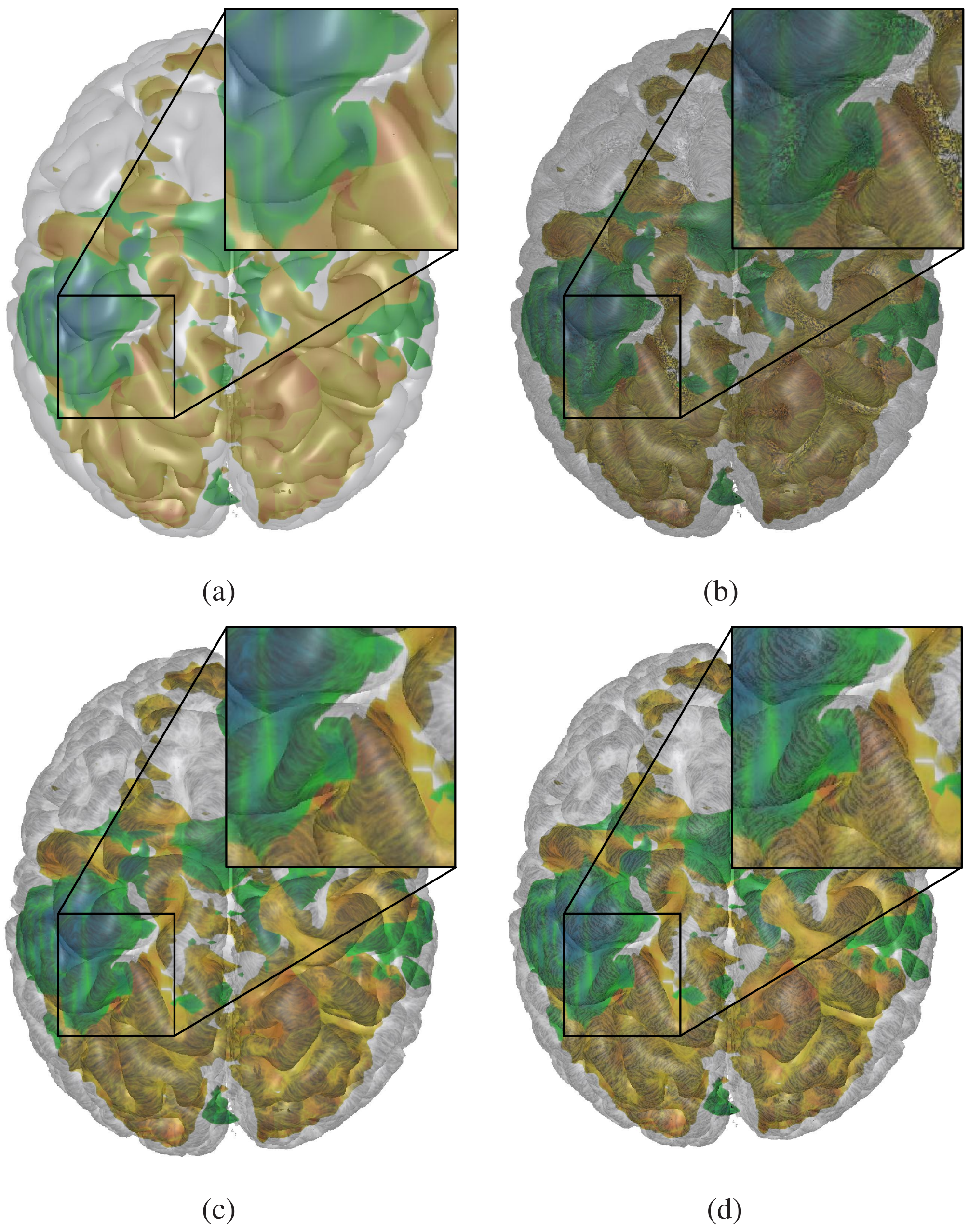


Figure 4.14: Combined visualization of functional and anatomical data. The brain activation is rendered with DVR while the anatomical brain structure is represented by an illuminated isosurface: (a) Isosurface without any line structures mapped onto it. (b) A high number of curvature lines without any masking; the short lines appear as points and influence the visualization negatively. (c) Smaller number of thicker curvature lines; in the areas of low curvature, the curvature lines are faded out completely. (d) Illuminated curvature lines. Only a small diffuse light contribution is used for emphasizing the lines' structure. The curvature lines appear more prominently without having changed their intensity.

w/o LIC	with LIC	with illum. LIC
42.17 fps	25.67 fps	25.57 fps

Table 4.2: Performance measurements of a combined visualization of DVR rendered brain activation and the anatomical brain surface, which is rendered as illuminated isosurface. Three different configurations for isosurface rendering are compared. All measurements are given in frames per seconds (fps), the viewport size is 800×600.

of the lines. If the diffuse part is emphasized too much, the lines appear as bumps on the isosurface, which makes it difficult to distinguish between the original curvature given by the isosurface and the visual ridges created by illumination.

Table 4.2.3 shows the visualization speed of several configurations measured on a PC with an AMD Athlon 64 X2 Dual 4400+ (2.21 GHz) CPU and 2 GB of RAM, and a NVIDIA GeForce 8800 GTX GPU with 786 MB of graphics memory. Considering the first two entries of the table, the lower frame rates for LIC rendering can be explained by the evaluation of the LIC integral. In this case, 20 integration steps in each direction are computed, which results in 40 texture lookups for each fragment. In contrast to the LIC evaluation, the gradient computation barely influences the rendering speed. Gradients are computed by central differences. Therefore, only 4 additional texture lookups per fragment are necessary, which makes it much faster than the LIC evaluation.

4.3 GPU-based Direct Volume Deformation

Fast and realistic soft-tissue deformation is an important feature for medical simulation environments. Those environments simulate specific surgical interventions and are for example applied in medical education. Physically-based deformation approaches, like *mass-spring systems* or *finite element methods* (*FEM*), usually employ an explicitly generated model, which is deformed by expensive computations. For visualization the deformed model has to be additionally transformed into a renderable representation. Altogether, the deformation process is very complex and time-consuming.

In this section a deformation technique is presented that aims to overcome these drawbacks in two ways. On the one hand, it directly acts on the originally acquired volume data of a patient. Thus, there is no need for expensive preprocessing. On the other hand it is completely performed on the GPU and seamlessly embedded into the standard volume visualization framework. This exploits the fast parallel computing capabilities of modern graphics hardware and avoids the expensive transfer of deformation information to the GPU for visualization. The proposed deformation approach is based on the physically-inspired *3D-ChainMail* algorithm, which was originally developed by Frisken-Gibson [36]. This algorithm acts on a regular grid of deformation elements. The basic idea is to iteratively propagate the displacement of an initially manipulated element across the grid. The displacement of the grid elements is thereby governed by several geometrical constraints. The advantages of the ChainMail technique are its low computational costs and the possibility to directly apply it to regular volume datasets. For GPU-based deformation the ChainMail algorithm was adapted for parallel execution and embedded into an integrated deformation and visualization pipeline.

The interactive manipulation and deformation of volumetric objects is an active research field. Chen *et al.* [20] give a good overview about this topic. They present several geometrical and physically-based approaches and show their usage in medical and non-medical applications. Schulze *et al.* [120] proposed a volume deformation technique that also employs the ChainMail algorithm. But in contrast to the approach presented here, the deformation is performed on the CPU, and the volume data on the GPU has to be updated after each deformation step. There is also some work on deformation computation on the GPU. E.g., Georgii *et al.* [40] presented a GPU-based mass-spring deformation system, which they combined with standard surface rendering. Tejada and Ertl [126] proposed a similar approach but perform direct volume rendering on a deformed tetrahedral grid.

The GPU-based volume deformation technique presented in this section was first published in [111]. The work was carried out in collaboration with Torsten Wolff, who realized the initial GPU-implementation of the ChainMail algorithm during the preparation of his diploma thesis. In the following, first, the original ChainMail algorithm is described; then, it is shown how the ChainMail deformation and the visualization of the deformed volume can be mapped to the GPU; finally, the GPU-based pipeline of manipulation, deformation and visualization is presented in detail and some results and performance measurements are shown.

4.3.1 3D ChainMail Algorithm

The 3D ChainMail algorithm [36] operates on deformation elements that are initially arranged on a three-dimensional regular grid. Each element is connected to its six nearest neighbors in x-, y-, and z-direction (see Figure 4.15 left). The relative position of a grid element to its neighbors is governed by several constraints that give the minimally ($minD_{x,y,z}$) and maximally ($maxD_{x,y,z}$) allowed distance and the maximally allowed shear ($maxS_{x,y,z}$) (see Figure 4.15 right). To simulate anisotropic deformation, these limits can differ along the three coordinate directions.

The algorithm works as follows. Starting with a single moved grid element, the neighbors of this element are checked if they still satisfy the ChainMail constraints. If not, the affected neighbors are minimally moved to fulfill the constraints again. For example, the new position of the left neighbor is determined in the following way:

$$\begin{array}{ll}
\text{if } (x - x_l) < minD_x, & x_l = x - minD_x; \\
\text{else if } (x - x_l) > maxD_x, & x_l = x - maxD_x; \\
 & \\
\text{if } (y - y_l) < -maxS_y, & y_l = y + maxS_y; \\
\text{else if } (y - y_l) > maxS_y, & y_l = y - maxS_y; \\
 & \\
\text{if } (z - z_l) < -maxS_z, & z_l = z + maxS_z; \\
\text{else if } (z - z_l) > maxS_z, & z_l = z - maxS_z;
\end{array} \tag{4.20}$$

$\mathbf{x} = (x, y, z)^T$ is the position of the currently investigated element, $\mathbf{x_l} = (x_l, y_l, z_l)^T$ is the position of the left neighbor. The displacement of the other five neighbors is done analogously. Then the algorithm goes on with the newly moved elements and tests their neighbors in the same way. This procedure is repeated until the list of moved elements is empty.

Frisken-Gibson has shown that each ChainMail element has to be touched only once if the ChainMail constraints are constant throughout the volume. However, the first-moved-first-processed propagation order does not create valid grid configurations for inhomogeneous deformation constraints. For this reason, Schill *et al.* [118] presented an enhanced ChainMail algorithm that first processes the elements with the largest displacement. This propagates the deformation information faster through stiffer material, analogous to the broadening of a shockwave.

After applying the ChainMail deformation, the grid elements fulfill the geometric ChainMail constraints, but the linear displacement does not adequately simulate natural soft body behavior. For this reason, Frisken-Gibson introduced an additional relaxation step that moves the grid elements to an energetic minimum. Thereby, the systems energy depends on the distances between the grid elements. There are several

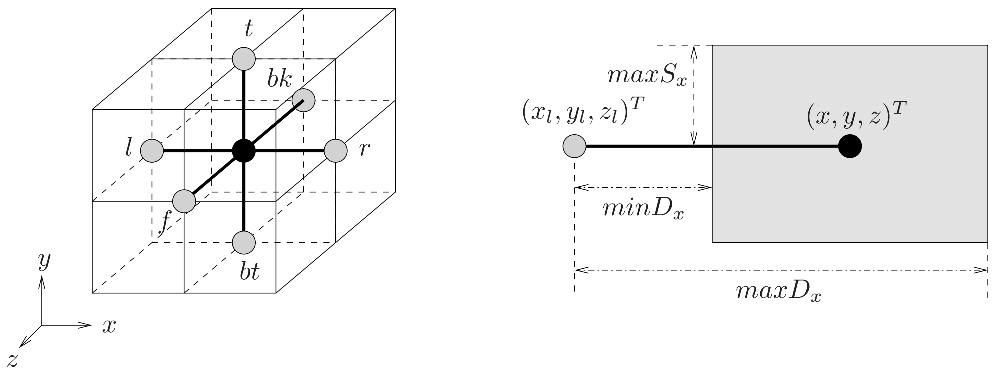


Figure 4.15: The 3D ChainMail grid: (left) the six neighbors (gray) of a grid element (black); (right) the area (grey rectangle) of valid positions of an element (black) relative to its left neighbor (grey).

different ways to determine the optimal grid positions for which the energy of the grid is minimal. In [37] Frisken-Gibson proposed to define the optimal position $(x, y, z)^T_{opt}$ of an element as the midpoint of its existing neighbors:

$$(x, y, z)_{opt} = \left(\frac{1}{N} \sum_{nghbrs} x_n, \frac{1}{N} \sum_{nghbrs} y_n, \frac{1}{N} \sum_{nghbrs} z_n \right), \tag{4.21}$$

where N is the number of existing neighbors of the element and $(x_n, y_n, z_n)^T$ is the position of the n-th neighbor. During relaxation each element is iteratively replaced to minimize the system's energy. However, with this method border elements tend to move inward and the object shrinks. To avoid this, Frisken-Gibson has reformulated the iterative relaxation in the following way:

$$(x, y, z)_{opt} = \frac{1}{N} \left(\sum_{nghbrs} (x_n - \Delta x_n), \sum_{nghbrs} (y_n - \Delta y_n), \sum_{nghbrs} (z_n - \Delta z_n) \right),$$

$$\begin{aligned}
\Delta x_n &= \begin{cases} -\Delta x, & n = l \\ +\Delta x, & n = r \\ 0, & \textit{all other neighbors} \end{cases} \\
\Delta y_n &= \begin{cases} -\Delta y, & n = bt \\ +\Delta y, & n = t \\ 0, & \textit{all other neighbors} \end{cases} \\
\Delta z_n &= \begin{cases} -\Delta z, & n = bk \\ +\Delta z, & n = f \\ 0, & \textit{all other neighbors} \end{cases}
\end{aligned} \tag{4.22}$$

where Δx, Δy and Δz are the optimal link lengths for left(l)/right(r), top(t)/bottom(bt), and back(bk)/front(f) neighbor pairs. This method produces the same results as the midpoint method for inside elements (Equation 4.21) but prevents border elements from moving inwards.

4.3.2 Mapping ChainMail Deformation to GPU

The goal of mapping the ChainMail deformation approach to GPU is two-fold. On the one hand, the performance should be improved by exploiting the parallel SIMD architecture of a GPU. On the other hand, the deformation process should be directly integrated into the GPU-based volume visualization pipeline. Therefore, it is required to find an adequate GPU representation of the ChainMail deformation grid, to adapt the serial ChainMail algorithm for parallel execution and to solve the problem of direct visualization of the deformed volume dataset.

GPU Representation of the Deformation Grid

The ChainMail elements are initially arranged in a regular grid and linked to their six direct neighbors along the x-, y, and z-axis. Thus, it is obvious to store the positions of the grid elements in a 3D texture on the GPU and to exploit the implicitly given neighborhood to the surrounding texels. To be independent of the volume's real extent, the element positions are stored relative to a normalized cube with edge length one. Thereby, a grid element's initial position is implicitly given by the texture coordinates of the associated texel. Further on, the resolution of the deformation grid can be chosen independently of the original volume dataset.

In the original ChainMail implementation the neighbors of an element are explicitly declared to allow the explicit definition of unregular object borders. However, if ChainMail deformation is combined with direct volume visualization, the shape of the deformed object is already implicitly given by the applied opacity transfer function. To avoid the explicit storage of neighborhood information, the object shape is instead dynamically determined by looking up the opacity of a grid element during deformation. If the opacity lies beyond a small threshold, it is assumed that the grid element does not belong to the visible structure, and it is ignored for further computations. In case that the deformation grid has a smaller resolution than the original volume dataset, an corresponding low-resolution volume dataset is computed by mipmap filtering, which provides a down-sampled presentation of the object shape.

For the simulation of inhomogeneous deformation behavior of different anatomical and pathological structures, a deformation transfer function is introduced that maps the intensity values of the underlying volume to the ChainMail constraints described in Section 4.3.1. For simplicity the transfer function maps an incoming intensity value i to a single normalized deformation constraint c in the interval $[0..1]$. The normalized deformation constraint $\bar{c}$ of a link between two neighboring elements is computed by averaging the deformation constraints c_1 and c_2 that are applied to the the two link elements:

$$\bar{c} = \frac{1}{2} \cdot (c_1 + c_2) \tag{4.23}$$

From the normalized deformation constraint $\bar{c}$ the final constraints for the minimal distance ($minD$) between two elements, the maximal distance ($maxD$) and the maximal shear ($maxS$) are computed depending on the initial distance d between two neighboring elements:

$$\begin{aligned} minD &= d \cdot (1 - \bar{c}), \\ maxD &= d \cdot (1 + \bar{c}), \\ maxS &= d \cdot \bar{c}. \end{aligned} \tag{4.24}$$

For the determination of the deformation constraints again the low-resolution volume datasat is used to look up a grid element's intensity value.

GPU-based ChainMail Algorithm

The original ChainMail algorithm processes the grid elements in a fixed serial order. This ensures that each element has to be touched only once. In contrast to that, a GPU shows its strength only if a huge number of elements are processed in parallel. For this reason, the ChainMail deformation is carried out in several consecutive passes. In each pass all elements of the grid are investigated. Therefore, the capability of modern GPUs (*Nvidia G80* series and newer) to directly render into a 3D texture is exploited. Namely, the deformation grid texture is processed slice-by-slice. In a loop one texture slice after the other is bound as render target and the element positions are updated by repeated rendering of screen-filling quads, as described in Section 2.3.2 about GPGPU rendering. To avoid inconsistencies during the parallel processing of the elements the deformation grid texture is duplicated, and the two textures are used alternately for reading and writing (*ping-pong rendering*).

Since the parallel pipeline of a GPU does not permit to write to any other fragment than the currently processed, the way of deformation propagation is inverted. Instead of displacing neighbor elements of the currently processed element, the element itself is displaced in case it violates a deformation constraint to any of its neighbors. Therefore, it is is first checked if one of its neighbors was moved in the previous pass, and, in the positive case, the validity of the deformation constraints relative to the respective neighbor is tested. If requested, the element is moved to a valid position due to Equation 4.20. Besides the speedup, the repeated investigation of all grid elements in parallel has the advantage that a single grid element can be displaced multiple times. Thus, the proposed approach automatically copes with inhomogeneous deformation constraints. The ChainMail deformation is stopped when no more element was moved in the current render pass.

The relaxation step can be performed on the same grid textures like the ChainMail deformation; again in several render passes. In each pass the position of the grid elements is updated due to an arbitrary relaxation rule, e.g. the one given in Equation 4.22.

The relaxation step can be stopped when the length of the displacement vector is falling below a small threshold for all elements. Then an energetic equilibrium is reached.

Direct Visualization of Deformed Volumes

The ChainMail deformation grid gives for a ChainMail element e that was originally placed at position $\mathbf{x_e}$ its new position $\bar{\mathbf{x}}_\mathbf{e}$ after deformation. The image position $\Phi(\mathbf{x})$ of an arbitrary point $\mathbf{x}$ inside the undeformed grid can be computed by trilinear interpolation:

$$\Phi(\mathbf{x}) = \sum_{i,j,k\in\{0,1\}} a_{ijk}(\mathbf{x}) \cdot \tilde{\mathbf{x}}_\mathbf{ijk} \tag{4.25}$$

where $\tilde{\mathbf{x}}_\mathbf{ijk}$ are the displaced positions of the eight surrounding grid elements of the point $\mathbf{x}$ and $a_{ijk}(\mathbf{x})$ are the respective trilinear interpolation weights obtained for position $\mathbf{x}$ from the original grid.

Based on the deformation grid, there are three different ways for rendering the deformed volume. The first possibility is to perform a pre-rendering step that resamples the deformed volume on a regular grid, as it is done by Schulze *et al.* [120]. Here, for each sampling point of the new regular volume the cell of the deformation grid is searched in which the respective sampling point lies. The corresponding sample value is then interpolated from the sample values at the cell vertices. An advantage of this method is that standard volume rendering algorithms for regular grids can be applied. But on the other hand, information contained in the original undeformed volume texture can be lost by resampling. This problem especially occurs when the resolution of the deformation grid is smaller than the one of the original volume dataset.

This problem can be avoided by alternatively applying the deformation during rendering. This can be done either in model space or in texture space. For the model space approach the deformation grid is directly used as proxy geometry for rendering. To prevent interpolation problems, the hexahedral grid cells consisting of eight adjacent grid nodes have first to be subdivided into tetrahedra, and then any method for volume rendering of tetrahedral grids can be applied, e.g. the one presented by Weiler *et al.* [138]

In contrast to the model space approach, the texture space approach keeps the proxy geometry undeformed and the deformation is instead applied to the texture coordinates just before the lookup of a volume sample. Rezk-Salama *et al.* [103] developed a slice-based volume rendering technique that subdivides object-aligned slices into small quads. The vertices of the quads get deformed texture coordinates applied, which are automatically interpolated during rendering. To prevent interpolation errors, the quads are further subdivided into four triangles by appending an additional vertex in the center of each quad. Brunet *et al.* [15] proposed an alternative technique that computes the deformed texture coordinates on a per-sample basis inside a shader.

In this work a volume rendering technique for deformed volumes was developed that is based on the approach of Brunet *et al.*. The advantage of this method is the fact that it can be easily combined with any GPU-accelerated volume rendering technique without the need to adapt the applied proxy geometry. However, since the deformation is applied in texture space, the forward deformation function Φ, which gives the deformation in object space, has to be inverted. Brunet *et al.* solve this problem by applying analytical deformation functions that have a well-defined inverse function. When ChainMail deformation is employed, the forward deformation is given at discrete positions and the displacement at intermediate positions is computed by trilinear interpolation (see Equation 4.25). Unfortunately, there is no closed solution for the inversion of this function. For this reason, a gradient-decent-like optimization algorithm for the computation of an inverse deformation grid was developed. The goal of this algorithm is to generate a regular grid that gives for each sample point the position where it was placed before deformation. Since all computations are performed in a normalized cube with edge length 1, this position directly maps to the displaced texture coordinates that are needed for rendering.

The inverse deformation position $\hat{\mathbf{x}}_{\mathbf{e}}$ of a grid element e is computed iteratively. In the beginning, $\hat{\mathbf{x}}_{\mathbf{e}}$ is initialized with the original undeformed position $\mathbf{x}_{\mathbf{e}}$ of the element e:

$$\hat{\mathbf{x}}_{\mathbf{e_0}} = \mathbf{x}_{\mathbf{e}} \tag{4.26}$$

Then the following update rule is iteratively applied to the current inverse deformation position of e:

$$\begin{aligned} \mathbf{frwd}_{\mathbf{e_i}} &= \Phi(\hat{\mathbf{x}}_{\mathbf{e_i}}); \\ \mathbf{diff}_{\mathbf{e_i}} &= \mathbf{x}_{\mathbf{e_i}} - \mathbf{frwrd}_{\mathbf{e_i}}; \\ \hat{\mathbf{x}}_{\mathbf{e_{i+1}}} &= \hat{\mathbf{x}}_{\mathbf{e_i}} + w \cdot \mathbf{diff}_{\mathbf{e_i}}; \end{aligned} \tag{4.27}$$

First, the forward displaced position $\mathbf{frwd}_{\mathbf{e_i}}$ of the inverse deformation position $\hat{\mathbf{x}}_{\mathbf{e_i}}$ after the i-th iteration step is computed. For this purpose, the trilinear displacement function Φ defined in Equation 4.25 is applied. Then, the difference vector $\mathbf{diff}_{\mathbf{e_i}}$ between the desired image $\mathbf{x}_{\mathbf{e}}$ of $\hat{\mathbf{x}}_{\mathbf{e_i}}$ and the current image $\mathbf{frwd}_{\mathbf{e_i}}$ is calculated. Finally, a new inverse deformation position $\hat{\mathbf{x}}_{\mathbf{e_{i+1}}}$ is generated by moving $\hat{\mathbf{x}}_{\mathbf{e_i}}$ slightly in the direction of $\mathbf{diff}_{\mathbf{e_i}}$. The amount of this movement is governed by the weight w. The iteration is stopped, when the length of $\mathbf{diff}$ falls below a small threshold. Figure 4.16 illustrates the iterative computation of $\hat{\mathbf{x}}_{\mathbf{e}}$.

The proposed iteration scheme is similar to gradient-descent optimization along the gradient of the error function

$$err(x_{e_i}) = |\mathbf{diff}_{\mathbf{e_i}}|, \tag{4.28}$$

which represents the distance between the current forward displacement of x_{e_i}, $\Phi(\hat{\mathbf{x}}_{\mathbf{e_i}})$, and the expected displacement position $\mathbf{x}_{\mathbf{e}}$. For gradient descent $\hat{\mathbf{x}}_{\mathbf{e_i}}$ is adapted along the negative direction of the gradient of err:

$$\hat{\mathbf{x}}_{\mathbf{e_{i+1}}} = \hat{\mathbf{x}}_{\mathbf{e_i}} - w \cdot \nabla err(\mathbf{x}_{\mathbf{e_i}}) \tag{4.29}$$

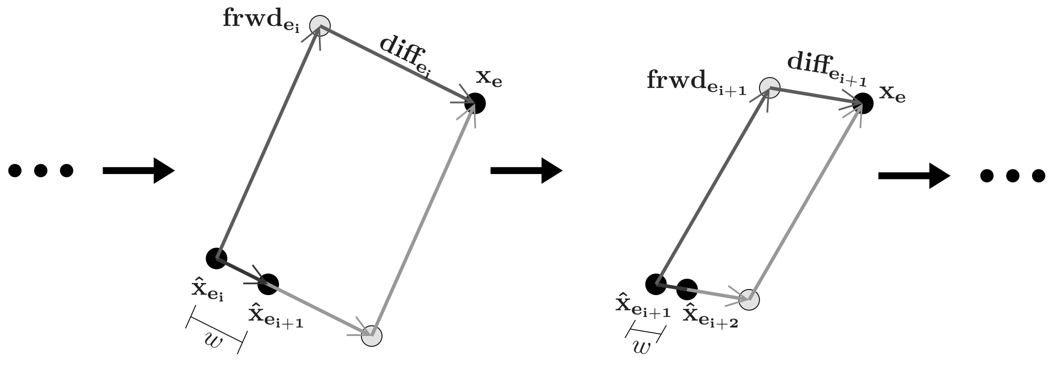

$\mathbf{frwd_{e_i}}$
$\mathbf{diff_{e_i}}$
$\mathbf{x_e}$
$\hat{\mathbf{x}}_{\mathbf{e_i}}$
$\hat{\mathbf{x}}_{\mathbf{e_{i+1}}}$
w
$\mathbf{frwd_{e_{i+1}}}$
$\mathbf{diff_{e_{i+1}}}$
$\mathbf{x_e}$
$\hat{\mathbf{x}}_{\mathbf{e_{i+1}}}$
$\hat{\mathbf{x}}_{\mathbf{e_{i+2}}}$
w

Figure 4.16: Iterative computation of the inverse deformation position $\hat{\mathbf{x}}_{\mathbf{e}}$ of a grid element e. In each step, first the forward displaced position $\mathbf{frwrd_{e_i}}$ of the current inverse position $\hat{\mathbf{x}}_{\mathbf{e_i}}$ is determined. Then, the difference vector $\mathbf{diff_{e_i}}$ between $\mathbf{frwrd_{e_i}}$ and the expected position x_e is determined, and $\hat{\mathbf{x}}_{\mathbf{e_i}}$ is slightly moved in this direction. This process is repeated until x_e is met.

This gradient descent leads to a local minimum of the error function. The gradient can, e.g., be computed by finite differences, which is an expensive operation that needs six additional lookups in the forward deformation grid. However, if it is assumed that the deformation behavior does only change slightly in a local area around $\hat{\mathbf{x}}_{\mathbf{e_i}}$, the difference vector $\mathbf{diff_{e_i}}$ represents a good approximation of the negative gradient, because the value of err is decreasing fast along $\mathbf{diff_{e_i}}$. Thus, the application of the iteration rule of Equation 4.29 will produce similar results like gradient descent with less cost.

4.3.3 GPU-based Deformation Pipeline

For evaluation of the GPU-based ChainMail approach a volume deformation pipeline was implemented that consists of five sequentially applied steps (see Figure 4.17). The implementation is based on OpenGL and GLSL and can be integrated easily into the multi-volume rendering framework presented in Chapter 3. Each pipeline step is realized by a specific GLSL shader. These shaders basically act on three data structures (3D textures): the original volume dataset, the forward deformation grid and the inverse deformation grid. For ping-pong rendering the forward deformation grid is duplicated. If the deformation grid has a lower resolution than the original volume dataset, an additional low-resolution volume dataset is generated by mipmapping. This low-resolution dataset is used for the determination of the deformation constraints. The data structures are either bound as input textures (read) or as render targets (write). In pipeline steps where the forward deformation grid is used for both, reading and writing, one of the two deformation grid textures is bound as input texture, and the other one is bound as render target. The roles of the two textures are changed after each render pass. In the following the five pipeline steps are explained in detail.

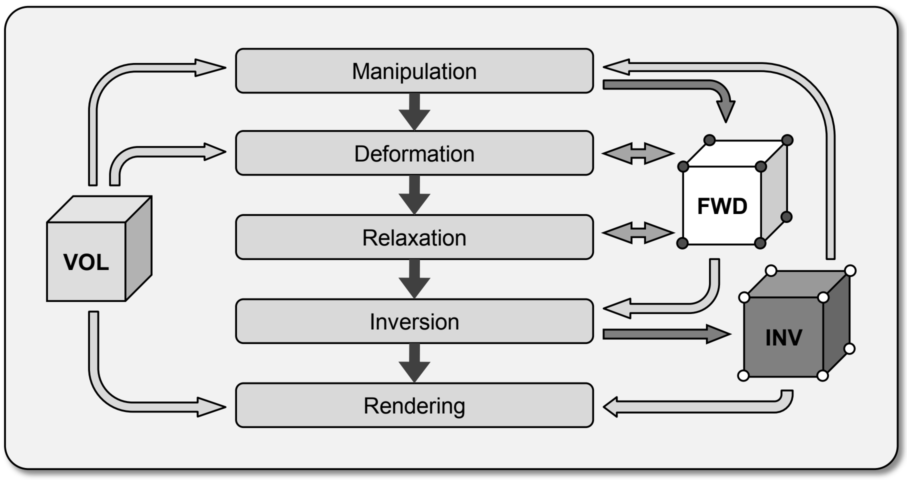


Figure 4.17: GPU-based deformation pipeline: The five pipeline steps access the three data structures – the original volume (VOL), the forward deformation grid (FWD), and the inverse deformation grid (INV), – either for reading, for writing, or for reading and writing.

Manipulation

In the manipulation step the current mouse movement is mapped into normalized 3D model space, and it is checked if a collision with the volume occurs. For this purpose, a single fragment is rendered, and the applied shader is casting a ray from the start to the end position of the mouse movement. At each sampling position, first, the deformed texture coordinate is looked up in the inverse deformation-grid texture. Then, the corresponding sample value is read from the original volume data set, and the related alpha value is looked up in the transfer function table. If the alpha value is greater than zero, a collision is found. In this case the deformation process is initialized by moving the eight grid elements of the forward deformation grid that surround the collision position accordingly to the mouse move.

Deformation

The ChainMail deformation step is realized in multiple passes. In each pass the forward deformation grid textures for reading and writing are interchanged. The current position of a grid element is stored in the x-, y-, and z-component of the related texel. The w-component is used to store the information if a grid element was currently moved. This information is used to evaluate if any neighbor of an investigated element was moved in the preceding pass. If not, the element is directly discarded. Otherwise, it is tested if the current element has to be moved to fullfill the chainmail constraints relative to a moved neighbor.

The chainmail deformation is stopped when no more elements are displaced in the current pass. This is tested by hardware supported occlusion queries which give the number of processed elements (fragments) in the current render pass. When all elements are discarded, the occlusion query returns zero and the deformation can be stopped. The deformation step can be optimized by exploiting the fact that it always starts with eight neighboring grid elements and that the region of possibly affected elements expands in a single pass about one along its six border faces. Thus, it is sufficient to investigate in each render pass only those elements that can be potentially moved, and to increase the region of investigated elements accordingly afterwards.

Relaxation

The relaxation step is performed in multiple passes as well. In each pass for each element the displacement is computed due to the relaxation rule presented in Equation 4.22. If the magnitude of the displacement is too small, the element is not moved and discarded instead. The relaxation is terminated when an per-pass occlusion query returns that no more element was processed.

For performance reasons, in each step only those elements are investigated that can be possibly moved. This is the case either if any of an element's neighbors or if the element itself was moved in the previous pass. In the first relaxation pass all elements that have been replaced sometimes during the deformation step are regarded as moved. Like for deformation, the region of possibly affected elements has to be increased in each pass by one along each border.

Inversion

The inverse displacement position of a forward grid element can be computed independently of the inverse displacement positions of the other elements. Thus, the grid inversion can be performed in a single rendering pass, while the iteration loop is placed inside the applied shader. Thereby, the forward deformation grid texture which was recently used for writing is bound as input texture and the inverse deformation grid texture is set as render target.

Rendering

For direct volume visualization of the deformed volume with the multi-volume rendering framework the *Scene Node*, namely the sub-node that is responsible for the volumes (see Section 3.3.1), is slightly adapted. Instead of directly looking up the volume sample at the current texture coordinate, the deformed texture coordinate is first looked up in the inverse deformation grid. Then, the sample value of the deformed volume is looked up in the original volume texture. Gradients of the deformed volume are computed on the fly by finite differences (see Section 2.4.2). Since the *Scene Node* makes the deformation process transparent for subsequent nodes, volume deformation can be combined with any existing or newly implemented render node. Further on, it

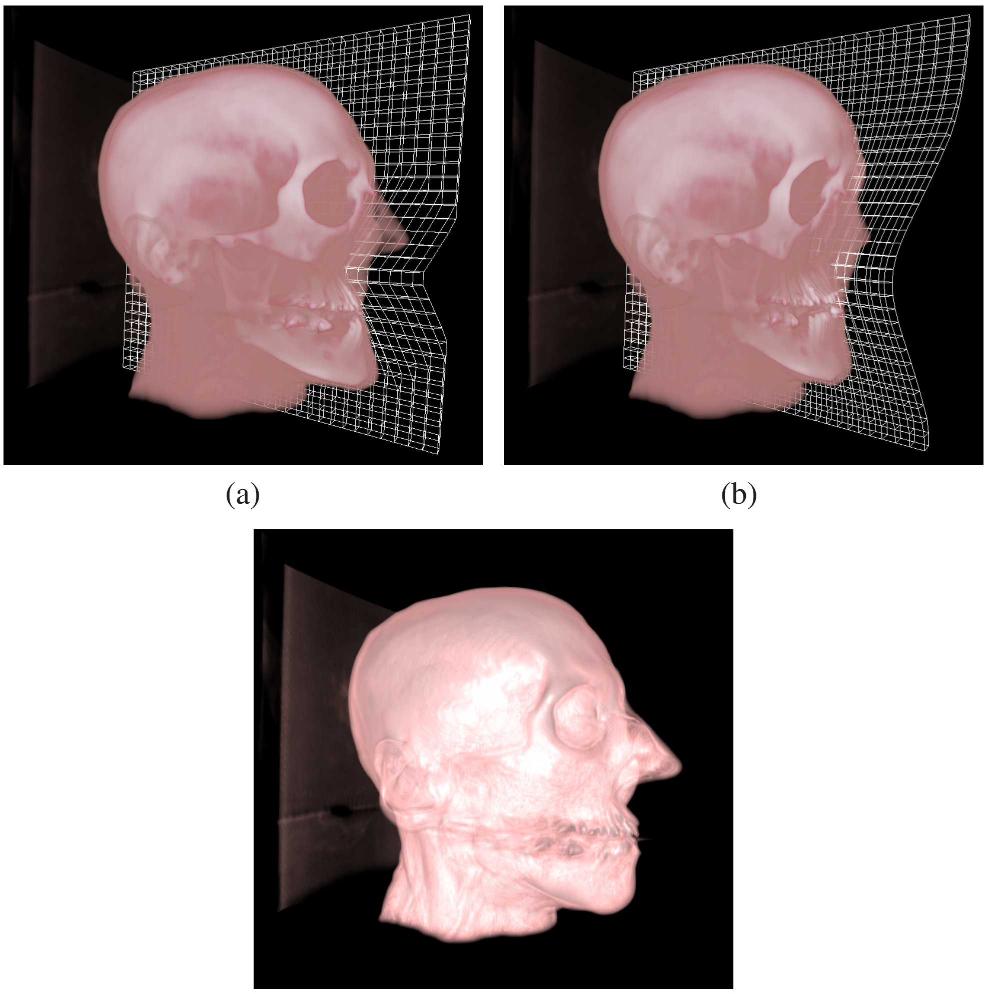


Figure 4.18: Three deformed CT scans of a human head (256×256×225) with different deformation algorithms and grid sizes. a) 32×32×32, ChainMail only; b) 32×32×32, ChainMail + Relaxation; c) 128×128×128, ChainMail + Relaxation + Illumination

is possible to apply either slice-based rendering (see Section 3.2.1) or ray casting (see Section 3.2.2).

4.3.4 Results and Discussion

The GPU-based ChainMail deformation approach was tested with a CT scan of a human head. Figure 4.18 shows three screenshots of the deformed head with different deformation grid resolutions and different stages of the algorithm applied. Figure 4.18(a) and (b) both present the deformation results for a grid resolution of 32×32×32 elements without (Figure 4.18(a)) and with relaxation (Figure 4.18(b)). It can be clearly seen, that the additional relaxation creates a much smoother deformation result. In Fig-

	Rendering only	16×16×16	32×32×32	64×64×64	128×128×128
CM	19.5	16.5	15.0	13.5	8.9
CM + R	19.5	15.5	14.5	11.5	7.0
CM + R + I	5.0	4.5	4.4	4.2	3.3

Table 4.3: Frame rates in frames per second (fps) for deformation of a CT dataset of a human head with resolution 256×256×225, viewport 512×512, GeForce 8800 GTX with 756 MB RAM. (CM $\hat{=}$ ChainMail, R $\hat{=}$ Relaxation, I $\hat{=}$ Illumination)

ure 4.18(c)) a deformation grid of 128×128×128 elements is used and the volume is additionally illuminated. Here, the deformation below the nose affects a smaller region of the head. Table 4.3 gives frame rates for deformation and rendering for different grid resolutions. The leftmost column shows the performance for rendering only. Standard volume rendering produces interactive frame rates, but additional illumination is four times slower because of the expensive on-the-fly gradient computation. The relaxation was restricted to five steps per frame. Instead, intermediate relaxation steps were introduced when no interaction happens.

A major advantage of the presented approach is that the resolution of the deformation grid is independent of the resolution of the original volume dataset. So the grid resolution can be chosen with respect to the desired locality of deformation and due to the required frame rate. However, the memory consumption increases significantly with higher grid resolutions. A deformation grid with 128×128×128 elements and four 32-bit floating point values per element needs 32 MB memory, a 256×256×256 grid needs 256 MB memory. Since the algorithm needs the grid three times, for ping pong rendering and for inverse deformation, the memory of the employed GPU is exceeded for a grid resolution of 256×256×256 elements. A possible solution to overcome this limit would be a bricking scheme like proposed in [120].

The frame rates in Table 4.3 show that the ChainMail-based deformation approach fits well to the parallel architecture of a GPU. Since it is directly combined with standard volume rendering, it can be easily applied to medical simulation applications. However, the ChainMail technique has still some drawbacks that should be eliminated. E.g., the achievable frame rates highly depend on the applied deformation constraints. If the constraints simulate stiff material, the deformation is propagated over a larger area of the grid than with soft tissue constraints. Hence, the ChainMail computation is getting significantly slower in this case. Nevertheless, the presented deformation pipeline allows to modify single steps without affecting the others. So deformation and relaxation could be replaced by more complex deformation models.

4.4 Conclusion

In this chapter several interactive medical volume visualization applications have been presented that illustrate the flexibility of the multi-volume rendering framework from Chapter 3. In Section 4.1 a generic multi-volume visualization tool was described that allows the interactive configuration of the render graph via the graphical user interface (GUI). Complex visualizations for arbitrary multi-volume scenes can be generated without the need to cope with underlying rendering functionality. Moreover, new render nodes can be implemented and seamlessly integrated into the system. Thus, the tool is perfectly suited to explore the capabilities of multi-volume visualization for new medical application fields.

In Section 4.2 a certain application field, the visualization of functional MRI images, was investigated. Primarily, a visualization tool was presented that was designed to support cognitive scientist in their daily work. This tool exploits the capabilities of the multi-volume rendering framework but provides an application-specific user interface. Further on, a technique for the combined visualization of functional and anatomical MRI volumes was presented that aims to improve the perception of the anatomical brain structure without occluding inside functional information. Therefore, the curvature of the brain surface is enhanced by line integral convolution (LIC). Since LIC computation is expensive, an extension of the multi-volume framework was proposed that supports deferred shading. Hereby, the costly calculations of the LIC integral can be restricted to the visible brain surface.

In Section 4.3 a direct GPU-based volume deformation approach was presented, which, for example, could be applied for surgery simulation. Based on the ChainMail deformation algorithm, a deformation pipeline was developed that is completely evaluated on the GPU. Besides user manipulation and deformation, this pipeline handles the direct visualization of the deformed volume. Before visualization the forward deformation field that is computed by the ChainMail deformation is inverted. The inverted deformation field provides deformed texture coordinates that allow the application of the standard volume rendering techniques that are provided by the multi-volume rendering framework.

The three visualization scenarios emphasize the suitability of the multi-volume rendering framework for a large variety of interactive medical-visualization applications. The flexibility of the render graph allows the creation of meaningful visualizations for differing tasks and datasets. For new visualization problems specialized render nodes can be easily integrated. With slight adaptations rendering concepts, like deferred shading or the rendering of deformed volumes, can also be realized. Even though new render nodes and rendering techniques are usually developed for a certain application case, they can be integrated into the basic framework and be employed for other visualization tasks.

Chapter 5 Automated Medical Volume Visualization

The interactive control of the visualization output is an important prerequisite for visual analysis in new medical application fields and for the examination of complicated cases. However, in clinical routine many patients with similar symptoms have to be examined. For those cases usually certain procedures for visualization and visual analysis have been established. The automation of these procedures could support the work of physicians and surgeons and improve the quality of the diagnosis results. On the one hand, a medical doctor could deal with other problems, while a computer performs the analysis and perpares 3D visualizations of the images acquired from a patient. On the other hand, the automation and, thereby, standardization of the visualization procedure allows the easy comparison of different cases, supports unexperienced medical doctors and eases the discussions with colleagues.

Automated medical visualization usually involves so-called *batch visualization*. Here, a number of images from different camera positions and for different scene configurations are rendered automatically due to a batch visualization script. These images are either stored seperately or assembled in one or several video sequences. After batch visualization the images or video sequences can be viewed with any standard viewer independent of the visualization system (offline). Figure 5.1 shows the adapted batch visualization pipeline. In contrast to interactive visualization, there is no feedback loop in which the user can directly manipulate the visual output. Instead, a number of parameters for the steering of analysis and visualization can be set before evaluation of the pipeline.

Systems for automated medical visualization should cover two aspects. On the one hand, the pre-rendered images should be provided in a way that allow an intuitive investigation of the visualized objects. On the other hand, the automated visualization procedure should be accessible for a large group of users and should have minimal demands on a user's hardware equipment. In this chapter a system for automated medical volume visualization is presented that incorporates both of these requirements.

In Section 5.2 an alternative batch visualization method is introduced that is based on so-called *3D object movies*. In contrast to images and video sequences, object movies provide the possibility to interactively navigate along predefined camera posi-

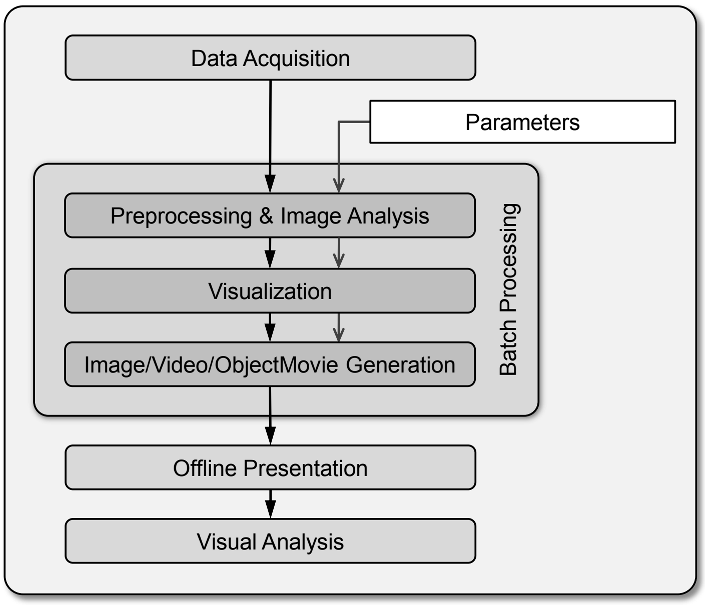


Figure 5.1: Pipeline for medical batch visualization: In the batch processing stage image preprocessing, image analysis and visualization are automatically performed due to a predefined script and some externally given parameters. The generated visualizations are persistently stored either as images, video sequences or object movies. The stored visualizations can be analyzed offline at any time with an appropriate viewer.

tions. For automated medical volume visualization a new *medical object movie* format and a corresponding viewer were developed that are especially adapted to the needs of medical analysis. The automated generation of medical object movies is integrated into a web service for standardized medical analysis and visualization, which is described in Section 5.3. This service provides a dynamic web interface with which a user can upload a medical volume dataset to a web server and initiate the generation of a couple of video sequences and object movies. The rendering of the videos and object movies is performed in parallel on a cluster of GPU-equipped computers. The advantages of the web service are, on the on hand, the possibilty to access it from any internet PC and, on the other hand, the fast availability of the results due to their parallel computation. The proposed concepts for automated medical volume visualization were evaluated with the use case of standardized analysis of *intracranial aneurysms*, which is detailed at first in Section 5.1.

The techniques for automated medical visualization that are described in this chapter were first published in [109] and [112]. They are based on earlier work from Sabine Iserhardt-Bauer [55; 56], who originally developed the web service for the standardized analysis of intracranial aneurysms. The work was carried out in collaboration with Matey Nenov and Torsten Wolff. Matey Nenov realized the offline visualization via object movies described in Section 5.2 during the preparation of his diploma thesis. Torsten Wolff implemented the parallel visualization web service presented in

Section 5.3 in the course of his study thesis. Peter Hastreiter from the Neurocenter at the University Hospital Erlangen, Germany, and Bernd Tomandl from the Bremen East Central Hospital, Germany, gave support about the medical background and provided the test datasets.

5.1 Use Case: Standardized Analysis of Intracranial Aneurysms

An *aneurysm* is a localized, blood-filled dilation of a blood vessel, which is caused by disease or weakening of the vessel wall. Aneurysms mostly appear in the *aorta* (the main artery coming from the heart) or in arteries at the base of the brain. Aneurysms in brain arteries are called *intracranial aneurysms* because they occur in vessels inside the skull (*cranium*). As the size of an aneurysm increases, there is an increased risk of rupture. The rupture of an intracranial aneurysm usually results in so-called *subarachnoidal hemorrhage* (bleeding inside the brain). Symptoms for subarachnoidal hemorrhage are the sudden onset of severe headache and neurological failures. In the worst case it can lead to the death of the patient.

For the radiological diagnosis of intracranial aneurysms and subarachnoidal hemorrhages frequently *computed tomography angiography* (CTA) is used. Here, the visibility of the vessels in the CT scan is increased by previously injecting a contrast agent. There are two alternatives for the treatment of intracranial aneurysms. For *surgical clipping* first the skull has to be opened and the aneurysm has to be exposed. Then, the base of the aneurysm is closed with a clip. In contrast to that, *endovascular coiling* is a minimally-invasive intervention. A catheter is inserted into the femoral artery in the patient's leg and navigated through the vascular system into the head and into the aneurysm. Then, platinum coils are pushed into the aneurysm and released. These coils initiate a clotting or thrombotic reaction within the aneurysm. Thereby, blood flow into the aneurysm is blocked and rupture is prevented.

Iserhardt-Bauer *et al.* [56] developed a standardized procedure for the analysis and visualization of intracranial aneurysms, which is mainly performed automatically without any user interaction. It aims to support radiologists and surgeons in diagnosis and treatment planning. Further on, the standardization should ease intra- and inter-patient comparison. The proposed analysis procedure first segments vessel structures from the CTA scan that hold a high risk for aneurysms . Then, these structures are visualized along predefined camera paths with direct volume rendering and the resulting images are assembled to several video sequences. Finally, the video sequences can be visually analazyed by the investigating medical doctor. Figure 5.2 shows the applied analysis pipeline, which consists of five steps. These steps are detailed below.

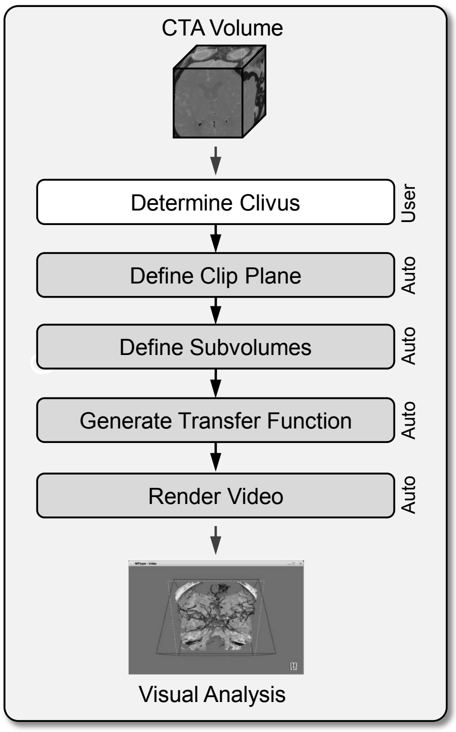


Figure 5.2: Pipeline for the standardized analysis of intracranial aneurysms: In the first four steps interesting vessel structures are determined and segmented. Then, a couple of volume rendered video sequences is generated, which can be visually analyzed with an arbitrary video viewer. The first step (white) has to be conducted by the user, the others are performed automatically.

Determination of Clivus Position

The clivus is a bony structure at the inside of the skull base. Important blood vessels are crossing here, and, thus, aneurysms often occur in the neighborhood of the clivus. For this reason the clivus is chosen as anatomical landmark for the subsequent analysis and visualization steps. Currently, the clivus position cannot be detected automatically, but has to be determined manually by the user. Nevertheless, the determination of the clivus position is the only pipeline step which requires user interaction.

Definition of a Clip Plane

The interesting arteries in an investigated CTA dataset are often hidden by veins located in the occipital part of the head. To overcome this problem, in this step a clip plane is defined that virtually cuts away disturbing vessels in the occipital half space of the volume. In a clinical study it was found that the clip plane is optimally positioned 30 mm in front of the clivus and rotated about $45°$.

Definition of Subvolumes

Intracranial aneurysms often grow at branches of blood vessels. To lay focus on those specific locations, four subvolumes are defined that enclose four critical points found at the tip of the *basilar artery*, at the left and the right *cerebral artery*, and at the *communicating artery*. For this purpose, the average distance vectors between the clivus and the four critical points have been evaluated in a clinical study, which was performed on a number of reference datasets. To determine the patient specific positions of the four critical points the predefined average distance vectors are added to the previously determined clivus position. The four critical points then act as the centers of the four subvolumes. The size of the subvolume ($60\ mm$ in each direction) is chosen in such a way that they overlap in any case. Thereby, it is not necessary to find the exact positions of the critical points.

Generate Transfer Function

After the reduction of the topological information by clip planes and subvolumes, it is additionally necessary to reduce the structural information contained in the investigated dataset. Besides the data inherent noise, the soft tissue that encloses the vascular structure prohibits a clear 3D representation. For this reason, the vessels are implicitly segmented by application of a predefined transfer function. This transfer function oppresses occluding structures by mapping the opacity of the according lower Hounsfield units to zero. Further on, the color values are chosen in a way that the vessel structure is emphasized. To handle differences that naturally occur in different datasets, the predefined transfer function is automatically adjusted to the patient specific dataset with a method proposed by Rezk-Salama *et al.* [102]. Here, a non-linear transformation is determined that maps the histogram of the dataset on which the template transfer function was defined to the histogram of the currently investigated dataset. Then, the same transformation is applied to the pre-defined template transfer function. By this means, a transfer function is generated that is optimized for the current dataset.

Render Dataset and Generate Video Sequences

Based on the applied segmentation operations the analyzed dataset is visualized with direct volume rendering. To show the data from different directions, the camera is moved around the volume along pre-defined paths, and at discrete positions images are computed. These images are assembled to a couple of video sequences that show the movement of the camera. At first, an overview video is generated that shows the whole volume with occluding structures clipped away by the previously defined clip plane. The applied camera path is chosen with respect to the way clinicians usually examine individual patient data. For the purpose of orientation, the camera follows in the beginning several circular paths around the complete volume . Then, the camera zooms in to take a detailed view from a closer distance. To allow a deeper investigation of critical vessel structures, four additional videos are generated that show the four

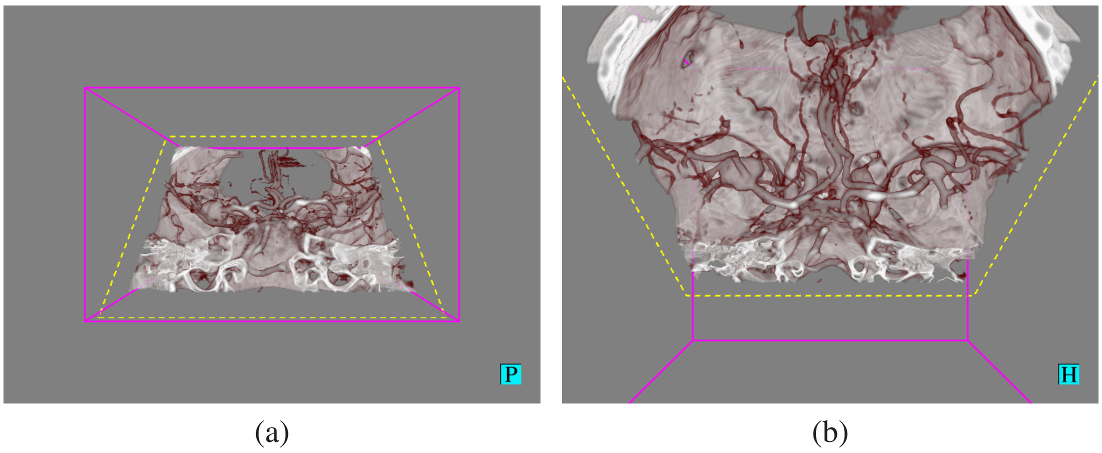


Figure 5.3: Two screenshots of the overview video sequence, which show the whole dataset, while disturbing structures are clipped away by an automatically placed clip plane. The left image (a) shows the initial view which presents the dataset from the back. In the right image (b) the camera is rotated to show the volume from the top. In addition it was zoomed in, to get a closer look of the data. On the left and the right handside of the dataset two aneurysms can be identified, which are focused in the videos of the subvolumes (see Figure 5.4).

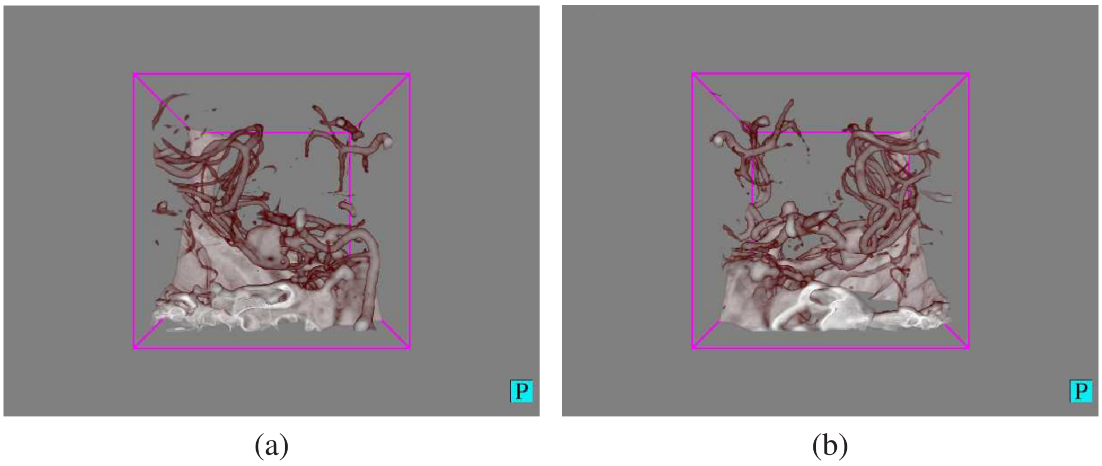


Figure 5.4: Two screenshots of the subvolume video sequences around the left cerebral artery (a) and around the right cerebral artery (b). They show two different aneurysms which are located in the centers of the two images.

previously determined subvolumes. Here, a simple circular camera flight of 360° is performed. The generated videos are stored persistently and can be visually analyzed at any time with an arbitrary video viewer. Figure 5.3 shows two screenshots from the overview video. In Figure 5.4 two screenshots of two of the subvolume videos are presented.

5.2 Automated Visualization with 3D Object Movies

While a clinical study [131] has proven the applicability of the system for standardized analysis and visualization of intracranial aneurysms in clinical routine, the video-based offline visualization has still some limitations. A major drawback is the fixed camera path and the restriction to linear navigation along a video's time line. Thus, it is difficult to get an impression of the spatial relationship between interesting details of the data set. Further on, no direct links between the different video sequences do exist.

An alternative technique for offline 3D visualization are *3D object movies*. Here, images are taken at fixed camera positions on a spherical hull around the visualized object (see Figure 5.5 (a)). Specialized viewers allow free navigation between these camera positions. Object movies have a close relationship to panorama movies in which the observer is placed in the center of a scene that can be interactively viewed by rotation of the camera. Panorama movies are often stitched together from $360°$ photographies of urban places or landscapes. There are several different formats and viewers for panorama and object movies. The most common is Apple's *QuickTime VR* [3] format in combination with the proprietary *QuickTime Viewer*. An alternative are the freely available *Panorama Tools* [25], which where originally developed by Helmut Dersch. They comprise software tools for the creation of panoramas and object movies as well as a Java Applet viewer.

Object movies have already been applied to medical data for educational purposes. For instance, *"Bones of the Skull"* [133] of the Hardin Library for the Health Sciences of the University of Iowa is an interactive learning tool for the anatomy of the human skull based on QuickTime VR object movies. Melin-Aldana and Scirotino [81] used QuickTime VR for an interactive atlas of pediatric liver pathologies. Both examples are based on photographies of real world objects. In contrast to that, Tiede *et al.* [130] combined QuickTime-VR panoramas and object movies to create interactive movies for virtual endoscopy. Chen *et al.* [19] applied the object movie technique for remote visualization of scientific datasets via the internet. To overcome a major limitation of classical object movies, the necessity of downloading the whole movie before the visualization can be started, they created a stand-alone viewer that downloads single images of an object movie on demand.

However, the existing object movie formats and viewers lack flexibility and do not fit to the demands of standardized medical visualization. Furthermore, they are difficult or impossible to extend. For this reason, in this work a new *medical object movie* format and an accompanying Java-based viewer were developed. Hereby, two objectives were pursued: on the one hand, all visualizations of an analyzed dataset should be integrated and interlinked in a single movie; on the other hand, it should be possible to access a movie in a fast and interactive way via the web.

5.2.1 Medical-Object-Movie Format

Design Concepts

The design of the *medical object movie* format was driven by the same ideas that were taken into account for the definition of video sequences for the analysis of intracranial aneurysms (see Section 5.1). There, one overview video of the whole dataset and several videos of dedicated sub-volumes in areas with a high risk for aneurysms are generated. The applied camera animation first follows several circular paths around the complete volume. Then, the camera zooms in to give a closer view of the data. Occluding structures are optionally cut away by a clip plane. To provide similar visual information as in the videos, the following visualization concepts have been integrated into the medical-object-movie format:

Tiles and Zoomed Views Usually, the views of an object movie are captured by rotating the camera around the vertical axis of the observed object. At discrete rotation angles (pan), the camera is tilted up and down to take several views at discrete tilt angles (see Figure 5.5 (a)). Obviously, the captured images can be arranged in a 2D grid in which the x-axis represents the pan angle and the y-axis the tilt angle (see Figure 5.5 (b)). The overall number of images – and thereby the size of an object movie – can be restricted by choosing large pan and tilt angles for the camera movement between consecutive views and by limiting the examined area to maximum allowed angles for pan and tilt.

However, for medical visualization the applicability of these measures is limited. On the one hand, the difference between consecutive views should be chosen in a way that the movement between them appears smooth. On the other hand, it should be possible to examine pathological structures from many different view angles all around the object. For these reasons, the medical object movie format provides the possibility to assemble a movie from several tiles that cover certain regions of the spherical hull on which the camera moves around the object (see Figure 5.5 (c)). This allows the combination of regions that have a wide tilt range with those that have a small tilt range.

Besides the restriction of an object movie's size, this feature can be used to constrict the examinable views to those which are interesting for the current medical analysis task. In addition, to each view an additional zoomed view can be attached, which provides a closer look from the same viewing direction. Typically, zoom views are appended for a subset of the available standard views to provide details for areas of high importance.

Sub-Movies and Links The medical object movie format allows the assembly of several so-called *sub-movies* in a single movie object. Thereby, visualizations of different details of an observed medical structure can be combined into an integrated visual presentation. E.g., in the context of analysis of intracranial aneurysms an object

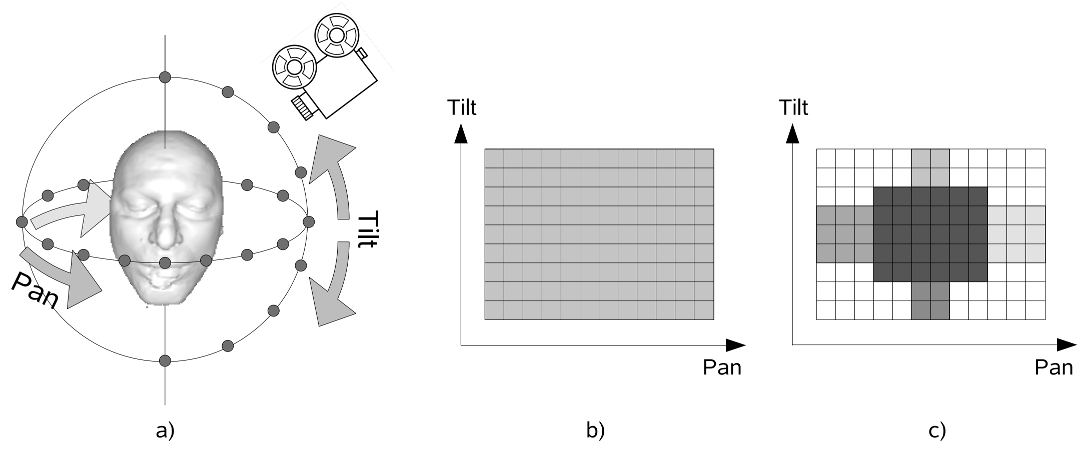


Figure 5.5: Generation and storing of a *medical object movie*: (a) The acquisition of the views on a spherical hull. (b) Rectangular array for picture storage in a traditional object movie. (c) Combination of different tiles (different grey values) in the new medical-object-movie format.

which gives an overview of the complete vessel structure can be joined with several object movies that represent the sub-volumes with a high risk for aneurysms (see Section 5.1).

Basically, the different sub-movies are independent from each other and can show completely different objects or structures. But if there is an overview-and-detail relationship between the sub-movies, links between these movies can be defined. Such a link specifies a 3D area, namely the center and radius of a sphere, which encloses the detail structure that is presented in the connected sub-movie. The object-movie viewer can use this information to allow interactive navigation between the different sub-movies. To give a visual hint about the existence and location of a link, the link area is usually emphasized in the pre-rendered views.

Visualization Modes For each sub-movie several different visualization modes can be applied. This feature allows emphasizing different aspects of a visualized object. For example, a volume can be rendered with different transfer functions or with and without a clip plane. For all visualization modes the same views are computed. Thus, the visualization mode can be switched at each viewing position

File Format and Image Storage

A medical object movie consists of two files, a *meta info file* in XML format that describes the structure of the movie and an *images file* that contains the image data. For the meta-info file an XML-format was developed that permits the assembly of object movies that follow the design concepts described above. Basically, the format allows the definition of several independent sub-movies, which are built of one or more tiles. A tile is defined by its width and height given in number of images, and its location relative to the 2D grid that represents the discrete viewing positions on the spherical hull around the observed object (see Figure 5.5). In addition, arbitrary zoom tiles can be defined, which should cover a subset of the un-zoomed tiles. For each sub-movie links to other sub-movies can be defined. For this purpose, each sub-movie has a unique id. A link is defined by the id of the linked sub-movie and by the center and the radius of a sphere that encloses the region that is shown by the linked sub-movie.

Besides the structure of the object movie, the meta-info file contains links to the related images, which are stored sequentially in the images file. For each tile of a sub-movie there is an image list that gives start positions and sizes in bytes of the related images in the images file. The images can be stored in any compressed or uncompressed image format that can be interpreted by the viewer. Usually, a lossy compression like JPEG is chosen to avoid that the size of the object movie is getting too large. If a sub-movie contains links, an image can be provided two times, once with the standard visualization mode and once with an additional accentuation of the link area. Thereby, the highlighting of a the link area, which may occlude important information, can be switched off. To keep the storage requirement of the two images as low as possible, the image with the emphasized link area only stores the color values of those pixels that differ from the original image. The others are set transparent. For this purpose, the difference image has to be stored in a format that supports transparency, e.g. PNG or GIF.

5.2.2 Medical Object Movie Viewer

For the observation of a medical object movie a Java-based viewer was developed, which can either be run as stand-alone application or as an Applet integrated into a website. The object movies can be loaded from any web location defined by an URL or from local disk. First, the meta-info file is read, the movie structure is analyzed and the viewer's GUI is configured accordingly. Then, the image download for the currently shown sub-movie is initiated. The interactive exploration of the movie can be started immediately. Figure 5.6 shows the basic architecture of the medical-object-movie viewer, which consists of four modules. The central *ObjectMovie Application Object* launches the viewer and initializes and interconnects the three other modules, which are responsible for the interpretation of the meta-info file (*Meta Info Module*), the download of the images (*Image Loader Module*) and the generation and controlling of the GUI (*GUI Module*). The functionality of the three modules is detailed in the following.

Meta Info Module

The *Meta Info Module* fetches the location of the object movie from the *ObjectMovie Application Object* and parses the XML-structure of the meta-info file. From this information it creates an object hierarchy that represents the meta structure of the complete object movie. This object hierarchy is used by the *GUI module* to build up the

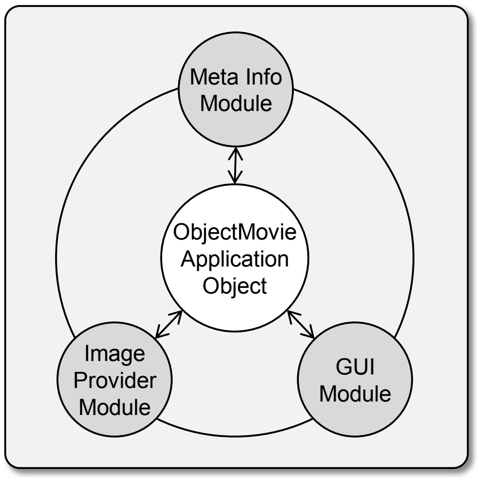


Figure 5.6: Architecture of the medical-object-movie viewer. The central *ObjectMovie Application Object* initializes three modules which are responsible for the analysis of the movie structure (*Meta Info Module*), download and caching of the images (*Image Provider Module*), and for the graphical user interface (*GUI Module*).

object-movie-specific GUI elements. Further on, the *Meta Info Module* serves the byte address and byte size of a certain view to the *Image Provider Module*. For fast access this information is stored in a hash table.

Image Provider Module

The *Image Provider Module* is responsible for downloading of the views from the images file. The image download is, on the one hand, performed on demand for the currently selected view. On the other hand, surrounding images are automatically downloaded in the background and stored in a client-side cache to speed up the navigation between different views.

The *Image Provider Module* consists of three sub-modules: the *Image Request Agent*, the *Image Loader* and the *Image Cache*. The *Image Request Agent* automatically generates image requests and adds these requests to the request queue of the *Image Loader*. It starts with a request for the currently selected view and goes on with requests for the surrounding views. For each request it is first tested if the requested image is already stored in the *Image Cache*. In this case the request is discarded. When the currently selected view is changed, the *Image Request Agent* restarts the generation of requests from the new view.

The *Image Loader* performs the download of the requested images. It handles several requests in parallel via image download threads, which are managed in a thread pool. Each time a thread has finished the download of an image, the *Image Loader* takes the next request from the queue, asks the *Meta Info Module* for the address and size of the requested image, and delegates the download of this image to the waiting thread.

The downloaded images are stored in the *Image Cache*. For fast access the images are stored in a hash map. The *Image Cache* holds different hash maps for the different visualization modes and sub-movies. If visualization mode or sub-movie is changed, the hash maps are also changed. The old hash-maps are stored for later reuse. If the size of the cache exceeds a predefined limit the least used hash map is deleted. To save memory, the images are stored as byte arrays in their compressed format and extracted on-the-fly when they are requested by the *GUI Module*.

GUI Module

The *GUI Module* initializes and controls the graphical user interface (GUI) of the viewer. Figure 5.7 shows the viewer's GUI with its different controls and panels. In the *MainView Panel* the currently chosen view is displayed. The camera can be moved by dragging the mouse but only in the range of precomputed views. The current camera position is shown in the *MiniMap Panel*. The 2D grid on which the views are arranged is visualized as a rectangle of image elements. The availability and the caching state of the views is shown by a specific color scheme. E.g., a view which is not contained in the object movie is visualized as a gray rectangle; an available view which has no zoomed view attached and which is not yet loaded is white; a view with an additional zoomed view is light blue. Furthermore, there are different colors that encode the current loading and caching state of a view.

Via the *Control Panel* different actions like reseting of the main view and switching to a zoomed view can be initiated. Furthermore, the visualization of the link areas can be activated and deactivated. The *ObjectModes Panel* allows switching between different modes of a single sub-movie. The *Objects Panel* shows which sub-movies are available. A sub-movie can either be chosen by selecting the corresponding symbol in the *Objects Panel* or by clicking to the emphasized link area in the *MainView Panel*. In Figure 5.7 the position of a single link area is indicated by a green bounding box.

When a new view is selected, the *GUI module* requests the image for the selected view from the *Image Provider Module*, more precisely the *Image Cache*. If the requested image is not yet available, the *Image Cache* returns a null pointer and the *MainView Panel* shows the accordingly rotated coordinate axis instead. At the same time the *Image Request Agent* is informed, which initiates the privileged download of the currently selected view.

5.2.3 Application

The new medical object movie format was applied to the standardized analysis of intracranial aneurysms (see Section 5.1). The first steps of the analysis pipeline shown in Figure 5.2 were kept the same, but instead of several video sequences a single standardized object movie is generated. The structure of this standardized object movie was designed in a way that it provides similar information like the video sequences.

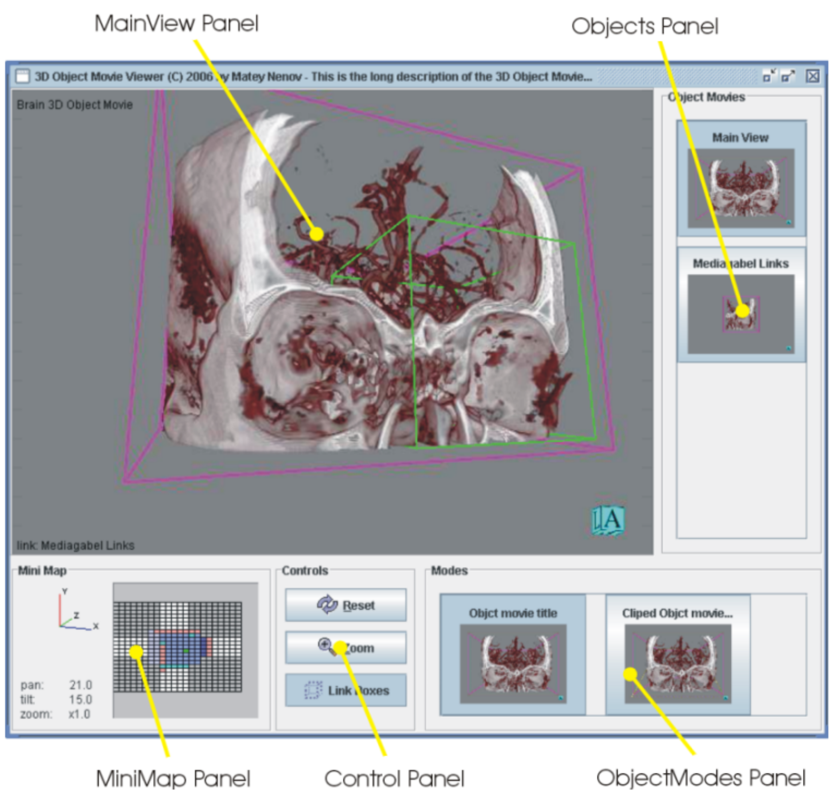
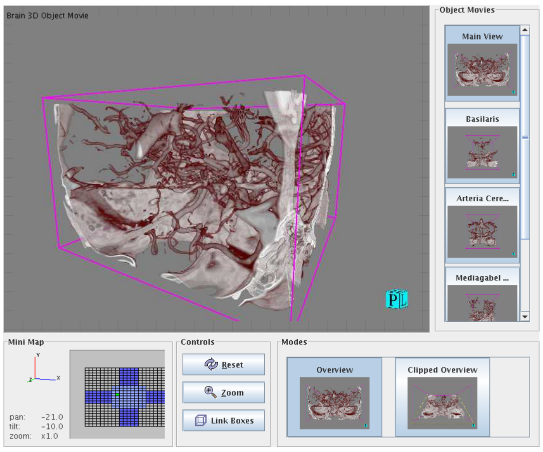


Figure 5.7: Graphical user interface (GUI) of the object movie viewer. In the *MainView Panel* the currently selected view is visualized; the *MiniMap Panel* shows the current camera position and the tile structure of the object movie; via the *Control Panel* different states can be changed; in the *Object Modes Panel* the visualization mode can be selected; the *Objects Panel* allows the selection of a sub-movie.

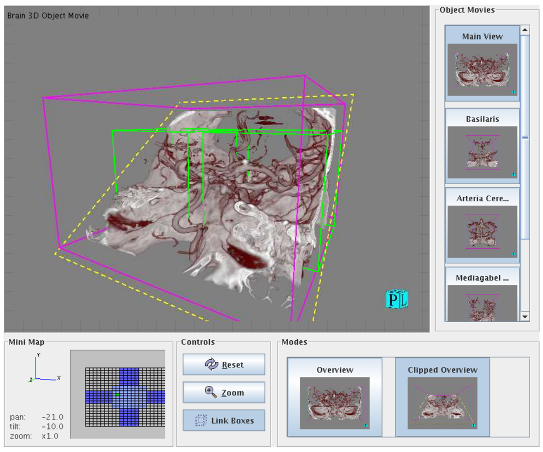
Therefore, it is assembled from five sub-movies. One sub-movie shows an overview of the complete CTA dataset. The other four sub-movies visualize the four sub-volumes that enclose regions with a high risk for intracranial aneurysms.

For the overview sub-movie two visualization modes are applied. One mode shows the complete volume; for the other mode occluding vessels are clipped away by the automatically adjusted clip plane. Hereby, it is possible to interactively check which structures are clipped away. The location of the sub-volumes is visualized by their bounding boxes, which are rendered in wireframe mode. In addition, link areas are defined that allow switching to the connected sub-movie by clicking with the mouse into the bounding-box area. Since the areas of the sub-volumes overlap, only links to the two sub-volumes around the left and right cerebral artery are attached.

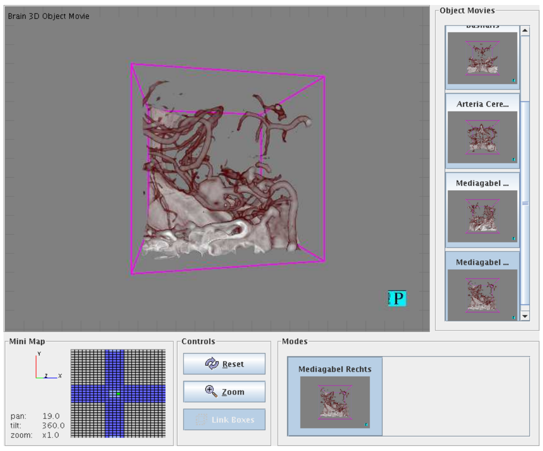
The overview sub-movie is assembled of five tiles. A large center tile shows the back side of the CTA volume. From this side the vessels are best visible, especially if some structures are cut away by the clip plane. For a detailed look to the tile zoomed



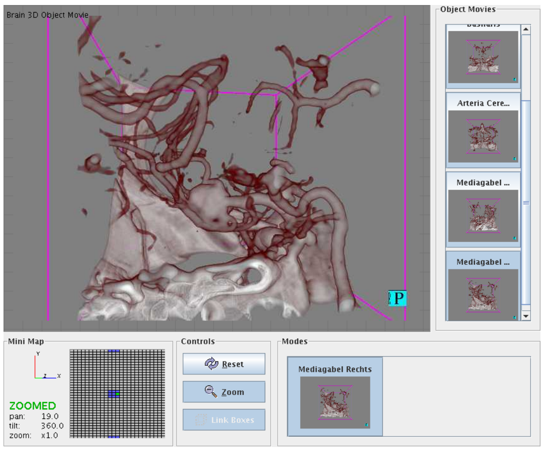
(a)



(b)



(c)



(d)

Figure 5.8: Four screenshots of a standardized medical object movie for the analysis of intracranial aneurysms. (a) Overview of the complete CTA dataset; (b) Overview with occluding structures cut away by an automatically positioned clip plane. In addition the bounding boxes of the sub-volumes around the left and right cerebral arteries are shown; (c) The sub-volume around the left cerebral artery; (d) A zoomed view which corresponds to the view in (c).

views are attached. The four other tiles allow rotations to left, right, top, and bottom. A full rotation about 360° is not supported because on the front side the vessels are usually occluded by the skull.

The four sub-movies that show the four sub-volumes all have the same structure. They have only one visualization mode that shows the sub-volumes completely. The tiles are arranged in a way that allows complete 360° rotation in pan and in tilt direction. Furthermore, there are small zoom areas on the front and the back side of the sub-volumes.

Figure 5.8 shows several screenshots of an automatically generated object movie. The images have a resolution of 640x480 pixels. As image format JPEG with average quality (quality factor 50) is used. The average size of a single view is about 30 KB. The difference images for the link areas are stored in PNG format. They have an average size of about 15 KB. The complete object movie contains 2292 images and has s size of about 50 MB. The average frame rate for the switch between images that are stored in the client side cache is about 10 frames per second on a standard PC. The download time for images depends on the location where the object movie is stored and on the speed of the applied network connection.

5.3 A Visualization Service for Standardized Medical Analysis

A fundamental requirement for the applicability of automated medical analysis and visualization is the easy accessibility of the functionality by a large group of clinical users. An obvious approach is to make the processing pipeline available as a service via the web that can be accessed by any workstation with connection to the internet. The costly data analysis and rendering procedures can thereby be performed on specialized high-performance hardware that is supplied by the service provider. Besides the speed-up of the batch analysis and visualization process, this concept brings the advantage that the initiation of the automated visualization can be carried out at another place than the investigation of the visual results. For example, the process could be started from a workstation that is directly connected to the imaging device where the data is acquired, and the generated visualization could be accessed from another workstation that is placed in the operation room.

The provision of visualization functionality as a service via the internet is a frequently applied concept in many visualization areas. Those services shift some or all stages of the visualization pipeline (see Section 2.2) to a server, in order to give access to data or functionality that is not available in the local environment of a user. A typical application field of visualization services are *geographic information systems* (*GIS*), which visualize spatially located data, like measurements and statistical values, in their geographical context [115; 146]. Other services aim to support the visualization of scientific data [6; 99]. For medical purposes also a number of service based applications have been proposed. Avis *et al.* [5] and Montagnat *et al.* [86] presented web services for remote 3D visualization of medical image data. Gibaud *et al.* [41] developed a service that matches patient specific image data with a brain atlas. The system of Kooper *et al.* [66] allows the reconstruction of a 3D volume from cross-sectional microscopic images. Moreover, some vendors of commercial medical visualization environments have integrated interfaces for remote access to data and functionality [127; 137].

Visualization services can be classified by the visualization pipline stages that they provide. Basically, there are two classes of services: those that delegate the rendering to the client computer (*client-based rendering*), and those that perform the rendering on the server (*server-based rendering*). Services of the first class usually act as a remote data source, e.g. for large data that can't be stored on the client computer or data which is acquired at the remote site. Furthermore, those services often provide specialized functionality for data preprocessing.

The second class of visualization services – the services with server-based rendering – can be further subdivided into those service that render the visualizations on the fly (*remote rendering*) and those services that prerender a couple of images in a batch process (*batch rendering*). While remote rendering services focus on the interactive manipulation of the visualization, batch rendering can be applied for automated visualization tasks. E.g., Chen *et al.* [19] applied the object movie technique for precomputing 3D visualizations of scientific datasets, which can be interactively investigated with a specialized viewer. Iserhardt-Bauer *et al.* [55; 56] developed a web service that provides the standardized analysis and visualization of intracranial aneurysms, as described in Section 5.1.

In this section a visualization service for standardized medical analysis is presented, which is based on the concepts proposed by Iserhardt-Bauer *et al.*. These basic concepts have been advanced in two ways. On the one hand, a service architecture was developed that abstracts from a certain medical application task. Thus, visualization services for different medical purposes can be easily realized and even provided by the same service hardware. On the other hand, a technique for parallel batch rendering on a GPU-cluster was integrated. This permits the generation of complex video sequences and object movies in a couple of minutes, and, thus, makes the service applicable for clinical routine.

5.3.1 System Architecture and Workflow

The visualization service system consists of two major components (see Figure 5.9). On the one hand, there is a web server that provides the web interface and manages the user data. It is based on a *Tomcat* HTTP server, an open source project of the Apache Software Foundation [2], which provides a runtime environment for *Java Servlets* and *java server pages* (*JSPs*). This offers an easy way for the implementation of dynamic HTML pages and the processing of user requests.

The other component is the *render server*, a GPU-cluster that performs the rendering and the encoding of videos and object movies. This cluster consists of eight standard PCs (4 GB RAM, 2 x *AMD Opteron*, 2.2 GHz), each containing a *NVIDIA GForce 6800 Ultra* graphics card with 256 MB RAM. The cluster nodes are connected with a fast *Infiniband* network. The inter cluster communication is performed with the *message passing interface* (*MPI*) [89], a special API for parallel processes. The web server and the cluster communicate via a *TCP/IP* connection with a standardized protocol for different analysis requests.

A typical workflow of standardized medical analysis performed with the visualization service is shown in Figure 5.10. In the beginning, a connection to the web server is established via the dynamic web interface. Then, the user uploads a pre-

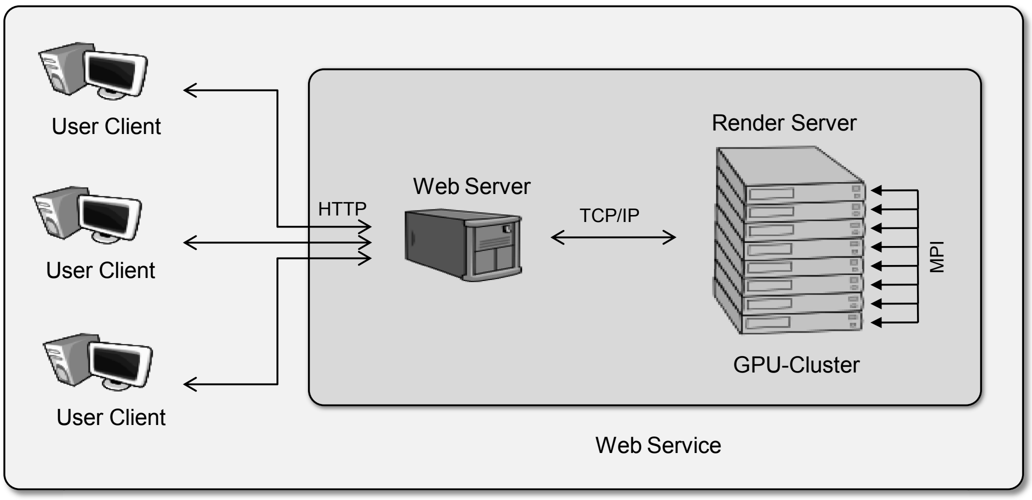


Figure 5.9: Architecture of the web service composed of a web server and a GPU-Cluster, which builds the render server

viously acquired volume dataset of the examined patient and provides, if necessary, some additional input parameters. Finally, the user can start the analysis by demanding the generation of either a couple of video sequences or of a medical object movie. The web server processes the demand and sends a standardized analysis request to the render server via the TCP/IP connection. At the render server a dedicated cluster node (*manager node*) takes the request, divides it into several jobs, and distributes them to the other cluster nodes (*render nodes*). The render nodes perform the jobs in parallel and notify the manager node when they have finished. When all jobs are terminated the manager node informs the web server, which then provides the videos or the object movie to the user via the web interface.

5.3.2 Render Server

The *render server* represents the rendering back end of the visualization service. It is responsible for the processing of analysis tasks and for the generation of standardized batch visualizations. For design and implementation of the render server two demands were taken into account. On the one hand, it should be applicable to different medical analysis tasks; on the other hand, it should easily be possible to integrate new analysis and visualization functionality.

To achieve these goals, the render server provides a standardized communication interface through which it accepts standardized requests. Hereby, it is possible that the render server can be accessed by several different clients, which potentially realize different visualization services for different medical purposes. The only prerequisite is that service clients and render server have access to a shared file system that can be used for data exchange (e.g. of the examined volume data set). Furtheron, the sup-

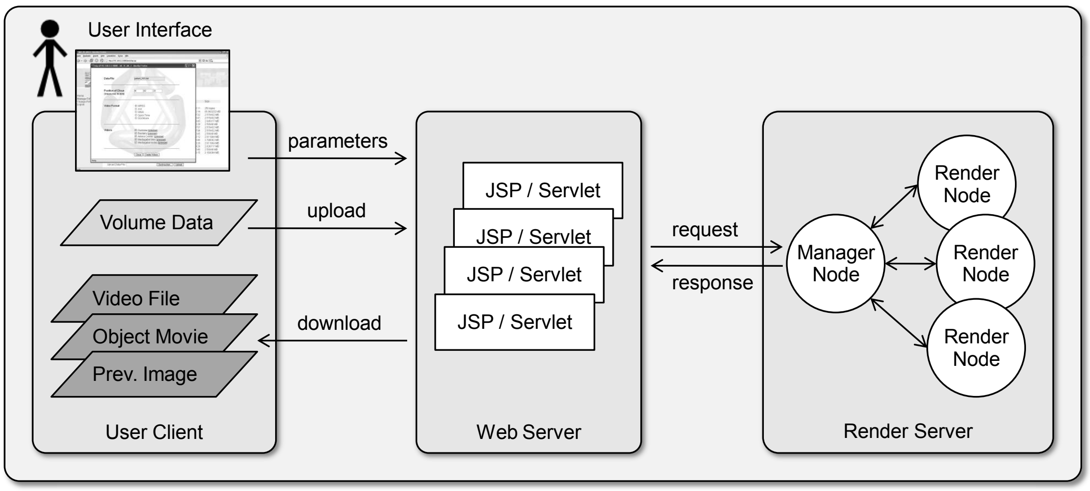


Figure 5.10: Workflow of automated visualization: A user demands the generation of video sequences or of a medical object movie via a web interface. The web server takes this request and sends it to the render server. There the request is distributed and performed in parallel. After finishing, the video sequences or the object movie are provided for download.

ported requests provide functionality on a level that abstracts from the current analysis task. Functionality which depends on the specific medical application, e.g. the determination of the clivus position (see Section 5.1), has to be contributed by the respective service client.

Parallel Processing of Analysis Requests

The render server is realized as an MPI program. This means that it acts like a single program consisting of several processes that run on different cluster nodes. The advantage of MPI is that it works with arbitrary hardware. Thus, an MPI program can either run on a cluster, a parallel computer or even on a single PC. When the program is started a predefined number of similar MPI processes are generated and automatically distributed to the available hardware. Since each process is executing the same program, a single MPI process has to distinguish its specific role by its unique rank. Usually, the process with rank 0 is acting as the so-called master that distributes the tasks to the other processes (the slaves) and merges the results. Figure 5.11 shows how the work is divided on the render server. The handling and distribution of render request is carried out by the manager process (master), whilst the render processes (slaves) perform the rendering and video encoding.

The manager process is composed of three major modules (see Figure 5.11 middle): the *TCP server*, the *job manager* and the *job dispatcher* . The TCP server module realizes functionality for external communication. It binds to a specified port and is waiting for incoming connections from different clients. If the TCP server has established a connection and has received a request, it passes this request to the job manager.

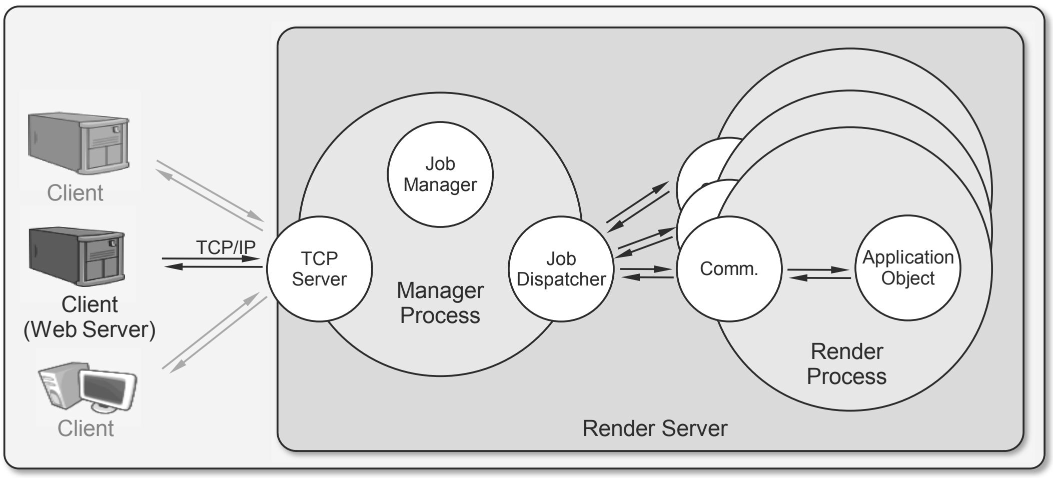


Figure 5.11: Architecture of the render server: It consists of a single manager process and several render processes, which actually perform the visualization requests.

This takes the request, splits it into a number of parallely executable jobs and stores them in a job queue. Finally, the job dispatcher distributes these jobs via MPI to the render processes. For this purpose, it continously monitors the states of the render processes and sends a job to the next one that is idle. This technique guarantees the optimal exploitation of the hardware and permits accepting of multiple requests at the same time because the requests are buffered until the required hardware is on hand.

Normally, the jobs are processed in the order they are queued and, if possible, executed in parallel. However, there are cases where a previous job has to be finished before a successor can be started. For example, the transfer function has to be generated prior to any batch visualization (see below). For this purpose, it is possible to define dependencies on preceding jobs, and a dependent job is put on hold until all predecessors are completed. Further, to each job processing priority can be applied that ensures its privileged treatment. Through this, jobs that need a fast response can be executed preferentially, even if the job queue is crowded with other long lasting requests. A typical example is the generation of a preview image, that should be provided to the user as fast as possible.

After the generation and distribution of the jobs, they are executed by the render processes, which consist of two modules (see Figure 5.11 right). On the one hand, there is the *communicator*, which acts as the counterpart of the *job dispatcher* at the manager process. It takes an assigned job via MPI, performs some preparations and delegates it to the other module, the *application object*. This is responsible for the execution of the job, e.g. the rendering of the frames of a video sequence. Thus, when new functionality should be added to the render server, basically the application object has to be extended.

Standardized Analysis Requests

Currently, the render server supports three types of analysis requests. One for the creation of a set of video sequences, one for the creation of object movies, and one for the rendering of preview images of the video sequences and object movies. The preview request is intended for giving the user a quick impression of the rendering result, before starting a time-consuming video or object movie generation request.

A request contains all parameters that are needed for the execution of the demanded task. To keep the network traffic low, the visualization data itself is not sent along with the request. Instead, a service client has to store the data on the shared file system and to give the corresponding file names via the request. The common parameter of the three supported request types is the file name of the observed volume data set. Moreover, the requests all apply the concept of automated transfer function generation, which was introduced in Section 5.1. To allow the application of different types of datasets and different transfer functions, the requests holds two extra parameters that give the filename of a specific template transfer function and of the associated template dataset. Since transfer function generation is an expensive task, an optional parameter can be defined that gives a file name for storage of the individually adjusted transfer function. When the dataset-specific transfer function already exists, it is loaded from the file system. Otherwise, it is generated and saved under the specified name.

Video Request A video request provides the generation of one or more video sequences of the observed dataset. Besides the common parameters for the specification of the dataset and the transfer function, the camera paths have to be given to the render server. They are stored in external files and describe the animation of the camera by the concatenation of six basic types of movement: translations along the three coordinate axis and rotations around the three axis. For each movement a value (length for translations, degree for rotations) and a duration in seconds has to be defined. Jumps between camera positions can be realized by movements with a duration of zero seconds. The render server, namely the application object, transforms the continuous camera path into a sequence of discrete camera positions. By the demanded frame rate of the video sequences, which has to be given as an additional request parameter, it is determined which camera positions have to be taken. Furthermore, the applied video codec can be chosen.

For each video sequence of a video request it can be defined if the visualizations should be generated from the complete volume dataset or from a previously extracted sub-volume. If a sub-volume should be taken, the center and the extent of the sub-volume has to be given. In addition, an individual clip plane can be defined for each video sequence which cuts away parts of the visualized volume dataset.

When the job manager receives a video request, it primarily checks if the dataset-specific transfer function already exists, or if it has to be generated first. In the latter case, the job manager creates a job for transfer function generation and appends it to the job queue. Then several jobs for the generation of the requested video sequences are created. To ensure that they are not executed before the transfer function exists, they are marked as dependent from the transfer-function job.

The number of created video generation jobs depends on the applied parallelization strategy. An obvious approach is the parallelization on the level of complete video sequences. Here, for each requested video sequence a separate job is created. Thus, if enough render processes are available, all requested videos can be rendered and encoded in parallel. But in case that there are less video-generation jobs than idle render processes, or if the different video sequences need different time for generation, the available hardware is not optimally exploited. To overcome these drawbacks, the generation of a single video sequence can be distributed to several smaller jobs. Each job is responsible for rendering and encoding of a certain part of a video sequence, and in the end the video parts are merged to a single video. Thereby, the work load of the available hardware resources can be better balanced. However, it has to be taken into account that the video parts have to be merged, which is a potentially time consuming task.

The render server supports both approaches for parallelization of the video generation. The subdivision of the video sequences is realized on the level of frames. Therefore, the complete number of frames of a video sequence is calculated, the video sequence is split into parts of fixed frame numbers, and for each part a video generation job with additional infomation about the start frame and the frame number is created. In addition, an extra job is created that is responsible for the merging of the video parts.

Object Movie Request An object movie request provides the generation of a single medical object movie. It is similarly structured like a video request, but instead of camera paths for video sequences the structure of the object movie has to be defined. For this purpose, basically the format of the meta info file (see Section 5.2.1) is used. The major difference is that the links to the byte addresses of the views are missing. In addition, for each sub-movie it has to be defined if it should visualize the complete volume dataset or just a sub-volume. Furthermore, to each visualization mode of a sub-movie an optional clip plane can be applied. Finally, the image format and the output quality (degree of compression) can be chosen.

Similar to a video request, an object movie request is split into one job for transfer function computation and several jobs for the rendering of the object movie views. For each tile in the object-movie a separate rendering job is created. The render processes store the images of the tiles in sequential order in temporary files. In the end, a finalize job merges the temporary files to a single image file and writes the byte addresses of the images to a newly created meta info file. Since a medical object movie usually consists of a large number of small tiles, a large number of small render jobs is created that are equally distributed to the available render processes. Thus, in the average case a good balanced work load is achieved.

Preview Request A preview request precomputes a single view from a set of video sequences or from an object movie. The base structure of a preview request is the same like that for the corresponding video or object movie request. In addition, it has to be defined which view should be prerendered. For video previews this is done by specifying the concerned video sequence and the point in time at which the preview should be taken. For object movie reviews the affected sub-movie and visualization mode, and an id that specifies the location of the view have to be defined. To give a fast feedback to the user, even if there are other requests in the queue, a preview request gets a higher processing priority than the other two request types.

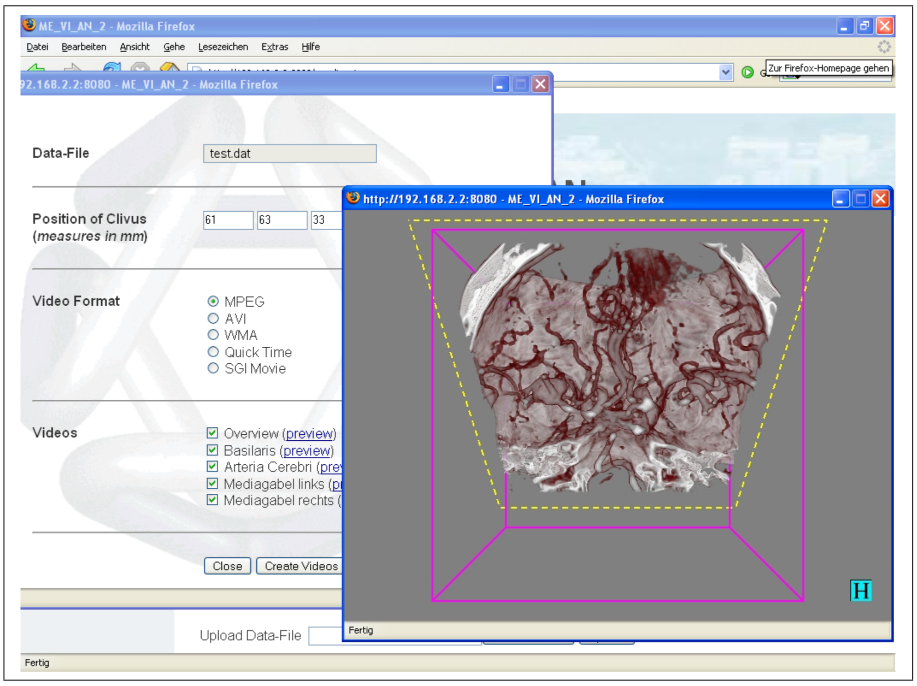
5.3.3 Web Application

As mentioned before, the render server can be accessed by arbitrary clients that realize a visualization service for a certain medical task. Currently, there is a single service client, a web server that provides a web application for the standardized analysis of intracranial aneurysms. The web application consists of a web-based user interface and a base module. The base module takes the user requests, maps them to requests for the render server and manages the communication with the render server. Furthermore, it is responsible for the maintenance of the user data and the batch visualizations that were created by the render server.

Each registered user of the web application can access the visualization service via the web interface. After login a user can upload patient specific volume datasets and start different analysis tasks on the data. Currently, three types of analysis tasks can be carried out: the generation of video sequences, the creation of an object movie and the rendering of a preview image.

Figure 5.12 shows a screenshot of the web interface when the generation of video sequences is requested. On the left hand side a web form can be seen in which the user has to specify several parameters that control the video generation task. The most important parameter is the position of the clivus, which is needed for the determination of the sub-volumes and the clip plane. Furthermore, it can be defined which video codec should be used, and it can be selected which video sequences should be generated. The service can generate up to five video sequences: one overview video, and four videos of sub-volumes that enclose areas with a high risk for intracranial aneurysms (see Section 5.1). For each of the video sequences the user can demand a preview. In this case the first frame of the respective video sequence is rendered by the service and immediately presented in an extra browser window (see Figure 5.12 right). This allows the user getting a first impression of the visualized object. Thereby it can be checked if a video will show relevant information, e.g. an aneurysm. If not, the respective video sequence can be deselected in the request form to save time for the overall analysis.

The web form for the request of an object movie looks similar to the form for video sequences. But in contrast the user can select here which sub-movies and visualization modes should be added to the object movie, whereby the object movie structure is chosen as it was described in Section 5.2.3. As for the video sequences, a preview


ME_VI_AN_2 - Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
Zur Firefox-Homepage gehen
92.168.2.2:8080 - ME_VI_AN_2 - Mozilla Firefox
http://192.168.2.2:8080 - ME_VI_AN_2 - Mozilla Firefox
Close
Create Videos
Upload Data-File
Fertig
Fertig
H

Data-File test.dat

Position of Clivus (*measures in mm*) 61 63 33

Video Format
[✓ MPEG]
AVI
WMA
Quick Time
SGI Movie

Videos
☑ Overview (preview)
☑ Basilaris (preview)
☑ Arteria Cerebri (pre
☑ Mediagabel links (pr
☑ Mediagabel rechts (

Figure 5.12: A screenshot of the web interface of the service for standardized analysis of intracranial aneurysms. In the form on the left hand side the user has to specify the clivus position and can choose the video format and the video sequences that should be generated. On the right hand side a preview image of the overview video is presented.

image can be demanded for each sub-movie and visualization mode.

When the web application gets a user request, the base module takes the clivus position and computes the positions of the sub-volumes like described in Section 5.1. Then, it creates a request where it explicitly defines parameters, like the volume dataset that should be visualized and the template transfer function that should be chosen. Due to the user input, further parameters are added which determine the structure of the requested video sequences or sub-movies. Finally, the request is send to the render server, which processes it in parallel (see Section 5.3.2). When the render server has finished the request, it informs the web application, which provides the visualization results to the user via the web interface. Generated video sequences can be downloaded and watched with an arbitrary video viewer; generated object movies are provided via the Java Applet viewer (see Section 5.2.2), which is integrated into the web interface. Thus, no explicit download of the object movie is required.

5.3.4 Performance

The performance of the visualization service was tested with the hardware configuration described in Section 5.3.1 As stated there, the utilized GPU-cluster consists of eight nodes. One node is acting as manager and seven nodes are actively processing

frames per job	600	300	200	100	50
1 $\times$ overview	94	52	37	21	23
1 $\times$ sub-volume	54	29	21	14	15
1 $\times$ overview + 4 $\times$ sub-volume	94	57	57	60	65

Table 5.1: Video generation times in seconds for different subdivision strategies from 600 frames per video part to 50 frames per video part. For each strategy the duration for the generation of the overview video, for the generation of a single sub-volume video, and for the combined generation of the overview video and four sub-volume videos was measured. All generated video sequences have a total length of 600 frames.

rendering jobs. In order to find the best approach for exploiting the available rendering hardware, several parallelization configurations have been examined. The tests were carried out with a CTA volume dataset that has an original resolution of $512\times512\times246$ voxels and a sub-volume size of $256\times256\times199$ voxels. Each generated video consists of 600 frames and a frame has a resolution of 640×480 pixels. The videos are encoded in *mpeg*-format. The applied camera paths were chosen like proposed in Section 5.1.

Time measurements have been taken for three different application scenarios: single generation of the overview video, single generation of a sub-volume video, and generation of all five videos in parallel. For each of these scenarios, five different subdivision strategies from 600 frames per video part (no subdivision) to 50 frames per video part (twelve parts per video) have been tested. The results are shown in Table 5.1.

For the simplest distribution approach, each video is rendered as a whole (600 frames) on a separate render node. In this case, the generation of the overview video takes 94 seconds, the generation of the sub-volume video takes 52 seconds. The total generation time for the set of five videos is also 94 seconds, which is equivalent to the generation time for the most time consuming overview job. Since there are seven render nodes and only five render jobs, the cluster is not exploited optimally. If the videos are split into parts of 300 frames, there are ten render jobs for the five videos, so some of the cluster nodes have to perform two jobs. Due to the better utilization of the hardware the total video generation time goes down to 57 seconds. When the videos are further subdivided there is no further improvement. The rendering times are even getting worse because of the increasing overhead for distribution and merging of the videos. In contrast to that, the performance for the rendering of a single video is improving down to the splitting of the videos into parts of 100 frames. The reason is that six jobs are built, which can be performed completely in parallel on the available render nodes. A further split into parts of 50 frames (twelve jobs) brings no advantage. In addtion, it can be noticed, that the generation time for the videos is not decreasing linearly with respect to the number of involved cluster nodes. This is once again due to the overhead of communication. As a result it can be stated that the hardware is optimally exploited, when the number of distributed jobs is nearby the number of available cluster nodes. However, if the execution time of the rendering jobs differ noticeably, like for the overview video and the sub-volume videos, a subdivision to smaller parts would guarantee a better balanced exploitation of the cluster nodes, whereas the extra effort for distribution management and the assembling of the final videos has also to be taken into account.

The performance for the generation of object movies is similar to that for video generation. E.g., an object movie of about 2300 images is rendered and assembled in i approximately 100 seconds. The parallelization of the object movie generation depends on the structure of the object movie. Thus, it is not possible to apply different sub-division strategies. A preview image is rendered in about three seconds. Since this is near real time the preview functionality can be used for direct control of the visualization result and interactive adaption of the visualization parameters, e.g. the clivus position. To get the complete time for an automated analysis one has to add the time which is needed for the generation of the transfer function. This procedure can not be parallelized and takes about 30 seconds. However, a once generated transfer function is stored by the service and re-used for subsequent visualizations of the same dataset.

5.4 Conclusion

In this chapter the field of automated medical volume visualization was addressed. Automated medical volume visualization provides standardized analysis and visualization procedures for certain medical tasks. These procedures are automatically performed without user interaction, and the results can be examined independently of the visualization system. Automated medical volume visualization requires, on the one hand, an adequate method for the presentation of the automatically generated visualization results and, on the other hand, an intuitive way to access the provided functionality. The system for automated medical visualization that was presented in this chapter takes both requirements into account.

For the presentation of prerendered visualization results 3D object movies were proposed. Object movies provide 2D images of a 3D object that were taken from different camera positions on a spherical hull around the object. With a specialized viewer a user can interactively navigate between these views. For the purpose of medical visualization a new medical-object-movie format was developed, that takes the needs of visual medical analysis into account. In this format visualizations that were generated with different parameter configurations (visualization modes) and visualizations of different parts of a dataset can be combined. Thus, all information that is relevant for a certain analysis task can be joined in a single movie. For presentation of a medical object movie a Java-based viewer was developed that can be integrated into a web page. Due to a sophisticated image download and caching concept a user can directly navigate along the prerendered visualizations that are provided by the movie.

The automated generation of medical object movies was integrated into a web-based visualization service system that performs standardized analysis and visualization procedures according to a predefined protocol. This system consists of a web server that provides the visualization functionality via a dynamic webinterface and a render server that performs the automated analysis and visualization of the data. The render server is build by a cluster of GPU-equipped PCs, which generate video sequences and object movies in parallel. The visualization-service system was successfully applied for the standardized analysis of intracranial aneurysms.

The system, namely the render server, was designed in a way that allows the easy adaption for other medical visualization tasks. However, it has to be stated out that the applicability of the visualization service is restricted to those task, for which the whole process of segmentation, registration and visualization can be completely automated. There are many segmentation and registration methods for specific medical problems that require interaction of the user. But even for those problems, the service could be applied to generate standardized and comparable visualizations.

Chapter 6 Iterative Development of Medical Volume-Visualization Solutions

The previous chapters addressed the problem of medical volume visualization on different levels of abstraction. Basis of all presented techniques is the multi-volume rendering framework that was introduced in Chapter 3. Thereby, the different tools and techniques for interactive and automated volume visualization represent different stages of an iterative development process of medical volume visualization solutions. Since most of the visualization methods were developed in a generic way, they can easily be adapted for other medical purposes. In the following, a common processing model for the iterative development of medical volume visualization solutions is proposed, which is based on the generic visualization concepts that were developed in the course of this thesis and which can be applied for differing medical tasks.

6.1 Four Stages of Medical Volume Visualization

The development model for medical volume visualization solutions consists of four stages (see Figure 6.1). It begins with 2D analysis of the acquired image data goes on with a generic 3D visualization prototype, is then followed by a specific interactive visualization application and ends with an automated visualization service that is applicable for a large group of users.

While the first stage represents the process that is carried out before the application of 3D visualization techniques, the other three employ hardware-supported direct volume visualization on different levels of specialization and standardization. These three stages can be further sub-divided into three phases: generation, application, and validation. In the generation step, a visualization solution of the current development stage is assembled due to the experiences that were gained in the previous stage. Then, the visualization solution is applied to a couple of real world cases, and, finally, it is evaluated if the analysis of the acquired data is supported in the desired way. Usually, the application and validation step are carried out repeatedly until the achieved visualization results fulfill the demands of the medical task. In this case, the generation of the visualization solution on the next development stage can be started. When the

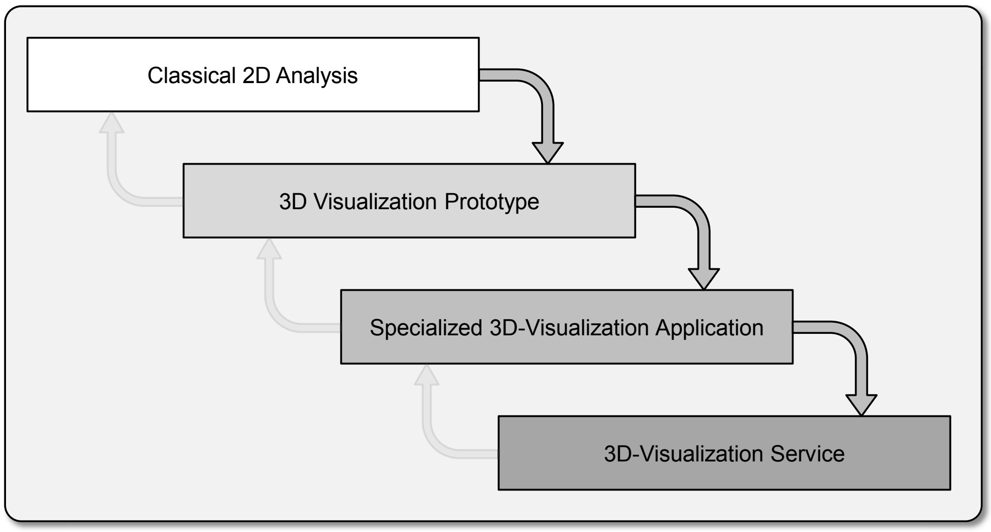

Classical 2D Analysis
3D Visualization Prototype
Specialized 3D-Visualization Application
3D-Visualization Service

Figure 6.1: Processing model for the iterative development of medical volume visualization solutions: The first stage represents the situation before the application of 3D visualization. The other three represent 3D visualization stages on different levels of specialization. In each stage the concepts that where developed for the previous stage are taken into account. When necessary, it can be returned to the previous stage to refine the there developed visualization solution.

desired objectives can not be accomplished on the current stage, it potentially makes sense to return to the previous stage on which the visualization solution can be adapted on a more general level.

Classical 2D Analysis

The development of a medical volume visualization solution always starts with a specific application case for which 3D visualization techniques should be applied. This can for example be the diagnosis of a special disease, the planning of a surgical intervention or the analysis of data that was acquired for research purposes. For this application case usually a workflow has been established that comprises the acquisition of the tomographic images and the analysis of the data in a classical slice-by-slice manner. Thus, the involved medical doctors or researchers have a clear idea about which images should be taken with a certain imaging modality and which information can be extracted from this data. 3D volume visualization techniques can support the analysis of this data in several ways. On the one hand, 3D visualization eases the examination of three-dimensional data, because the spatial relationship between different structures is directly displayed. Thus, the analysis process can be accelerated. Furthermore, 3D visualization provides new insights into the data and allows the application of complex interaction methods.

3D Visualization Prototype

In the first stage of 3D visualization a prototypic volume visualization solution for the acquired image data is developed. For this purpose, the generic volume visualization tool that was presented in Section 4.1 can be employed. This tool allows the interactive manipulation of the render graph and, thus, its application for different visualization purposes. At first, a medical expert, who is familiar with the medical application case, and a visualization expert, who knows the generic volume visualization tool, explore which render nodes have to be combined to achieve the desired 3D presentation of the data. If necessary, new render nodes are implemented and integrated into the system like described in Section 4.1.2. When a visualization is found that best fits to the data, the corresponding render graph can be serialized and stored as a XML file for later reuse. Depending on the application task, several render graphs for different combinations of the acquired datasets can be arranged.

In the application and validation phase the predefined render graphs are applied to a number of selected cases of clinical practice or reasearch. Thereby, it can be evaluated if the developed visualization patterns meet the requirements of the medical task. Usually, the render graphs will be iteratively adapted and extended until the medical experts are satisfied with the results. Further on, it could happen that the prototypic 3D visualizations show that the acquired medical images do not provide adequate information for visualization. In this case, it should be returned to the previous evolution stage to primarily adapt the image acquisition workflow.

Specialized 3D Visualization Application

In the next development stage a customized volume visualization application is developed that is based on the visualization patterns that were designed in the previous prototype stage. This application can utilize the multi-volume rendering framework and combine it with a task-specific user interface. The aim is to generate a visualization tool that supports researchers and medical doctors in their daily work. Therfore, the underlying details about the employed render graphs should be hidden, and instead interaction mechanisms should be provided that fit to the application domain. Thereby, the tool should allow the flexible analysis of standard and non-standard cases. When necessary, the underlying rendering framework can be extended by new rendering techniques that permit a better optimization of the visualization process.

The task-specific 3D visualization tool is applied to all cases that occur in clinical or research routine. Since the tool provides a domain-specific user interface it can be employed by researchers and medical doctors who have experience with the medical application but who must not be familiar with the underlying visualization techniques. Thereby, best-practice visualization workflows can be elaborated that support the analysis task in an optimal way. If it is discovered that the specialized application does not provide all visualization concepts that are necessary for an adequate visual analysis of the data, one can return to the previous stage, refine the prototype and integrate the improved techniques into the specialized visualization tool.

3D-Visualization Service

Based on the visualization workflows that were elaborated in the previous stage, in the last stage of the processing model a 3D visualization service is developed that automates and standardizes the visualization process. Therefore, the visualization service system that was presented in Section 5.3 can be employed. The application object (see Section 5.3.2) has to be extended by the new analysis and visualization techniques that were developed for the specific application case. Furthermore, structures for video sequences and object movies have to be designed that represent the analyzed datasets. When necessary, new analysis requests for the new tasks have to be developed and to be integrated into the system. Finally, a task-specific web interface has to be composed that provides intuitive access to the service functionality.

The service can be employed by a large group of users. At first, medical doctors and researchers can use it to improve and speed-up the analysis of the medical data. Since most steps of the analysis and visualization process are automated, the service can also be operated by other medical personnel who has less knowledge about the medical details. The service functionality can even be exploited by doctors in smaller medical facilities who are not experts for the specific medical task. The service further allows the incorporation of experts from remote locations.

There are two situations which can necessitate the return to the previous evolution stage. First, it can be found that a visualization workflow is not suited for automation. Then, the workflow has to be appropriately adapted and evaluated with the corresponding interactive visualization tool. The other situation can occur during the regular application of the service when the predefined workflow does not provide an adequate visualization for the current data. In this case, a medical expert should additionally examine the data with the interactive 3D visualization tool. Consequently, the visualization service does not replace the related interactive visualization application, but the two complement one another. The service can be used for standard cases, and when problems occur, the interactive application can be additionally consulted.

6.2 Discussion

The proposed processing model covers the complete life cycle of a medical volume visualization solution. Beginning with the initial request for the application of 3D volume visualization techniques, in each step the solution is becoming more specialized and the visualization workflow is becoming more standardized. With the increasing specialization and standardization the number of potential users and the amount of analyzed cases can also become larger. Table 6.1 summarizes these aspects and illustrates the different application fields of the different evolution stages.

As mentioned before, the different evolution stages consist of three phases: the

#	Stage	Developers	Users	Application	Workflow	Tools / Examples
1	**2D Analysis**	Medical researchers	Medical researchers	Medical and clinical research, few research or clinical cases	individual	Analysis of functional brain images (Section 4.2.1), analysis of intracranial aneurysms (Section 5.1)
2	**3D Prototype**	Medical researchers, visualization expert, render-graph programmer	Medical experts	Research hospital, selected clinical cases	defined	Generic multi-volume visualization tool, (Section 4.1), direct volume deformation (Section 4.3)
3	**3D App**	Medical experts, visualization expert, application programmer	Medical doctors	Large hospitals, clinical routine, clinical studies	specialized	Visualization of functional brain images (Section 4.2)
4	**3D Service**	Medical doctors, visualization expert, service programmer	Medical personnel	Hospitals and other medical facilities, medical routine	standardized	Visualization-service system, standardized analysis of intracranial aneurysms (Chapter 5)

Table 6.1: Comparison of the four development stages of medical volume-visualization solutions

development of the visualization solution, the application to a number of cases, and the evaluation of the benefit of the solution. The development is done by a number of medical specialists, who have knowledge about the medical task, an expert for medical visualization and a programming expert, who can implement the visualization solution for the current evolution stage. The visualization expert acts as intermediary between the medical specialists and the programming expert. The medical specialists who are incorporated in the development process can be any of the users of the visualization solution on the previous evolution stage.

For the first stage – the 2D analysis stage – the developers and the users are the same group of persons. Usually, a small group of medical researchers have developed an imaging and analysis concept for a certain medical task and at the same time they are the ones who apply these concepts to a couple of research or clinical cases. The visualization workflow is not yet fixed and can be individually handeled by different researchers. Examples for this visualization stage are the analysis of functional brain images by cognitive neuroscientists (see Section 4.2.1) and the examination of intracranial aneurysms by radiologists and neuro surgeons (see Section 5.1).

The medical researchers that have established the first development stage can develop the 3D visualization prototype from the second stage in cooperation with a visualization expert and a programmer who has deep knowledge about the render graph framework. The visualization prototype can be used by a number of medical doctors or researchers who are experts for the specific application task. They have to evaluate the applicability of the visualization prototype with selected clinical cases. The visualization workflow is predefined by the visualization prototype but can be easily adapted when required. Usually, the generic multi-volume visualization tool (see Section 4.1) can be applied for this stage. Another example for a 3D visualization prototype is the direct volume deformation technique that was presented in Section 4.3.

The specialized 3D visualization application of the third stage is implemented by an application programmer, who takes into account the experiences that the medical experts have made in the previous stage. This application can be used by any medical doctor who wants to perform the specific medical task. It is, thereby, not restricted to the research hospital in which it was developed but can be employed by any hospital with specialists for the medical application for which it was designed. The tool can be used for all cases in clinical routine. It restricts the visualization workflow due to its specialized user interface but still allows the adaption of the workflow. When a certain workflow was established, its applicability can be evaluated in a clinical study. An example for a specialized 3D visualization application is the tool for visualization of functional brain images that was described in Section 4.2.

For the fourth stage a service programmer integrates the established and evaluated visualization workflows into the visualization service system (see Section 5.3). Because of the automation of the visualization process, the service can be applied by personell who must not be experts for the medical application. Thus, it can also be used by smaller medical facilities, e.g. to find out if a patient has a certain disease and if he or she has to be send to a specialized hospital. Further on, the service provides the possibility to collect data from a large group of patients. This data could, for example, be used to get a deeper knowledge about a disease or about the success of a therapy. An example is the standardized analysis of intracranial aneurysms, that was detailed in Section 5.1.

Depending on the medical application, a visualization solution may not pass through all 3D visualization stages. For example, when a visualization is only needed for a specific research topic, the development of a visualization prototype can be sufficient. In those cases the users are usually willing to cope with the complexity of the generic volume visualization tool, and it does not make sense to develop a specialized application. Furthermore, there can be cases where it may not be possible to automate the visualization workflow. For example, there can be medical problems where each case needs an individual analysis by an expert. Then it is not necessary to provide a visualization service for this task.

Since all visualization solutions on the different development stages are based on the flexible multi-volume rendering framework, new visualization methods that were developed for a certain medical application can be reused for other tasks. The reuse of newly developed render nodes is obvious, but also specialized rendering techniques can be employed for other purposes. E.g., the technique of deferred multi-volume shading, which was introduced in Section 4.2.3, could be employed for performance improvements in many medical visualization applications. The method for direct visualization of deformed volumes, which was proposed in Section 4.3.2, could for example also be used for the direct visualization of local registration results. Consequently, the framework provides a flexible and extensible platform for most medical volume visualization purposes.

In practice, the described processing model supports (and demands) the close collaboration of medical researchers and medical practitioners. A successful example for such a close collaboration between medical research and medical practice is given by the research institute *Fraunhofer MEVIS* [35] and the affiliated MeVis Medical Solutions AG [83]. There, a process for the transfer of research results in medical image analysis and medical visualization is established. While Fraunhofer MEVIS is responsible for basic research, MeVis Medical Solutions AG produces software products for medical analysis and visualization that incorporate the research results of Frauenhofer MEVIS. Basis for the work of both institutions is *MeVisLab* [84], which is a software framework for medical image processing and visualization. It provides, on the one hand, a flexible GUI with which a sophisticated user can interactively combine existing functionality via a graph. On the other hand, it can be used as a platform for the development of task-specific medical applications. Thus, it can applied for the fast exploration of new application fields, and the findings can be easily integrated into new software products.

Summarizing, MeVisLab can be employed in a similar way like the flexible volume rendering framework that was developed for this thesis. Morevover, MeVis Medical Solutions AG provides *MeVis Distant Services* [82] for remote preparation of liver surgeries. However, as far as we know, MEVIS has not introduced a fixed processing model for the development of medical analysis and visualization solutions as the one proposed here, and the analysis and visualization service for liver surgery is not automated but provided manual by a team of medical specialists and visualization experts.

CHAPTER 7 CONCLUSION

This thesis addressed the problem of volume visualization for medical applications on three different levels of abstraction. On the lowest level – the level of rendering – a flexible framework for GPU-based rendering of multiple volume datasets was introduced. With the so-called render graph complex visualizations of arbitrary multi-volume scenes can be designed. The render graph is build of a number of render nodes that represent different stages of the shading process, like clipping of a volume against a plane or mapping a sample to a color due to a transfer function. Each render node provides the information that is needed for the evaluation of its specific responsibility. From this information a shader generator assembles a number of shader programs for GPU-accelerated rendering. The modular design of the framework allows the combination of the render graph concept with different volume rendering techniques. Here, two different techniques for GPU-based multi-volume rendering have been proposed: slice-based multi-volume rendering and GPU-based multi-volume ray casting.

The multi-volume rendering framework provides a high degree of flexibility, which allows its employment for a large variety of medical visualization purposes. On the second level of abstraction – the level of interactive medical volume visualization – several techniques have been presented that not only demonstrated the framework's flexibility but which have also shown that new rendering techniques can easily be integrated by slight adaptions. First, a tool for generic multi-volume visualization was introduced that allows the direct configuration of the render graph on a graphical level. Furthermore, it provides a standardized interface for extension by new render nodes and can, thus, easily be applied for new visualization problems.

The second application considered the visualization of functional MRI (fMRI) images of the human brain. Those fMRI volumes are, for example, acquired for studies in cognitive neuro science. The challenge here was to provide the anatomical brain structure without occluding the functional information. Besides several standard multi-volume visualization techniques, like clipping and rendering of semi-transparent iso-surfaces, the application of line integral convolution (LIC) was proposed as a solution of this problem. With this flow visualization technique the curvature of the anatomical brain surface can be emphasized while simultaneously the opacity can be reduced. For fast computation of the LIC integral a technique for deferred multi-volume shading was integrated into the multi-volume rendering framework, which permits the restriction of expensive shading operations to the effective visible surface of a volumetric object.

As a third interactive volume visualization technique a method for direct volume deformation was presented. This method employs the physically-inspired 3D ChainMail algorithm and performs the complete pipeline of deformation and visualization on the GPU. Therefore several GPGPU shaders are evaluated that deform the volume with respect to the current mouse movement. From the deformed volume an inverse deformation volume is generated by a gradient-decent-like inversion algorithm. The inverse deformation volume allows the direct visualization of the deformation by the standard rendering techniques that are provided by the multi-volume rendering framework.

On the highest abstraction level techniques for automated medical volume visualization were considered. These techniques automatically generate a couple of predefined visualizations of a medical volume and present these to a medical doctor in an intuitive way. First, a concept for the intuitive presentation of the pre-rendered visualizations was presented, which is based on 3D object movies. For the purpose of medical volume visualization a new object movie format was developed that allows the incorporation of sub-movies from different parts of a volume and of sub-movies that were taken with different visualization configurations. For the automated generation of medical object movies and video sequences a visualization service system was developed. Here, the user can upload a volume dataset via a task-specific web interface and request the generation of an object movie or a couple of video sequences. Then, the object movie or the video sequences are rendered in parallel on a cluster of GPU-equipped PCs. Thereby, the visualization results are available after a few minutes and the user can access them via the web interface. The service system was designed in a way that allows the application for different medical purposes. It was successfully tested on the standardized analysis of intracranial aneurysms.

The visualization techniques that were developed for this work do not only cover different levels of abstraction, but also represent different stages of development that can be reached by a medical visualization solution. To emphasize this fact a processing model for the development of medical volume-visualization solutions was presented, and it was demonstrated that this thesis provides for each development stage a basic visualization technique that can be adapted for a specific medical purpose. The processing model starts with a specific medical task for which medical specialists have already established an informal workflow for image acquisition and classical 2D analysis of the data. From these experiences an initial 3D visualization prototype on the basis of the generic multi-volume visualization tool is generated. Different visualization concepts can be evaluated with this prototype. Then, an interactive visualization application is developed that provides a task-specific user interface. This application can be applied for all cases in clinical practice and different visualization workflows can be elaborated. In the last development stage these workflows are automated and integrated into the web service system. Then, the visualization solution is available for a large group of users in medical facilities of different sizes. Even non-experts can employ it, and when needed, a specialist can be consulted.

Summarizing, in this thesis a volume visualization concept was presented, that covers the whole life cycle of a medical volume visualization solution. The framework for GPU-based multi-volume rendering builds the foundation of this concept. Due to its modularity and its extensibility, the framework can be employed for a wide range of medical applications. Visualization and rendering algorithms that were developed for a certain medical task, can be integrated into the framework and applied for other purposes. Thus, even sophisticated GPU-based volume visualization techniques can be quickly established in medical practice. All in all, the proposed visualization concept helps bridging the gap between modern volume visualization techniques and medical applications.

However, the medical visualization pipeline does not only consist of the rendering step but incorporates several preprocessing and analysis steps, like filtering, segmentation and registration (see Section 2.2). Since these steps often process the volume data in a similar way like rendering, it is obvious to also exploit the parallel computing power of modern GPUs for these tasks [119]. GPUs have already been successfully employed for medical image processing tasks, but usually those solutions are isolated and not integrated into the visualization process. Thus, the future challenge of medical visualization is the integration of fast GPU-based algorithms into the complete medical visualization pipeline and the seamless combination of preprocessing, analysis and rendering of medical volume data. For this purpose, the GPU-based volume processing techniques have to be encapsulated, and access patterns on a more abstract level have to be provided. Then, the different techniques can be quickly combined and new medical volume processing and visualization concepts can be easily investigated. Herewith, a processing model similar to the one for the development of medical volume visualization solutions that was proposed in this thesis can be established for the complete medical volume visualization pipeline. Beginning with the investigation of new GPU-based volume processing techniques, interactive and task-specific applications can be developed and, finally, the complete volume analysis and visualization process can be standardized and automated by a service.

BIBLIOGRAPHY

[1] G. D. Abram and T. Whitted. Building block shaders. *SIGGRAPH Computer Graphics*, 24(4):283–288, 1990.

[2] Apache Software Foundation. Apache Tomcat. Web site: `http://tomcat.apache.org`, 2009.

[3] Apple Inc. QuickTime VR. Web site: `http://developer.apple.com/documentation/Quicktime/InsideQT_QTVR/insideqt_qtvr.pdf`, 2009.

[4] B. Aubert-Broche, M. Griffin, G. B. Pike, A. C. Evans., and D. L. Collins. Twenty new digital brain phantoms for creation of validation image data bases. *IEEE Transactions on Medical Imaging*, 25(11):1410–1416, 2006.

[5] N. J. Avis, I. J. Grimstead, and D. W. Walker. Grid enabled remote visualization of medical datasets. *Studies in Health Technology and Informatics (Proc of MMVR 13)*, 111:22–28, 2005.

[6] W. Bethel, C. Siegerist, J. Shalf, P. Shetty, T. J. Jankun-Kelly, O. Kreylos, and K.-L. Ma. VisPortal: Deploying grid-enabled visualization tools through a web-portal interface. In *Third Annual Workshop on Advanced Collaborative Environments*, 2003.

[7] J. Beyer, M. Hadwiger, S. Wolfsberger, and K. Bühler. High-quality multimodal volume rendering for preoperative planning of neurosurgical interventions. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1696–1703, 2007.

[8] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, 1977.

[9] C. Boyd. DirectX 11 Compute Shader. In *ACM SIGGRAPH 2008 Classes: Beyond Programmale Shading*, 2008.

[10] V. Boyer. Automatic and dynamic generation of GPU programs to mix renderings and enhance informations on high scale scenes. In *Proceedings Visualization, Imaging, and Image Processing (VIIP 2007)*, 2007.

[11] K. Brodmann. *Vergleichende Lokalisationslehre der Grosshirnrinde in ihren Prinzipien dargestellt auf Grund des Zellenbaues*. Leipzig, Translated by Laurence Garey as Localisation in the Cerebral Cortex (1994), London: Smith-Gordon, new edition 1999, London: Imperial College Press. edition, 1909.

[12] S. Bruckner, S. Grimm, A. Kanitsar, and M. E. Gröller. Illustrative context-preserving exploration of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1559–1569, 2006.

[13] S. Bruckner and M. E. Gröller. Exploded views for volume rendering. *IEEE Transaction on Visualization and Computer Graphics*, 12(5):1077–1084, 2005.

[14] S. Bruckner and M. E. Gröller. Volumeshop: An interactive system for direct volume illustration. In *Proceedings of IEEE Visualization 2005*, pages 671–678, 2005.

[15] T. Brunet, K. E. Nowak, and M. Gleicher. Integrating dynamic deformations into interactive volume visualization. In *Eurographics / IEEE VGTC Symposium on Visualization (EuroVis) 2006*, pages 219–226, 2006.

[16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[17] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 263–270, 1993.

[18] W. Cai and G. Sakas. Data intermixing and multi-volume rendering. *Computer Graphics Forum*, 18(3):359–368, 1999.

[19] J. Chen, I. Yoon, and E. W. Bethel. Interactive, internet delivery of visualization via structured, prerendered multiresolution imagery. *IEEE Transactions in Visualization and Computer Graphics*, 14(2):302–312, 2008.

[20] M. Chen, C. Correa, S. Islam, M. W. Jones, P.-Y. Shen, D. Silver, S. J. Walton, and P. J. Willis. Manipulating, deforming and animating sampled object representations. *Computer Graphics Forum*, 26(4):824–852, 2007.

[21] M. Chen and J. V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, 2000.

[22] R. L. Cook. Shade trees. *SIGGRAPH Computer Graphics*, 18(3):223–231, 1984.

[23] T. F. Cootes, C. J. Taylor, and J. G. D. H. Cooper. Active shape models - their training and application. *Computer Vision and Image Understanding*, 61(1):38–59, 1995.

[24] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, 1994.

[25] H. Dersch. Panorama Tools. Web Site: `http://panotools.sourceforge.net/`, 2009.

[26] O. Dössel. *Bildegebende Verfahren in der Medizin: Von der Technik zur medizinischen Anwendung*. Springer-Verlag, 1 edition, 2000.

[27] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of the 11th IEEE Visualization Conference (VIS 2000)*, pages 195–202.

[28] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-salama, and D. Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006.

[29] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (HWWS) 2001*, pages 9–16, 2001.

[30] A. Evans, D. Collins, S. Mills, E. Brown, R. Kelly, and T. Peters. 3D statistical neuroanatomical models from 305 MRI volumes. In *Proceedings of IEEE Nuclear Science Symposium and Medical Imaging Conference 1993*, pages 1813–1817, 1993.

[31] T. Fangmeier, M. Knauff, C. C. Ruff, and V. Sloutsky. fMRI evidence for a three-stage model of deductive reasoning. *Journal of Cognitive Neuroscience*, 18(3):320–334, 2006.

[32] M. Ferre, A. Puig, and D. Tost. A framework for fusion methods and rendering techniques of multimodal volume data: Research articles. *Computer Animatation and Virtual Worlds*, 15(2):63–77, 2004.

[33] N. Folkegård and D. Wesslén. Dynamic code generation for realtime shaders. In *Proceedings of SIGRAD 2004*, pages 11–15, 2004.

[34] P. T. Fox and M. E. Raichle. Stimulus rate determines regional brain blood flow in striate cortex. *Annals of Neurology*, 17:303–305, 1985.

[35] Fraunhofer MEVIS. Fraunhofer MEVIS - Institute for Medical Image Computing, Bremen. Web site:
`http://www.mevis.de/mre/en/Fraunhofer_MEVIS.html`, 2009.

[36] S. F. Frisken-Gibson. 3D ChainMail: a fast algorithm for deforming volumetric objects. In *Proceedings of Symposium on Interactive 3D Graphics 1997*, pages 149–154, 1997.

[37] S. F. Frisken-Gibson. Using linked volumes to model object collisions, deformation, cutting, carving, and joining. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):333–348, 1999.

[38] K. J. Friston, J. T. Ashburner, S. Kiebel, T. E. Nichols, and W. D. Penny, editors. *Statistical Parameteric Mapping: The Analysis of Functional Brain Images*. Academic Press, London, 2007.

[39] J. Fung. Computer vision on the GPU. In M. Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computations*, pages 649–665. Addison-Wesley, 2005.

[40] J. Georgii, F. Echtler, and R. Westermann. Interactive simulation of deformable bodies on GPUs. In *Proceedings of Simulation and Visualization (SimVis) 2005*, pages 247–258, 2005.

[41] B. Gibaud, O. Dameron, E. Poiseau, P. Toulouse, and P. Jannin. Implementation of atlas-matching capabilities using "web services" technology: Lessons learned from the

development of a demonstrator. In *Proceedings of Computer Assisted Radiology and Surgery (CARS) 2005*, pages 266–271, 2005.

[42] F. Goetz, R. Borau, and G. Domik. An XML-based visual shading language for vertex and fragment shaders. In *Proceedings of the Ninth International Conference on 3D Web Technology (WEB3D 2004)*, pages 87–97, 2004.

[43] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, 1998.

[44] gpgpu.org. GPGPU - General-Purpose Computation Using Graphics Hardware. Web site: `http://www.gpgpu.org`, 2009.

[45] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. Flexible direct multi-volume rendering in interactive scenes. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2004*, pages 379–386, 2004.

[46] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 301–308, 2003.

[47] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[48] M. J. Harris. Fast fluid dynamic simulation on GPUs. In R. Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, pages 635–667. Addison-Wesley, 2004.

[49] P. Hastreiter. *Registrierung und Visualisierung medizinischer Bilddaten unterschiedlicher Modalitäten*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999.

[50] P. Hastreiter and T. Ertl. Integrated registration and visualization of medical image data. In *Proceedings of Computer Graphics International (CGI) 1998*, pages 78–85, Washington, DC, USA, 1998. IEEE Computer Society.

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

[52] G. Hounsfield. A method and apparatus for examination of a body by radiation such as X-ray or gamma radiation. US Patent, 1972.

[53] G. Hounsfield. Computerized transverse axial scanning (tomography). 1. description of system. *British Journal of Radiology*, 46(552):1016–1022, 1973.

[54] V. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 109–116, 1997.

[55] S. Iserhardt-Bauer, P. Hastreiter, T. Ertl, K. Eberhardt, and B. Tomandl. Case study: Medical web service for the automatic 3D documentation for neuroradiological diagnosis. In *Proceedings of IEEE Visualization 2001*, pages 425–428, 2001.

[56] S. Iserhardt-Bauer, P. Hastreiter, B. Tomandl, N. Köstner, M. Schempershofe, U. Nissen, and T. Ertl. Standardized analysis of intracranial aneurysms using digital video sequences. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI) 2002, Part I*, pages 411–418, 2002.

[57] W. M. Jainek, S. Born, D. Bartz, W. Straßer, and J. Fischer. Illustrative hybrid visualization and exploration of anatomical and functional brain data. *Computer Graphics Forum*, 27(3):855–862, 2008.

[58] F. Jargstorff. A framework for image processing. In R. Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, pages 445–467. Addison-Wesley, 2004.

[59] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *SIGGRAPH Computer Graphics*, 18(3):165–174, 1984.

[60] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, V1(4):321–331, 1988.

[61] Khronos Group. *The OpenCL Specification - Version 1.0*, 2009.

[62] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 513–520, 2003.

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

[64] J. J. Koenderink. *Solid shape*. MIT Press, 1990.

[65] A. König, H. Doleisch, and M. E. Gröller. Multiple views and magic mirrors - fmri visualization of the human brain. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1999.

[66] R. Kooper, A. Shirk, S.-C. Lee, A. Lin, R. Folberg, and P. Bajcsy. 3D medical volume reconstruction using web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS) 2005)*, pages 709–716, 2005.

[67] J. Krüger, J. Schneider, and R. Westermann. ClearView: An interactive context preserving hotspot visualization technique. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):941–948, 2006.

[68] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization (VIS '03)*, pages 287–292, 2003.

[69] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[70] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering (NPAR 2000)*, pages 13–20, 2000.

[71] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[72] H. Lehmann, D. Geller, J. Weese, and G. Kiefer. Efficient hardware accelerated rendering of multiple volumes by data dependent local render functions. In *Proceedings of SPIE Medical Imaging 2007: Visualization and Image-Guided Procedures*, pages 6509Z 1–11, 2007.

[73] T. Lehmann, W. Oberschelp, E. Pelikan, and R. Repges. *Bildverarbeitung für die Medizin*. Springer-Verlag, 1 edition, 1997.

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

[75] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, 1987.

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

[77] McConnel Brain Imaging Center. BrainWeb: Simulated brain database. Web site: `http://www.bic.mni.mcgill.ca/brainweb/`, 2009.

[78] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[79] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi. Abstract shade trees. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D) 2006*, pages 79–86, 2006.

[80] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARD II: a reconfigurable interactive volume rendering system. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS) 2002*, pages 137–146, 2002.

[81] H. Melin-Aldana and D. Sciortino. Virtual reality demonstration of surgical specimens, including links to histologic features. *Modern Pathology*, 16(9):958–961, 2003.

[82] MeVis Medical Solutions AG. MeVis Distant Services. Web site: `http://www.mevislab.de`, 2009.

[83] MeVis Medical Solutions AG. MEVIS Medical Solutions. Web site: `http://www.mevis.de/mms/en/index.html`, 2009.

[84] MeVis Medical Solutions AG. MeVisLab: medical image processing and visualization. Web site: `http://http://www.mevis.de/mms/en/Distant_Services.html`, 2009.

[85] Microsoft Corporation. DirectX: Advanced graphics on windows. Web site: `http://msdn.microsoft.com/en-us/directx/default.aspx`, 2009.

[86] J. Montagnat, E. E. Davila-Serrano, and I. E. Magnin. 3D objects visualization for remote interactive medical applications. In *Proceedings of 3D Data Visualization, Processing, and Transmission (3DPVT) 2002*, pages 75–48, 2002.

[87] H. P. Moreton. Simplified curve and surface interrogation via mathematical packages and graphics libraries and hardware. *Computer-Aided Design*, 27(7):523–543, 1995.

[88] E. Mortensen, B. Morse, W. Barrettand, and J. Udupa. Adaptive boundary detection using 'live-wire' two-dimensional dynamic programming. In *Proceedings of Computers in Cardiology 1992*, pages 635–638, 1992.

[89] MPI Forum. The Message Passing Interface (MPI) standard. Web Site: `http://www-unix.mcs.anl.gov/mpi/index.htm`, 2009.

[90] D. R. Nadeau. Volume scene graphs. In *Proceedings of IEEE Symposium on Volume Visualization 2000*, pages 49–56, 2000.

[91] NVIDIA Corporation. *NVIDIA CUDA^{TM} Programming Guide - Version 2.1*, 2008.

[92] S. Ogawa, T. Lee, A. Kay, and D. Tank. Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 87(24):9868–9872, 1990.

[93] OpenGL. OpenGL - The Industry's Foundation for High Performance Graphics. Web site: `http://www.opengl.org`, 2009.

[94] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Larry Seiler. The VolumePro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 251–260, 1999.

[95] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[96] J. Plate, T. Holtkämper, and B. Fröhlich. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1584–1591, 2007.

[97] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof. High performance direct gravitational n-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650, 2007.

[98] B. Preim and D. Bartz. *Visualization in Medicine: Theory, Algorithms, and Applications (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 1 edition, 2007.

[99] W. Qiao, M. McLennan, R. Kennell, D. Ebert, and G. Klimeck. Hub-based simulation and graphics hardware accelerated visualization for nanotechnology applications. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization 2006)*, 12(5):1061–1068, 2006.

[100] M. E. Raichle. *Handbook of Functional Neuroimaging of Cognition*, chapter Functional neuroimaging: A historical and physiological perspective, pages 3–26. MIT Press, 2001.

[101] G. Ramachandran and A. Lakshminarayanan. Three-dimensional reconstruction from radiographs and electron micrographs: Application of convolutions instead of fourier transforms. *Proceedings of the National Academy of Sciences of the United States of America*, 68(9):2236–2240, 1971.

[102] C. Rezk-Salama, P. Hastreiter, J. Scherer, and G. Greiner. Automatic adjustment of transfer functions for 3D volume visualization. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2002*, pages 357–364, 2000.

[103] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast volumetric deformation on general purpose hardware. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 17–24, 2001.

[104] K. Rohr. *Landmark-Based Image Analysis: Using Geometric and Intensity Models*. Kluwer Academic Publishers, 2001.

[105] W. C. Röntgen. Über eine neue Art von Strahlen (vorläufige Mitteilung). *Sitzungsberichte der physikalisch-medizinischen Gesellschaft zu Würzburg 1895*, pages 132–141, 1896.

[106] C. Rorden. MRIcro software guide. Web site:
`http://www.sph.sc.edu/comd/rorden/mricro.html`, 2009.

[107] F. Rößler, R. P. Botchen, and T. Ertl. Dynamic shader generation for flexible multi-volume visualization. In *Proceedings of IEEE Pacific Visualization Symposium (PacificVis) 2008*, pages 17–24, 2008.

[108] F. Rößler, R. P. Botchen, and T. Ertl. Dynamic shader generation for GPU-based multi-volume raycasting. *IEEE Computer Graphics & Applications*, 28(5):66–77, 2008.

[109] F. Rößler, M. Nenov, S. Iserhardt-Bauer, P. Hastreiter, and T. Ertl. Investigating 3D object movies for web-based medical visualization. In *Proceedings of 6. Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie (CURAC '07)*, pages 209–212, 2007.

[110] F. Rößler, E. Tejada, T. Fangmeier, T. Ertl, and M. Knauff. GPU-based multi-volume rendering for the visualization of functional brain images. In *Proceedings of Simulation and Visualization (SimVis) 2006*, pages 305–318, 2006.

[111] F. Rößler, T. Wolff, and T. Ertl. Direct GPU-based volume deformation. In *Proceedings of 7. Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie (CURAC '08)*, pages 65–68, 2008.

[112] F. Rößler, T. Wolff, S. Iserhardt-Bauer, B. Tomandl, P. Hastreiter, and T. Ertl. Distributed video generation on a GPU-cluster for the web-based analysis of medical image data. In *Proceedings of SPIE Medical Imaging 2007: Visualization and Image-Guided Procedures*, pages 650903 1–9, 2007.

[113] R. J. Rost. *OpenGL® Shading Language (2nd Edition)*. Addison-Wesley, 2005.

[114] T. Saito and T. Takahashi. Comprehensible rendering of 3- D shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, 1990.

[115] A. Sayar, M. Pierce, and G. C. Fox. Developing GIS visualization web services for geophysical applications. In *Proceedings of ISPRS International Society for Photogrammetry and Remote Sensing Workshop*, 2005.

[116] T. Schafhitzel, F. Rößler, D. Weiskopf, and T. Ertl. Simultaneous visualization of anatomical and functional 3D data by combining volume rendering and flow visualization. In *Proceedings of SPIE Medical Imaging 2007: Visualization and Image-Guided Procedures*, pages 650902 1–9, 2007.

[117] T. Schafhitzel, D. Weiskopf, and T. Ertl. Interactive investigation and visualization of 3D vortex structures. In *Electronic Proceedings International Symposium on Flow Visualization (ISFV) '06*, 2006.

[118] M. A. Schill, S. F. F. Gibson, H.-J. Bender, and R. Männer. Biomechanical simulation of the vitreous humor in the eye using and enhanced ChainMail algorithm. In *Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI '98)*, pages 679–687, 1998.

[119] T. Schiwietz. *Acceleration of Medical Imaging Algorithms Using Programmable Graphics Hardware*. PhD thesis, Technische Universität München, 2008.

[120] F. Schulze, K. Bühler, and M. Hadwiger. Interactive deformation and visualization of large volume datasets. In *Proceedings of International Conference on Computer Graphics Theory and Applications 2007*, pages 39–46, 2007.

[121] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[122] K. Singh. MRI3DX. Web site: `http://imaging.aston.ac.uk/mri3dX/index.shtml`, 2009.

[123] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.

[124] R. Stokking, K. Zuiderveld, and M. Viergever. Integrated volume visualization of functional image data and anatomical surfaces using normal fusion. *Human Brain Mapping*, 12(4):203–218, 2001.

[125] J. Talairach and P. Tournoux. *Co-Planar Stereotaxic Atlas of the Human Brain: 3-Dimensional Proportional System : An Approach to Cerebral Imaging*. Thieme Medical Publishers, 1988.

[126] E. Tejada and T. Ertl. Large steps in GPU-based deformable bodies simulation. *Simulation Practice and Theory. Special Issue on Programmable Graphics Hardware*, 13(9):703–715, 2005.

[127] TeraRecon Inc. AquariusNET Server. Web Site: `http://www.terarecon.com/products/aq_net_1_prod.html`, year = 2009.

[128] D. Terzopoulos, A. Witkin, and M. Kass. Constraints on deformable models: recovering 3D shape and nongrid motion. *Artificial Intelligence*, 36(1):91–123, 1988.

[129] U. Tiede, T. Schiemann, and K. H. Höhne. High quality rendering of attributed volume data. In *Proceedings of IEEE Visualization 1998*, pages 255–262, 1998.

[130] U. Tiede, N. von Sternberg Gospos, P. Steiner, and K. H. Höhne. Virtual endoscopy using QuickTime-VR panaorama views. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI) 2002. Part II*, pages 186–192, 2002.

[131] B. Tomandl, P. Hastreiter, S. Iserhardt-Bauer, N. Köstner, M. Schempershofe, W. J. Huk, T. Ertl, C. Strauss, and J. Romstock. Standardized evaluation of CT angiography with remote generation of 3D video sequences for the detection of intracranial aneurysms. *Radiographics*, 23(2):e12, 2003.

[132] M. Trapp and J. Döllner. Automated combination of real-time shader programs. In *Proceedings of Eurographics 2007 - Short Papers*, pages 53–56, 2007.

[133] University of Iowa, Hardin Library for the Health Sciences. The Bones of the Skull: A 3-D Learning Tool. Web site: `http://www.lib.uiowa.edu/commons/skullvr/index.html`, 2009.

[134] F. Vega Higuera, P. Hastreiter, R. Naraghi, R. Fahlbusch, and G. Greiner. Smooth volume rendering of labeled medical data on consumer graphics hardware. In *Proceedings of SPIE Medical Imaging 2005: Visualization, Image-Guided Procedures, and Display*, pages 13–21, 2005.

[135] I. Viola, A. Kanitsar, and M. E. Gröller. Importance-driven volume rendering. In *Proceedings of IEEE Visualization 2004*, pages 139–145, 2004.

[136] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. *International Journal of Computer Vision*, 24(2):137–154, 1997.

[137] Visage Imaging Inc. Visage PACS. Web site: `http://www.visageimaging.com/products/visage_pacs.php`.

[138] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, 2003.

[139] D. Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[140] D. Weiskopf, K. Engel, and T. Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[141] D. Weiskopf and T. Ertl. A hybrid physical/device-space approach for spatio-temporally coherent interactive texture advection on curved surfaces. In *Proceedings of Graphics Interface 2004*, pages 263–270, 2004.

[142] Wellcome Trust Centre for Neuroimaging. SPM - Statistical Parametric Mapping. Web site: `http://www.fil.ion.ucl.ac.uk/spm/`, 11 2008.

[143] L. Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990.

[144] O. Wilson, A. VanGelder, and J. Wilhelms. Direct volume rendering via 3D textures. Technical report, 1994.

[145] T. S. Yoo, editor. *Insight Into Images 'Principles and Practice for Segmentation, Registration and Image Analysis'*. A K Peters, Ltd., 2004.

[146] D. A. Yuen, Z. A. Garbow, and G. Erlebacher. Remote data analysis, visualization and problem solving environment (PSE) based on wavelets in the geosciences. *Visual Geosciences*, 9(1):29–38, 2004.