Technische Universität Wien

Dissertation

Real-Time Volume Visualization on Low-End Hardware

ausgeführt
zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Leitung von
A. o. Univ.Prof. Dipl.-Ing. Dr.techn. Meister Eduard Gröller,
Institut für Computergraphik und Algorithmen (186),

eingereicht
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik,

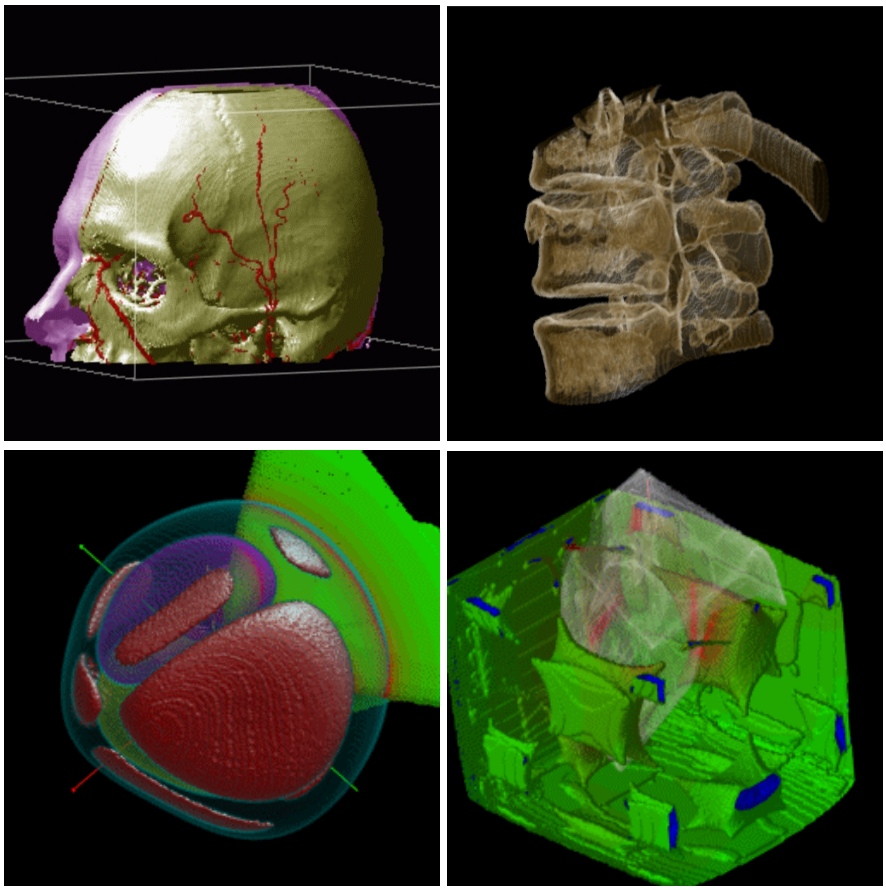
von
Dipl.-Ing. Lukas Mroz,
Matrikelnummer 9125411,
Leystraße 19-21/14/27,
A-1200 Wien, Österreich,
geboren am 10. Okt. 1972 in Warschau, Polen.

Wien, im Februar 2001.

Lukas Mroz

Real-Time Volume Visualization on Low-End Hardware

(PhD Thesis)



http://www.cg.tuwien.ac.at/~mroz/diss/
http://bandviz.cg.tuwien.ac.at/basinviz/
mailto:mroz@cg.tuwien.ac.at

Abstract

Volume visualization is an important tool for investigating and presenting data within numerous fields of application. Medical imaging modalities and numerical simulation applications, for example, produce huge amounts of data, which can be effectively viewed and investigated in 3D. The ability to interactively work with volumetric data on standard desktop hardware is of utmost importance for telemedicine, collaborative visualization, and especially for Internet-based visualization applications.

The key to interactive but software-based volume rendering is an efficient approach to skip parts of the volume which do not contribute to the visualization results due to the visualization mapping in use and current parameter settings. In this work, an efficient way of skipping non-contributing parts of the data is presented. Skipping is done at a negligible effort by extracting just potentially contributing voxels from the volume during a preprocessing step, and by storing them in a derived enumeration-like data structure. Within this structure, voxels are ordered in a way which is optimized for the chosen compositing technique, and which allows to efficiently skip further voxels as they become irrelevant for the result due to changes to the visualization parameters. Together with a fast shear/warp-based rendering and flexible shading based on look-up tables, this approach can be used to provide interactive rendering of segmented volumes, featuring object-aware clipping, transfer functions, shading models, and compositing modes defined on a per-object basis, even on standard desktop hardware. In combination with a space-efficient encoding of the enumerated voxel data, the approach is well-suited for visualization over low-bandwidth networks, like the Internet.

Kurzfassung

Die Visualisierung von Volumsdaten ist ein wichtiges Werkzeug zur Untersuchung und Präsentation von Daten innerhalb zahlreicher Anwendungsgebiete. Bildgebende Verfahren innerhalb der Medizin sowie numerische Simulationen, liefern beispielsweise Datenmengen, die ohne Visualisierung im 3D kaum zu bewältigen wären. Die Möglichkeit interaktiver Manipulation von Volumsdaten auf desktop-PCs ist insbesondere in Hinblick auf Anwendungen in der Telemedizin, kollaborativer Visualisierung sowie der Visualisierung über das Internet von großer Bedeutung.

Eine Vorbedingung für interaktive, softwarebasierte Volumsdarstellung ist effizientes Ausschließen von nicht relevanten Volumsbereichen von der Projektion (nicht relevant sind Daten, die keinen sichtbaren Einfluß auf das Ergebnis der Visualisierung haben). In dieser Arbeit wird ein diesbezügliches Verfahren vorgestellt, das ein Überspringen nicht relevanter Voxel während der Darstellung faktisch ohne zusätzlichen Aufwand ermöglicht. Dazu werden während eines Vorverarbeitungsschritts potentiell relevante Voxel identifiziert und in einer abgeleiteten Datenstruktur gespeichert. Durch entsprechende Anordnung der Voxel innerhalb dieser Datenstruktur, können durch interaktive Veränderung von Visualisierungsparametern irrelevant gewordene Voxel ebenfalls effizient übergangen werden. Zusammen mit einer schnellen, shear/warp-basierten Projektion und einer auf flexibler Kombination von Look-up-Tabellen basierenden Schattierung, können die derart vorbereiteten Volumendaten interaktiv auf PC Hardware dargestellt werden. Da innerhalb der extrahierten Voxeldaten einzelne segmentierte Objekte unterschieden werden können, bietet das Verfahren weitreichende Flexibilität bei der Visualisierung: Wegschneiden einzelner Objekte, individuelle optische Parameter, Transferfunktionen und Beleuchtungsmodelle, sowie die objektweise Wahl des Compositing-verfahrens. Eine effiziente Kompression der Voxeldaten ermöglicht den Einsatz des Verfahrens zur Visualisierung über Netzwerke mit geringer Bandbreite, wie z.B. das Internet.

Related Publications

This thesis is based on the following publications:

- L. Mroz, A. König, and E. Gröller: **Real-Time Maximum Intensity Projection**. Published in *Data Visualization '99, Proc. of the Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization 1999*, pp. 135-144, Vienna, Austria, May, 1999. An extended and revised version republished as
- L. Mroz, A. König, and E. Gröller: **Maximum Intensity Projection at Warp Speed**. Published in *Computers & Graphics*, **24**(3), pp. 343-352, June, 2000.
- L. Mroz, H. Hauser, and E. Gröller: **Interactive High-Quality Maximum Intensity Projection**. Published in *Proc. of the EUROGRAPHICS 2000 Conference*, pp. C-341—C-350, Interlaken, Switzerland, August, 2000.
- L. Mroz, R. Wegenkittl and E. Gröller: **Mastering Interactive Surface Rendering for Java-Based Diagnostic Applications**. Published in *Proc. of the IEEE Visualization 2000 Conference*, pp. 437-440, Salt Lake City, USA, October, 2000.
- H. Hauser, L. Mroz, G-I. Bischi, and E. Gröller: **Two-Level Volume Rendering – Fusing MIP and DVR**. Published in *Proc. of the IEEE Visualization 2000 Conference*, pp. 211-218, Salt Lake City, USA, October, 2000. Revised and extended version accepted for publication in *IEEE Transactions on Visualization and Computer Graphics.*
- G.-I. Bischi, L. Mroz, and H. Hauser: **Studying Basin Bifurcations in Nonlinear Triopoly Games by Using 3D Visualization**. Accepted for publication in *Journal of Nonlinear Analysis*, 2001.
- L. Mroz, H. Hauser: **Space-Efficient Boundary Representation of Volumetric Objects**. Accepted for publication in *Data Visualization 2001, Proc. of the Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization*, Ascona, Switzerland, June 2001.
- L. Mroz and H. Hauser: **RTVR – a Flexible Java Library for Interactive Volume Rendering**. Available as technical report TR-VRVis-2001-003 from `http://www.vrvis.at/`.
- B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller: **Fast Visualization of Object Contours by Non-Photorealistic Volume Rendering**. Available as technical report TR-VRVis-2001-002 from `http://www.vrvis.at/`.

Contents

Chapter 1

Introduction

The need to analyze and visualize volumetric data arises in many fields of application. In medicine, imaging modalities like CT or MR scanners acquire volumetric data sets which are examined using 2D and 3D visualization [13] as a part of routine work. In geo-sciences, volumetric data which is obtained from seismic measurements is used to analyze structure and composition of ground [6]. Numerical simulations (like computational fluid dynamics, for example) produce huge amounts of data which is usually also defined in the 3D domain [12]. Generally speaking, phenomena within a 3D domain can be discretized and represented by a volumetric data set of samples, like, for example, the objects and structures within the phase space of 3D dynamical systems [4].

Depending on the main goal of visualization – ranging from data exploration to presentation – different requirements are put on interactivity and image quality. Interactivity is crucial for efficient exploration and analysis of data. Complex data sets require careful and frequent tuning of visualization parameters to obtain meaningful visualization results. The specification of a proper transfer function [3, 23, 27, 29, 34], i.e., the assignment of optical properties to data values within the volume, is a complex task which profits greatly from immediate visual feedback by interactive rendering. Thus, during the explorative stages of visualization, interactivity and immediate feedback are more important than a high visual quality of the resulting image. For the presentation of the results obtained from data analysis the emphasis is reversed. Often the creation of high-quality visualization results for later presentation can be performed off-line, without user interaction.

Still images or animations are often not sufficient to communicate the complex findings of the analysis process to a viewer. Visualization results are in many cases easier to understand, if the viewer is able to manipulate the visualization

output to a certain degree, by changing at least a restricted sub-set of visualization parameters [54]. This may range from simple manipulation of viewing parameters like camera position and zoom factor, to changes in transfer functions or to clipping of parts of the data. Again, interactive rendering is crucial for providing this possibilities to a viewer efficiently.

The main obstacle for interactive volume rendering is simply the amount of data to be processed for generating an image from a volumetric data set. Typical volume sizes in medicine range from 256^3 voxels for MR data to $512^2 \times 2000$ voxels for data acquired with recent multi-detector CT scanners. For a straightforward approach, this would mean shading and compositing 16-500 million voxels for each single image – a tough task, even for multi-processor hardware. Simple straight forward implementations of volume rendering are only competitive in terms of performance, if directly implemented in hardware – like the VolumePro (vp500) volume rendering board from Real Time Visualization [48].

The usual approach for software-based rendering is to use auxiliary data structures for efficiently skipping of parts of the volume, which do not contribute to the visualization results (totally transparent regions, or inner parts of opaque objects). This approach has an additional advantage: while the effort for the brute-force approach grows linearly with the number of the voxels, and thus is in general $O(n^3)$ for a n^3 sized volume, methods which manage to limit the rendering to a "thick" surface-like neighborhood of the depicted objects, may have an effort in the order of $O(n^2)$ (depending on the transfer function settings).

A special technique may be used, if the display of (iso-)surfaces within the volumetric data is desired. Instead of directly rendering the volume data using an appropriate rendering method and transfer function, an intermediate polygonal representation of the surface is created (for example, using the marching cubes algorithm [33]). The representation of the surface can then be rendered exploiting polygon-rendering hardware. The main disadvantages of this approach are the amounts of time and memory required to extract and store the surface polygons, and the large number of geometric primitives generated by this kind of approach (typically several hundreds of thousands of triangles, for 256^3 data sets). Really interactive rendering of polygonal models of this size is currently not possible on common 3D hardware.

If visualization is carried out within a networked environment, for example, for performing remote diagnosis, collaborative investigation of data, or simply to exploit remote computational resources, the bandwidth required to transmit volume data and/or visualization results poses an additional problem. Sending entire volume data sets over low-bandwidth networks like the Internet is in most cases not feasible. As an alternative, either a reduced resolution volume is rendered

at the client, or the visualization is entirely carried out at a server and just the resulting images are transmitted to the client. Both solutions suffer from problems. Reducing the resolution of the data destroys information. Performing the visualization remotely on a server implies, that even the slightest change in visualization or viewing parameters – like changing the camera position – requires the transmission of new images over the (slow) network, thus hampering interactivity. For further information on interactive volume rendering techniques and their application in networked environments please refer to chapter 2.

Within this work, a novel solution to interactive rendering of volumetric data is presented, which is also well-suited for use in networked environments due to a compact data representation. Although data defined on Cartesian grids is required for rendering, data defined on other types of grids can be transformed to a Cartesian representation (by resampling) for rendering.

Several distinguishing features make the presented method a fast and flexible solution to interactive, software-based volume rendering for low-end hardware:

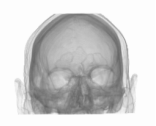
- **preprocessing**: during a preprocessing step, voxels which potentially contribute to a visualization result are identified.
- **voxel enumeration**: possibly contributing voxels are extracted from the volume and stored in a derived data structure, which is basically a list of individual voxels.
- **compact representation**: the extracted voxels are well-suited for efficient compression and can be transmitted over a network for visualization at a remote computer.
- **voxel ordering**: the extracted voxels can be ordered in a way which is optimized for rendering using specific compositing modes and visualization parameter settings.
- **fast rendering**: A fast shear/warp-based rendering [28] is used to project the extracted voxels.
- **object awareness**: if segmentation information is available, extracted voxels can be assigned to individual objects. Visualization parameters can be defined on a per-object basis, allowing to individually adjust opacity and color transfer functions, shading models and even compositing modes, without much impact on rendering performance.

The voxel extraction approach can be seen as a hybrid approach between direct volume rendering, which directly operates on the original volume data, and approaches like marching cubes, which derive a polygonal representation of objects

within the volume for rendering. On one hand, only a secondary data representation, which represents the volume, is used for rendering – the list of potentially contributing voxels. On the other hand, the voxel data within this data structure is just a space-efficient storage representation for a sparsely populated volume.

The basic concepts of the selection of relevant voxels, data representation and rendering are explained in chapter 3. Chapter 4 demonstrates the application of the concepts to implement interactive maximum intensity projection (MIP) and the rendering of iso-surfaces. Furthermore, a general approach for mixing MIP, surface, and direct volume rendering based on opacity weighted blending of voxels (DVR) within a single visualization is presented. The implementation of different shading models (Phong, non-photorealistic shading, ...) is also described. Chapter 5 presents an efficient encoding scheme for compact storage and transmission of extracted voxels. RTVR, a Java library for real-time volume rendering is presented in chapter 6. The library exploits the techniques presented here and combines them with additional features to provide an extendible basis for the creation of flexible visualization tools.

Chapter 7 presents three sample visualization applications which benefit from the presented methods – a Java-based medical viewing and diagnostic workstation, a simple general purpose volume viewer, which can also be included as a volume presentation applet into web pages, and a simulation and visualization application for the investigation of 3D non-invertible maps (discrete dynamical systems).



Chapter 2

State of the Art

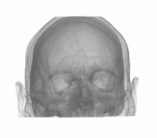
> The researchers of many commentators have already thrown much darkness on this subject, and it is probable that, if they continue, we shall soon know nothing at all about it
>
> *Mark Twain (1835–1910)*

Recent literature related to the methods presented in this work can be subdivided into three major topics: volume rendering in general, volume compression, and network-based volume visualization. In addition to these three major topics which will be treated in the following sections, literature related to specific aspects of the work, for example, maximum intensity projection, will be discussed in the corresponding chapters.

2.1 Volume Rendering

In the following, single data samples within the volume which are given at well-defined positions in three-space will be referred to as *voxels*. Each voxel has a position (x, y, z) and one or more scalar or vector attributes, like density, pressure, and gradient. A scalar attribute will be referred to as *data value*. A *cell* is built up from a set of neighboring voxels which are located at the cell's vertices. When considering a volume as being built up from cells, attribute values within a cell are obtained by interpolation of attribute values at the vertices of the cell. For volumes defined on a Cartesian grid, cells are regular hexahedra. The following descriptions will focus on the rendering of rectilinear grids.

Since the first approaches for direct rendering of volumetric data in the early 1980s, four major groups of techniques have emerged: ray casting [26, 29], splatting [60], shear/warp projection [28], and hardware-assisted rendering based on



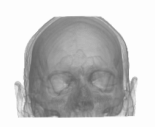
texture mapping [59]. As a special case, the rendering of surfaces from volume data can be performed by constructing a polygonal representation of the surface first and rendering it using polygon rendering hardware [33].

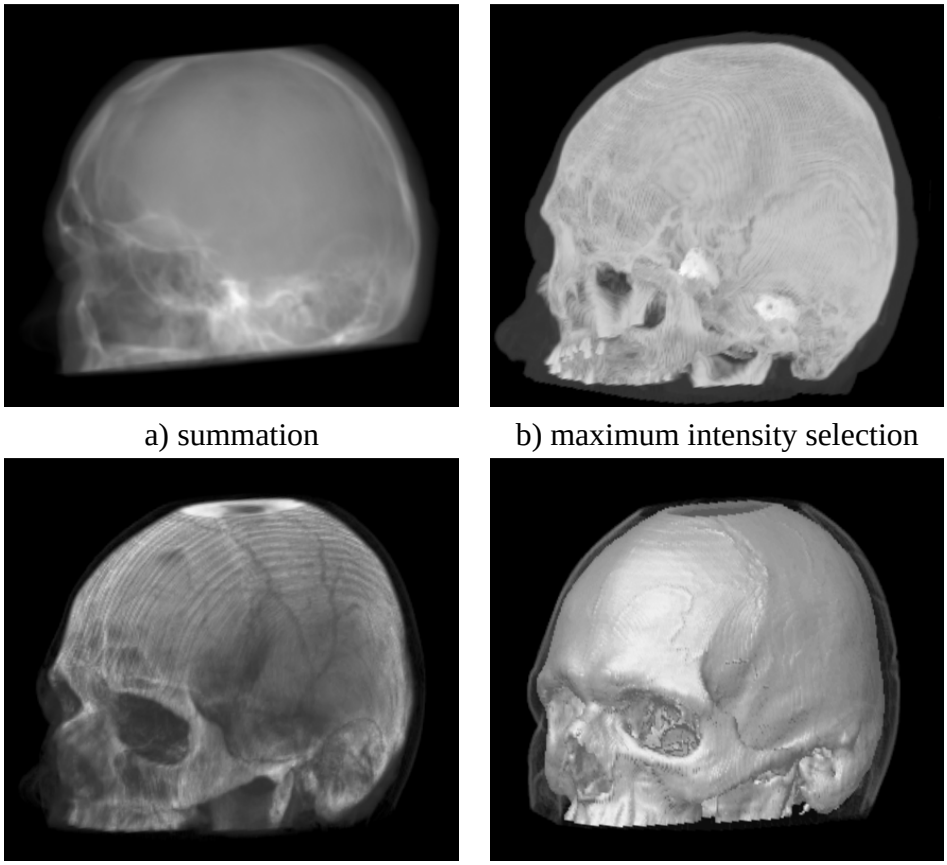
Ray casting is a straight-forward, image-order algorithm. A ray is shot from the eye through each pixel of the image into the volume. Along the ray's intersection with the volume several operations can be performed to obtain the color of the pixel. The operation may be a simple summation of data values along the ray to obtain X-ray like images (figure 2.1a), or the selection of the maximum value along each ray (maximum intensity projection, MIP, figure 2.1b). The most commonly used operation is the integration (or weighted summation in the case of sampled volumes) of color contributions along each ray [35] (figure 2.1c, d). Each data sample within the volume is assigned a set of optical properties (color, opacity, emission and reflection coefficients, ...by the use of so called transfer functions), which determine the contribution of data samples to pixel values.

Ray casting is a rather time consuming method of volume rendering. Performance of ray casting algorithms can be significantly improved, if regions which do not contribute to the image are skipped from rendering. Such regions are parts of the volume, which contain only entirely transparent voxels, or inner parts of objects with a high opacity. Transparent parts of a volume are skipped, for example, by encoding at each voxel of the volume the distance to the closest non-transparent voxel [8, 55, 66]. This information can be used to efficiently skip empty regions. Data within opaque regions can be easily omitted, if the ray is tracked from the eye towards more distant regions. By keeping track of the opacity of the data encountered so far, the ray can be stopped as soon as the cumulative opacity is close to total – further samples would not be visible (early ray termination [29]).

In contrast to ray casting, which computes one pixel of the image at a time, splatting is an object-order algorithm – the contributions of each voxel to all pixels of the image are computed at a time. The area affected by the projection of a voxel (footprint) is usually a circle (for parallel projection) or an ellipsoid (perspective projection). Within the affected area, the voxel contributes to the color of pixels according to a Gaussian distribution (or similar) around the center of the footprint. Empty (transparent) regions of a volume can be easily skipped during splatting. Skipping of opaque, invisible regions (interior parts of opaque objects) is more difficult, as a voxel may not contribute to some pixels of the footprint, but may contribute to others.

Both, ray casting and splatting are considered to be high-quality methods, capable of generating images at arbitrary view parameters, image sizes and quality. The rendering times of splatting and ray casting are comparable, with a better performance of splatting for volumes with large amounts of transparent data. On





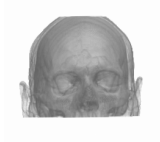
a) summation

b) maximum intensity selection

c) opacity weighted blending (without shading)

d) opacity weighted blending (shaded, surfaces emphasized)

Figure 2.1: Some of the most important image compositing methods for volume rendering



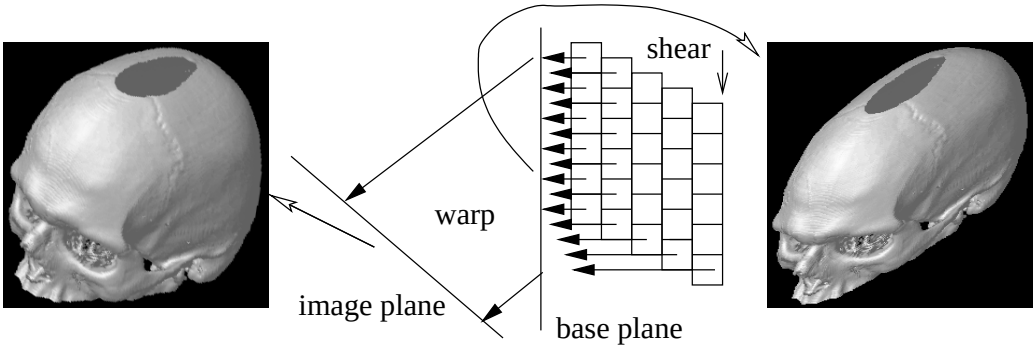


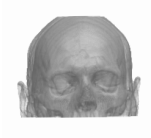
Figure 2.2: Parallel projection using a shear/warp-factorization of the viewing transformation

the other hand, ray casting is perfectly suited for parallel implementation [47], as pixel values are computed independently of each other.

Approaches which utilize shear/warp-based projection (like the one presented in this work) are the fastest software-based methods for volume rendering. The usually quite costly process of transforming data from the volume coordinate system into image coordinates for projection is split into two shears along axes of the volume (plus a scaling operation if perspective projection is performed), and a 2D warp operation. The data is sheared and projected onto one of the faces of the volume (*base plane*, a plane which is normal to an axis of the volume coordinate system). The cheap shear-based projection is performed for all voxels of the volume, creating a distorted version of the rendered image. The warp (which can be, for example, efficiently done by texture mapping hardware) transforms the base plane image into the final image (see figure 2.2).

As the decomposition of the projection into two separate steps requires to perform resampling twice – first as voxels are projected onto the base plane, and second during the warp step, images produced using this technique are more blurred as compared to the results of ray casting or splatting. Usually, no scaling is performed during the projection to the base plane. Each voxel is projected onto an area of approximately one pixel. Thus, zooming into the volume is performed by zooming into the base plane image during the warp step, which leads to stronger blurring as the zoom factor is increased. Usual approaches to accelerating shear/warp-based volume rendering use run-length encoding for sequences of voxels with similar optical properties, for example for transparent regions. All pixels covered by the projection of a run can be treated equally, thus accelerating the rendering.

The texture mapping capabilities of recent polygon-rendering hardware can be exploited to perform rendering of volumetric data sets. Two different approaches



can be distinguished here. Their applicability depends on the capabilities of the used rendering hardware. If the application of 3D textures to polygons is supported, a set of polygons perpendicular to the viewing direction can be placed within the volume and textured using color and opacity information from the volume [59]. By blending the textured polygons in a back-to-front order, the volume is rendered. The quality of the image depends on the number of slices rendered, and is in general lower than the output of software-based rendering. If no 3D texture support is available, single slices of the volume can be mapped as 2D textures on polygons. Three perpendicular sets of polygons and textures are required to avoid viewing the polygons edge-on [50]. The set of polygons which is most perpendicular to the viewing direction is rendered. For small volumes (up to 256^3) hardware-based approaches can achieve frame rates of up to 30Hz even on consumer 3D hardware. Hardware-based approaches usually provide just a subset of the capabilities of software-based renderers, for example, the user may have to choose between color rendering and shading of the volume, but can not apply both simultaneously.

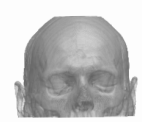
A more detailed comparison of the four classes of volume rendering techniques discussed above has been published by Meißner and others [36].

2.2 Volume Compression

Compression algorithms for volumetric data can be classified according to various criteria. Compression may be lossy, or lossless, the method may use a general compression algorithm or an approach which exploits specific characteristics of volumes. Compression may be based on hierarchical approaches, and suitable for progressive rendering even if only a part of the data is available. Some techniques allow direct rendering of compressed data, others require a decompression first.

The efficiency of compression, of course, depends on the characteristics of the data. In the following, compression rates are given as examples which are typical for data sets from medicine. Lossless compression provides significantly lower compression rates (around 2:1) than lossy compression, which usually allows to choose the desired compression ratio, i.e., quality degradation. Acceptable rendering quality can be obtained for lossy compression by factors of 5–50:1. Despite of the superior compression rates of lossy compression, many applications, like the diagnosis of medical data, prohibit changes to the volumetric data and thus require lossless methods.

Compressing volume data by the use of general purpose tools, like `zip` [18], just exploits coherence in one of the three dimensions of the data. Fowler and



Yagel [17] presented a technique which uses prediction based on the data values of neighbors of a voxel in all three dimensions and which stores just a Huffman-encoded prediction error. Although coherence is exploited better compared to `zip`, the compression ratios are just insignificantly better.

Ning and Hesselink [46] presented a lossy compression method based on vector quantization. They group neighboring voxels into so-called bricks. Attributes of voxels within a brick (data value, gradient, ...) are the elements of a vector, which is then quantized by mapping to the closest representative within a codebook. The compression rates depend on the size of the codebook, and are typically around 5:1, if acceptable quality should be preserved. Rendering can be performed without decompression, by projecting precomputed templates of the codebook entries.

An approach which is similar to the methods used by JPEG compression [57] has been presented by Chiueh and others [7]. The volume is subdivided into bricks which are then transformed into frequency domain using a discrete Fourier transform (JPEG uses the discrete cosine transform in contrast). The coefficients in frequency domain are quantized and the results are entropy encoded. For compression factors close to 30 the rendering results exhibit an acceptable image quality. Rendering can be performed without prior transformation into spatial domain. Within the bricks, frequency domain rendering [30] is performed. If summation rendering (X-ray like attenuation) is performed in-between the bricks, a difference to rendering of the uncompressed volume is hardly noticeable. Other compositing techniques between the bricks lead to visible artefacts.

Lippert and others [32] presented a volume compression technique based on wavelet decomposition, which is well-suited for progressive rendering and transmission over networks. Coefficients of the wavelet representation are quantized and stored in a way, which allows to reconstruct and render a preview of the volume even if only a part of the data is available. In general, hierarchical (for example, wavelet-based) approaches are the most flexible form of volume compression. On one hand, sufficiently accurate coefficient information can be stored to allow lossless reconstruction of a volume. The compression rates achieved in this case are comparable to other lossless techniques. On the other hand, a proper arrangement of the coefficient information in the compressed data file or stream, allows to approximately render a volume using just a small fraction of the whole data.

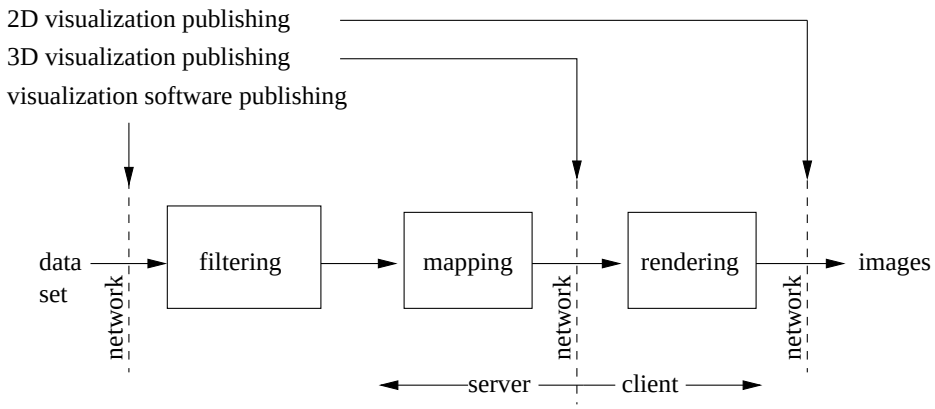


Figure 2.3: Client/server visualization pipeline with network

2.3 Networked Volume Visualization

After the appearance of the first easy to use web browsers, the importance of networks in general and especially the Internet for providing visualization to remote users has been soon realized. Considering the visualization pipeline [21] (figure 2.3) as a general model for visualization, three major approaches for network inclusion can be distinguished [25, 61]:

- **2D visualization publishing**

  In this case, all computations are performed at a visualization server. Visualization parameters are set by the user within an HTML page or in an applet and sent to the server, which recomputes the visualization and sends the resulting image to the client for display. The implications of this solution are, that even the smallest change in the parameters, which might affect only a few stages of the visualization pipeline, like a change of the viewpoint, for example, requires computational resources at the server and leads to a retransmission of a whole visualization image over the network. Due to this fact, this approach is in most cases not suited for interactive data exploration, except for cases when special rendering hardware of the server should be exploited [65] and sufficiently fast network connections are available.

- **3D visualization publishing**

  As for the previous approach, the largest part of the visualization is performed on a server [31, 56, 61]. Instead of sending images to the client, an intermediate representation of the visualization is created and transmitted

(usually a polygonal representation of objects within the visualized data). As rendering of the model is performed at the client, a more efficient and interactive response to changes of viewing parameters can be achieved. In case of volume visualization this approach can be applied to visualize iso-surfaces generated using the marching cubes algorithm or some similar approach. Rendering geometry at the client bears two main problems: First, the memory and computational resources available at the client pose a limit to the complexity of the visualization and strongly influence the ability to work interactively. Additionally, interactions which lead to changes at stages deeper within the visualization pipeline, which are executed at the server, impose delays and require sufficient computational resources at the server.

- **Visualization software publishing**

  The entire software needed to create and display the visualization is transmitted to the client in advance and runs, e.g., as a Java applet within a web browser. Publishing visualization software greatly reduces the demands put on the server and has the potential for more interactive data exploration by tightening the loop between the user and the visualization system [37, 62, 63, 64]. This approach also allows to create easy-to-use visualization software which is able to reach large user communities and run on various platforms without the need for local installation. The main drawbacks of this solution with respect to volume visualization are the huge amounts of "raw" volume data which have to be transmitted and processed at the client.

The 2D and 3D visualization publishing schemes are also called thin client solutions, while visualization software publishing is also known as a fat client solution, reflecting the demands put on the user's workstation [25].

One of the problems common to all three approaches above is the difficulty to provide the user the possibility to interactively explore the data without requiring high-performance clients and networks. As interactivity is a main factor to the efficiency of data exploration [16], ways have to be found to facilitate it in Internet-based visualization tools.

As volume visualization is considered to be a memory and computationally intensive task, fat client approaches have not been considered for the first approaches to volume visualization over large scale networks [1]. An additional problem is the heterogeneous nature of clients, thus making a deployment of portable client software difficult. After the appearance of Java and it's integration

into Web browsers, it was soon shown that the new technology could be useful to provide fat client solutions even for volume visualization [37].

If the visualization is restricted to (iso-)surfaces – which can be represented using a polygonal model, the surface extraction can be performed at the server. Rendering, and thus also the response to changes in viewing parameters are performed at the client. Only after changes of parameters which influence the geometry of the surface, the new polygonal model has to be retransmitted from the server. Again, the major problem of this approach is the size of the polygonal model which has to be transmitted and rendered – typically several hundreds of thousands of triangles. To overcome this problem, Engel and others [15] place the data set on a server and use progressive transmission and progressive refinement to allow interactive surface extraction and viewing. They also presented an approach for providing direct volume rendering (DVR) at low-end clients [14]. First a small, subsampled version of the data set is transmitted to the client. During interactions which influence the rendered image, the local copy of the data is rendered using texture-mapping capabilities of consumer 3D hardware. After finishing the interaction, a high-quality rendering of the full-resolution data set is computed on a server and transmitted to the client. Although these approaches work well for a limited number of users who share the same server, they can not be applied if an interactive visualization is published to a large group of viewers, for example, over the Internet.

An approach which is more suited for "public" distribution of visualization results has been presented by Höhne and others [54]. A multi-dimensional array of images is rendered and stored using an extended Quicktime VR format. The viewer can browse through different views of the data, imitating an interactive rotation, dissection, or segmentation, for example. Additional object label data allows selection of objects and can be used for retrieval of additional information on the selected object. While this approach provides high-quality images on low-end hardware, the user interaction is restricted by the "hidden" browsing mechanism (in between pre-computed views). Furthermore, the size of even small-scale movies already becomes a limiting factor for viewing over low-bandwidth networks.

The approaches for volume rendering and transmission which are presented in this work can be used to implement a scenario which is located in between the two methods discussed above. The amount of data which actually has to be transmitted to the client for visualization is very low (about the size of several images), especially in comparison to the Quicktime VR approach. Using the presented rendering algorithms, the viewer is not restricted to pre-computed views and has full control over visualization parameters. The only restriction for rendering is that just those parts of the volume which have been classified as relevant and

pre-selected for presentation and transmission can be rendered. When used in a distributed client-server scenario, the software-only rendering approach provides much more flexibility in terms of rendering parameters than volume previewing using texture mapping hardware, still at comparable or even lower costs in terms of bandwidth requirements.

Chapter 3

Basic Concepts

> . . . in which it is shown that Tiggers don't climb trees
> *A. A. Milne (1882–1956), "The House at Pooh Corner"*

The approach to volume rendering which is presented in this work aims on providing interactive frame rates even on low-end hardware. To achieve this goal, a set of optimizations and efficient techniques are combined for data preprocessing and rendering. The algorithms are based on an interpretation of the volume as a set of voxels defined on a Cartesian grid. Key features of the concept are

- **preprocessing**: during a preprocessing step, voxels which potentially contribute to a visualization result are identified. All others are excluded from further processing.
- **efficient data representation**: possibly contributing voxels are extracted from the volume and stored in a derived data structure, which basically is a list of individual voxels. The extracted voxels can be ordered in a way to optimize the data layout for rendering with respect to specific compositing modes and visualization-parameter settings.
- **fast rendering**: a fast shear/warp-based rendering is used to project the extracted voxels.

Data sets as dealt with in the context of this work are considered to be composed of objects, i.e., semantically distinct sub-sets of the data. Therefore, each voxel of the volume is considered to belong to an object, i.e., a spatial structure within the data. The representation of extracted voxels allows a very flexible handling and assignment of visualization and rendering parameters, on a per-object basis. This

feature can be exploited to subdivide the volume in a way which corresponds to the internal structure of the contained objects. Each object can be rendered using the color, opacity, shading, and compositing method which is best-suited to obtain the desired visualization goal. The flexibility of visualization-parameter tuning and especially the ability to mix different shading models and compositing modes within a single visualization is not achievable with many other volume rendering approaches. Especially in combination with the interactivity of rendering, this poses a unique feature of our approach.

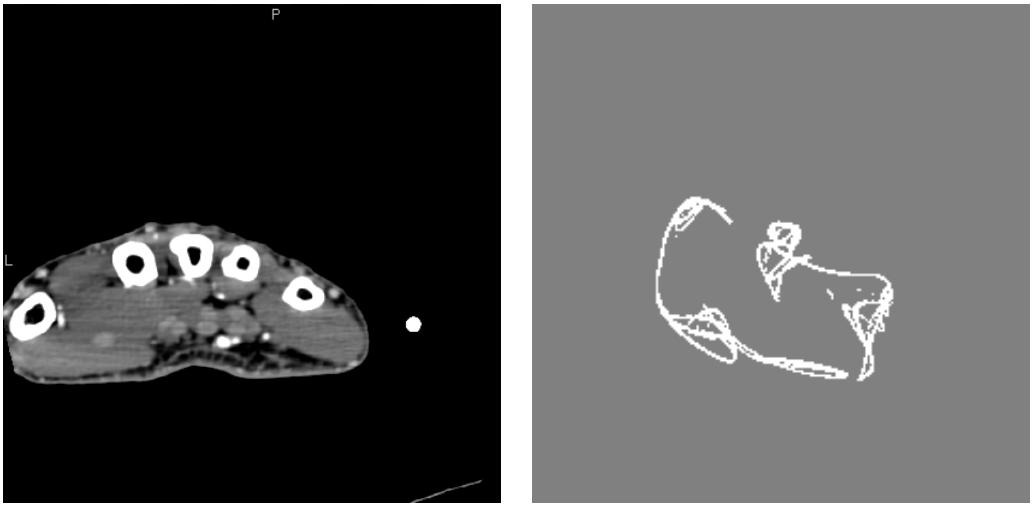
3.1 Preprocessing

Given a set of visualization parameters such as compositing technique in use, etc., the goal of the preprocessing step is to classify voxels of the volume into voxels which possibly contribute to an image and voxels which do not contribute to an image. The classification criteria depend on the chosen opacity transfer function, the desired compositing method, and the degree of freedom which should be provided for further manipulation of the transfer function. Generally speaking, the more voxels are classified as irrelevant, the fewer data has to be processed during projection, and the faster the rendering gets.

If an iso-surface should be rendered (characterized by a sharp transition between transparent and opaque voxels at the iso-value), only voxels close to the surface of the iso-value transition are relevant. Voxels with lower values are entirely transparent, voxels with higher values which are located inside the surface do not contribute, as they are occluded by opaque voxels at the surface. Of course, this relevance condition is bound to a specific iso-value – specifying a new iso-value to view another surface makes reclassification of the voxels necessary.

A higher flexibility with respect to transfer function tuning is obtained by extracting all voxels of an object. Volumetric data sets often contain objects of interest which are surrounded by irrelevant data, for example, body parts surrounded by air in medical data sets (figure 3.1a), or attractors surrounded by “empty” regions of phase space in the case of dynamical system data (figure 3.1b).

The viewing direction can also be included into the classification function. By subdividing the range of all possible viewing directions into several clusters, and performing classification for each cluster separately, several sets of relevant voxels are obtained. Each set is usually significantly smaller than the set of relevant voxels without considering viewing directions.



a) CT scan of a human hand b) section through chaotic attractor

Figure 3.1: Sections through volumetric data sets. A significant amount of voxels does not belong to any object of interest.

As the criteria are highly dependent on the actual target visualization scenario, they will be presented in more detail together with the description of the rendering methods in chapter 4. The time required for classification is as variable as are the criteria. Simple conditions, like the identification of voxels which belong to a surface, require typically less than a second for an entire data set. Complex conditions, like applied for maximum intensity projection, may take several minutes of preprocessing.

3.2 Data Representation

After the preprocessing step, the voxels which are classified as relevant make up a more or less sparse volume, which usually consists of intermixed clusters of relevant and irrelevant voxels of various size each. As irrelevant voxels are treated as "empty" or transparent during rendering, one of the already established schemes for space leaping could be used to accelerate rendering. However, the efficiency of those approaches is based on the assumption of a large granularity of empty and "full" regions, and a low degree of intermixing between them. This assumption does not hold for some of the relevance criteria. Additionally, common space leaping schemes still require some extra effort for actually avoiding empty space during rendering.

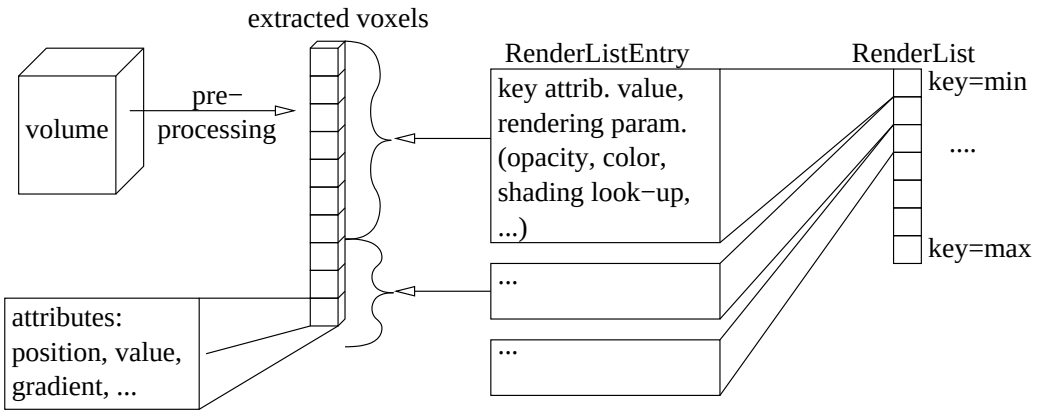


Figure 3.2: Data structure for efficient volume rendering. During preprocessing, potentially relevant voxels are extracted from the volume and stored into a list (an array). For each voxel, the position, and a set of attributes are stored. The voxels are ordered by one of the attributes, called the "key" attribute. This might be, for example, data value, or one of the coordinates. Groups of voxels with an identical key value are joint into so-called `RenderListEntry`s. All `RenderListEntry`s are sorted by the key value to form a so-called `RenderList`.

Instead of relying on established schemes, a novel approach is proposed for space leaping. All voxels which have been classified as relevant are extracted from the volume and stored in a secondary data structure. The structure is simply a list, or array, containing attribute information for the extracted voxels. Each entry corresponds to one extracted voxel, and holds the voxel's position, data value, gradient information and/or other attributes. If different objects are distinguished within the volume, separate lists are created for the voxels of each object.

For rendering, only information contained within the list is used. Some compositing methods require a specific ordering of the voxels. For DVR, for example, a consistent front-to-back or back-to-front order is required. For this reason, and to allow further optimizations tailored to specific rendering modi, the extracted voxels are ordered in a render-mode specific way (See chapter 4). For example, for maximum intensity projection, the voxels should be ordered by decreasing data value. This allows to skip with just a single test all voxels which are mapped to black (due to a change of the transfer function) and thus do not contribute to an image.

As the voxels are sorted by one of the attributes (for example, data value, or one of the coordinates), voxels within the list can be grouped into blocks (so-called `RenderListEntry`s), with the same value of a "key" attribute (figure 3.2). To save memory, the key attribute has to be stored just once for all voxels of the

group. All `RenderListEntry`s, sorted by the value of the key attribute, form a so-called `RenderList`.

In contrast to the original representation of a volume, the extracted voxel data structure requires to explicitely store position information for each voxel. As usually a significant portion of the volume's voxels is classified as irrelevant, the overall storage for the extracted voxels is less or comparable to the storage requirements of the original volume representation.

The possibility to reduce the volume to a small set of relevant voxels makes this representation well-suited for efficient network transmission. An efficient compression scheme for this purpose is presented in chapter 5.

3.3 Fast Rendering

By storing just relevant voxels in the `RenderList` structure, neighborhood information, which is implicitly present in a conventional volume representation, is not available. It is not trivial to obtain spatial neighbors of a voxel from the list to perform interpolation. This means, that voxels have to be treated individually, performing splatting (or something similar) for projection.

The fastest software-based rendering method is shear/warp projection. Voxels from the `RenderList` can be efficiently projected onto the base plane using look-up tables for determining projected positions. To maximize performance, nearest-neighbor interpolation is used during projection of voxels. Each voxel is projected onto the center of exactly one pixel of the base plane. The voxel affects only the value of the pixel it has been projected to. The calculation of the contribution depends on the compositing mode used. If optical properties of voxels are stored at the corresponding `RenderListEntry`s (figure 3.2), it is easily possible to assign different parameters to each group of voxels. In practice, all `RenderListEntry`s which store voxels of the same object will have identical parameters, like color and opacity transfer functions.

Projection is performed, by merging the `RenderList`s of all distinct objects, and sequentially processing all `RenderListEntry`s of the joint `RenderList` which represents the volume. Within each `RenderListEntry` all voxels are also projected sequentially. The strictly sequential order of voxel projection which does not depend on the viewing direction leads to an excellent cache-efficiency of this approach. In contrast, traversal of the usual, spatially ordered volumes, suffers from severe performance degradation due to frequent cache misses, when viewing the volume from certain directions. Although there are methods to reduce this

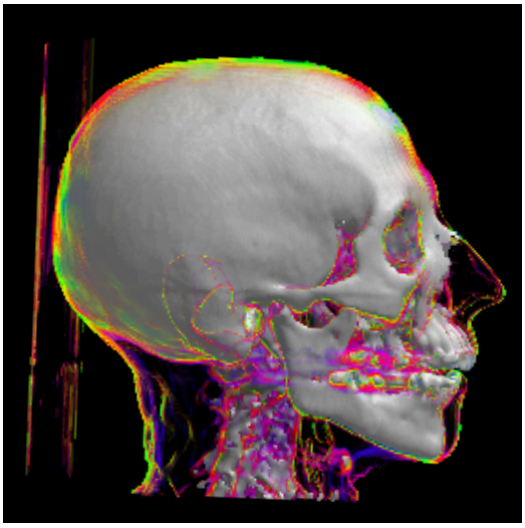


Figure 3.3: Mixing different lighting models: Phong shading for the bone, non-photorealistic contour enhancement for the skin

effect [51] by reordering the volume into bricks, they do not reach the effectivity of the presented, purely sequential approach.

If rendering parameters are changed, some of the extracted voxels may become irrelevant for the visualization. An effective way to skip them during rendering without much effort, is to reorder voxels belonging to each `RenderListEntry` in a way, that irrelevant voxels are moved to the end of the group. This allows to render the block of relevant voxels at the beginning of the group, and to skip the remaining, irrelevant ones efficiently. Clipping or removal of parts of the volume can be done in a similar way. If voxels which are clipped away are moved to the end of the list of each `RenderListEntry`s voxels, they can be skipped in the same way as irrelevant voxels. for details, please refer to chapter 6.

For fast evaluation of lighting models, a look-up table based approach can be used [19]. For this purpose, the gradient vector is quantized to 12–16 bit, and stored as an attribute with the voxels. The compact representation of the gradient vector is used as an index into a look-up table which stores precomputed shading information. The content of the look-up table has to be recomputed whenever a factor which influences lighting is modified, for example, a light source is moved, or the volume is rotated. Due to the small size of the table (4096–65536 entries, depending on the gradient quantization), various shading models can be applied on a per-object basis at interactive frame rates. Phong shading [49] and non-photorealistic shading models [10, 11] can be easily mixed within a single image (figure 3.3).

To ensure constant quality of shear/warp-based rendering, voxels should be isotropic, i.e., equally sized in all dimensions. If the original volume does not fulfill this condition, as, for example, the sampling distance in z direction is larger than in x and y – resampling can be performed during voxel extraction to obtain isotropic voxels in the `RenderList`. The resampling can be performed on-the-fly, and does not require the creation of a whole resampled, and thus possibly much larger copy of the original volume.

3.4 Two-Level Volume Rendering

Good visualization strongly depends on what data is to be visualized, what structure this data consists of, as well as on the visualization goals of the user. Depending on these prerequisites several useful approaches exist, and individual decisions (what rendering method to choose) have to be made for specific applications. Different parts of a volume (objects) might require different rendering methods to best depict their structure. If segmentation information is available, the presented approach can be used to provide this functionality at interactive frame rates.

The basic idea of two-level volume rendering [22] is to investigate a viewing ray into the data set for every pixel, and detect what objects are intersected. For every object intersected, a meaningful and representative contribution is computed using an object-specific compositing method (for example, MIP or DVR). These object representatives are finally composed into a pixel value by combining them using a global compositing method (usually DVR compositing).

The principles of two-level volume rendering can be easily explained using a ray casting based approach: a 3D segmentation mask specifies which regions of the data set belong to which objects. The subdivision of the data set into objects also segments viewing rays into sets of distinct segments (figure 3.4).

During ray traversal, two simultaneous tracks of rendering are processed. For every segment of the ray, local rendering is performed using the object's compositing strategy, to compute an object representative associated to the segment (rendering at object level). On the scene level a global track of rendering is computed which combines the object representatives to final image values. Whenever the ray leaves an object and enters a new one, the local value of the old object is merged into the global rendering track using the global compositing method.

For an example see figure 3.5. In this case, DVR rendering gives good results for ray segments within vessels and bones. MIP rendering works best for ray segments in soft tissue regions. This is mainly due to the fact that MIP generates rather equal transparency regardless of the object thickness.

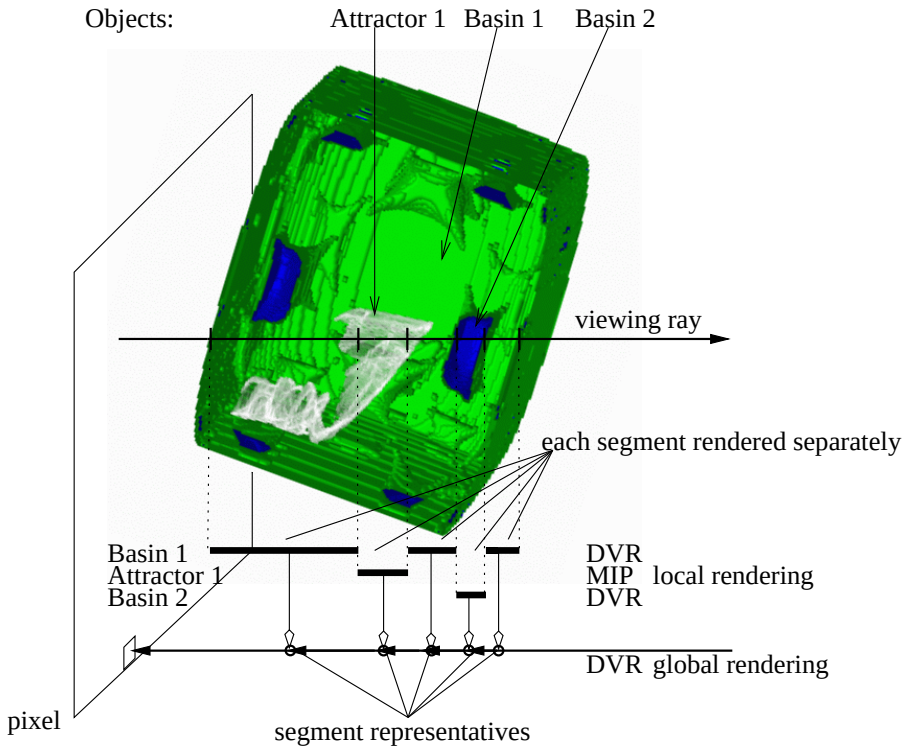


Figure 3.4: object segmentation implicitly yields viewing rays to be partitioned into segments (one per object intersection).

Usually, using DVR-compositing on the global level seems to be appropriate. The only exception, where MIP seems to be more useful instead, is if all objects in the data set are rendered by the use of MIP themselves, also. In contrast to standard MIP, this “MIP of MIP” approach allows to easily distinguish between different objects within the scene, as different transfer functions and thus colors can be assigned to different objects.

For implementing the two-level rendering approach based on the `RenderList` structure, two sets of buffers are used for the base plane image. An object buffer is used for performing rendering within an object, while a global buffer is used to perform inter-object rendering. In addition to intermediate pixel values, each pixel of the object buffer additionally stores a unique ID for the currently front-most object. If a voxel is projected onto the intermediate image, it’s ID is compared with the stored ID in the object buffer. If both IDs match, the value in the object buffer is updated using an operation which corresponds to the local rendering mode of the object (maximum selection or blending of the voxel value with the buffer content). If the ID of the voxel differs from the ID of the pixel in the buffer, the viewing ray though this pixel must have entered a new object.

Figure 3.5: joining MIP and DVR – a simple example (bones and vessels: DVR; skin: MIP).

The content of the object buffer pixel has to be combined with the corresponding global buffer pixel using an operation which depends on the global rendering strategy (MIP or DVR). Afterwards the object buffer pixel is initialized according to the voxel of the new object and the new local rendering mode.

After all voxels have been projected, the contribution of the front-most segment at each pixel has to be included by performing an additional scan of the buffers and merging the segment values left in the local buffer into the global buffer. See `http://bandviz.cg.tuwien.ac.at/basinviz/two-level/` for sample images and animations produced using this technique.

Chapter 4

Interactive Rendering

This chapter presents the application of the ideas presented in chapter 3 to obtain interactive MIP, DVR, and surface display from volumetric data. As the strategies for the detection of relevant voxels for various rendering methods are quite different, each technique will be discussed in detail within an own section.

4.1 Real-Time Maximum Intensity Projection

The ability to depict blood vessels is of enormous importance for many medical imaging applications. CT and MR scanners are used to obtain volumetric data sets, which then allow the extraction of vascular structures. Especially data sets originating from MR, which are most frequently used for this purpose, exhibit some properties which make the application of standard volume visualization techniques like ray casting [26] or iso-surface extraction [33] difficult. MR data sets usually contain a significant amount of noise. Inhomogeneities in the sampled data make it difficult to extract surfaces of objects by specifying a single iso-value.

MIP exploits the fact, that within angiography data sets the data values of vascular structures are higher than the values of the surrounding tissue. By depicting the maximum data value seen through each pixel, the structure of the vessels contained in the data is captured. A straight-forward method for calculating MIP is to perform ray casting and search for the maximum sample value along each ray instead of the usual compositing process done in volume rendering. In contrast to direct volume rendering, no early ray termination is possible and the whole volume has to be processed. Depending on the quality requirements of the resulting image, different strategies for finding the maximum value along a ray can be used.

- **Analytical solution:** for each data cell which is intersected by the ray the maximum value encountered by the ray is calculated analytically. Trilinear blending of the cell vertices is usually assumed for data values within a cell, thus the maximum of a third degree polynomial has to be calculated. This is the most accurate but also computationally most expensive method [51].

- **Sampling and interpolation:** as usually done for ray casting, data values are sampled along a ray using trilinear interpolation. The cost of this approach depends on how many interpolations can be avoided that will not affect the ray maximum [51, 67].

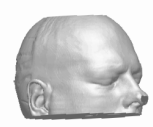
- **Nearest neighbor interpolation:** values of data points closest to the ray are used as maximum estimations. In combination with discrete ray traversal this is the fastest method. As no interpolation is performed, the voxel structure is visible in the resulting image as aliasing [5].

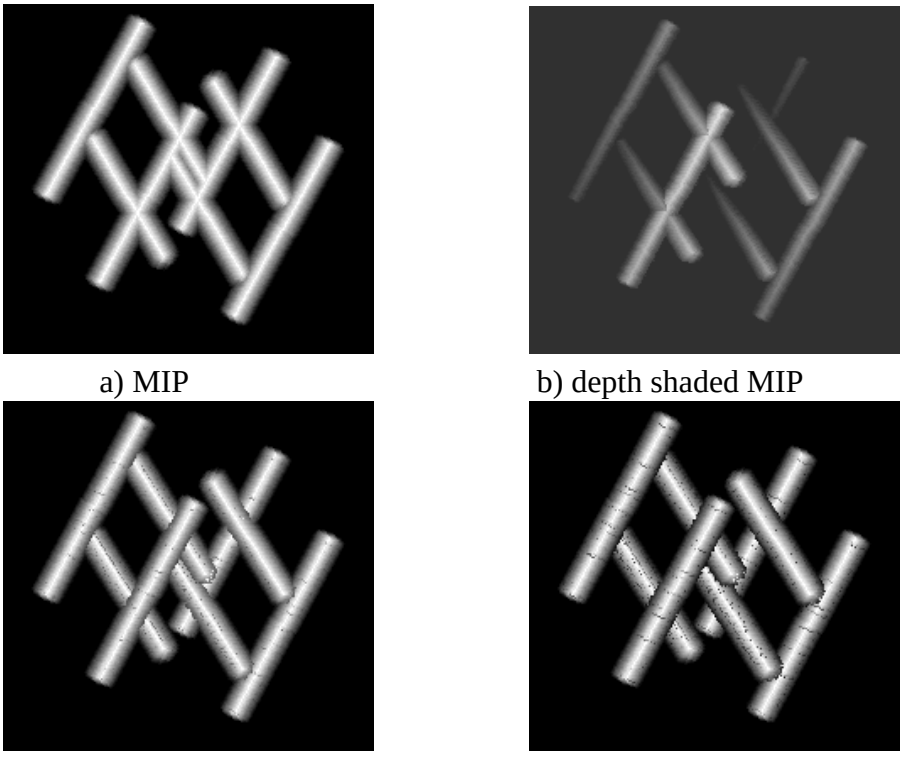
Recent algorithms for MIP employ a set of approaches for speeding up the rendering:

- **Ray traversal and interpolation optimization:** Sakas and others [51] interpolate only if the maximum value of the examined cell is larger than the ray-maximum calculated so far. For additional speed-up they use integer arithmetic for ray traversal and a cache-coherent volume storage scheme. Zuiderveld and others [67] apply a similar technique to avoid trilinear interpolations. In addition, cells containing only background noise are not interpolated. For further speed-up a low-resolution image containing lower-bound estimations for the maximum of clusters of rays is generated before the projection. Cells with maximum values below this bound can be skipped when the final image is generated. Finally a distance-volume is used to skip empty spaces efficiently.

- **Using graphics hardware:** Heidrich and others [24] use conventional polygon rendering hardware to simulate MIP. Several iso-surfaces for different threshold values are extracted from the data set. Before rendering, the geometry is transformed in a way, that the depth of a polygon corresponds to the data value of its iso-surface. MIP is approximated by displaying the z-buffer as a range of gray values. Recently the VolumePro board [48] became available as a purely hardware-based solution for MIP. The board is able to produce medium-quality MIP using shear/warp-based projection at interactive frame rates.

- **Splatting and shear/warp:** Several approaches [5, 9] exploit the advantages of shear/warp rendering to speed up MIP. Cai and others [5] use an





a) MIP b) depth shaded MIP

c) LMIP – high threshold d) LMIP – low threshold

Figure 4.1: a: MIP of a test data set containing soft bounded cylinders; b: although depth shading provides some depth impression, the cylinders seem to intersect; c, d: While LMIP provides more information on the spatial structure, the results are sensitive to the threshold.

> intermediate "worksheet" for compositing interpolated intensity contributions for projection of a single slice of the volume. The worksheet is then combined with the shear image to obtain the maxima. Several splatting modes with different speed/quality-tradeoffs are available, run-length encoding and sorting of the encoded segments by value are used to achieve further speed-up.

As a MIP image contains no shading information, depth and occlusion information is lost (see figure 4.1a). Structures with higher data values lying behind a lower valued object appear to be in front of it. The most common way to ease the interpretation of such images is to animate the viewpoint while viewing (interactive frame-rates are essential here). Another approach is to modulate the data values by their depth to achieve a kind of depth shading [24] (see figure 4.1b). As the data values are modified before finding the maximum, MIP and depth-shaded MIP (DMIP) of the same data may display different objects.

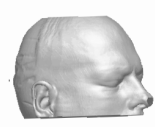
Depth shading provides some hints on the spatial relation of objects. However it's performance is rather poor especially for tightly grouped structures. In such cases Closest Vessel Projection [51] or LMIP [52] can be used. Similar to MIP, for LMIP the volume is traversed along a ray, but the first local maximum which is above a user-defined threshold is depicted. If no value above the threshold is found along a ray, the global maximum along the ray is used for the pixel. With a high threshold value this method produces the same result as MIP, with a carefully selected one, less intense structures in front of more intense background are depicted, producing an effect similar to shading (see figures 4.1c, d). As this method is very sensitive to the setting of the threshold, the ability to interactively tune this parameter is extremely important.

In the following a novel approach to generate MIP images really fast is presented [42, 43]. In Section 4.1.1 several algorithms for preprocessing the volume data to eliminate voxels, which will not contribute to a MIP, are discussed. Furthermore, a volume storage scheme which is based on `RenderLists` is presented which is optimized for skipping non-contributing voxels during rendering. In section 4.1.4 extensions of the algorithm for generating depth-shaded MIP and LMIP using the optimized data structure are discussed.

4.1.1 Voxel Elimination

Most approaches to optimize the performance of MIP rendering aim at excluding voxels from the traversal and rendering process, which contain less-important information like low-valued background noise. In fact, in addition to this low-importance data, there is usually a remarkable amount of regular-valued voxels which never contribute to a MIP image. A voxel V does never contribute to a MIP and can be discarded if all possible rays through the voxel hit another voxel W with $d(W) \geq d(V)$ either before or after passing through V, where $d(V)$ is the data value at voxel V. This fact can be exploited when original voxel-values are used for rendering using nearest neighbor interpolation, as it is done within the presented approach.

In the following, two algorithms for identifying non-contributing voxels are presented. The first approach performs classification based on the local neighborhood of a voxel, identifying voxels invisible from any viewing direction. The second algorithm groups possible viewing directions into several clusters and produces a set of potentially contributing voxels for each cluster. The view-point dependent elimination achieves much better elimination rates at the cost of storing several sets of voxels for rendering.



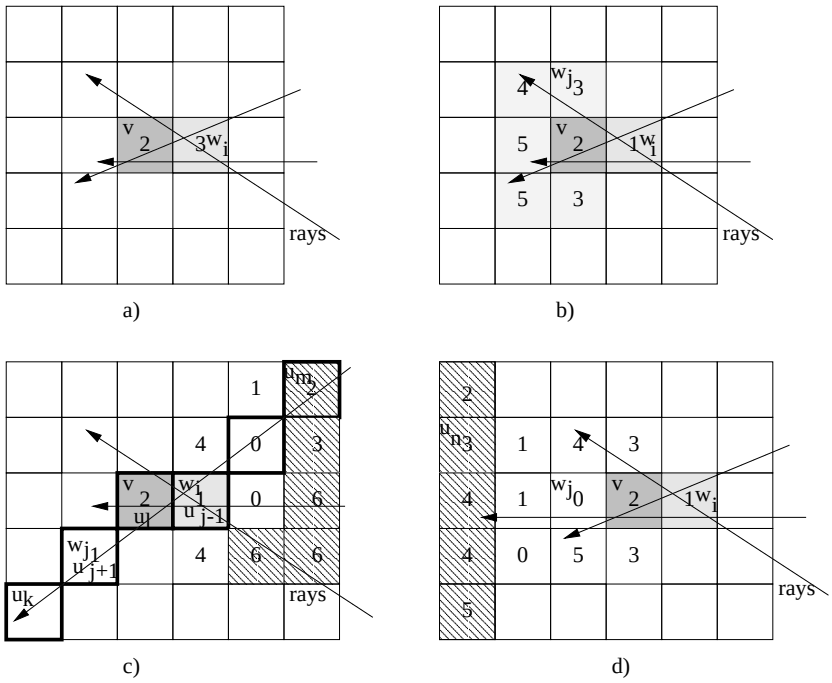
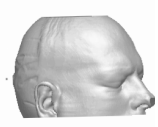


Figure 4.2: Detection of voxels V which do not contribute to any MIP. For simplicity the different cases are shown in 2D. a) V irrelevant for rays through W_i as $d(W_i) \geq d(V)$ b) V irrelevant for rays through W_i. c) V irrelevant for rays through W_i as ray maxima are determined by voxels U_m. d) V irrelevant for rays through W_i, as ray maxima are determined by U_n.

Neighborhood-based Elimination

A simple approach for the identification of "hidden" voxels is to examine direct neighbors W_i of a voxel V. If V does not influence the maximum of any ray passing through V and any of it's neighbors W_i, it can be removed. Obviously, this is the case if all rays through V pass through a neighbor W_i with $d(W_i) \geq d(V)$ either before of after passing through V.This condition can be tested by examining the values of all 26 direct neighbors W of V:

- If for all i: $d(W_i) \geq d(V)$, all rays through W_i and V will have at least the value $d(W_i)$ and are not influenced by V (see figure 4.2a).
- If for some i: $d(W_i) < d(V)$, but all rays through W_i and V pass through another neighbor W_j of V with $d(W_j) \geq d(V)$, the ray-values will at least be $d(W_j)$. Thus V has also no influence on rays through W_i (see figure 4.2b).



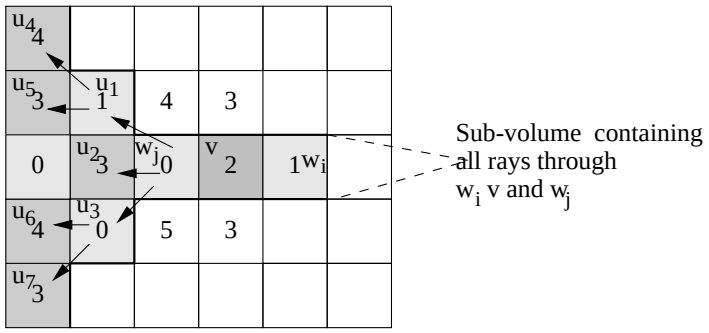


Figure 4.3: Tracking the influence of distant parts of the volume on rays through W_i, V and W_j. As all of them pass either through U_1, U_2 or U_3 and $d(U_2) > d(V)$, V does not influence rays through U_2, W_j, V and W_i. As $d(U_1) < d(V)$ and $d(U_3) < d(V)$, further tracking of rays passing through U_1 or U_3 is required.

Voxel elimination which is based on these two tests can be performed rapidly (see table 4.1) and just requires access to direct neighbors of each voxel. If the above tests fail for a group of rays through W_i, V, and W_j, with $d(W_i) < d(V)$ and $d(W_j) < d(V)$, the influence of a larger neighborhood of V on the value of these rays can be checked. Each ray passes through a sequence of voxels U_k with $U_l = V$, $U_{l-1} = W_i$ and $U_{l+1} = W_j$ as the volume is traversed. If each of those rays passes through some voxel U_m with $m < l - 1$ and $d(U_m) > d(V)$, V has no influence on the values of the rays through W_i and W_j (figure 4.2c). Similarly, if all rays through W_i and W_j pass through voxels U_n with $n > l + 1$ and $d(U_n) > d(V)$, V has no influence on the values of those rays (figure 4.2d).

The check of the influence of more distant volume areas can be realized by recursively propagating bundles of rays starting at direct neighbors of V (see table 4.1). If a voxel U is hit by a bundle of rays and $d(U) \geq d(V)$ the tracking of the bundle is terminated, as V has no influence on the rays. If $d(U) < d(V)$ the bundle is split and tracked in a subset of the neighbors of U (figure 4.3).

To improve efficiency and avoid unnecessary recursive checks, values of voxels scheduled for removal are replaced by the minimum of their obscuring values (which is always larger or equal to their original value). To compensate for noise in the data and to increase the number of rejected voxels a user specified tolerance value ε can be included in the comparison process, which artificially lowers the value of each checked voxel V by ε. For voxel reduction rates using different elimination efforts and tolerance values ε please refer to table 4.1. Although recursive elimination requires significantly more time, it may be the method of choice if data is prepared on a high-end machine for viewing with low-end hardware over a network.

data set	*size*	*Fast* ε *=0%*	*Full* ε *=0%*	*Fast* ε *=1%*	*Full* ε *=1%*	*direction* dependent
angio	$256^2 \times 64$	25% 5s	47% 270s	29% 5s	53% 270s	73% 72s
mr01	$256^2 \times 74$	33% 6s	46% 208s	45% 6s	55% 208s	72% 76s
mr03	$256^2 \times 124$	29% 10s	58% 562s	29% 10s	58% 562s	70% 120s

Table 4.1: Volume reduction: Fast optimization ("Fast") considers direct neighbors, full optimization ("Full") performs recursion up to a distance of 10 voxels. Direction dependent optimization produces 24 view-dependent sets of voxels. The results show the percentage of voxels removed and the time required for preprocessing on a PII/450 PC.

Shadow Sweep Elimination

Although the above approach is able to remove approximately half of a data set, many voxels have to remain in the volume even if they only contribute to a MIP for a narrow range of viewing directions. If enough memory is available for redundant storage of the data, more efficient preprocessing can be performed by subdividing all possible viewing directions into disjoint view-sets in a way which minimizes dependencies among voxels (figure 4.4). The highest efficiency of removal in 3D can be achieved if 24 such view-sets are used. For each view-set a set of potentially visible voxels is computed using a two-pass procedure:

- **Shadowed voxel sweep:** Mark all voxels V in shadow of (local) maxima as non-contributing. This is the case when $d(V)$ is lower than the smallest ray maximum of rays through V collected before reaching V. This can be easily performed by sweeping through the volume and processing voxels in the order in which they are hit by an advancing front of rays. During this process a front is propagated which contains the smallest maximum values for rays entering each voxel directly ahead of the front. Considering the example in figure 4.5a, all viewing-rays of the given view-set through voxel V have first to pass either through V_1 or V_2. The lowest possible values of rays at V_1 and V_2 respectively are $min_1 = d(V_1)$ and $min_2 = d(V_2)$, as both V_1 and V_2 are on the border of the volume. Thus the lowest possible value of a ray arriving at V is $arr_V = min(min_1, min_2)$. As $d(V) > arr_V$, V may influence some of the arriving rays, and can not be removed. The value of the rays after passing V equals $max(arr_V, d(V))$ and influences the values of rays arriving at the successive voxels V_3 and V_4.

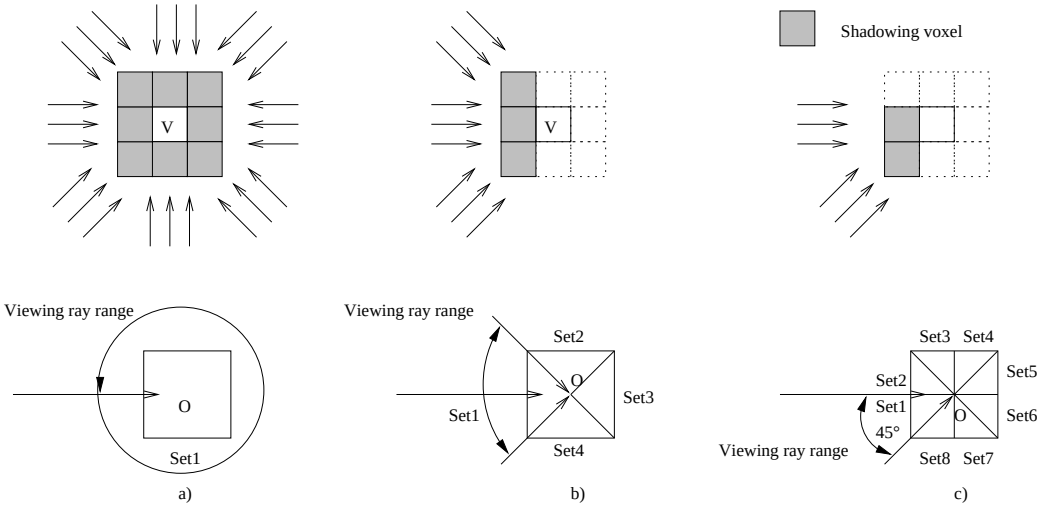


Figure 4.4: Removal of hidden voxels in 2D: a) viewpoint-independent preprocessing: relevance of voxel V depends on rays passing through eight neighbors. b) 4 view-sets: relevance of V depends on three neighbors. c) 8 view-sets: dependency minimized (two neighbors).

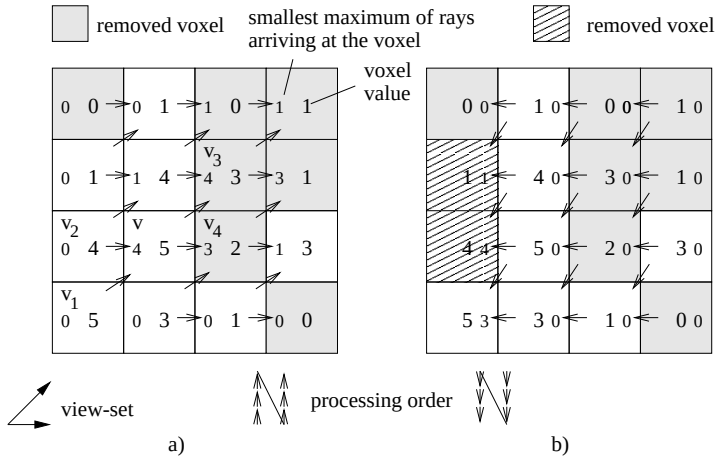


Figure 4.5: View-point dependent voxel removal (2D). a) Shadowed voxel sweep. b) Leading voxel sweep.

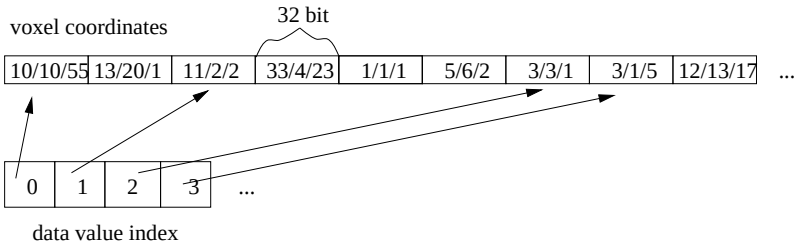


Figure 4.6: Optimized voxel storage for MIP: relevant voxels are sorted according to data value. Their position in space is stored in an array (as the only individual attribute). The voxels are grouped into `RenderListEntry`s which hold voxels with identical data values. The data value is stored only once at each `RenderListEntry`

- **Leading voxel sweep:** During the second pass all low-valued voxels in front of voxels containing unremoved (local) maxima are removed. The removal is performed in the same way as step 1 with the difference that the scan is performed opposite to the direction of the viewing rays (figure 4.5b). Note, that voxels removed during the shadowing sweep have no influence on ray maxima during the second sweep.

Direction dependent preprocessing allows to remove about three quarters of all voxels for each view-set (see table 4.1). During rendering, voxels of the view-set which contains the current viewing direction are selected and rendered.

4.1.2 Voxel Storage

Voxels which have been classified as relevant during the preprocessing step are extracted from the volume, by storing just their positions into an array. Within the array, voxels are sorted according to data value. After sorting, voxels with the same data value occupy subsequent positions in the array, and can be easily grouped into `RenderListEntry`s. The data value has to be stored just once for every group (see figure 4.6) at the `RenderListEntry`. The required sorting of all voxels according to their data value can be performed in linear time, as the limited range of possible data values allows to use histogram-based sorting. The coordinates of each voxel within the volume can be packed into a 32 bit integer, allowing the encoding of volumes of up to $2048 \times 2048 \times 1024$ voxels. A straightforward conversion of a 16 bit/value volume to a 32 bit position representation would mean to double the memory cost. Omitting the voxels which have been marked by the preprocessing step as irrelevant leads to a factor of 0.8 to 1.5 in storage size compared to the original data for direction independent data. For

direction-dependent data, approximately one quarter of the original data has to be stored for all 24 view-sets. The overall memory cost for direction dependent preprocessing is thus 12 times higher than the cost of the original volume.

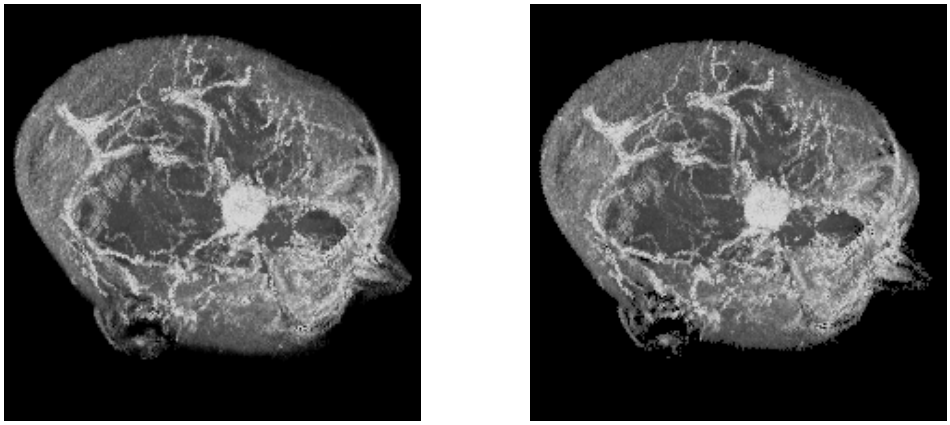
By sorting voxels by value several important advantages for MIP are gained:

- The voxels can be splatted in the order of ascending data values as the array is traversed. Therefore *comparing* the value of the actual voxel with the screen content is *not necessary* at all. The value of the actual voxel is always larger or equal to the content of the screen, thus it's projection can be written into the image. Omitting the comparison during projection results in about 10% better performance.

- A general advantage of the voxel list approach is that the array is traversed in the same way independent of the viewing direction. *Optimal cache coherency* is always achieved. The same code accessing data in a linear way is up to eight times faster than in the case of extremely misaligned access.

- As blood vessels are represented by high data values, lower data values usually contain less important information. If an interactive display of the full data set is not possible on the given hardware, lower intensity values can be simply skipped (see figure 4.7) to achieve interactivity during user interactions. The algorithm can be adjusted to display MIP at *an arbitrary frame rate*. The number of voxels which can be displayed at a specific frame rate can be easily derived automatically from a performance measurement of the number of voxels which can be rendered per second.

- Instead of directly mapping data values to a linear ramp of gray values for viewing ($d_{min} \Rightarrow g_{min}, d_{max} \Rightarrow g_{max}$), medical data sets are usually viewed using a "window" to improve contrast and to focus on certain details. The window is defined by a center value c and a width w which maps all data values below $c - w/2$ to black, all data values above $c + w/2$ to white, and the data in between to a gray ramp. As realistic window functions as used by doctors map significant portions of the data to black, the corresponding parts of the sorted voxel array can be skipped during rendering at almost zero cost. This results in an additional *speed-up of up to 500%* compared to rendering without windowing *under real-world conditions*.



(a) (b)

Figure 4.7: a) MIP of a head data set (MR), 2.6M voxels. b) Interactive preview of the same data at 10 fps on a P233MMX with approximately 25% of the volume data displayed. Only slight differences are noticeable

4.1.3 Projection

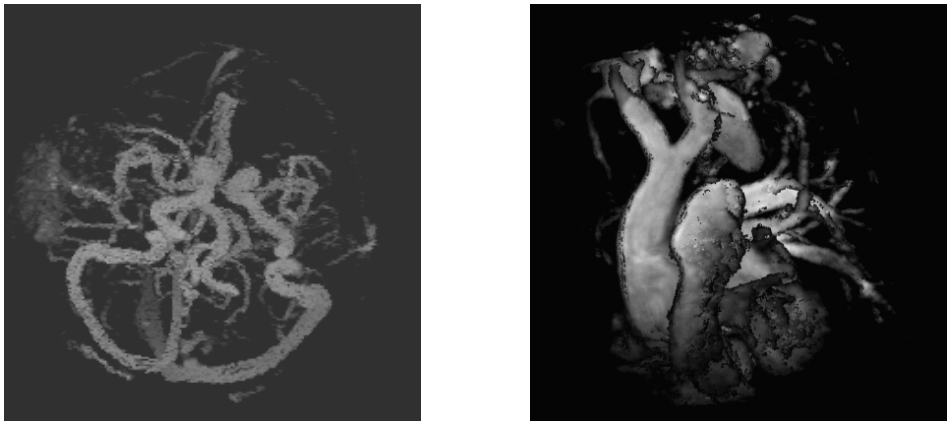
As no scaling is performed during (shear/warp) projection to the base plane and nearest-neighbor interpolation is used, voxels are projected onto integer coordinates within the base plane. Assuming that z is the main viewing axis, and thus the base plane is the xy-plane, the projected position (index within the image buffer) of a voxel (x, y, z) is

$$\mathbf{P} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \text{offset}[z] + y * width_{baseplane} + x$$

offset is the offset of the voxel with $x = 0, y = 0$ within a volume slice at depth z, $width_{baseplane}$ is the width of the base plane image in pixels. The resulting value can be directly used as an index to access the affected pixel of the base plane.

4.1.4 Extensions – LMIP and Depth-Shaded MIP

The algorithm for MIP of value-sorted voxels can be easily extended to generate depth-shaded images (figure 4.8a). Three depth-templates are generated to calculate the depth of each voxel's projection from it's coordinates by look-up. The intensity value of each voxel is modulated by it's depth and written into the base plane image only if the value of the pixel at this position is smaller than the modulated value. As the depicted maximum values do not directly correspond to data



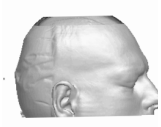
(a) (b)

Figure 4.8: a) MIP of an angiography with depth shading. b) LMIP of a stenosis of the aorta

values, applying this method to preprocessed volume data may produce results slightly differing from projecting non-preprocessed data.

The second extension of the algorithm is capable of generating LMIP images providing the possibility to interactively adjust the threshold parameter. For generating a MIP image, the order of examining samples along a ray is not relevant. Straight-forward LMIP requires the samples to be processed front to back along the viewing ray in order to find the first local maximum. Using templates for voxel-depth calculation and two z-buffers per pixel allows to extract the closest local maximum from samples arriving in arbitrary order. The `z[pix]` buffer stores the depth of the currently visible voxel, while `zb[pix]` stores a "back-clipping" distance for each pixel which is used to skip voxels belonging to maxima behind the currently closest one. As the voxels are processed in order of ascending data value, all voxels below the LMIP-threshold can be first projected using the simple and fast MIP algorithm without z-calculation. Among all voxels above the threshold which are processed later, closest local maxima have to be found. At the beginning, z[] is initialized to contain the maximum possible distance. With `z` and `v` containing the depth and value of a projected voxel, the closest local maximum along a ray is found by

```
if (z<z[pix])
  screen[pix]=v; zb[pix]=z[pix]; z[pix]=z;
else if (z<zb[pix])
  screen[pix]=v; z[pix]=z;
```

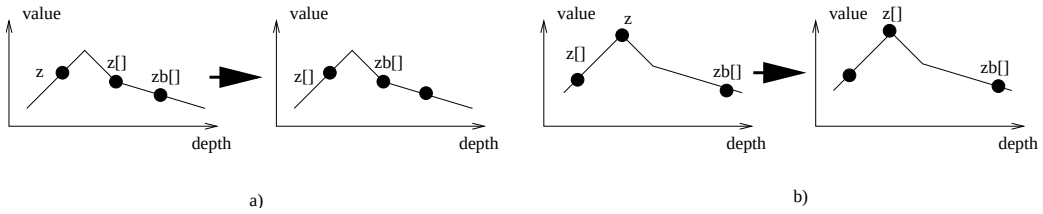


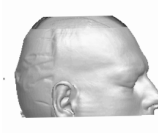
Figure 4.9: Local Maximum Intensity Projection for non-sequential samples a) new voxel in front of `z[]` b) new voxel between `z[]` and `zb[]`

The first condition detects voxels closer than the currently displayed voxel. As they have at least the same intensity as the currently displayed one, they are entered into the `screen` and `z[]` buffers, the back clipping plane is moved forward to the previous `z[]` position (figure 4.9a). The second condition takes care of new voxels which are placed between the currently displayed voxel and the back clipping distance and are at least of the same intensity as the current one. In this case the `screen` and `z[]` are set to the new voxel. Voxels behind the back clipping distance are ignored (figure 4.9b). As the value at `zb[]` is always older than the value at `z[]` it is also smaller due to the ascending processing order of the voxels. Thus, any voxel which arrives later during processing and is located behind `zb[]` can be ignored , as it does definitely not belong to the closest maximum.

4.1.5 MIP Results

The MIP algorithms discussed above have been implemented as a Java applet (*http://www.cg.tuwien.ac.at/research/vis/vismed/RT-MIP/*). The rendering times in table 4.2 have been measured on a PII/450 PC using Sun's Java virtual machine of JDK 1.1.6 for Windows with a just-in-time compiler. The first value gives the time for rendering data which was preprocessed independent of the viewpoint, the second value is the time for rendering direction-dependent preprocessed data. As timings for LMIP depend on the chosen threshold, measurements for the worst case are given. A C++ version of the MIP rendering algorithm has shown an approximately 30% better performance than the Java version.

Figure 4.10 is provided for a visual comparison of the discussed variations of MIP. It shows a 256x256x100 angiography data set of a hand rendered with MIP, DMIP, and LMIP using the presented voxel extraction approach.



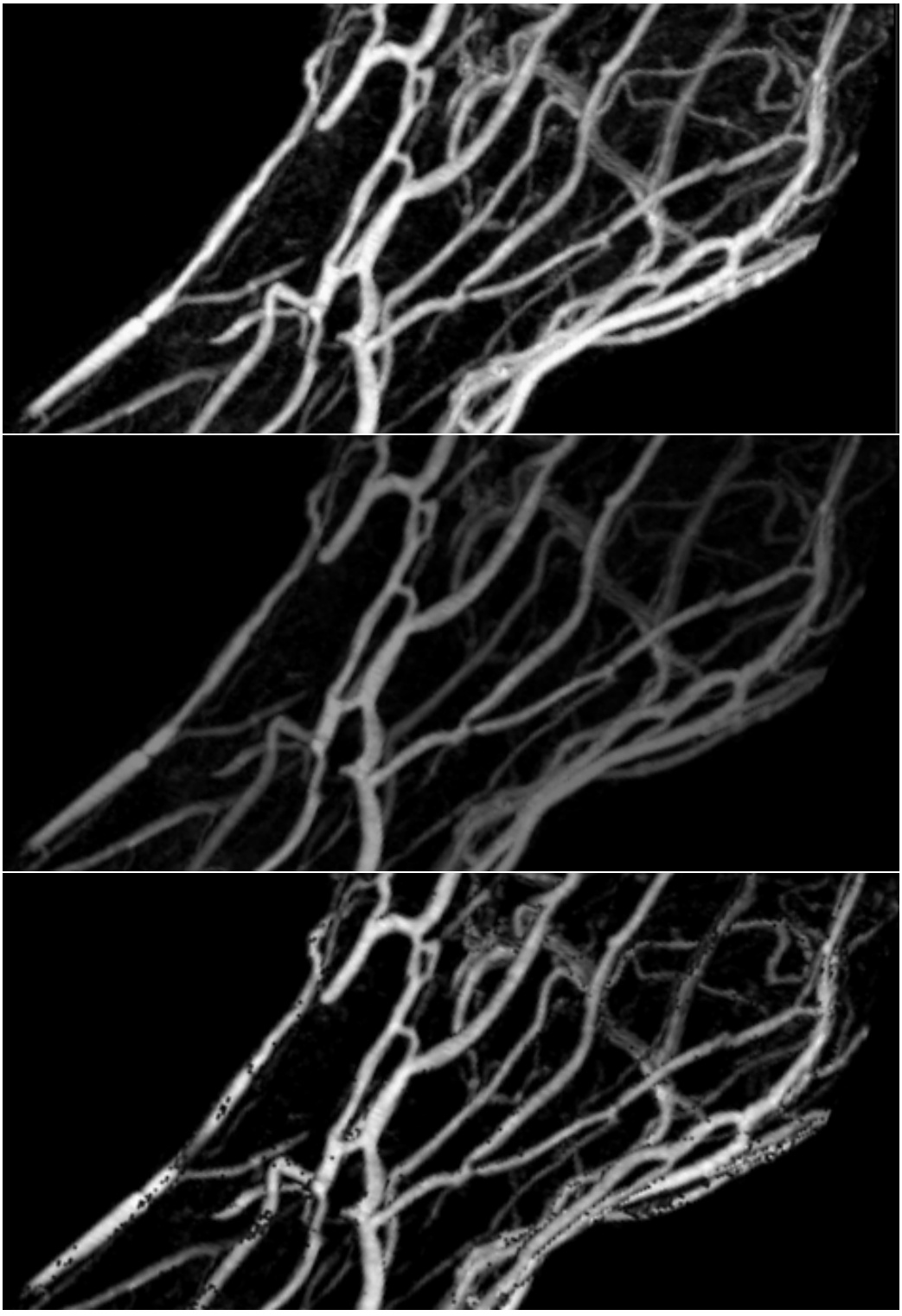
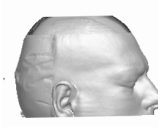


Figure 4.10: MIP variations of the vessels in a hand: MIP, Depth-shaded MIP, LMIP.



data set	*MIP*	*MIP windowed*	*depth-shaded*	*LMIP*
mr_angio $256^2 \times 64$	169ms	27ms	209ms	<305ms
	94ms	21ms	118ms	180ms
mr01 $256^2 \times 74$	163ms	42ms	224ms	<288ms
	106ms	32ms	146ms	198ms
mr03 $256^2 \times 124$	254ms	13ms	338ms	<455ms
	219ms	11ms	267ms	370ms

Table 4.2: Frame rendering times (with direction-independent/direction-dependent preprocessing.

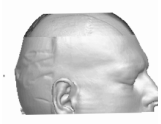
4.1.6 Discussion

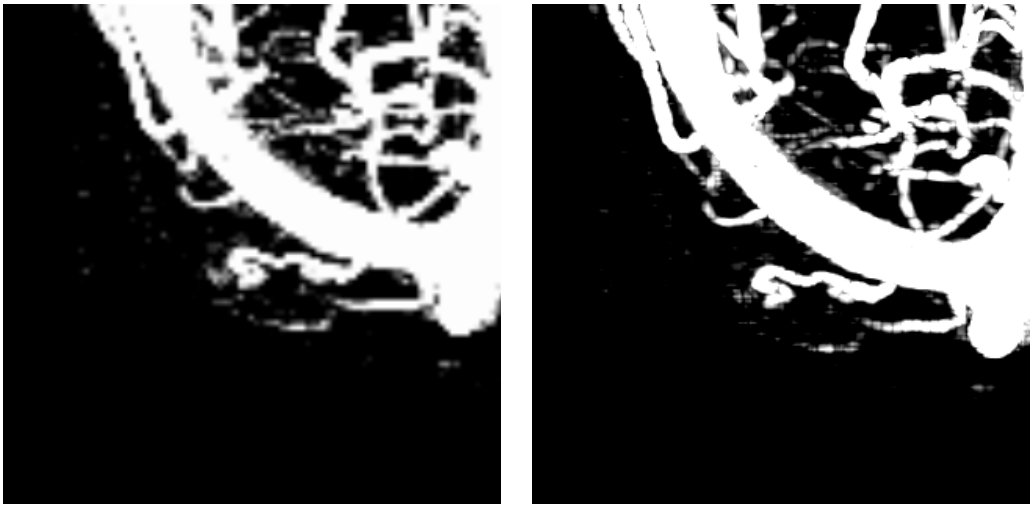
The presented new rendering algorithm for maximum-intensity projection (MIP) runs at real-time frame rates. In combination with the preprocessing step which removes parts of the volume which do not contribute to a MIP image and the sorting of the remaining voxels by value within the `RenderList` storage scheme, overhead for traversing empty regions is eliminated. Compared to other optimized MIP algorithms, up to an order of magnitude is gained in speed, compared to a brute-force ray casting approach even two orders of magnitude. A preview mode for interactions on low-end hardware is provided basically "for free" by skipping voxels with low data values. Two extensions of the algorithm allow to generate depth-shaded MIP and LMIP at comparable speed.

4.2 Interactive High-Quality MIP

For exact depiction of even small features using MIP, there is need for algorithms which produce high-quality MIP in real-time [41]. In contrast to interactive MIP techniques, which perform just zero-order (nearest-neighbor) interpolation, more accurate evaluation of ray maxima is required for the generation of high-quality MIP (trilinear interpolation, or higher order methods) . At the cost of significantly longer computation times, this allows to create much more detailed images and accurate animations (figure 4.11).

Two major limitations to the performance of current software-based high-quality MIP can be identified. First, although regions of the volume which do not contain meaningful data can be identified in advance and skipped during rendering by the use of distance volumes or similar structures [67], the overhead associated with evaluating the additional information and stepping over these cells significantly limits the possible speed-up of volume traversal. Secondly, despite



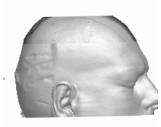


(a) (b)

Figure 4.11: MIP with a) shear/warp projection with nearest-neighbor interpolation b) ray casting and trilinear interpolation.

of space-leaping approaches, the number of interpolations which are actually performed is still far from the optimum. As the volume is traversed in a spatially ordered manner along viewing rays, local maxima are usually encountered and evaluated before the global ray maximum is reached. Moreover, lots of unnecessary evaluations are performed on the rising slopes of data values which precede a maximum.

Based on the principles from chapter 3, a new algorithm for the generation of high-quality MIP (parallel projection) is presented, which is approximately one to two orders of magnitude faster than other software-based approaches with comparable quality. In contrast to the previously described voxel-based MIP, the approach works on cells, which allows to achieve a high quality of the image by performing trilinear interpolation during projection. The projection is done using a ray casting approach. In Section 4.2.1 the preprocessing scheme is presented, which can (but not necessarily has to) be applied to identify and exclude non-contributing cells from the volume. To maximize the amount of cells which do not contribute to any MIP, 12 sets of cells are generated, corresponding to 12 clusters of viewing directions. Usually, more than two thirds of all cells can be eliminated from the data, no longer causing any overhead to identify them later and skip over them. This is achieved by resorting the cells according to their maximum value (Section 4.2.2). As the spatial order of processing cells is not relevant for maximum evaluation, cells containing high data values are evaluated first. This reduces the number of evaluations required for MIP significantly. To avoid even



more of the relatively expensive trilinear interpolations, a fast method for estimating the maximum value along a ray-cell intersection is used (section 4.2.3). Using these techniques, the number of trilinear interpolations required per image pixel is greatly reduced, achieving interactive high-quality MIP.

4.2.1 Preprocessing of Volume Data

The methods described in the following sections are based on an interpretation of the data set as a regular grid of cells, each one defined by eight data samples of the volume located at the cell's vertices. Within cells trilinear interpolation is assumed. For image generation continuous rays are traced through the pixels of the image, allowing arbitrary image sizes as well as oversampling.

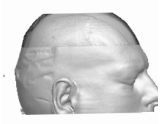
For trilinear interpolation within cells the cell maximum is always located at one of the vertices. Besides skipping pre-identified empty regions of the volume using distance volumes, another simple way to increase efficiency is to perform maximum evaluations only within cells with a cell maximum C_{max} greater than the current ray maximum. This technique avoids the evaluation of cells reached by rays after processing a (local) maximum. If, in addition, a good lower-bound estimation of the ray maximum can be obtained before rendering, also cells encountered before the maximum can be skipped, if their maximum is lower than the lower-bound estimation [67].

Although these methods reduce the number of required evaluations significantly, the lower-bound estimation has to be performed for each new viewing direction and much time is spent during rendering on identifying and then skipping cells and empty regions.

To increase time savings during rendering, cells which do not contribute to any MIP should be identified and removed during a preprocessing step.

Cell Removal

A cell C does not contribute to MIP from any viewing direction (and therefore should not to be considered for rendering), if all rays passing through it collect a higher value by passing through other cells D either before or after C is processed. The first approach is to investigate direct neighbors D_i of a cell C only. C does not contribute to any ray through it if $\forall i | (C_{max} \leq D_{imin})$. As we assume continuous rays to be traced through the volume, only face-connected neighbors of C have to be considered (A ray entering C through an edge or vertex at least "touches" some face-connected neighbors).



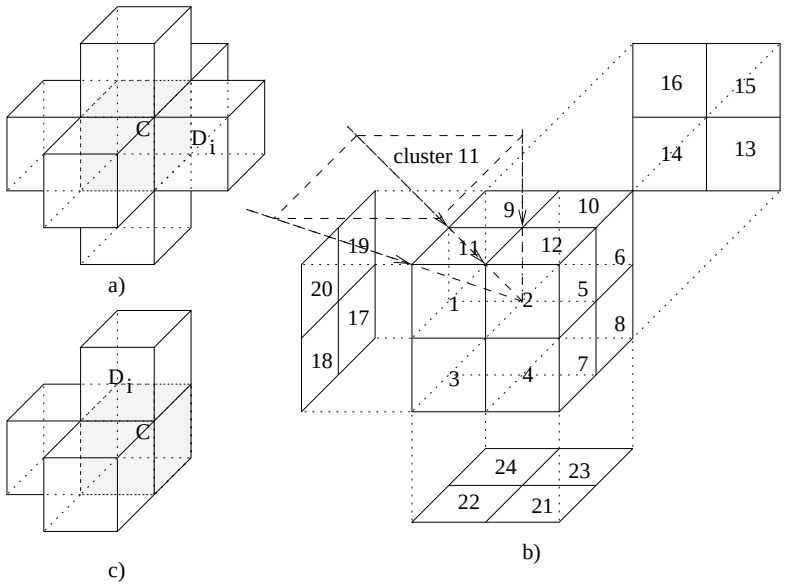
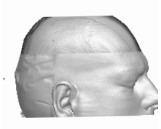


Figure 4.12: a) relevance of cell C depends on values of 6 face-connected neighbors D_i. b) decomposition of the viewing-domain into 24 (12) clusters of viewing directions. c) for any viewing direction within cluster 11 relevance of C depends on the values of at most 3 direct neighbors

Applying such a unified relevance detection for cell elimination, which considers the whole domain of viewing directions, is very ineffective and leads to low cell removal rates. Significantly better results can be achieved if several distinct clusters of similar viewing directions are distinguished. For rendering, the set of cells corresponding to the cluster which contains the current viewing-direction is selected and used for MIP. To minimize the number of neighbors which have to be checked for elimination, a decomposition of all viewing directions into 24 clusters is performed first (in the same way as for the voxel-based MIP). Each cluster of viewing directions corresponds to a quarter-face of the directional cube as depicted in figure 4.12b. Considering just viewing directions out of one cluster, a cell's relevance to MIP depends on just three neighbors (figure 4.12c) instead of six as in the case of viewpoint-independent preprocessing (figure 4.12a).

As the direction of ray traversal is not relevant for MIP and a MIP generated from a certain viewing direction produces exactly the same image as a MIP from the opposite direction, sets of possibly contributing cells have only to be calculated for 12 of the 24 clusters of viewing directions. Furthermore, as can be seen in figures 4.12b and c, three clusters of viewing directions around each corner of the directional cube result in the same set of direct neighbors on which a cell's relevance depends (for example clusters 1, 11 and 20). Unfortunately, the clus-



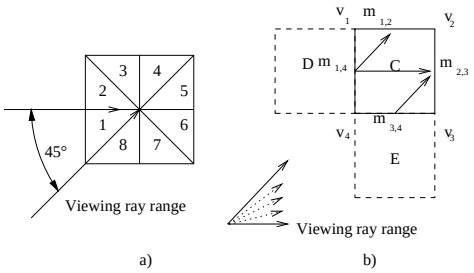


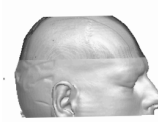
Figure 4.13: a) Viewing domain decomposition in 2D. b) face-maximum propagation for cluster 1

ters differ in the way in which the estimations for rays entering through cell faces combine to the estimations for exiting faces. As the removal algorithm uses face estimations for determining non-contributing cells, the clusters can not be combined without sacrificing removal efficiency.

To achieve effective cell elimination, not only the influence of direct neighbors, but also the influence of more distant cells on rays has to be investigated. This can be done by applying a simple two-pass scheme for each cluster of viewing directions. For reasons of simplicity, the procedure will be explained in 2D. The extension to 3D is straight-forward.

For cell removal in 2D, the viewing domain is decomposited into 8 clusters, each one covering a range of viewing directions spanning 45 degrees (figure 4.13a). Rays within cluster 1 can enter cell C only through edge $\overline{v_1v_4}$ or $\overline{v_4v_3}$. Considering just the values at the cell's vertices, cell C has no influence on the maximum of rays through $\overline{v_1v_4}$, if $min(v_1, v_4) \geq max(v_2, v_3)$ – in this case, the maximum contribution of the cell to any ray entering through $\overline{v_1v_4}$ and leaving through $\overline{v_1v_2}$ or $\overline{v_2v_3}$ is located on the edge $\overline{v_1v_4}$. As this edge is shared with cell D which is traversed by the rays earlier, C does not contribute to the rays. Similarly, the cell has no influence on rays through $\overline{v_4v_3}$ if $min(v_3, v_4) \geq max(v_1, v_2)$ due to the contribution of cell E – the condition for $\overline{v_4v_3}$ has to consider the influence of v_1, as a large data value at v_1 may influence the data gradient within the cell in a way that rays through $\overline{v_4v_3}$ obtain a larger value within the cell than at $\overline{v_4v_3}$ or $\overline{v_2v_3}$.

By preprocessing the volume in a spatially consecutive order, the influence of cells on ray maxima can be propagated with an advancing front approach. In the example in figure 4.13b, cells D and E are processed before cell C is reached. For both of them, lower-bound estimations of the maximum for rays which leave the cells have been calculated (for this cluster, rays leave cells either through face $\overline{v_2v_3}$ or through face $\overline{v_1v_2}$). This means, that at the time C is processed, lower-bound estimations $m_{1,4}$ for rays entering through v_1v_4 and



$m_{3,4}$ for rays entering through v_3v_4 are available, which already include the influence of more distant (preceding) cells and the influence of the edges themselves ($min(v_1, v_4)$ and $min(v_3, v_4)$). Thus, the revised test for the irrelevance of cell C is $m_{1,4} \geq max(v_2, v_3)$ and $m_{3,4} \geq max(v_1, v_2)$. If both conditions are true, C is removed and never ever considered for MIP anymore. After classifying the cell, the lower-bound estimations for the maxima of rays leaving the cell have to be computed. As for the investigated cluster of viewing directions only rays which have entered the cell through $\overline{v_1v_4}$ can leave through $\overline{v_1v_2}$, the estimation for rays through $\overline{v_1v_2}$ is $m_{1,2} = max(min(v_1, v_2), m_{1,4})$. As rays entering through both, $\overline{v_1v_4}$ and $\overline{v_3v_4}$ can leave through $\overline{v_2v_3}$, the estimation for $v_2v_3 = max(min(v_2, v_3), min(m_{1,4}, m_{3,4}))$.

While the first sweep allows to identify and remove low-valued cells located behind higher-valued parts of the volume, a second sweep in the opposite direction is required to propagate the influence of high-valued cells to cells reached earlier by the rays of this cluster. The second sweep is identical with the first sweep with the exception of the inverted orientation of the rays and thus an inverted volume scan order. Cells which have been removed during the first sweep do not influence ray values during the second sweep. Considering also cells removed during the first sweep would result in mutual elimination of cells and would lead to holes in the volume.

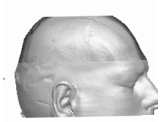
The two-pass method presented above can be extended to 3D in a straightforward way. For a decomposition of the viewing domain into 24 (12) clusters, rays may enter and leave cells through three faces for each cluster. Lower-bound estimations have to be propagated for faces instead of edges.

To even further increase the efficiency of removal and to compensate for noise which is usually present in MR data sets, the removal process can be modified to remove also cells which violate the exact criteria by a factor which does not exceed a user specified threshold (removal tolerance). After the preprocessing with a 1% tolerance, on average only about 30% of all cells remain as possibly contributing for a single view-set. For detailed results please refer to section 4.2.4 and table 4.5.

4.2.2 Cell Storage

In the following, the vertex coordinates $(x, y, z) = (min(x_i), min(y_i), min(z_i))$, with (x_i, y_i, z_i) being the eight vertices of a cell, will be used as reference coordinates of a cell.

Cells which have been identified during preprocessing as relevant cells are extracted from the volume and stored into a similar `RenderList` structure as



described in section 4.1.2. Cells are sorted according to descending cell-maximum (C_{max}) values within the cell array. For every cell its reference coordinates are stored into the corresponding 32 bit entry in the array. C_{max} is stored as the key attribute at `RenderListEntry`s.

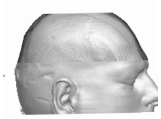
Using this alternative storage scheme, it is trivial to start MIP with high-valued cells. Speed-up factors of 10–20 compared to other optimized high-quality MIP approaches [5, 51, 67] are gained by just this part of the approach, in combination with the usual optimizations like evaluating only cells with C_{max} >current ray maximum, and noise skipping (see table 4.3, row 3 for detailed results).

Although sorting all cells by C_{min} would most effectively reduce the number of interpolations required, sorting them by C_{max} has several advantages. The encoding of C_{max} in the `RenderListEntry` allows to access it in an efficient way for testing a cell's relevance during projection. Within a group of cells with the same C_{max} sub-sorting is done according to descending C_{min}. Due to the small range of different values, sorting by C_{max} and C_{min} can be done using fast histogram-based sorting (complexity $O(N)$). During projection, cells with high C_{max} and C_{min} are processed first.

The chosen cell order is also well-suited for efficiently skipping cells which have been mapped to black due to windowing. If cells are sorted by C_{max} and rendering is started with highest values of C_{max}, rendering can be stopped after reaching the first cell with C_{max} mapped to black. For realistic window definitions (as used by medical doctors) this can significantly speed up the rendering (up to several times faster).

In addition to the fast computation of MIP due to re-sorting, another big advantage is gained: progressive refinement is achieved automatically, as cells which are most relevant to MIP are projected first. Projection can be stopped any time – the result will always be optimal for the given time-constraints. Also, computation of cheap previews is simple. Since interaction is crucial for using MIP, this advantage is also very important for practical use.

In the following a rough comparison between the traditional way of volume storage and the `RenderList`-based representation of cells is given: Storing the position instead of data values doubles the amount of memory required. As roughly 30% of all cells remain after the preprocessing step (cell removal), the amount of memory required is about 0.6 * original volume size (per cluster of viewing directions). For 12 clusters this results in an approximately sevenfold increase in memory size compared with the original data set. Considering data sets from medical applications and regular hardware resources, the storage requirements are high, but acceptable. If, nevertheless, the memory resources are a limiting factor, the data set can also be transformed into a single cell array without



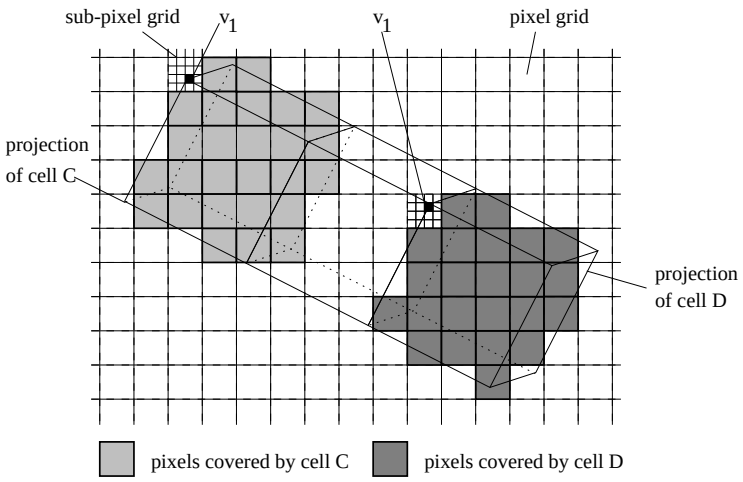


Figure 4.14: Cell projection templates. As the projections of cell C and cell D (position of v_1) have different sub-pixel offsets, the shape of pixel-sets affected by the cells differs. Thus C and D require different templates for projection.

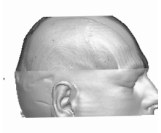
prior view-dependent preprocessing. This, of course, slightly increases rendering time (see table 4.7), but still gains results significantly faster than conventional approaches of comparable image quality, while requiring just twice as much memory as the original data set.

4.2.3 Rendering

To achieve high rendering performance, precalculated templates are used to determine all pixels of the image which are covered by a cell's projection. A fast parallel projection is used to calculate the position of a cell's projection in the image. Finally, the order of traversal of the cell array and a fast heuristic estimation of the upper-bound for the maximum value along a cell-ray intersection reduce the number of more costly and accurate trilinear maximum evaluations required.

Template Calculation

The main purpose of the template is to allow fast identification of the pixels affected by the projection of a cell. At each of these pixels, the cell's influence on the maximum of the ray through this pixel and thus to the pixel value has to be evaluated. A sufficiently accurate calculation of this contribution requires several steps of trilinear interpolation along the ray within the cell. To save time during cell projection, it is quite useful to pre-calculate entry and exit coordinates of the



ray for each pixel of the template. Interpolation weights (u, v, $w \in [0,1]$ for trilinear interpolation) of the entry and exit points are stored for this purpose.

As only parallel projection is used, the shape and size of the projected images of all cells is identical in a continuous image space. Due to arbitrary scaling and rotation of the volume for viewing and the discrete nature of a pixelized image, images of cells differ by an individual sub-pixel displacement with respect to the pixels of the image (see figure 4.14), which also leads to differing sets of pixels covered by a cell's image. To account for this shift with sufficient accuracy, the projection has to be performed with sub-pixel accuracy, allowing to place a cell's image in between image-pixel positions. The placement on a 4x4 grid within a pixel produces satisfying results.

The placement of cell images on sub-pixel positions leads to slightly different shapes of the templates for different x/y-displacements and also requires the calculation of individual ray entry/exit-positions for each of the templates. The resulting (4x4) array of templates can be directly accessed during rendering using the sub-pixel displacements of a cell's projection.

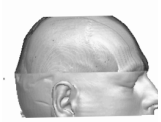
To optimize the rendering performance each element (=pixel) of the template stores a set of values:

- the offset of this element's pixel from the pixel containing the projection of the cell's origin (cell vertex v_1). As the image of v_1 can be located anywhere within the cell's image depending on the viewing direction, the pixel offsets may also become negative.
- the interpolation weights (u,v,w) for ray entry and exit coordinates at this element.
- the number of interpolation steps required along the ray/cell intersection
- A (du,dv,dw) vector defining a step along the ray.

Before rendering, the template is optimized to speed-up access. To avoid the necessity of skipping non-covered pixels within a template, just a list of covered template entries is stored instead of a 2D array. Thus, each template is just an array of image offset and ray-information elements for locations covered by a cell's projection.

Projection

As the parallel projection of a point can be performed by independently projecting it's x, y, and z coordinates, the projected positions x_{img} and y_{img} can be precalculated for the projection of all possible x, y, and z cell coordinates within the



volume. This results in six arrays – one for each x_{img}, y_{img} position of each x, y, and z coordinate.

For performance reasons, integer values are used to represent positions. As the summation of 3 integer values to obtain the x_{img} or y_{img} position of a cell's projection introduces an error of up to 1.5 pixels in x_{img} and y_{img}, the arrays have to contain sub-pixel coordinates to keep the error below one pixel.

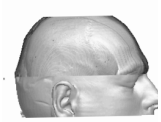
In combination with precalculated templates, the projection of a cell becomes simple and efficient. `Cmax` is obtained in a way described in the following paragraph, `img_width` is the width of the image in pixels. `.pixel` is the x or y coordinate of an image pixel, `.subpix` is a sub-pixel offset.

```
(xp,yp)=projection(cell.v1);
imgpos=xp.pixel+yp.pixel*img_width;
template=subtemplate[xp.subpix, yp.subpix];
for all elements in template
{
  if(image[imgpos+template_element.imgoffset]<Cmax)
    evaluate cell contribution at this pixel
}
```

Evaluation of the Maximum

Compared to the other stages of a MIP computation, the evaluation of maximum values within cells (trilinear interpolation) is by far the most expensive part. The reduction of the number and effort of evaluations which are required to generate an image is crucial for the performance. Performance can be improved on the one hand by using a less expensive (but usually also less accurate) evaluation method to approximate the maximum. On the other hand, more evaluations can be omitted if a good guess for the ray-maximum can be found early. The sorted cell-array allows to access and render most promising cells first. If the rendering is started with the projection of cells which have the highest cell maximum (and minimum), the probability of having to evaluate successive cells projected on the same pixels is significantly reduced. As can be seen in table 4.6, only about 2–4% of the cells of the original data set require the use of trilinear interpolation to evaluate their possible contribution to a MIP. The evaluation of the remaining cells is stopped either after checking C_{max} or after performing the slightly more expensive maximum estimation described below.

Values of pixels which are covered by the current cell and which are lower than the cell maximum potentially have to be replaced by a higher value. An



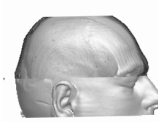
analytical solution for the maximum along the ray through the pixel is extremely expensive, and even a few trilinear interpolation steps along the ray are also quite costly. A cheap estimation of an upper bound for the ray maximum which is more restrictive than the cell maximum C_{max} can greatly reduce the number of more costly and exact evaluations of the maximum. The following observations can be used to present such a heuristic:

- If the maximum is located on the entry or exit point of the ray: applying two bilinear interpolations on the entry and exit faces of the cell gives the exact value of the maximum, i.e., $max(entry, exit)$.
- If for this ray the maximum is located within the cell, there must be some positive deviation from a linear evolution between the entry and exit point of the ray. If a cheap approximative guess for this nonlinearity within the cell can be found, the upper-bound of the ray can be estimated as $min(C_{max}, max(entry, exit) + deviation)$.

A fast approximative estimation of this deviation is $deviation = Max(0, Max(v_i + v_j)/2 - c)$ with v_i, v_j being data values at the vertices located at the ends of the four space-diagonals of the cell and c being the trilinearly interpolated value at the center of the cell (cheap evaluation, as a special case). Although this estimation is not a strict upper bound in all cases (just in about 99% of the cases), no visible impact on images of real-world data sets has been found. On average, this estimation for the ray-maximum within a cell is 30% lower compared with C_{max} as an estimation. When using this estimation about 60% less (of significantly more expensive) trilinear evaluations have to be performed (table 4.6). As only 25–30% of the trilinear evaluations actually lead to a change of a pixel value, a more tight upper bound estimation could gain even more performance. If the estimated upper bound for the ray is above the value of the examined pixel, several steps of trilinear interpolation within the cell are performed utilizing information stored in the templates to obtain the ray maximum.

Projecting high-valued cells first and using an additional estimation of the ray/cell-maximum is very efficient, as the value of each non-background pixel of the image is set only 1.3 to 4 times as compared to 10–25 times for MIP using conventional ray-casting with optimizations.

For the following pseudo-code summary of the projection of the cell array, `gray[]` stores the mapping from data-values to gray levels as defined by the windowing function.



levels of proposed approach	*preprocessing time*	*memory factor*	*speed-up compared to*		
			brute force	*+ $C_{max} > max$*	*+ignore noise*
`RenderList`	6s	2.0	1.3–1.4		
+ $C_{max} > max$	6s	2.0	6.1–7.4	3.9–5.3	3.1–4
+ ignore noise	6s	2.0	20–40	14–25	11–20
+ cell elimination	93.6s	3.6–12	28–43	18–27	14–22

Table 4.3: Comparison of ray-casting based MIP (trilinear interpolation, columns 4 and 5 with optimizations) with the proposed approach (`RenderList` of cells and optimizations).

method	*interpolations*	*cells*	*pixels set*
brute-force	100%	100%	24
+ $C_{max} > max$	26%	29%	24
+ ignore noise	7.7%	7.8%	13
cell array	1.3%	2.5%	1.65

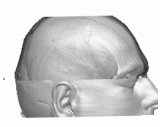
Table 4.4: Comparison of interpolation efficiency: brute-force (row 1) interpolates at every step along ray (col. 1), within every cell (col. 2). Each pixel changes 24 times (col. 3). Row 2: interpolation only if $C_{max} > max$. Row 3: also ignore low-valued cells. Row 4 shows the results for the proposed approach.

```
cells=get_cell_set(viewmatrix); // out of 12 preproc.
calculate_template(viewmatrix);
calculate_projection_arrays(viewmatrix);
for Cmax=highest to 0
  if(gray[Cmax]>0)
    for cell=cells.first_cell_with_Cmax
           to cells.last_cell_with_Cmax
      project(cell);
```

4.2.4 Results

The MIP algorithm presented in this paper has been implemented as a Java applet. The applet and further high-resolution results are available at (*http://www.cg.tuwien.ac.at/research/vis/vismed/CMIP/*). A C implementation is expected to exhibit a 30–40% better performance.

Table 4.3 depicts the speed-up achieved by the `RenderList`-based method compared to brute-force ray casting with trilinear interpolation (integer arith-



data set	*size*	*toler.*	*sweep 1*	*sweep 2*	*total*
hand	$256^2 \times 100$	1%	81%	3%	84%
hand	$256^2 \times 100$	2%	93%	2%	95%
kidney	$256^2 \times 69$	1%	56%	5.5%	61%

Table 4.5: Cell elimination results

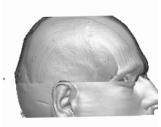
metic) and to optimized ray-casting (no interpolation in cells with $C_{max} \leq max$, skip background noise which is windowed to black). The second row (sorted cells, evaluation only if $C_{max} > max$) clearly shows the advantage of projecting high-valued cells first (factor 3–7). Even more performance is gained by the ability to skip low-valued cells without any overhead (row 3). This results in a speed-up factor of 11–40. Finally, by eliminating cells during the preprocessing step, factors of up to 43 compared to brute-force rendering and up to 22 compared to optimized ray-casting can be achieved.

Rendering times as compared in table 4.3 may depend on the optimization of the implementation. To obtain a less implementation-dependent measure for the efficiency of the approach, the number of interpolations performed, cell-, and image-access statistics of the algorithm are compared to ray casting in table 4.4. Compared to even optimized ray casting, the cell-array requires five times less interpolations. Also, updates of ray-maxima and thus of pixel values occur significantly less frequently during the rendering process. Due to the difference in the number of interpolations performed, it can be concluded, that a speed-up factor of approximately five is related to the avoidance of cell evaluations. The remaining portion of the speed-up is based on efficient volume traversal (strictly linear access to the cell array) and cell skipping.

Table 4.5 shows cell removal statistics for the preprocessing of the hand data set depicted in figures 4.15 and 4.16 and the kidney data set of figure 4.17a. As the second sweep eliminates just few additional cells, it can be optionally omitted to shorten the preprocessing. Increasing the tolerance for preprocessing mainly affects low-gradient areas which usually do not contain vessel data (figure 4.16).

Table 4.6 presents statistical information on the number of cells involved in the image creation, the number of upper-bound estimations for ray-cell intersections and the percentage of intersections which require a more accurate trilinear interpolation to evaluate the maximum within a cell. Rows 1 and 2 represent the hand data set for two different image sizes, Row 3 shows data for the kidney data set.

Finally, table 4.7 presents rendering times for MIP using the cell array approach. The rendering times have been measured on a PIII/750 PC.



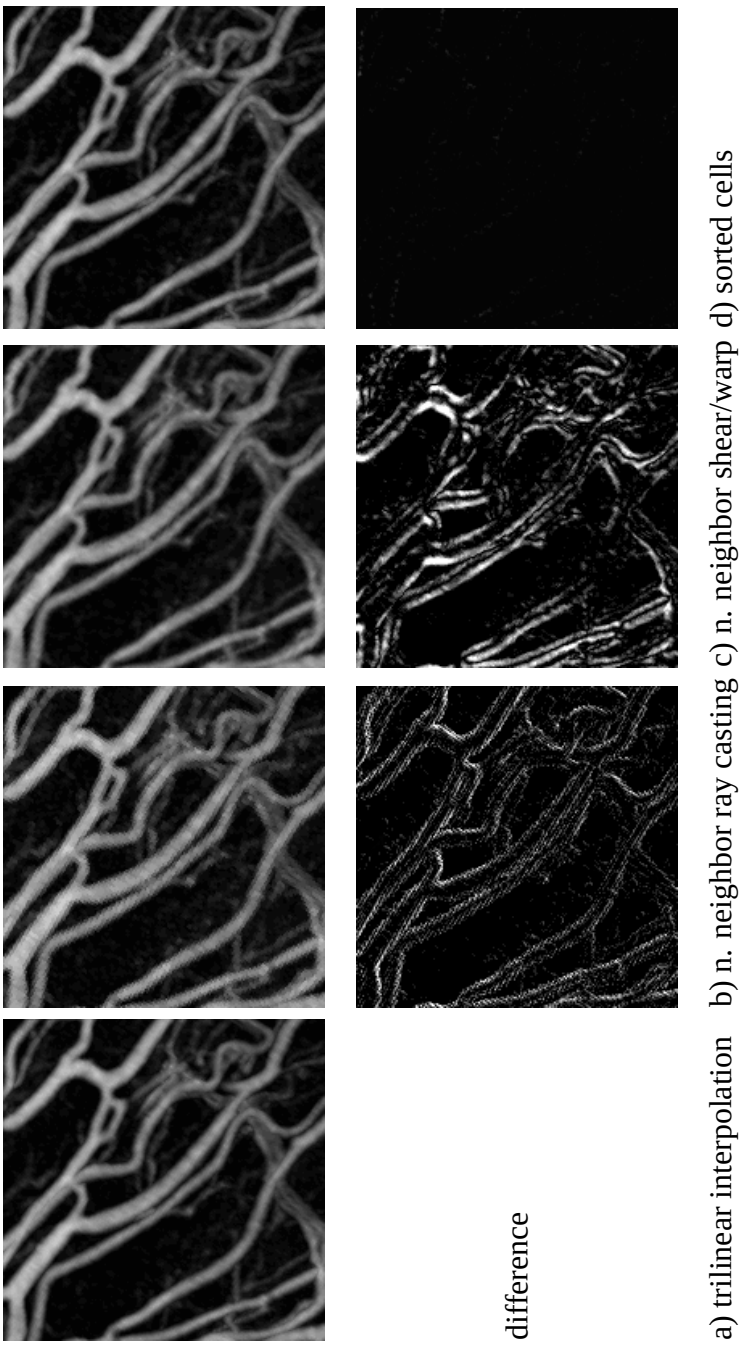
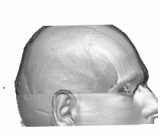
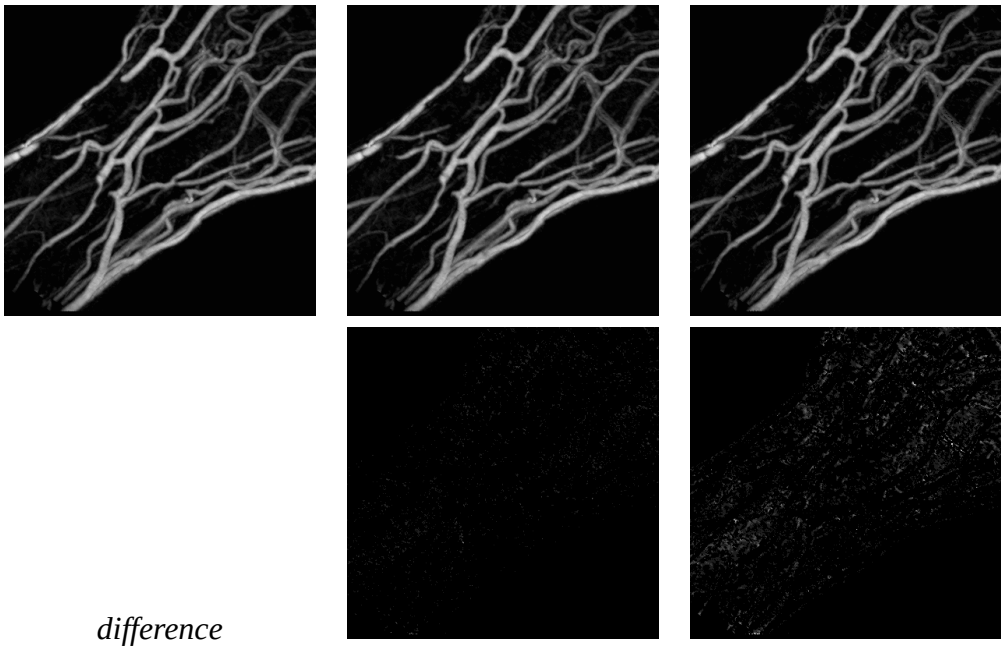


Figure 4.15: Quality comparison of MIP which was produced using a) ray casting with trilinear interpolation, b) ray casting with nearest neighbor interpolation, c) shear/warp projection with nearest-neighbor interpolation, d) `RenderList` and trilinear interpolation. The difference images depict amplified differences with respect to ray casting with trilinear interpolation.



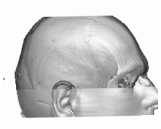


difference tolerance 0% *tolerance 1%* *tolerance 2%*

Figure 4.16: Results for `RenderList`-based MIP with different tolerance values for preprocessing (hand data set). As the tolerance is increased, deviations appear mainly within areas containing low-valued noise.

data	*image size*	*cells accessed*	*approximations*	*evaluations*	*pixel set*
hand	465^2	2.5%	1518k	26%	1.65
hand	512^2	4.1%	2439k	27%	1.5
kidney	465^2	3.2%	1077k	29%	1.46

Table 4.6: Efficiency of maximum-value evaluation: Column “cells accessed” gives the percentage of cells involved in the evaluation process. “approximations” is the total number of approximation operations, “evaluations” is the percentage of approximated ray intersections which required more exact evaluation. “pixel set” is the number of writes to a non-background pixel.



data set	*size*	*img size*	*without preprocessing*	*with preprocessing*
hand	$256^2 \times 100$	256^2	295ms	215ms
		512^2	615ms	450ms
kidney	$256^2 \times 69$	256^2	180ms	150ms
		512^2	350ms	315ms
head	$256^2 \times 124$	256^2	180ms	170ms
		512^2	380ms	365ms

Table 4.7: Rendering times for different data sets and image sizes. All data sets have been preprocessed with a 1% tolerance threshold. The timings depend on the used window definition. Some of the resulting images can be seen in figures 4.15d and 4.17a, b. User hardware: PIII/750PC.

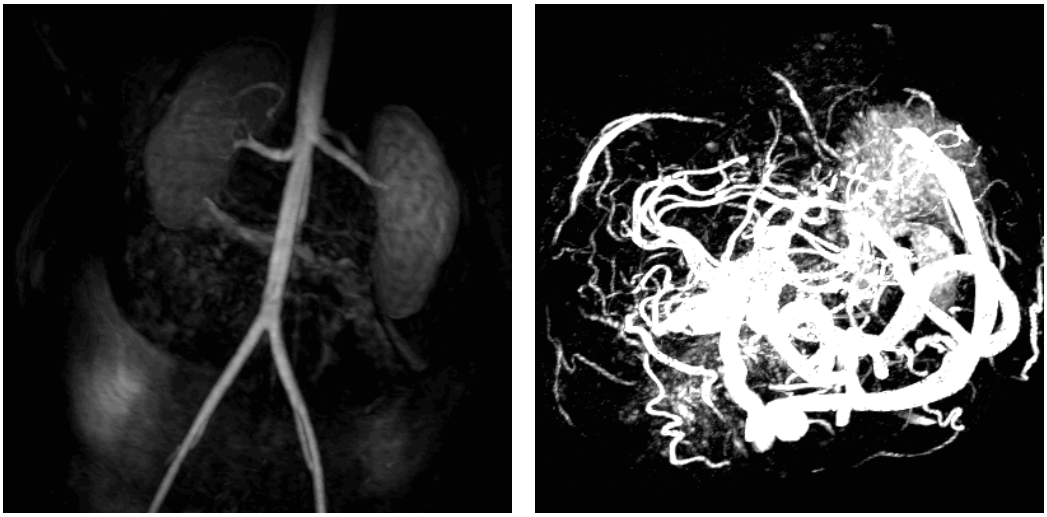
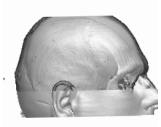


Figure 4.17: a) kidney data set b) head data set



4.2.5 Discussion

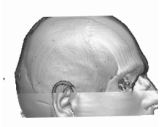
The proposed method provides interactive frame-rates on high-end PCs and still several frames per second on standard desktop hardware, and is well-suited to depict small details within the data which may be missed or deformed using other fast, but less accurate methods. Compared to high-quality MIP based on ray casting, the method avoids trilinear interpolations in a significantly more efficient way and achieves 20 times better performance, while providing better quality than other (shear/warp or hardware-based) approaches.

4.3 Display of Iso-Surfaces

A standard method to do surface rendering is first-hit ray casting for depicting surfaces implicitly contained in volumetric data sets (ray terminates when opaque surface is hit). Although (first hit) ray-casting is a versatile method which is able to provide high-quality images, it is not suited for interactive investigation of real-life data sets on consumer hardware. Although various approaches to improve efficiency of ray-casting exist, high-end multi-processor hardware is required to achieve interactive frame-rates [58]. Purely hardware-based solutions which exploit either 3D-textures or dedicated volume rendering hardware [48] can not be taken into consideration for a portable cross-platform application. Due to the fact, that fast polygon-rendering hardware is available on many platforms, explicit surface extraction methods like marching cubes [33] become more and more the method of choice for commercial applications. Depending on the capabilities of the rendering hardware the speed and the quality of the images differ from system to system and comparable performance can not be guaranteed for different platforms. Also for storing even optimized triangle-based surface representations huge amounts of memory are required. Mainly the high memory consumption makes a polygonal model less suited for surface rendering within a portable software.

The demands for a versatile and highly portable surface rendering algorithm can be summed up as following:

- software only approach: ensures highest portability and provides a comparable rendering speed on comparable platforms.
- low memory requirements: low memory requirements for the surface representation allow (in combination with software-based rendering) to operate the software under low-resource conditions (like on notebooks).



- high rendering speed: more then 10 frames per seconds on a standard PC for interactive navigation
- stacked surfaces: to provide context around objects of interest the algorithm should be capable of displaying several semi-transparent surfaces.

4.3.1 Preprocessing

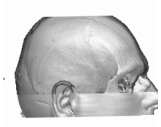
To achieve real-time performance on affordable hardware, the advantages of explicit surface extraction and representation are unified with the advantages of fast rendering using shear/warp projection. A voxelized surface representation is created during the preprocessing step, and stored in a `RenderList` for later rendering.

The creation of an explicit surface representation prior to rendering eliminates the need for surface detection during rendering as required by first-hit ray casting. To obtain a representation for the iso-surface, the volume is scanned for transitions between voxels with a value below and above the threshold, i.e., the iso-value. If a voxel has a data value greater or equal to the threshold and if it has at least one 26-connected neighbor below the threshold, it belongs to the surface and is considered to be relevant. The relevant voxels are extracted form the volume and stored into a `RenderList` / `RenderListEntry` structure. The voxels which have been extracted during the scan correspond to a 6-connected surface within the volume at the specified threshold value. The 6-connectedness of the surface voxels is required to ensure that no holes appear during shear/warp-based rendering due to displacement of voxels between successive slices.

For each surface voxel, it's position and the gradient vector are stored as attributes. To obtain a memory-efficient representation, the gradient vector is converted to polar coordinates instead of the usual (x, y, z) representation and stored using 14 bits (7 bits per angular coordinate, as a tradeoff between shading quality and computational effort for precomputing shading). The storage scheme allows to code 16384 distinct normal vectors which is sufficient for smooth shading of real-life data.

4.3.2 Rendering

For rendering, the surface voxels are replicated into three `RenderList` structures, one for each principal viewing direction. Within each one, the voxels are sorted according to the coordinate which corresponds to the axis most parallel to the viewing direction (For simplicity referred to as z coordinate. The procedure



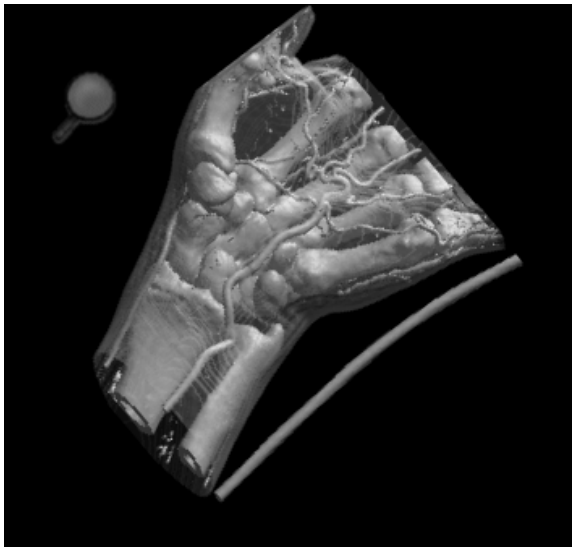


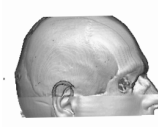
Figure 4.18: Surface rendering depicting two surfaces: opaque bones and vessels and semi-transparent skin.

for the other viewing-axes is analogous). As the sorting produces groups of voxels with equal z coordinates such that they are stored at consecutive positions within the array, the z coordinate does not have to be stored for each voxel. Instead, voxels with the same z coordinate are grouped into `RenderListEntry`s which store the z coordinate once for all contained voxels.

For rendering, the `RenderListEntry`s of all surfaces (as several surfaces could be viewed simultaneously, see figure 4.18) are merged and sorted by the z coordinate into a single list. To be able to distinguish objects during rendering, additional properties like the opacity of the surface are stored in each `RenderListEntry`.

During projection, the `RenderListEntry`s are processed in back-to-front order and their voxels are projected onto the base plane. For performance reasons, each voxel is projected onto exactly one pixel of the intermediate image plane, without interpolation. As the scaling component of the viewing matrix is not considered during this step of the projection, neighboring voxels within a slice of the volume are projected onto neighboring pixels of the intermediate image plane. During projection, compositing is performed, blending the old pixel value with the value of the current voxel according to the opacity stored at the currently processed `RenderListEntry`.

Clipping planes (see figure 4.19a) which operate on the `RenderList` voxel storage scheme can be implemented in a very efficient way. A



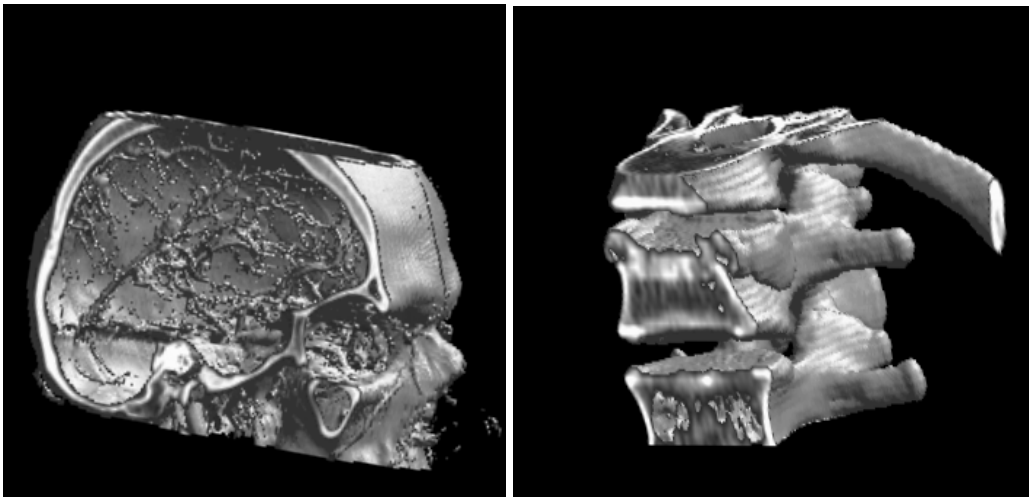


Figure 4.19: Interactive clipping planes allowing a) insight into objects b) displaying density data within an object

`RenderListEntry` stores voxels with the same depth with respect to the intermediate image plane. The order in which the voxels are projected onto the plane is not relevant, as each of them is projected onto a unique pixel of the intermediate image plane. Thus, if a clipping plane is applied, the voxels within a `RenderListEntry` can be resorted in a way, that voxels, which are clipped and become invisible, are moved to the end of the array section which belongs to this `RenderListEntry`. By setting a pointer to the last voxel which is not removed at the `RenderListEntry`, the clipped voxels can be efficiently skipped during rendering. The resorting of the voxels into clipped/non-clipped voxels requires just a single scan of the array with swapping of pairs of voxels. This approach is able to handle clipping at arbitrary planes or even non-planar objects.

Common visualization scenarios require not only the clipping and removal of parts of the data, but also to display original data values at the clipping plane within the context of the 3D object (see figure 4.19b). The display of original data on planar sections through the scene can be easily integrated into the presented surface-rendering approach. Usually, only voxels with values above the threshold which defines the surface have to be displayed, providing information on data within the object. When a new position for a clipping plane is defined, voxels which contain the plane and have a value above the threshold are extracted from the volume and stored in a similar way as surface voxels. Instead of storing gradient vectors, the data value at the voxel is stored as an attribute. The extracted voxels are again sorted according to their z coordinate and grouped into `RenderListEntry`s, which are merged with the remaining objects of the

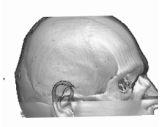


fig.	*volume size*	*voxels*	*shear*	*warp*	*frame*	*extract*
4.18	$256^2 \times 232$	232k	18ms	15ms	50ms	2.0s
4.19a	$256^2 \times 158$	376k	30ms	15ms	61ms	2.2s
4.19b	$256^2 \times 147$	253k	20ms	15ms	53ms	1.8s
4.19a	$512^2 \times 316$	1.8M	90ms	100ms	220ms	15s

Table 4.8: Rendering times for iso-surface rendering (rows 2 and 4 show the results for the same data set, with and without downsampling)

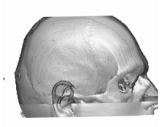
scene. During rendering, a flag at each `RenderListEntry` is evaluated to decide whether the attribute stored at it's voxels is a gradient and should be used for shading or if it is a data value which should be immediately written into the base plane.

4.3.3 Results

The time needed for rendering and surface extraction have been measured on an PII/400 PC using the Java virtual machine 1.1.8 from Sun with a Symantec just-in-time compiler (for rows 1–3 of table 4.8). The corresponding data sets have been resampled to fit into a 256^3 cube. The size of the images is 512^2 pixels. While the time required for the shear step depends only on the number of projected voxels, the time required for warping the intermediate image plane into the final image and also the time required by Java to transfer the image to the screen buffer (included in the overall frame time in column 6) depends mainly on the size of the image. As interpolation is only performed during the warp step, large images (1024^2 for example) generated out of 256^3 data sets are blurred due to strong scaling during interpolation. Row 4 of table 4.8 shows the time for rendering a data set at original resolution ($512^2 \times 128$) on an AMD Athlon 600 PC. As equal voxel size is required in all dimensions, the data set is treated as a $512^2 \times 316$ volume during surface extraction. Using data at the original resolution gives satisfactory results for 1024^2 images. The time required to recompute the shading table is negligible.

In table 4.8 the column "voxels" gives the number of voxels extracted from the volume to represent the surface(s). The time required for extraction includes the computation of gradient vectors using the central difference method.

The memory requirements for storing a surface are moderate. For each principal viewing direction each voxel stores the two other coordinates ($2 * 8$ bits) and the gradient vector (14 bits). For all three principal directions this sums up to 90 bits per voxel. This can be compared to the requirements of a polygonal model



of a surface which uses triangle strips: Making a fair assumption, that after an optimization of the surface representation there are approximately just as many vertices in the model as voxels in the presented surface representation, the model requires at least 24 bytes (192 bits) for each vertex (3*float coordinates, 3*float normal to allow handling by graphics hardware). Additional memory is required for referencing the vertices within the strips (indexed triangle strips). Even without considering the memory required to store this connectivity information for a geometric model, `RenderLists` require significantly less memory.

4.3.4 Discussion

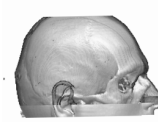
The `RenderList`-based algorithm for surface display has been designed for providing real-time frame rates for viewing medical data within a Java-based diagnostic application. By using purely software-based rendering, the algorithm achieves similar performance on different hardware platforms. By exploiting the advantages of explicit surface extraction and shear/warp projection, high frame rates are achieved at low memory cost. The only non-interactive step of the visualization process is the extraction of surface voxels during the preprocessing step (1-2 seconds for a typical 256^3 data set). Surface extraction has to be repeated each time a new iso-value is specified.

Compared with a polygonal representation of the boundary surfaces, this approach preserves the full accuracy of the data set at much lower memory cost, and allows interactive rendering on low-end hardware.

4.4 Extended shading models

As already mentioned before, various shading models can be applied at interactive frame rates to render the `RenderList`-based volume representation, if a quantized representation of the gradient is stored as an attribute with every voxel. Gradients which are usually given using 3 float (=96 bit), or at least byte coordinates (=24 bit), are first quantized to 12–16 bit to obtain a compact representation. As the number of distinct gradient directions is rather low after quantization, a good distribution of the quantized directions over the unit sphere is necessary.

If parallel projection is performed, and only directional light sources (located at infinity) are used, the evaluation of common lighting models like Phong shading does not depend on the position of the voxel, but only on the gradient vector, viewing direction, and light direction. For shading models where this assumptions



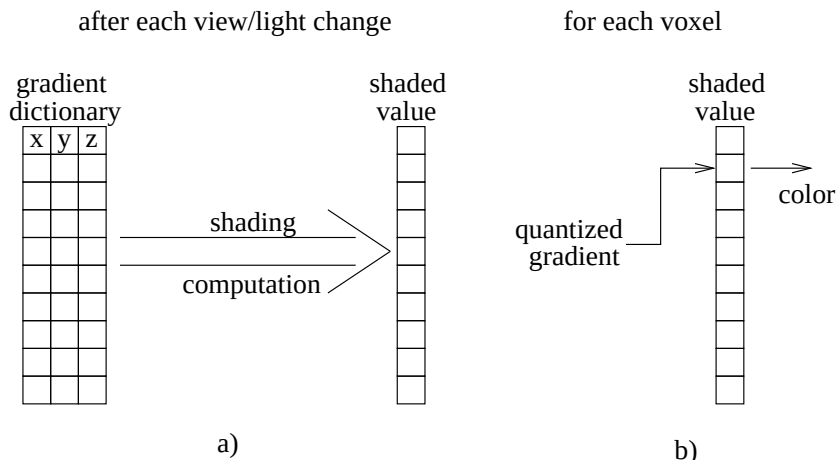


Figure 4.20: Shading look-up table creation (a) and application (b)

hold, the quantized gradients can be used as indices to access an array of precomputed shading values during rendering (figure 4.20). The table has only to be recomputed, if one of the influencing factors (light or viewing direction) changes. A gradient dictionary table is used to store non-quantized representations of each possible quantized gradient (required for the computation of table content). To update the table, the specific lighting equation is evaluated for each gradient vector in the gradient dictionary and the result is stored into the corresponding entry of the look-up table. The value stored in the look-up table is the shaded voxel color. A more flexible approach, based on storing an intensity value instead, which can be used to modulate the color of voxels is described in chapter 6.

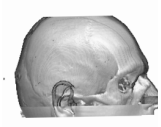
The limited number of evaluations required to update the look-up table, allows to apply even complex shading models without impact on the interactivity. The simplest model used is the Phong shading model, with the intensity I_p depending on gradient direction, viewing direction and light source position

$$I_p(P, V, L) = I_a + I_d G(P) \cdot L + I_s (R \cdot V)^n$$

with $G(P) = \nabla d|_P$ being the gradient at voxel P, L being the directional vector of the light and V the viewing direction. R is the main reflection direction

$$R = 2(G(P) \cdot L)G(P) - L$$

I_a, I_d, and I_s are the coefficients for ambient light, diffuse, and specular reflection, respectively, n controls the sharpness of the specular highlight. Due to the quantization of the gradient vector directions, n should be rather low to prevent visible artefacts of a very narrow highlight – values of 4–8 provide good results (figure 4.21a, b).



The look-up table based shading also allows to implement various non-photorealistic shading methods, for example, contour enhancement [11, 10]. The model assigns high intensity (and opacity) values to voxels with gradients most perpendicular to the viewing direction. Lower values are assigned to voxels with gradient vectors facing towards or away from the viewer:

$$I_c(P, V) = g(|G(P)|) \cdot (1 - |G(P) \cdot V|)^n$$

$g()$ modulates the intensity depending on gradient magnitude to provide higher values for voxels in regions of high gradient magnitude, which are usually parts of a surface. The exponent n controls the sharpness of the contour, and should (due to the quantization of the gradients) not exceed 8. The result of this shading technique can be seen in figures 4.21e and f.

As the application of the pure contour enhancement method provides only a sketch-like representation of the objects, without much information on shape details, both approaches, Phong and contour enhancement, can be combined to obtain a color C which depends on both models:

$$C = C_c I_c + C_p I_p (1 - I_c)$$

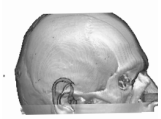
C_c is the color to be used for the display of contours, C_p is the color to be used for Phong shading. I_c and I_p are intensities obtained from the above lighting equations. The overall color is a weighted sum of the contour and the Phong color. Voxels located at object contours are rendered using the contour color only, voxels facing to the viewer are rendered with Phong shading only. If the contour color is set to black, this results in a darkening of object contours, which allows to distinguish internal edges of an object more easily (figure 4.21c, in comparison to pure Phong shading in figure 4.21a). Choosing white as contour color, brightens the contours, which is especially useful if the object is rendered semi-transparently (figure 4.21d, in comparison to pure Phong shading in figure 4.21b).

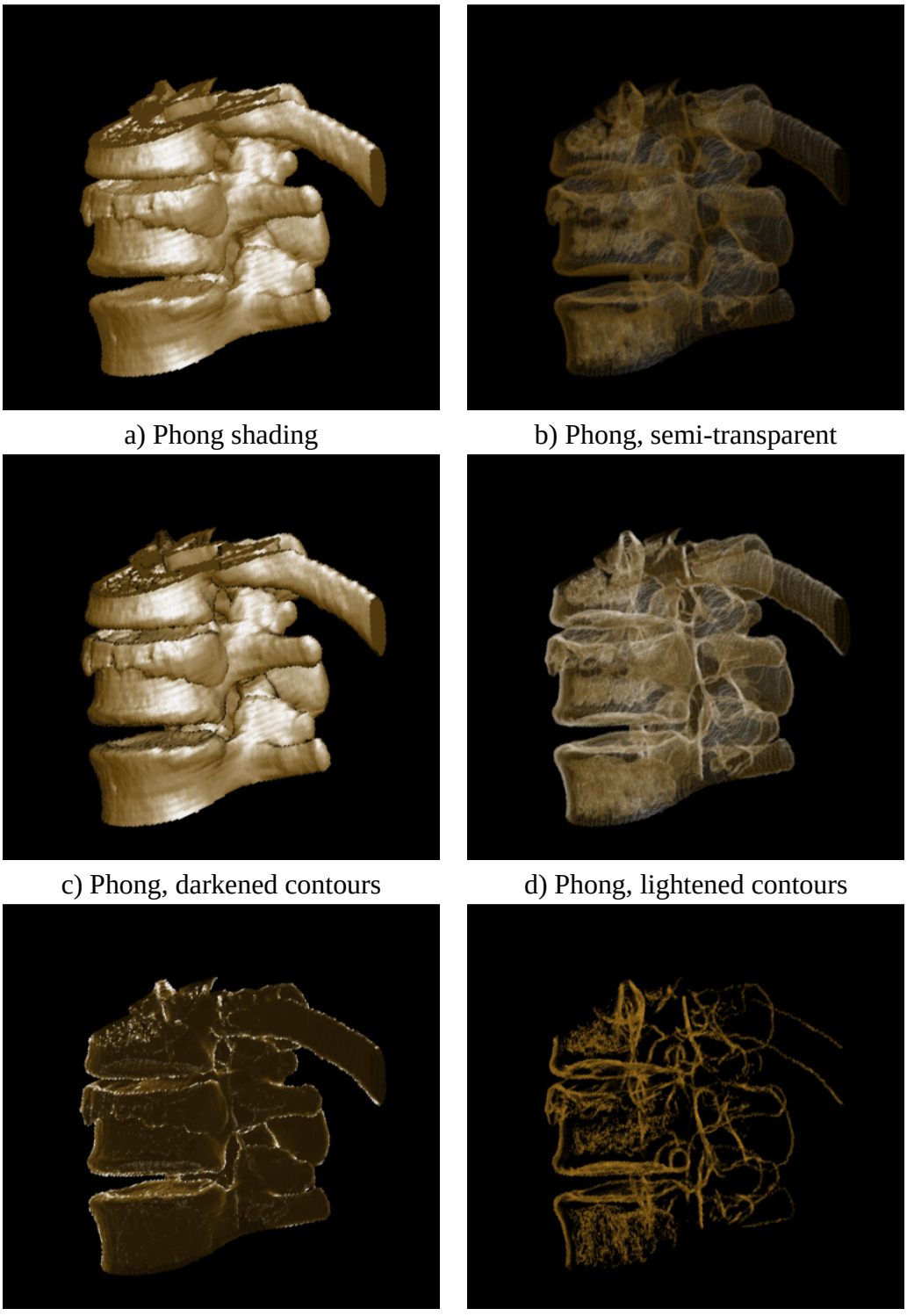
4.4.1 Optimized Preprocessing for Contour Rendering

In the following, two approaches for volume preprocessing and voxel extraction are discussed, which are optimized for contour rendering.

MIP of Contours

If contours should be rendered, voxel intensities are not constant data values from the volume as commonly used for MIP projection. Intensities I_c result from a





a) Phong shading

b) Phong, semi-transparent

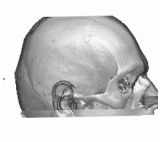
c) Phong, darkened contours

d) Phong, lightened contours

e) contour enhancement

f) contours only

Figure 4.21: Different lighting models applied to a scan of human vertebrae



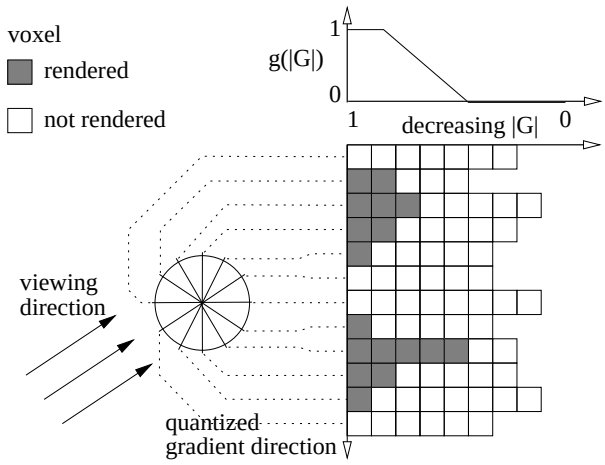
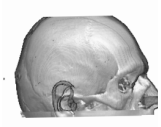


Figure 4.22: Voxel ordering for MIP (2D). Voxels are grouped by (quantized) gradient direction into `RenderListEntry`s. Within a group, voxels are sorted by gradient magnitude. Only groups with $(1 - |G(P) \cdot V|)^n > \varepsilon$ are rendered, within a group rendering is stopped after the first voxel with $I_c < \varepsilon$ is encountered.

function which depends on the viewing direction and gradient magnitude. This property makes a global pre-sorting of voxels by intensity impossible. However, proper ordering of the voxels can be used to group and efficiently skip groups of voxels mapped to black either by windowing of the gradient magnitude (modifying $g()$) or by the influence of the current viewing direction.

For maximum intensity projection, the order of projecting voxels is not relevant as $max(a, b) = max(b, a)$. Thus, voxels do not have to be ordered and projected in spatial order. Instead, voxels with the same or a similar gradient direction are grouped. This exploits the fact, that voxels which are not part of a contour for the current viewing direction, are mapped to low-intensity values. Entire groups of voxels with a similar gradient direction can be skipped, if the intensity $(1 - |G(P) \cdot V|)^n$ of a representative of this group is below some threshold ε (see figure 4.22). The quantization of gradient vectors for rendering leads to the required clustering of voxels into groups with the same gradient representation. For typical data sets, over 75% of all voxels can be skipped by exploiting just this scheme. Furthermore, within a group of voxels with the same gradient representation (a `RenderListEntry`), voxels can be sorted by gradient magnitude. If projection of voxels within a group starts with voxels with the highest gradient magnitude, processing of the `RenderListEntry` can be stopped as soon as the first voxel with an intensity I_c below ε has been projected. This arrangement of voxels allows to skip non-contributing parts of the data with utmost efficiency. The disadvantage of this optimization is the restriction of the compositing pro-



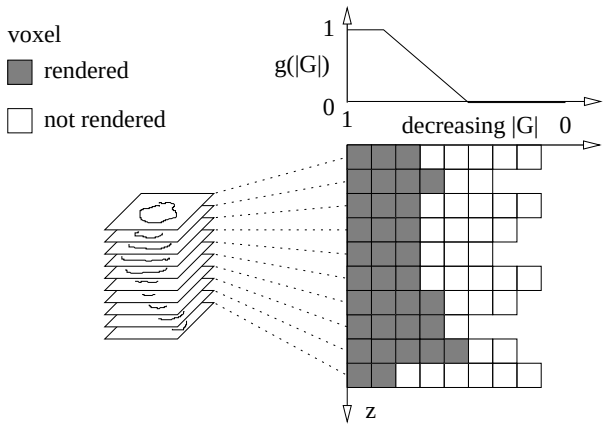


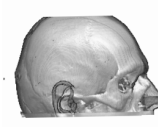
Figure 4.23: Voxel ordering for back-to-front rendering: Voxels within each slice (=`RenderListEntry`) are sorted by gradient magnitude. Voxels which are mapped to 0 by $g(|G(P)|)$ can be skipped efficiently.

cess to maximum intensity selection. Due to the arbitrary spatial ordering of the voxels, blending of voxel contributions is not feasible with this scheme.

Back-to-front Compositing of Contours

To maintain full flexibility in the choice of compositing operations, like local MIP or alpha-blending, a spatially consistent ordering of the projected voxels has to be maintained. Voxels with a gradient magnitude below a specific threshold do not provide a useful contribution to an image rendered using the contour shading model. Only about 25% of all voxels have a sufficiently high gradient magnitude, and are thus included into the `RenderList` structure, thus keeping the memory requirements at a reasonable level (For back-to-front rendering using shear/warp, three copies of the data are required, the MIP approach described before requires just one copy, as spatial ordering is not relevant).

Within a `RenderListEntry`, voxels are sorted according to gradient magnitude. During rendering, only voxels which are not mapped to black due to their gradient magnitude (see figure 4.23) have to be considered. Voxels mapped to black due to the currently used $g(|G(P)|)$ are located at the end of a `RenderList`'s voxels and can be efficiently skipped. Compared to the MIP-only ordering of voxels described earlier, significantly more voxels have to be rendered. Voxel skipping is only based on the gradient magnitude of a voxel, but not on the view-dependent property of being part of a contour.



	volume size	*voxels rendered*	*time*
head/MIP	$256^2 \times 225$	102k	85ms
head/back to front		366k	150ms
screws/MIP	$256^2 \times 241$	337k	130ms
screws/back to front		942k	270ms

Table 4.9: Rendering times for contour rendering

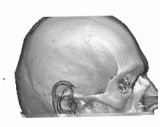
4.4.2 Results and Discussion

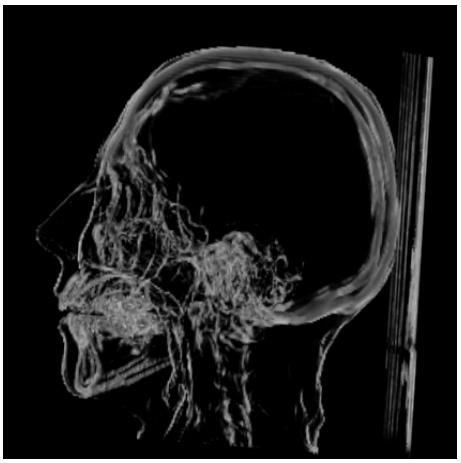
The most important factor for the quality of contour images is the accuracy of gradient vectors. Quantizing gradients to a few bits for interactive rendering, limits the size of exponent n. High values of n result in very sharp and thin contours. The quantization error of gradients close to the contour is therefore amplified and results in too bright or too dark voxel contributions. For a quantization to 12 bit, an exponent of approximately 4 provides a sufficiently narrow contour without producing disturbing artefacts (figure 4.24).

Due to more efficient skipping of black voxels and a simpler compositing operation for projecting a voxel, rendering using MIP is faster (see table 4.9) than when blending of voxel contributions is used. Although MIP allows to depict the most significant features of a volume (see figure 4.24e, f), the lack of occlusion and depth information in MIP images may be a disadvantage. The high interactivity of non-photorealistic rendering using MIP compensates for this disadvantage by adding time as an additional degree of freedom for the visualization (i.e., interactive view-point changes).

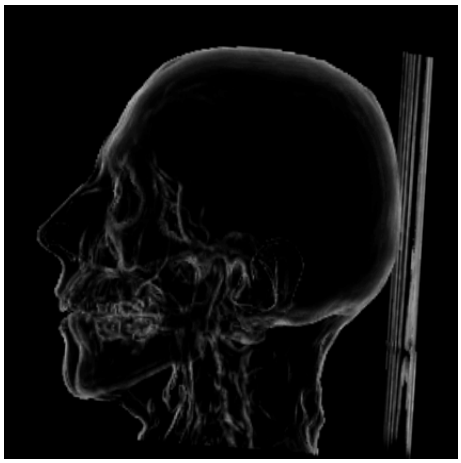
More flexibility is gained by using DVR (back to front compositing). The rendering times are acceptable (table 4.9), although slower than for MIP. Depending on the choice for voxel opacity, different effects can be achieved. By setting the opacity equal to I_c (figure 4.24a) an effect similar to MIP is achieved, with the difference, that occlusion and spatial ordering of the voxels is taken into account. Contours in areas with a higher gradient magnitude are depicted brighter than in areas with lower gradient magnitude. If opacity is derived from $g(|G(P)|)$ only, the resulting image displays a blended set of surfaces with lighted contours (figure 4.24b). This approach can also be used with good results to enhance contours [11] in addition to Phong shading for surface rendering (figure 4.24c). For segmented data which allows to distinguish between different objects, non-photorealistic methods can be easily combined with other rendering methods, for example with conventional surface rendering (figure 4.24d).

The rendering times in table 4.9 have been measured using a Java implementation of the algorithms on a PII/400MHz PC with Sun JDK 1.3 for Windows.

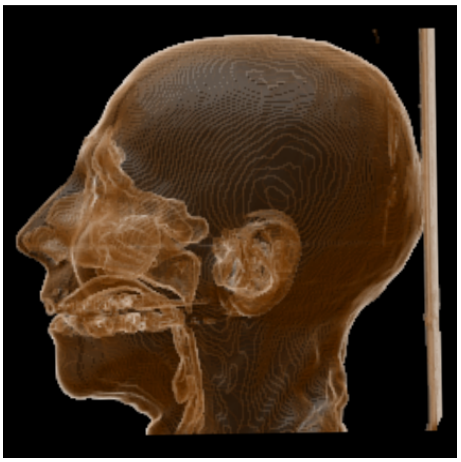




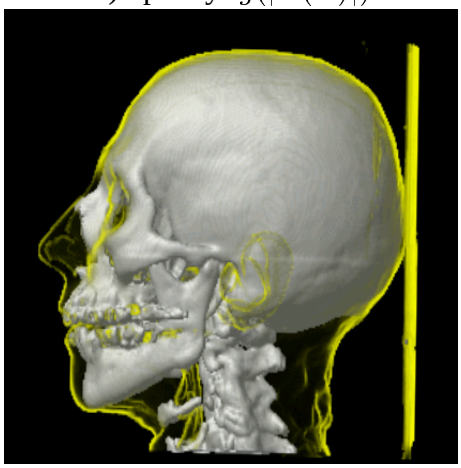
a) opacity=I_c



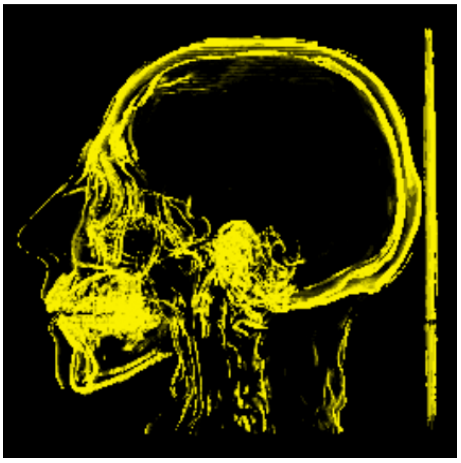
b) opacity=$g(|G(P)|)$



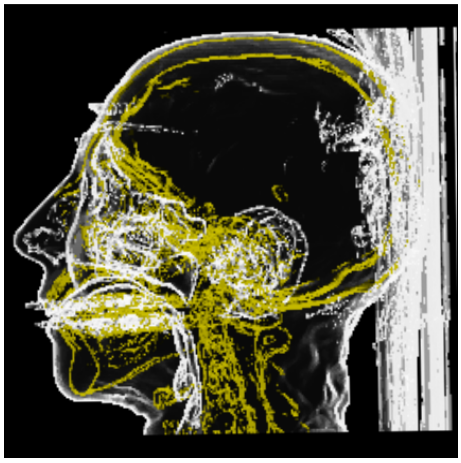
c) weighted sum of Phong and I_c



d) contours combined with an iso-surface

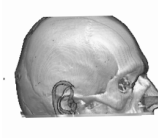


e) MIP of I_c



f) MIP of I_c with color transfer function

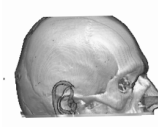
Figure 4.24: Interactive non-photorealistic rendering results



4.5 Summary

The previous sections presented the application of the proposed techniques, i.e., the identification of relevant voxels and their extraction and storage in `RenderList`s, to various rendering techniques. Depending on the desired compositing technique, lighting model, and transfer function setup, different optimizations and voxel ordering schemes can be applied to optimize performance. The bit resolution used to store voxel coordinates (8-11 bit per coordinate) and quantized gradient vectors (12-16 bit) for shading also varies between the described techniques, to optimize performance for a given application.

In chapter 6 a unified implementation will be presented which combines the presented techniques within a single framework, using similar bit resolution and voxel ordering for all techniques to allow fast and flexible parameter modification and rendering mode changes. Although the unified implementation is achieved at the cost of slightly lower rendering performance than highly optimized special-purpose implementations, the rendering speed is still interactive even on standard desktop hardware.



Chapter 5

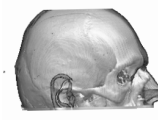
Space-Efficient Object Representation for Network Transmission

One major challenge of visualization in general is to deal with large amounts of data. Especially in volume visualization, sizes of common data sets range between several hundreds of Kilobytes, at the minimum, up to Gigabytes of uncompressed data. In medical visualization, for example, volumetric data sets of size $256^3 \times 16$ bit, i.e., 32 MBytes in total, are quite usual. If standard compression like `gzip` [18], for example, is applied, data sets usually shrink to about 30–60 percent of the original size – which is still in the range of MBytes.

Processing huge data sets itself poses high-performance requirements on the visualization software, but also storage and transmission of volumetric data sets easily produce bandwidth problems, especially if multiple data sets are to be treated. From medical applications, for example, we know that archiving 3D data sets, which accompany diagnosis data, significantly stresses storage devices currently available in common clinical setups.

Visualization over the Internet is even more critical, concerning the size of volumetric data sets, and concerning storage problems. Web applications like remote diagnosis suffer from low transmission rates, even over local networks (LANs).

The applicability of the more flexible fat-client solution to volume visualization strongly depends on the effectivity of the compression techniques used for transmission of the data set. Lossless compression techniques – for general data [18] as well as especially for volumetric data [17] – usually achieve rather low compression ratios (around 2), which is not sufficient to significantly widen

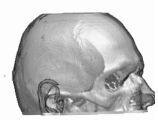


the bandwidth bottleneck. Using lossy compression [7, 32, 46], reduction ratios in the range of 5:1 to 50:1 can be achieved while maintaining acceptable quality of the visualization results. On the other hand, medical applications, typically prohibit changes to the accuracy of the data, as induced by lossy compression methods. Hierarchical methods, like wavelet compression [32] combine advantages of lossy and lossless compression. By transmitting and considering just a small fraction of the coefficients (around 5%) images of acceptable quality can be generated. On the other hand data values of the original volume can be reconstructed if all coefficients are considered. A useful property of wavelet compression and many lossy compression techniques is the ability to render compressed data directly, without prior expansion and decompression.

Polygonal representations of structures within the volume (e.g., of isosurfaces) can be used to realize solutions which are a compromise between a pure thin- and fat-client approach. The volume is kept at the server, just the polygonal surface model is transmitted and rendered at the client. Changes of viewing parameters require local rendering only, just changes affecting the shape of the model require a recomputation at the server and transmission of surface data over the network. To reduce the bandwidth required to transmit the model and to improve the interactivity of rendering at low-end clients, progressive refinement as well as focus-and-context techniques can be used [15], trading quality of representation (in less relevant regions of the volume) for speed.

Pure thin-client solutions on the other hand, allow to perform visualization on low-end clients making at the same time shared use of special purpose hardware at the server (multiple CPUs and/or VolumePro board [48], for example).

One approach to determine the effectiveness of compression techniques for volumetric data sets and their suitability for Internet-based visualization is to compare the size of compressed volumes versus the size of images of the same data. This comparison is useful as it directly corresponds to the trade-off between thin- and fat-client solutions. If sizes of compressed volume data sets are in the same range as sizes of images thereof, and given the client to provide sufficient computational performance to carry out most of the visualization steps itself, then fat-client solutions become feasible even via the Internet. The proposed technique [40] achieves compression rates such that, given a 256^3 data set as well as 512^2 images (24 bits per pixel) in compressed GIF-format, about 2–5 images already are bigger in size than the compressed volume data set.



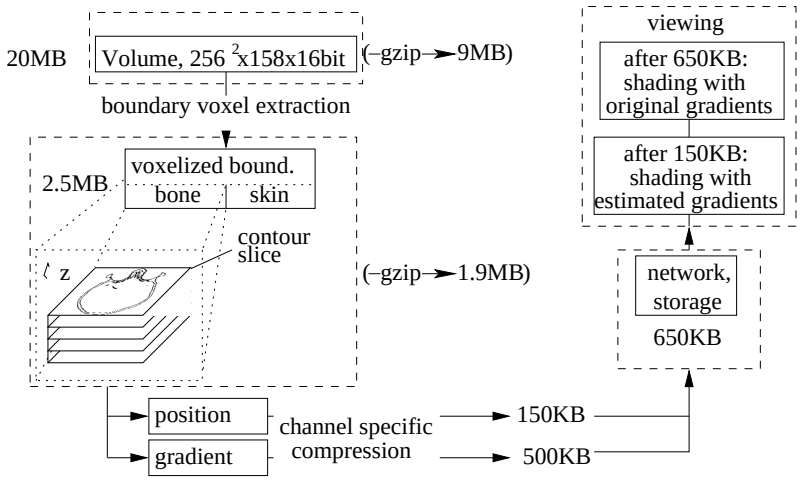
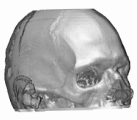


Figure 5.1: Boundary extraction, compression, and visualization pipeline

5.1 The Basic Idea

The effectivity of the presented approach is based on the observation that for the vast majority of applications, especially in medical visualization, volumetric data is rendered by displaying either iso-surfaces [33] or surface-like structures defined by areas of high gradient magnitude [29]. In both cases, the result of the visualization is determined by contributions of just a small fraction of all data samples. By just coding those voxels of an object, which actually contribute to its visual appearance, i.e., the voxels stored within the `RenderList`, the size of the data set is greatly reduced. Thereby, a small-scale boundary representation of volumetric objects is generated (figure 5.1, Sect. 5.2). Compression of the `RenderList` representation, which exploits spatial coherence among neighboring voxels, produces a very compact object representation (Sect. 5.3) which is well-suited for network transmission (Sect. 5.4). The information contained within this representation of objects allows interactive rendering at a client without any dependency on hardware-support, and with more flexibility regarding visualization parameters than polygonal surface representations (a demonstration applet is available from `http://bandviz.cg.tuwien.ac.at/basinviz/compression/`).

The first step to obtain an efficient representation of bounded objects within a volumetric data set is the identification and extraction of voxels which contribute to the object's visual representation, i.e., the boundary of the object. This is performed during a preprocessing step using one of the techniques described in chapter 4. Best compression results are obtained if object surfaces (iso-surfaces in general) are created and stored into `RenderLists`. Usually just 5–10% of all voxels belong to the boundary representation.



Within the `RenderList`, voxels are grouped into slices sharing the same z coordinate (see figure 5.1). Within a slice, the boundary voxels form contours of the object – a set of connected sequences of voxels. Exploiting spatial coherence of the contour, the positions of voxels within the slice are efficiently encoded into a compressed data stream. Voxel gradients are compressed in the same order as the corresponding positions, using a special compression scheme. Additional streams of voxel attributes (= data channels), like data value, gradient magnitude, etc., can be optionally encoded in a similar way. The output of the compression step is a boundary representation of volumetric objects, typically compressed by a factor of 10–100 compared to the original volume.

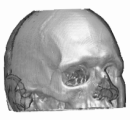
By transmitting the data channels in a specific order, for example, position data first, gradients last, a preview of the objects with full spatial accuracy can be displayed (figure 5.4) after transmitting just a few Kilobytes of data (using estimated gradients for shading).

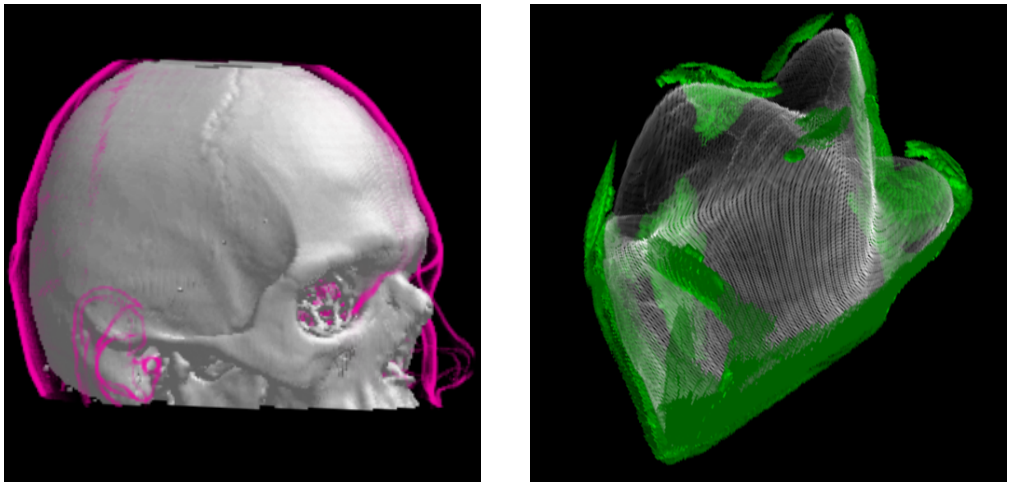
The decompressed boundary representation can be directly converted to a `RenderList` and rendered. Compared with a polygonal representation of the boundary surfaces, this approach preserves the full accuracy of the data set at much lower memory cost, allows interactive rendering on low-end hardware and provides more flexibility with respect to rendering parameters. Transparency, non-photorealistic shading, and the fusion with truly volumetric objects are easily possible without performance degradation (see figure 5.2 for examples).

5.2 Extraction of Boundary Voxels

For boundary voxel specification, either the iso-surface metaphor, or a predefined and explicit segmentation mask is used. In the first case, the technique described in chapter 4.3 is used. The 6-connectedness of the resulting sets of boundary voxels is useful for exploiting coherence during compression of the contours. Boundaries of objects which are defined using a segmentation mask, can be extracted in a similar way and also result in 6-connected sets of voxels.

Although best compression efficiency is achieved for surface-like voxel sets, truly volumetric objects can be extracted and compressed in the same way. This is especially useful for the visualization of spatially complex structures, like vessels in medical angiography data sets or complex chaotic attractors in the field of dynamical systems [4].





(a) (b)

Figure 5.2: a) By adjusting the visualization mappings at the client, the skin surface has been rendered using a non-photorealistic technique on top of the Phong shaded skull. b) A data channel containing distance information has been used to modulate opacity of the basin surface to emphasize areas of almost-contact between the surface and the attractor contained within.

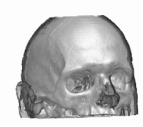
For the extracted voxels, attributes like voxel position, data value, gradient direction and magnitude, and application specific attributes are stored. When only the display of shaded surfaces is desired, storing voxel position and gradient direction is sufficient.

5.3 Data Compression

Individual objects within the volume are compressed separately. Voxels of each object are grouped into slices of voxels with the same z coordinate which are processed sequentially (see figure 5.1). To ensure effectivity, different data channels (attributes) have to be compressed using specialized compression methods.

5.3.1 Compression of Position Data

Boundary voxels within a single z-slice form object contours which consist of face-connected voxels (see figure 5.3). Exploiting spatial coherence and connectivity, voxels can be grouped into "*sequences*" which spatially follow the object contour. During compression, the slice is scanned for non-encoded voxels. When-



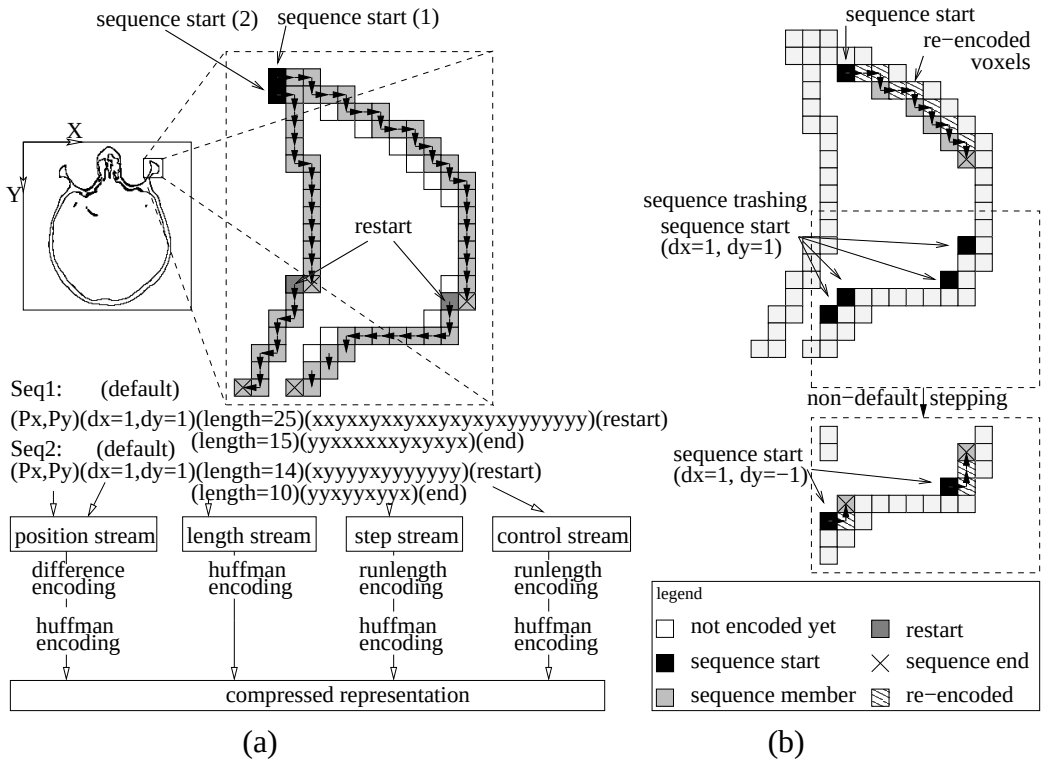
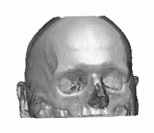


Figure 5.3: Encoding of voxel positions: slice scanned from top left to bottom right a) long sequences and sequence continuations b) re-encoding of voxels and non-default stepping direction to reduce number of sequence starts and thus position specifications

ever one is found, a new sequence is started and the position of the voxel (P_x, P_y) is stored. The sequence is continued, by selecting and appending one of the neighbor voxels at it's end. As the contour voxels are face-connected, potential candidates for continuation are located at $(P_x + dx, P_y)$ or $(P_x, P_y + dy)$ with dx, dy being respectively -1 or 1. Encoding the selection of one of the four neighbors as a successor would require 2 bits. If the choice is restricted to two neighbors by using constant values of dx and dy for a whole sequence, each voxel continuing a sequence can be specified by a single bit, which defines whether a step by dx or dy is used. Although this restriction reduces the flexibility and thus the average length of sequences, the cost per voxel within a sequence is cut by half, outweighting the disadvantage of shorter sequences.

In cases where a direct neighbor of the trailing voxel of a sequence is present, but can not be reached using the current (fixed) dx and dy values, a *sequence restart* can be performed, continuing the sequence at this neighbor with a new value for dx or dy. To realize this, each sequence is followed by a command code



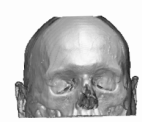
which specifies whether the sequence ends, or restarts with a different stepping direction. The presence of a restart code implicitly defines the position of the start voxel of the new sequence. As the previous sequence had to be terminated, no successors of it's last voxel are present in it's dx and dy direction. One of the remaining two neighbors is the second but last voxel of the interrupted sequence, so the other one necessarily is the starting voxel of the new sequence. The new values of dx and dy are derived from dx and dy of the old sequence. Depending on whether the last step of the sequence was dx or dy either dx or dy is inverted. Although being more restrictive than with an explicit specification of dx and dy, this strategy still allows encoding of cyclic structures with a single position specification and restart commands within a *chain of sequences*.

For each combination of dx and dy values, one of the possible stepping directions is preferred, whenever both ways can be taken. The preference is chosen in a way, that a clockwise processing of closed objects will stay as close to the outer border as possible. For example, for $dx = 1$ and $dy = 1$ like in the first sequence of figure 5.3a, steps by dx are preferred.

After the creation of long sequences, usually groups of short sequences or even non-connected voxels remain. Starting a new sequence for each of these voxels is expensive. Usually most of these voxels can be encoded at a lower cost by joining them into sequences re-using voxels already encoded earlier in the process (figure 5.3b). In general, a sequence has to be continued, reusing already encoded voxels, if this allows to reach non-encoded voxels at a cost which is lower than a "sequence end" and the start of a new sequence.

As the scan for non-encoded voxels within a slice is performed in ascending x and y direction, using $dx = 1$ and $dy = 1$ as a default stepping direction for newly started sequences is usually a good solution – voxels with smaller x and y coordinates compared to the current one are already encoded in this case. In some cases however, keeping $dx = 1$ and $dy = 1$ as default directions tends to generate a lot of short sequences "sequence trashing" (figure 5.3b). Instead it is better to first search "backwards" from the identified non-encoded voxel (using $dx = -1$, $dy = 1$) and to start the new sequence using $dx = 1$ and $dy = -1$ at the last voxel found (for reasons of simplicity no backward scan was performed for the sequences of figure 5.3a). At each sequence start one bit is used to store, whether the default stepping direction $(1, 1)$, or the direction of the backward scan $(1, -1)$ is used.

For further compression, the sequence data is separated into four streams. The *position stream* stores starting positions and stepping directions. Positions are stored using Huffman encoded differences between successive coordinate values (Typically 12 bits per starting code). The *length stream* stores information about



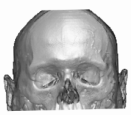
sequence lengths (Huffman encoded, 5 bits per sequence). The *step stream* stores the information for building up sequences (one bit per voxel). As dx and dy steps tend to cluster due to the presence of a preferred stepping direction, this information is run-length encoded, using again Huffman encoding for the run-lengths. The *control stream* is used for the sequence control information (end/restart, 1 bit per sequence). As many restarts at the beginning of encoding a slice are followed by short sequences collecting isolated voxels towards the end of encoding, which leads to clustering of restart and end commands, run-length encoding combined with Huffman encoding is also used here. Combining all those streams, an average of 2 bits is required to encode the position of a single voxel.

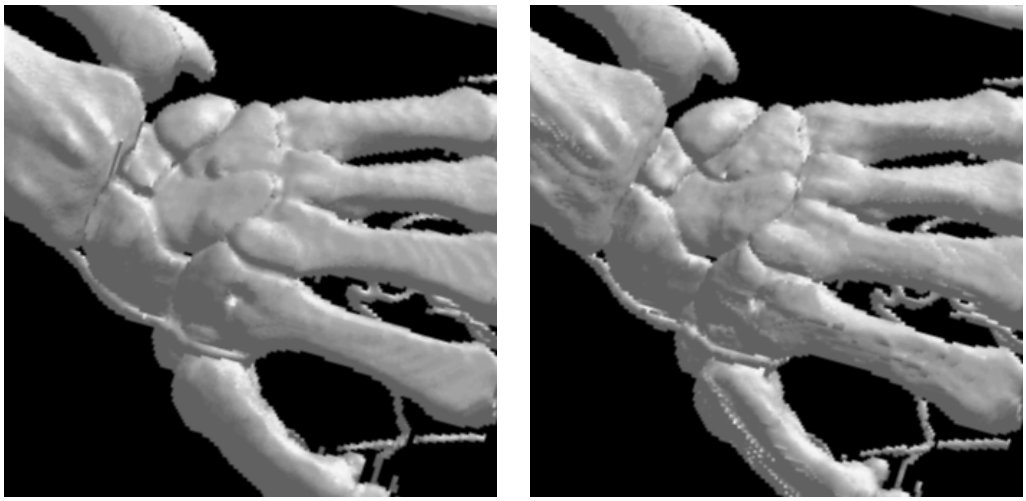
Within all other data channels, voxels are encoded in the same order as their position data. This ordering allows to exploit spatial coherence within voxel sequences also for attribute encoding. For subsequent occurrences of re-encoded voxels, of course, no attribute information is stored.

5.3.2 Compression of Gradient Direction Data

As a first step, gradient vectors are normalized, transformed to polar coordinates and quantized to 2x6 bits. The same representation is used by the rendering algorithm for interactive shading. By exploiting spatial coherence within the encoded stream of voxels the gradient information is reduced to 3–8 bit per voxel, depending on the smoothness of the boundary. Both polar coordinates are encoded into separate streams, storing differences between coordinates of successive voxels. As most of the difference data consists of sequences of values in the range of $[-1, 1]$ which are occasionally interrupted by larger values or clusters thereof, the encoder switches between two different coding schemes. The first scheme is used to encode sequences of differences in the range of $[-1, 1]$ using 1 bit for 0 (most common), 2 and 3 bits for -1 and 1, and a 3 bit code to switch to the other encoding scheme. Larger differences are encoded using Huffman coding with an extra symbol to switch to the encoding scheme for small values. A switch to the code for small values is only performed to encode sufficiently long sequences of small values (keeps cost of switching low).

The use of prediction techniques for estimating gradients and the encoding of the prediction error instead of encoding gradient differences seems to promise good results at first glance. Nevertheless, tests performed using linear regression [45] with a diameter of influence of 3 and 5 for gradient estimation, indicate that compression rates obtained using this technique are worse than with the above approach. One explanation of this observation might be that the given difference data actually exhibits very little spatial coherence in the data stream.





(a) (b)

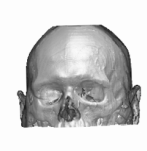
Figure 5.4: Estimated gradients (a) are used for shading until the original gradient data has arrived (b)

5.3.3 Compression of Other Data Channels

Additional data channels, like gradient magnitude, data value, etc., are compressed in the same order as the positions of the voxels to exploit spatial coherence also. Huffman encoding of differences of successive values and additional `zlib` compression (for further reduction of uniform areas) is used.

5.4 Data Transmission and Decompression

The compressed data set consists of two parts: a header, which contains control-information about the objects and their position within the data, information about additional data channels and how to use them for rendering. The body contains voxel positions and other data channels for all objects. The data within the body is arranged in a way which allows to obtain a view on the data as early as possible during loading. Objects and data channels which are more significant for the preset visualization mappings are stored and transferred earlier than less significant data (like actual gradient information). Data channels are subdivided into blocks of a few Kilobytes each. As soon as an entire block has arrived, it can be decompressed and displayed while the following data is arriving. This allows voxel data to be rapidly updated, without having to wait for the arrival of the entire channel. Finally, as gradient information usually accounts for most of the data to



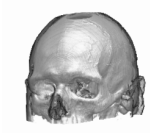
data set	*volume size*	*obj. voxels*	*bit/ pos*	*bit/ gradient*	*bit/ voxel*	*file (w/o gradients)*	*ratio to* `gzip`*ped volume*
head-bone	$256^2 \times 158$	378k	2.0	7.0	9.0	430k (95k)	1:22 (1:97)
head-skin	$256^2 \times 158$	231k	2.1	5.8	7.9	229k (60k)	1:40 (1:154)
hand-bone	$256^2 \times 232$	191k	2.5	7.8	10.3	246k (60k)	1:45 (1:186)
hand-skin	$256^2 \times 232$	170k	2.0	4.0	6.0	126k (41k)	1:89 (1:273)
engine	$256^2 \times 110$	298k	1.7	5.1	6.8	253k (64k)	1:13 (1:51)
teapot	256^3	152k	1.7	3.4	5.1	80k (28k)	1:4 (1:11)
attractor	256^3	769k	1.8	4.9*	6.7	639k (170k)	–**
basin	256^3	292k	2.2	0.6*	2.8	104k (80k)	–**

Table 5.1: Compression survey. * Scalar value channel instead of gradients. ** The attractor and basin data sets have been extracted from a volume with a vector of several scalar values at each voxel directly within the simulation application. No explicit volumetric representation is available outside the application.

be transmitted (See table 5.1), for boundary objects a locally computed gradient approximation (linear regression [45] with a filter size of 5 while interpreting the data as a binary object) can be displayed before the original gradient data arrives (see figure 5.4). For inherently binary objects, like basins of attraction within the phase space of a dynamical system [4] the locally computed gradients can entirely replace the transmission of gradients, significantly decreasing the amount of transmitted data.

5.5 Results

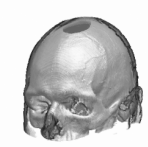
Table 5.1 presents the compression rates obtained by applying the technique to a collection of data sets from different application fields. The head and hand data sets are CT scans containing objects typical for medical applications. Bone and skin surfaces extracted from the data are usually made up from 1–4% of all vox-



els. Using our compression scheme the boundary data is compressed by a factor of 20–90 compared to the original volume when compressed with `gzip`. If gradient information is not stored but approximated at the client the compression factor increases to 100–270. The cost of compressing voxel positions within such data sets is relatively independent of the surface shape (approximately 2–2.5 bit/voxel). The cost for storing gradients depends on the smoothness and curvature of the surface and varies between 4 and 8 bit/voxel. For objects with artificial, "well-behaved" surfaces like the CT scan of an engine block or the voxelized teapot, better compression is achieved for both voxel position and gradient data. The attractor and basin-of-attraction data, obtained from the simulation of a dynamical system, is also effectively compressed – especially as the basin boundary is derived from a binary classification of space and no gradient information has to be stored – it can be reconstructed from the surface shape at the client. Compression for each of the examples mentioned above takes approximately one second on a PIII/733 PC. Decompression timings for locally stored data are similar on the same PC. An applet which implements the described techniques and all compressed data sets discussed and depicted here are available at
(`http://bandviz.cg.tuwien.ac.at/basinviz/compression/`).

5.6 Discussion

Many applications of volume visualization require the display of object boundaries. Using the presented compact volume representation, volume visualization becomes feasible even over the Internet, while still providing full spatial accuracy of the data. Representing just the boundary voxels of objects dramatically reduces the amount of data to be transmitted or stored. By exploiting known properties of the boundary voxels (like spatial coherence and inter-voxel connectivity) the data is further compressed. The resulting data representation is smaller by a factor of 20-250 than the volume compressed with `gzip`. The location of voxels within the volume is compressed very efficiently to about 2 bit/voxel. The compression rates for gradient data are lower, in the range of 3-8 bit/voxel, as gradient data is derivative information compared to the original data, and thus containing less spatial coherence. Using a proper gradient reconstruction scheme, gradients can be estimated from voxel positions only, allowing to display objects just after the well-compressed position data has arrived, instead of waiting for the original gradient information.



Chapter 6

RTVR – a Flexible Java Library for Interactive Volume Rendering

Data exploration and interactive presentation with low demands on computational and/or networking resources have been the driving factors for the development of the RTVR (Real Time Volume Rendering) library [39]. It unifies the techniques described in the previous chapters into a flexible framework which can be used to provide volume visualization on PC-hardware and can be easily extended to include new approaches and techniques. In this chapter some of the distinguishing design issues of RTVR are described which are responsible for its high efficiency with respect to real-time volume rendering and flexibility.

In contrast to established volume visualization toolkits like VolVis [2] or VTK [53], which cover a very broad range of data representations and applications, RTVR is focused on providing interactive visualization for rectilinear volumes on desktop hardware by relying on preprocessing, voxel extraction, and the `RenderList` as a flexible internal data structure which is well-suited for fast rendering [42, 43, 44]. The data structure itself and also the fast shear-warp rendering which is used by RTVR are not suited to perform high-quality rendering. On the other hand, z-buffer output from the method could be used to accelerate high-quality rendering which is based on ray-casting.

The memory-efficient way of handling volumetric data makes RTVR well-suited for remote rendering of volumetric data even over low-bandwidth networks [40], either for interactive presentation of data which has been generated offline, or within a split client/server approach for online visualization. The amount of data which actually is transmitted to the client for visualization is very low (about the size of several images, see chapter 5). The user is not restricted to pre-computed views and has full control over visualization parameters. The only

restriction for rendering is that just those parts of the volume which have been pre-selected for presentation and transmission can be rendered.

When used in a distributed client/server scenario, the software-only rendering approach of RTVR provides much more flexibility in terms of rendering parameters than volume previewing using texture mapping hardware, still at comparable or even lower costs in terms of bandwidth requirements.

In the following, visualization capabilities and the internal structure of the RTVR library are presented. Section 6.1 gives a short overview over the rendering features and visualization techniques which are implemented by RTVR. Section 6.2 presents RTVR's internal data structure, its handling of user interactions, and the rendering algorithms used. Timings for typical application scenarios are given in section 6.3.

6.1 Features of RTVR

One characteristic feature of RTVR is its way of handling and rendering of volume data, which is highly optimized to provide interactive feedback during the manipulation of viewing, rendering, and data mapping parameters. A high rendering performance is achieved by efficiently excluding non-relevant parts of the data from the rendering process. In common applications, like the visualization of medical data from CT or MR scanners, usually only a small portion of the data actually belongs to the object of interest. Furthermore, meaningful settings for rendering parameters may render the inner parts of objects opaque – a fact exploited by early ray termination techniques and also by the preprocessing included in RTVR.

RTVR interprets the input volume as being composed of objects, like bones, vessels, and other tissue making up, for example, a data set of a human hand (figure 6.1). The necessary segmentation information is either obtained together with the volume data itself from an external data source, or is interactively computed using simple threshold-based segmentation. For rendering, data mapping and rendering parameters can be individually assigned object by object. Besides the usual manipulation of object properties like opacity and color transfer function, also the shading model which is used for rendering can be defined individually for each object. This allows, for example, to combine objects which are rendered by using a standard shading model like Phong shading with objects that are rendered by the use of non-photorealistic shading [10, 11]. In addition to the shading model, the way in which voxels are blended to the image plane can be defined in an object-aware way. Most volume rendering packages only allow to

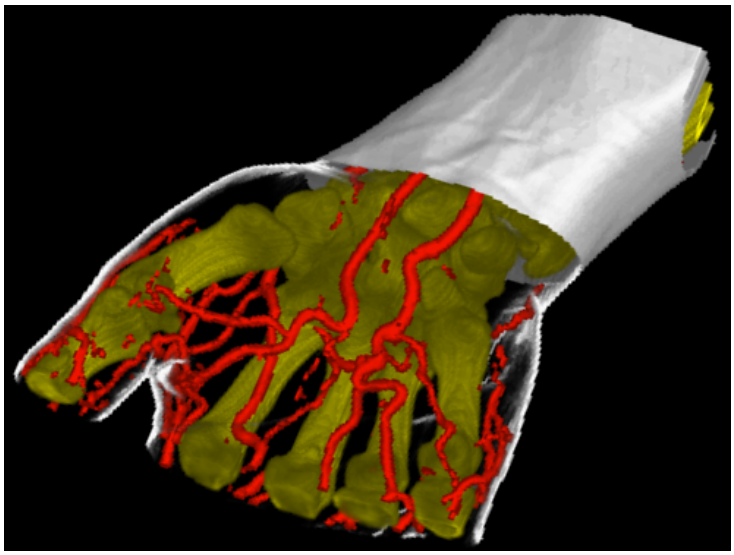


Figure 6.1: Visualization of a human hand created with RTVR: surface rendering of vessels, combined with direct volume rendering of bones, and a surface rendering of the skin. The skin is clipped into two parts, one part shaded using Phong shading, the other part using a non-photorealistic rendering model which emphasizes contours.

render a whole data set using either the usual opacity-blended compositing [26], or surface rendering [29, 44], or maximum intensity projection (MIP) [42]. In RTVR, compositing modes can be selected on a per-object basis and combined with another inter-object compositing mode (two-level volume rendering [22]). This allows to choose the most appropriate rendering and compositing parameters for each object, depending on the structure of the data and the goal of the visualization.

Another feature of RTVR is to support the visualization of time series of volumetric data and of multi-dimensional parameter-series of volumes from simulation. The large memory demands of such data are compensated by the fact, that data extracted from a volume and used by RTVR for rendering is usually much smaller than the original volume. Only extracted data of the current volume has to be kept in memory for rendering, remaining parts of the volume and data which belong to other time (parameter) steps are placed into a combined memory/disk cache.

Among other “standard” features of volume visualization systems, RTVR provides the ability to display data on planar sections through the volume, enables object and position picking by clicking into the rendered image, and supports the clipping of volumes or sets of individual objects at planes and more complex

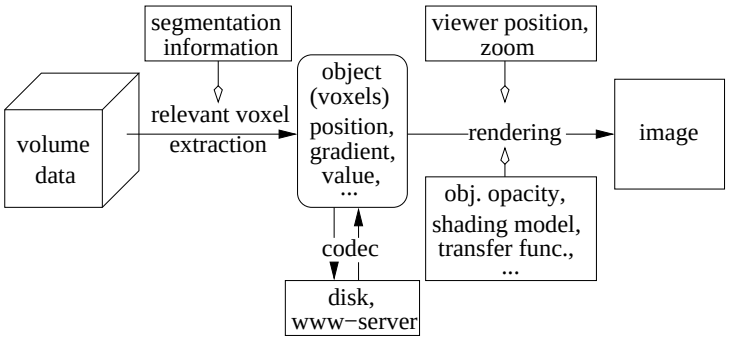


Figure 6.2: Volume data flow within RTVR: first, voxels with actually contribute to the visualization are extracted, then this representation of volumetric objects is used for fast and flexible rendering.

structures. Data which has been "clipped away" can be omitted from rendering – which is the most common approach – or rendered using different rendering parameters, for example, more transparent than the non-clipped part of the object, or using a different shading model, which for example, displays just contours.

6.2 RTVR Intrinsics

The basic object and rendering primitive of RTVR is a voxel, i.e., a single data sample within the volume. During a segmentation and data extraction step (figure 6.2), voxels which actually are relevant for the user-defined visualization are extracted and stored (object by object) within a `RenderList` data structure. The voxel extraction step usually leads to a significant data reduction. First, only a portion of the original volume actually belongs to objects of interest. Second, depending on the desired visual representation of the object, only a subset of the object's voxels has to be considered for rendering. If surface rendering (using a fixed iso-value) is performed, for example, a thin layer of voxels is sufficient as a representation of the object. When rendering an object by using opacity transfer-functions which depend on gradient magnitude [29], voxels with a low gradient magnitude do not noticeably contribute to an image and therefore can be omitted.

The voxel sets which result from object extraction are the basic data structures of RTVR. For visualization, this data is inserted into a scene graph and rendered. An intermediate representation, which can be used to store visualization results for later interactive viewing is produced by transforming extracted voxel data into a space-efficient compressed format (see chapter 5) [40] and storing it on disk together with the currently used visualization and rendering parameters. For ren-

dering, RTVR uses fast shear/warp projection as described in chapter 4, which requires the data to be given as isotropically spaced voxels. Fortunately, this does not really restrict the use of RTVR for visualization. Data which is given on non-Cartesian grids, can be resampled on-the-fly during the extraction of object voxels.

After extraction, the next step in the visualization process is the assignment of voxel attributes to optical properties (transfer-function mapping). One or two voxel attributes can be selected to influence a voxel's contribution to the visualization result. This attributes usually are the data value, the gradient direction, and/or the gradient magnitude. The restriction to two arguments per transfer function is imposed for performance reasons. For rendering, both values have to fit into a 16 bit field, which is typically subdivided into a 12 bit main channel for a more significant data value, and a 4 bit channel for a second, additional value.

The data values are used to index look-up tables to obtain and modulate color and opacity values in a way which is defined by the selected rendering mode. The look-up tables are used to implement different transfer functions and shading models in a very effective way.

Depending on the visualization parameters in use, some object voxels may not contribute to a visualization at all – as they may be, for example, totally transparent after application of the transfer function. A background thread identifies those voxels during idle-time and rearranges the data in a way that later no effort is spent on skipping them during the next rendering pass. This is especially useful for accelerating the rendering of "fuzzy" objects, where no exact information about object shape is available at the time of extraction.

The object data contained within a single scene graph can be simultaneously displayed in several views – a 3D view and several sections through the volume, for example. Parameter changes which influence the results of the visualization can be carried out either by using GUI components, or by directly interacting with objects within the rendered view. GUI components for parameter adjustments are automatically derived from the visualization pipeline and grouped into a "control panel". Within a (3D) view, objects can be selected by clicking on them, parameters like the camera position, zoom factor, light source position, or object opacity and transfer function can be manipulated by dragging the mouse.

6.2.1 `RenderList` as Data Representation

During the object extraction process, the volume is scanned slice by slice, producing for each object within the volume a distinct `RenderList` which is in fact an array of `RenderListEntry` objects, each one containing the object's voxels

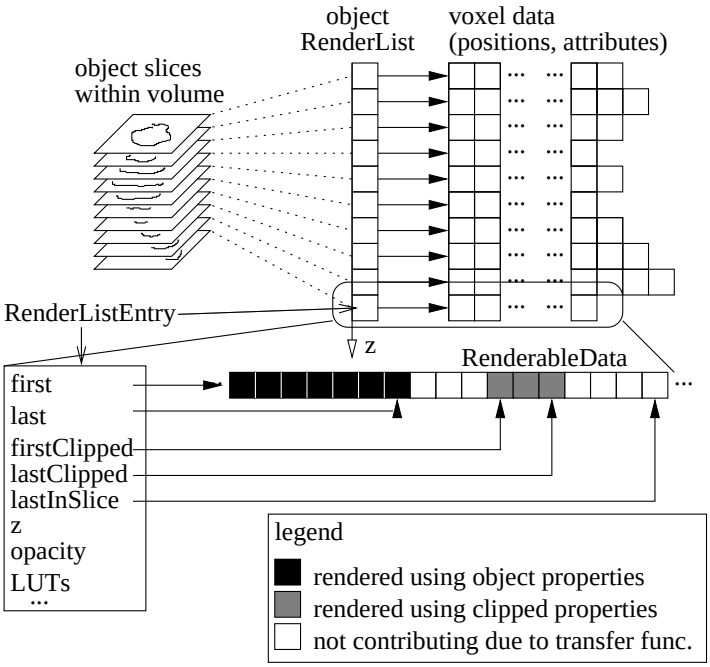
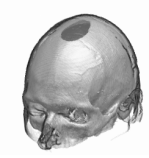


Figure 6.3: Volumetric object representation: voxels which are relevant for rendering an object are extracted slice by slice from the volume and stored into `RenderList`s.

within a slice (figure 6.3). Note, that the original (implicit) spatial arrangement of data values within the 3D array is sacrificed for an object-aware enumeration scheme of arbitrarily arranged voxels. For each voxel it's position within the slice and a user-definable set of attributes are stored. As gradient computation from extracted voxels is not trivial due to the lack of connectivity information within the `RenderList` structure, gradient direction and gradient magnitude are usually precomputed during the extraction step and stored with the `RenderList`. Typically, also the original data values (one or more scalar values) are added as attributes here. All the attributes of a voxel are stored in separate arrays, the `RenderListEntry` just stores information which is required for rendering:

- object-level opacity for clipped and non-clipped voxels
- look-up tables for clipped and non-clipped voxels
- specification of rendering mode for clipped and non-clipped voxels
- a reference to an array which contains a renderable representation of voxel data (derived from voxel attributes). Within this array, voxels between `first` and `firstClipped` belong to the non-clipped part of



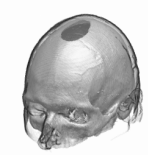
an object, voxels between `firstClipped` and `lastInSlice` belong to the clipped part. Only voxels between `first` and `last`, respective `firstClipped` and `lastClipped` have to be rendered, voxels between `last` and `firstClipped`, and `lastClipped` and `lastInSlice` have been identified by the optimizer-thread as non-contributing for the current transfer function setting and are not rendered.

The "blocking" of voxels into non-contributing, clipped, etc., as shown in figure 6.3, is achieved by simply reordering the voxels within `RenderListEntry`s during clipping and optimization operations. This may be done, as the voxel order within a slice is not relevant for the fast shear/warp algorithm in use for rendering.

For fast rendering, position and attribute information for each voxel is fitted into a single 32 bit integer. The x and y coordinates of the voxel are stored using 8 bit each, the z coordinate is identical for all voxels within a `RenderListEntry` as they are all extracted from the same slice of the volume and thus it is stored just once. Using just 8 bits per coordinate limits the maximum extent of an object to 256^3 voxels. Larger volumes and objects are internally split into 256^3 pieces and the missing high bits of the coordinates are encoded into an offset, which is also stored once at the `RenderListEntry`. The remaining 16 bits are typically split into a 12 bit and a 4 bit field which store the data attributes used for rendering as previously described. This "renderable" voxel representation is attached as an additional array to each `RenderListEntry`. Reordering of voxel data during clipping and optimization has to be performed synchronously on all attribute arrays as well as on the derived renderable data array.

Although the common coordinate stored at the `RenderListEntry` for all voxels is referred to as z for reasons of simplicity, in fact three copies of the data and thus three `RenderList`s are required for the shear/warp algorithm – each one grouped and sorted by one of the three coordinates.

Although the limitation to two voxel attributes with an overall of 16 bit for rendering is clearly a limitation with respect to flexibility and accuracy, the compact representation is perfectly suited for very fast rendering. Together with the ability to re-order voxels within a `RenderListEntry` the rendering process turns into a "streaming" of sequential chunks of voxels – an optimal scenario for caching and prefetching as implemented by recent processors. The problem of the low bit resolution of data attributes for rendering can be addressed by applying intelligent remapping when copying voxel attribute data into it's renderable form: instead of clipping low bits of an attribute, a logarithmic remapping can be performed, or a certain sub-range of attribute values can be mapped to the range of values available for rendering.



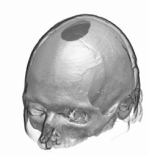
Java Peculiarities

Due to the specific way of memory management as employed by current Java virtual machines (VM), a special data handling and caching functionality is used by RTVR to support the visualization of huge data sets (dozens to hundreds of volumes), which are produced, for example, by numerical simulations [4]. The maximum amount of memory which is available to a VM has to be fixed at initialization time. As the garbage collection and object allocation mechanism sweeps through the entire address space of the VM, allocating more memory to the VM than physically available would lead to excessive paging and strong performance degradation. Instead of allocating sufficient memory to fit even the largest data sets, RTVR uses a separate memory and disk cache for space-demanding parts of it's data structures, i.e., the original volume data, the extracted voxel attributes, and the renderable voxel data. Data, which is currently not used for rendering, is placed into the memory cache and thereby potentially written to disk by a background thread. Requests for recently used data can usually be satisfied out of the memory cache, whereas reading less recently accessed data may require to fetch it from disk. The cache feature is used when large data sets are visualized locally, and is disabled when RTVR is used within a web-browser.

6.2.2 The RTVR Scene Graph

After extraction, the `RenderList`s and attribute data of volumetric objects are encapsulated into so-called `VolumeObject`s and added to a common scene graph for rendering (figure 6.4). A common task of all types of nodes within the scene graph is to deliver up-to-date `RenderList`s which represent the content of their subgraphs. In the following a short overview on the most important node types is given:

- `VolumeObject`: Holds the `RenderList` of a single object as well as information on all parameters which affect the appearance and visualization mappings for this object.
- `GroupNode`: The shear/warp renderer performs a back-to-front rendering of `RenderListEntry`s. In addition to providing a simple way of handling multiple objects, the main purpose of the `GroupNode` is to merge and sort the `RenderList`s of it's sub-graphs into a single list which is sorted by the current main viewing axis (z).
- Depending on the value of a selection parameter, the `SwitchNode` provides the `RenderList` of one of it's children. `SwitchNode`s allow to



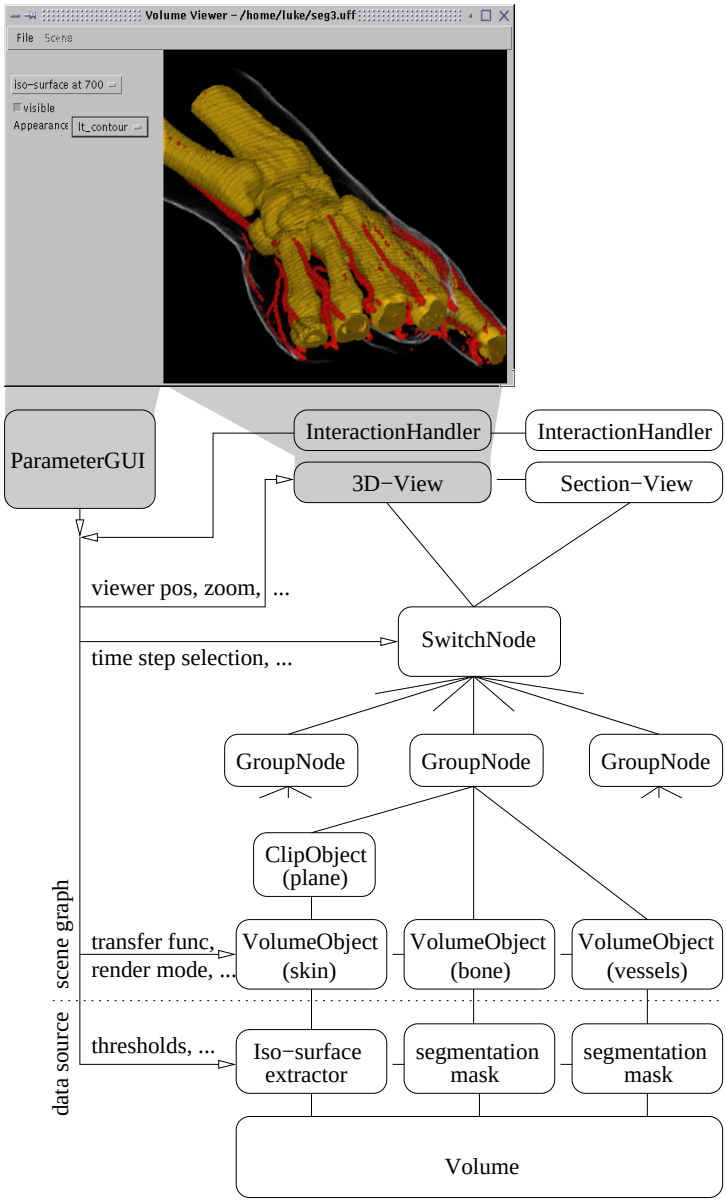
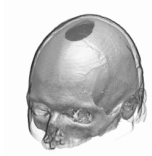


Figure 6.4: RTVR scene graph and user interaction handling



browse through multi-dimensional arrays of volumes, like time series, or parameter-dependent simulation results.

- `ClipNodes` filter and reorder the voxels of it's child nodes to implement clipping.

Each node is responsible for tracking changes of parameters which affect it's content and for performing appropriate actions according to changes. The actual update of renderable data to reflect parameter changes is carried out as late as possible, i.e., when a request for the affected voxel data is issued for rendering (lazy evaluation). Keeping just the currently visible data up-to-date improves the responsiveness of the visualization during interactive parameter changes significantly. Furthermore, keeping every voxel which is affected by a parameter change immediately up-to-date would spoil the efficiency of any caching scheme. An example for this would be a clipping operation applied to the representations of an object in 100 time steps of the data.

The volume information which is contained within a single scene graph can be displayed within multiple views. For multiple 3D views just the viewing parameters, like camera position may vary, remaining parameters are encoded into the `RenderList`, which is shared among all views. Additional 2D views on the data may display sections through the volume, histograms, etc.

6.2.3 User Interaction

The philosophy of data manipulation within RTVR is object-oriented. One of the objects within the currently displayed volume is selected to be the "active" object. The selection is done by clicking on the object within the rendered image, or by selecting it through the GUI. From practical experience, it appears, that direct interaction with an object within the 3D scene is the most efficient way of working with data. Especially expert users prefer this way of interacting, compared to using standard GUI components. The most important properties of the active object can be changed by pressing one of the mouse buttons and dragging. Transfer function shape (contrast), opacity, and color can be changed directly within the 3D view. Furthermore, camera position, light source position, and zoom factor can be set within the view. The mapping of mouse actions to parameter changes is performed by an `InteractionHandler` component, which can be adapted to meet the needs of specific applications (for example to implement stream line integration from the position of a mouse click for a flow-visualization application).

As a supplement to parameter manipulation within the rendered view, all parameters of the active object can be adjusted using GUI components which are

automatically generated by RTVR. This parameter panel allows an explicit selection of the active object and adjustment of it's parameters, and can (but does not have to) be used within any application which utilizes RTVR for visualization. A real-time screen capture of an interactive visualization session, which was performed by a viewer based on the RTVR library, can be downloaded from web page `http://www.vrvis.at/vis/research/rtvr/`.

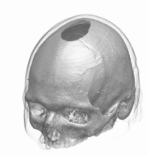
Java Peculiarities

for Java-based graphical user interfaces, basically two APIs are available: the AWT, available in it's present form since Java version 1.1, and the more sophisticated SWING 1.1, which is part of the Java runtime system since version 1.2. The front end (GUI and rendering output) of RTVR is provided using both, either AWT or SWING. As most web-browsers currently provide a 1.1 virtual machine only, an AWT implementation is provided for compatibility reasons, despite of all it's inconveniences and deficiencies. The rendering performance of the SWING implementation benefits, for example, from a faster image handling (`BufferedImage`) introduced in Java 1.2.

6.2.4 Rendering

To achieve interactive rendering rates even on standard desktop hardware, fast shear/warp-based parallel projection is used. Rendering to the base-plane is performed using a back-to-front compositing of voxels by the use of nearest-neighbor interpolation. In comparison to previously presented versions of this fast algorithm [44], RTVR includes an extended version, which provides more flexibility for mapping voxel attributes to color and opacity. Three look-up tables (typically 1×4 bit, 2×12 bit) are available at each `RenderListEntry` for implementing shading and transfer functions. A set of combination patterns for the voxel attributes and look-up tables is provided by RTVR (See figure 6.5) and selected by choosing an appropriate rendering mode for an object. This scheme of combining LUTs allows efficient processing while still enabling various ways of selectively applying visualization techniques to objects within the data. The `RenderListEntry` can also be extended to provide user-defined rendering functionality for it's voxels, which finally allows to implement any desired operation on voxel attributes and look-up tables.

Shading operations are performed using a look-up table based approach, with a 12-bit representation of the gradient vector as an index. Using this approach



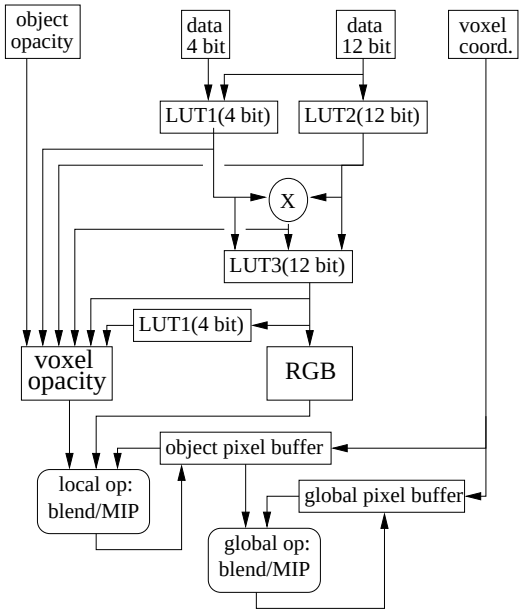
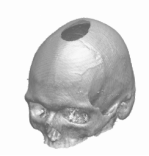
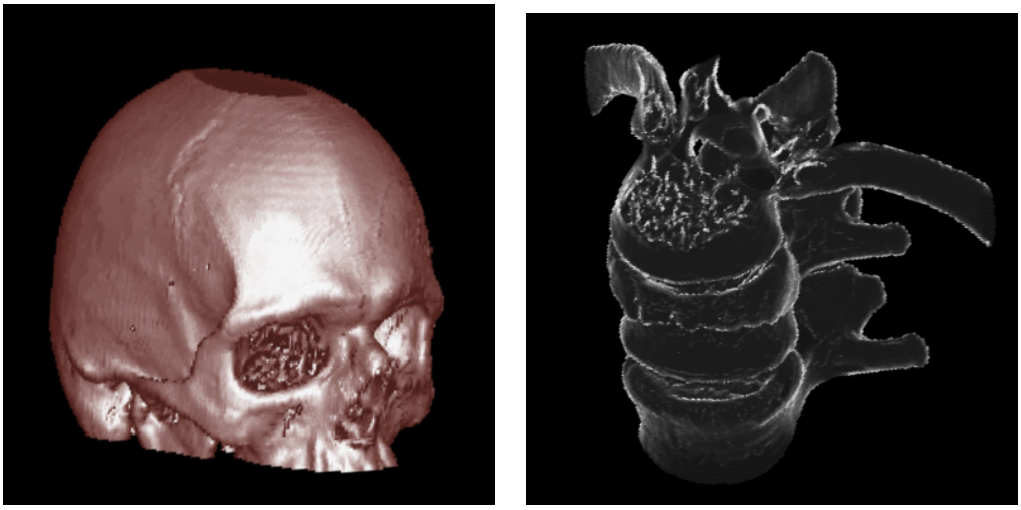


Figure 6.5: Voxel rendering within RTVR: LUT1 can be used either for color or opacity computations. For color computations, it can be indexed by the 4 bit attribute or by the upper bits of the 12 bit attribute. LUT2 is addressed by the 12 bit attribute. LUT3 is addressed using the output of LUT1 or LUT2 (which may be an identity mapping), or a combination of both.

various shading models can be implemented efficiently and with acceptable quality and applied even on a per-object basis. Two shading models are provided by RTVR: a Phong shading table (figure 6.6a) and a non-photorealistic shading table (figure 6.6b) which enhances the contour of an object [11]. The shading tables have to be re-computed after every change of viewer or light source position, which is not time critical due to their small size (4096 entries). For rendering, the shading table is placed into LUT2 (figure 6.5), and indexed by the 12 bit data channel which contains the gradient vector. The output of the look-up is not an RGB color but an intensity value, which is then used to access the color transfer function in LUT3. Although it would be possible to combine lighting and transfer function mapping within a single look-up into LUT2 (like described in section 4.4), splitting it into two stages allows to reuse the same shading table for objects with different color transfer functions.

The opacity of a pixel is influenced by several sources. An all-object opacity value is always included into the computation and can be used to tune the overall opacity of entire objects, independently of individual per-voxel opacity calculations. The individual opacity of each voxel can be derived from various combi-





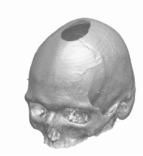
a) surface representation of a skull *b) contour enhanced vertebrae*

Figure 6.6: RTVR rendering examples

nations of data channel and look-up operations. In the following, a few sample color and opacity calculation setups will be discussed, which implement different volume rendering approaches.

- *display of (iso-)surfaces*: the surface voxels of an object have to be shaded and blended using the object opacity. A Phong shading table is put into LUT2, the resulting intensity value is used to access a color transfer function in LUT3. The transfer function is a ramp of object color values starting with maximum lightness and minimum saturation (white, or more generally, the color of the light source) and evolving towards maximum saturation and minimum lightness (object color, ambient light, see figure 6.6a). By just rendering a thin layer of voxels which form the surface, the object opacity can be used to influence the transparency of the surface in the same way as an alpha-value influences the appearance of a polygonal surface model.

- *Opacity weighted by gradient magnitude*: if voxels in areas with high gradient magnitude are rendered as being rather opaque, material transitions become visible as surfaces. The usual approach for volume rendering with this type of transfer function requires access to three values for each voxel: data value (for color and opacity from transfer function), gradient direction (for shading) and gradient magnitude (for opacity modulation). As only two attributes can be stored in the renderable data array of a `RenderListEntry`, two of the above three values have to be chosen.



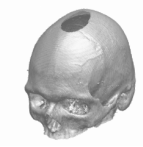
If the voxel data value is stored in the 12 bit channel, and a properly transformed gradient magnitude in the 4 bit channel, an unshaded volume can be rendered. LUT1 is indexed by gradient magnitude which yields voxel opacity, the data value indexes LUT3 which holds the color transfer function. As an alternative, the gradient direction can be stored into the 12 bit channel instead of data value, and used to compute shaded voxels using a shading table in LUT2 and a color transfer function in LUT3. The result is a transfer function with opacity solely dependent on gradient magnitude, and not on data value. To obtain a better control over the appearance of the rendered data, a threshold-based pre-segmentation can be applied to obtain independent control over the parameters for different voxel value ranges.

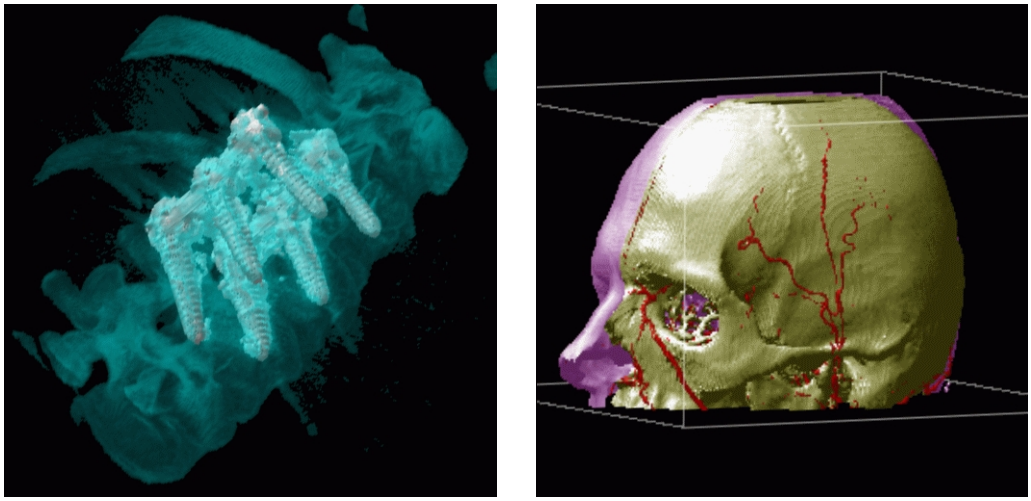
- *Bright object outlines*: LUT2 is loaded with a shading table which maps the angle between viewing direction and gradient direction to intensity. LUT3 contains a transfer function which is used to tune contrast and color for the specific object. Results of this technique can be seen in figure 6.6b. If the result of the LUT2 look-up is also used as voxel opacity, the object becomes almost entirely transparent – except for the contours which remain opaque (figure 6.1, clipped skin).

- *Colored contour outlines*: contours are colored differently from the remaining (Phong shaded) parts of the object (similar to the method presented by Ebert and Rheingans [11]). A special shading table which encodes Phong shading information into lower bits, and the "contourness" of a voxel into higher bits of the resulting value, is generated and placed in LUT2. An accordingly designed color transfer function is placed into LUT3.

In addition to color and opacity values, also the compositing mode can be individually defined for each object – for example, maximum intensity projection, or the usual opacity-weighted blending (DVR). The action to be performed for compositing in-between objects can be defined independently from the object-compositing modes (two-level volume rendering [22], figure 6.7a). Object-aware compositing requires the use of two separate pixel buffers, one for compositing within an object and one for compositing of the global image (figure 6.5).

Clipping of objects is handled in a way which differs from the usual approach. Instead of simply not displaying parts of objects which have been clipped, clipped data is rendered using a different set of attributes. Separate values can be set for clipped-object opacity, rendering mode (LUT configuration) and look-up table content. The compositing mode has to remain the same for clipped and non-clipped parts of an object. By setting clipped object opacity to zero, the usual effect of removing clipped data is obtained (figure 6.7b). By using, for example,





a) MIP for bone, DVR for screws *b) object-aware clipping of skin surface*

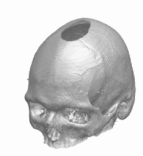
Figure 6.7: RTVR rendering examples

Phong shading for non-clipped voxels and a contour-only rendering for clipped parts, insight into an object can be given, while still providing a sketch of the most significant features of the clipped part as a context (figure 6.1).

To obtain high frame rates, despite of the flexibility of color and opacity calculation and compositing mode selection, optimized routines are implemented for frequently used rendering modes and compositing mode combinations. Scenes which require only MIP or DVR (within and in between objects), can be rendered with the usual approach and do not require two pixel buffers. If pure MIP is used, voxels can be sorted and grouped into `RenderListEntry`s by value instead of the z coordinate [42]. In this case, projecting sorted voxels from lowest valued to highest valued ones eliminates the need for maximum search. However, if MIP is combined with other compositing techniques within the scene, back-to-front rendering and thus sorting by the z coordinate is required also for objects composited by MIP, as they may interleave with other objects rendered with different techniques.

6.2.5 Data Optimization

Depending on the compositing method in use, on the content of look-up tables, and on the rendering mode which defines the usage of the tables, a significant percentage of an object's voxels may not contribute to a rendered image at all. For example, when MIP is used for compositing, black voxels (after the transfer



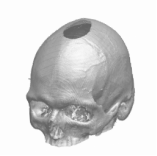
data set	*size*	*scene voxels*	*MIP*	*DVR*	*DVR/ mono*	*mixed MIP/DVR*
hand & vessels	$256^2 \times 124$	380k	80ms	135ms	80ms	170ms
head & vessels	$256^2 \times 158$	640k	100ms	185ms	110ms	290ms
attractor & basin	256^3	1M	130ms	250ms	140ms	333ms

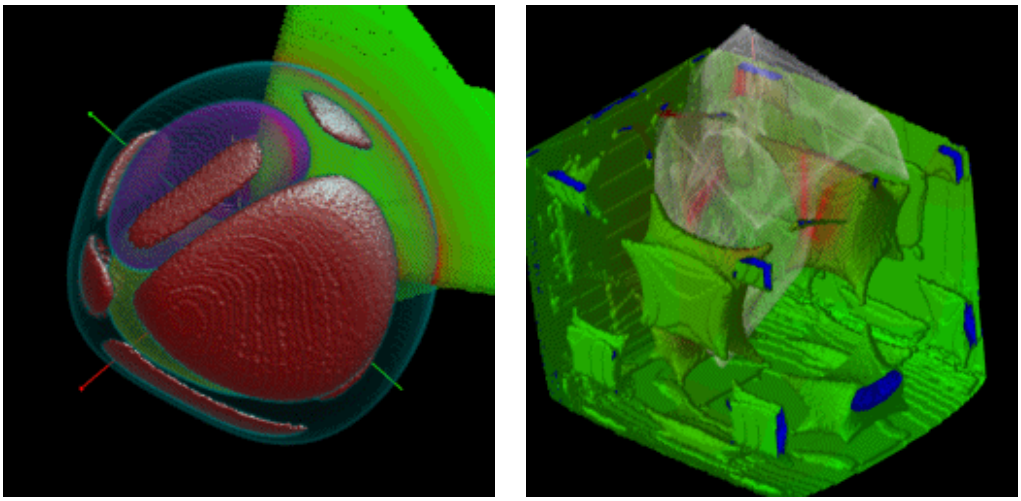
Table 6.1: Timings for various data sets and rendering modes

function mapping) do not contribute to the result. If DVR is used, (almost) totally transparent voxels provide no visible contribution. RTVR utilizes a background thread, which is activated whenever the application is idle, to identify those voxels and to reorder data in a way which allows to simply skip non-contributing parts from rendering. The currently visible `RenderList` is scanned periodically for `RenderListEntry`s which have not been optimized yet (or which have changed since the last optimization and thus may have to be optimized again). Depending on the rendering parameters of the `RenderListEntry` a classification of the voxels is performed. Voxels which have been identified as non-contributing are moved to the end of the sequence of clipped respective non-clipped voxels, two pointers are set to indicate the end of voxels which actually have to be rendered (see figure 6.3). For some rendering modes, like the rendering of contours only, opacity and color of voxels change for every new viewing position. In this case the content of the shading LUT is not considered as a criterion for optimization.

6.3 Performance

High responsiveness of a visualization system to user actions is a crucial factor for the effectivity of data exploration and analysis. The rendering times for the surface rendering [44], MIP [42] and two-level volume rendering approach [22] used by RTVR can be found in chapter 4. Thus, instead of broadly surveying the behavior of each method, a comparison of the measured times for rendering the same data set with RTVR using various methods is given in table 6.1. The measurements have been carried out on a PII/400MHz PC using the virtual machine of JDK1.3 from Sun and the AWT frontend of RTVR. The size of the rendered images is 512^2. The first row shows timings for the data set shown in figure 6.1. Skin, bones, and vessels are represented by their surface voxels. The rendering is carried out using MIP, DVR, a gray-scale DVR view, and a combination of DVR for the vessels and MIP for bones and skin. The second row displays timings for the head data shown in figure 6.7b, with bone, skin and vessels represented as surfaces. The data set in row 3 is similar to the one depicted in figure 6.8b. The basin





a) basins of attraction and a critical surface (green)

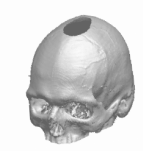
b) contact bifurcation – one out of a sequence of 40 volumes

Figure 6.8: RTVR rendering examples

is represented by it's surface voxels, the chaotic attractor is a highly complex structure, and is thus treated as a truly volumetric object.

The pure rendering time reflects the rendering performance for most interactions. These include interactive changes of the viewing parameters (viewer position and zoom), changes to content of look-up tables (moving light source, changing transfer function), and changes to the parameters and rendering modes of objects. Clipping operations require scanning and reordering of object voxels. During simple clipping of all objects at an axis aligned plane, the response time increases by approximately 40% compared to when changing viewer position. Time required for clipping at more complex objects depends on the complexity of the test which has to be performed for each voxel. Clipping of a complex scene at an oblique plane, for example, can be done with 1–2 frames per second. During browsing through large (time or parameter) series of volumes, voxel data may have to be fetched from disk cache, thus increasing the response time by the time required to read the data. Depending on the size of the scene, this may range from few milliseconds, to more than one second. The time for extraction of new objects from a volume depends on the complexity of the segmentation criteria and on the amount of voxels selected (gradient computation). The extraction of an iso-surface from a 256^3 volume for example requires approximately 1.5 seconds.

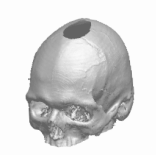
The choice of the virtual machine used to execute the application has severe impact on the performance. Among the tested runtime environments, fastest execution and rendering has been observed for the VMs (1.1.6++, 1.2, 1.3) from



Sun on Windows and (1.1.8, 1.2, 1.3) from IBM on Windows and Linux. Virtual machines provided by web-browsers are in general slower, probably due to additionally performed security checks. Worst results are obtained by the VM which is used by Netscape browsers (Version <= 4.7.4) on Linux – the results are more than ten times slower than the timings in table 6.1.

6.4 Discussion

Using an efficient data representation and a fast rendering method volumetric data can be displayed at an average desktop PC at frame rates which are comparable with those which are achieved by consumer 3D hardware. The software approach provides significantly more flexibility, like object-wise transfer functions, shading models and compositing methods (MIP, DVR, ...). Taking into account peculiarities of Java, all those capabilities can be made available to users with standard desktop hardware using different operating systems. Using a compact volume representation, the RTVR library can be also exploited to provide highly interactive and flexible presentations of visualization results over networks, like the Internet. For sample visualizations using RTVR see `http://www.vrvis.at/vis/research/rtvr/`.



Chapter 7

Sample Applications

The presented techniques have proven their use within several projects from the fields of medical data visualization and the visualization of dynamical systems (discrete 3D maps).

7.1 Interactive Surface Rendering for Java-Based Diagnostic Applications

The surface rendering and MIP approaches described in chapter 4 have been implemented into the Java-based diagnostic software called *J-Vision/Diag* by Tiani Medgraph [44]. As pure software approaches, they are not limited to specialized hardware and run on any platform which provides a Java 1.3 runtime environment.

7.1.1 System Overview

J-Vision/Diag is a Java-based viewing and diagnostic workstation for a Tiani Medgraph PACS (Picture Archiving and Communication System). The software is used by radiologists to view and diagnose images from different modalities like computer tomographs, magnetic resonance tomographs and ultrasonic devices. In addition to basic two-dimensional viewing features, e.g.; zooming, filtering, and windowing, the software also interprets stacks of images as volumes and displays arbitrary cuts through the volume (MPR = multi-planar reconstruction), maximum intensity projections (MIP) and surface renderings. Efficient work flow is achieved by a user interface which makes extensive use of so called hot-regions (clickable areas within the image display area which directly invoke frequently

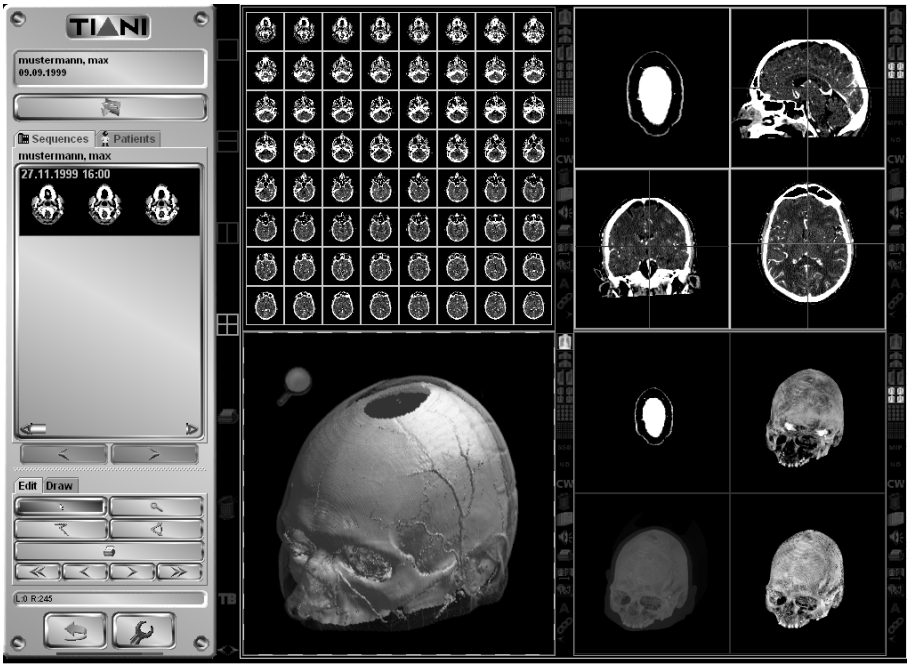
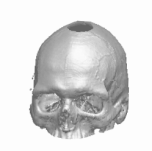


Figure 7.1: J-Vision user interface

used operations on the data). Figure 7.1 shows a screenshot of a typical work bench with simultaneous views of a data set as a set of slices (top left), using MPR (top right), the surface rendering algorithm (bottom left) and various variants of MIP (bottom right).

7.1.2 Interaction Techniques

After loading the images from the PACS system, a volume of interest for surface extraction should be defined within the data set. This can be intuitively achieved by zooming and panning into the desired section of the original images and by deselecting those slices, which should not be involved in the 3D display process. If no volume of interest is defined, the whole image sequence is used. The surface display is initialized by selecting the corresponding display mode. As no interpolation is used during projection to the intermediate image plane, the voxels should be equal-sized in all dimensions to avoid blurring due to stronger stretching in one direction during the warp step. If necessary, to ensure equally sized voxels and interactivity, the volume of interest is resampled to obtain a new volume with a size of at most 256^3 voxels. If a more accurate view of some part of the volume is required, another volume of interest can be constructed at this position and displayed within another view of the workbench.



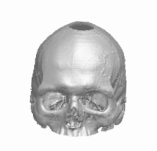
To improve the efficiency of working with the data, all frequently used interactions can be carried out by clicking into specific portions of the image. For supporting novice users, graphical tool tips appear if the mouse is moved over such a sensitive "hot region". Figure 4.18 shows, for example, the looking-glass symbol for the zooming region. Clicking on it and dragging the mouse zooms into and out of the data set. To avoid to distract expert users, the display of the hot-region symbols can also be disabled. A large region at the center of the image can be clicked to rotate the object, regions at the top of the image allow to specify the threshold and opacity for the surfaces. As the extraction of a new surface is the only non-interactive operation on the scene (see section 4.3.3 for details) the threshold is displayed numerically until the interaction is finished, the corresponding surface appears shortly after finishing the interaction. As mostly CT-data, which has a well-defined tissue-to-value correspondence, is visualized using surfaces, providing feedback only after the interaction has been finished poses no big problem. In addition, the surface threshold can be exactly specified using a numerical entry field. Currently, the parameters for two simultaneous surfaces can be defined using hot-regions within the image, which is sufficient for most applications.

Clipping planes can be defined and moved using six hot-regions at the bottom of the image. Two clipping planes perpendicular to each axis are defined, one for clipping data above, and one for clipping data below a specific coordinate value.

In addition, the windowing mechanism, which is familiar to radiologists, can be used to enhance contrast and emphasize features of the original volume data which is displayed on the clipping planes. x/y-movements of the mouse are used to define and modify the window (c, w) which maps all data values below $c - w/2$ to black, all values above $c + w/2$ to white, and the values between $c - w/2$ and $c + w/2$ to a uniform ramp of gray values.

7.2 RTVR-Based Volume Viewer

The RTVR library has been used as a basis for a volume viewer, which can be used for fast volume viewing and exploration. Visualization settings created with the viewer (extracted objects and visualization parameters) can be stored using a compact representation (typically just a few hundred kilobytes [40], chapter 5) for later interactive viewing or for publication on the Internet. The viewer supports all features of RTVR, providing full control over the contents of look-up tables and their configuration, separate rendering and shading modes for all objects, as well as the per-object application of arbitrary clipping planes, and the possibility to capture visualization sessions for the creation of animation sequences.



An applet version of the viewer which provides the above described functionality, except for the extraction and creation of new objects, can be used to view previously stored data within web pages. Some pages which use the applet for the presentation of volume data can be found at `http://www.vrvis.at/vis/research/compression/` and `http://www.vrvis.at/vis/research/rtvr/`.

7.3 Visualization of 3D Maps

A second application which makes use of the capabilities of RTVR is a visualization and analysis system for 3D dynamical systems (discrete non-invertible maps [38]) [4]. The application is used to analyze and visualize structures and events (bifurcations) within the phase-space of the systems.

Objects of interest are attractors (often complex or even chaotic), their basins of attraction (i.e., the set of all system states which are attracted by them in the limit) [20] and critical surfaces (surfaces which separate regions of phase space with different properties). Up to now, the most commonly used visualization technique for investigating this type of structures within phase space are simple planar cross-sections. Although they well depict the boundaries of a basin of attraction, for example, it is difficult to convey the 3D shape of the basin and even more the shape of a complex attractor by just viewing a set of sections (figure 7.2). The ability to view the objects in 3D with the option to interactively change viewing parameters and to manipulate objects, for example, to obtain cross sections, is of great help during investigation of the data. As attractors are contained within their basins of attraction, and frequently also basins are nested within each other, efficient tools to deal with occlusion are required, like the ability of object-wise tuning of opacity, render mode and lighting model which are provided by RTVR. The feature of mixing MIP with other compositing methods has proven to be especially useful for visualizing chaotic attractors. Their complex internal structure is well captured using MIP while producing little occlusion. At the same time, the attractor's basin of attraction can be rendered as a shaded surface.

7.3.1 Data Acquisition

For visualization, a voxelized representation of phase-space and the objects of interest have to be obtained. As this requires extensive iteration of the difference equations which define the map, simulation and voxelization are performed off-line. The simulation process creates a set of volumes (256^3 voxels each), which

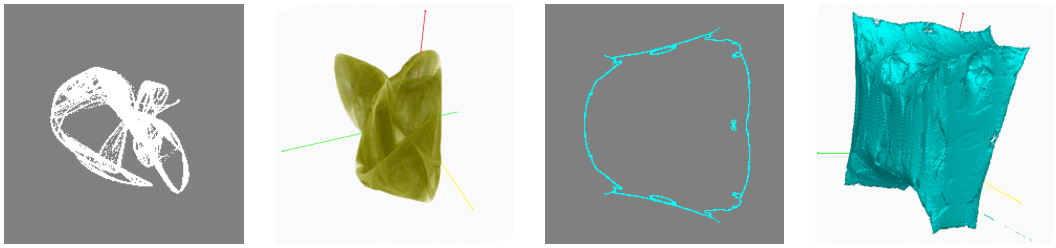


Figure 7.2: Attractor, basin, and cross-sections through them: Deriving the 3D shape of the object just from sections is difficult.

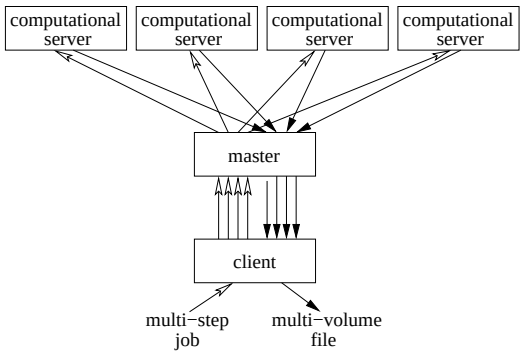
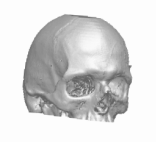


Figure 7.3: Computation of a parameter sequence simulation using a computational cluster

contain basin-labeling information (which basin does the voxel belong to?), visit-count information for attractor voxels (how frequently is the voxel visited during iteration?), distance volumes which store the distance to the closest attractor voxel, and further volumes storing information required for the construction of critical surfaces. A post-processing step extracts objects' voxels (attractor voxels, basin-boundary voxels and critical-surface voxels) for storage and later viewing. The time to perform a single simulation is 3–10 minutes on a PII/500 PC with sufficient memory. Frequently, the creation of whole parameter sequences of data sets is required, varying the value of some dynamical system parameters for each step. To efficiently carry out simulation in such cases, a cluster of distributed computational servers is used (figure 7.3). As the simulation of different parameter steps can be performed independently of each other, the computation of each step can be assigned to any free computational server. Scheduling of the jobs is performed by a master server, which collects the results of the simulation and returns them to the requesting client. Even though extracted voxel data is stored instead of entire volumes, simulation sequences with tens or hundreds of parameter steps can easily produce 100–300 MB of data.



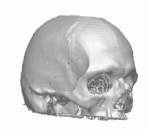
7.3.2 Study of Bifurcations

Bifurcations are events within parameter space of the dynamical system, where the behavior of the system changes. For example, if a parameter of the system is continuously changed from value a to b, the topology of the basins of attraction may change at some point (figure 7.6). Bifurcations are often caused by contacts between structures as some parameter of the dynamical system is changed. Frequently, the process can not be investigated analytically, thus numerical simulation is required. In 3D the detection of where and how such contacts happen is extremely difficult, if only 2D sections are available as a visualization method.

For the analysis of bifurcations, sequences of up to hundreds of volumes are computed for different values of the bifurcation parameter. For investigation, the data produced by the simulation is loaded into the viewer (the disk-cache implemented by RTVR is extremely useful for large sequences of volumes). To ease the detection of contacts between objects, distance information can be mapped to voxel opacity or color, as shown in the sequence depicted in figure 7.4, which illustrates a contact bifurcation between an attractor and the boundary of it's basin.

Figures 7.5, 7.6, and 7.7 show the visualization of another type of contact bifurcation – the contact of a basin with a specific part of a critical surface ($\widehat{CS}$), and the resulting change in topology, namely the creation of disjoint basin parts (figure 7.6). Figure 7.5, top left shows the basins of four attractors (three inner basins and one outer basin) together with three sheets of critical surface structures $\widehat{CS}_{-1}$. If $\widehat{CS}_{-1}$ is iterated once, by the application of the difference equations which define the map, $\widehat{CS}$ is obtained, which is the folded structure depicted in 7.5 top right. The sheets of $\widehat{CS}$ subdivide phase space into regions Z_n with different properties (a different number of pre-images for all the inner points). The crossing of a basin boundary into one of this regions (a region with more pre-images) may cause the appearance of disjoint parts of the basin.

Identifying the contact from 2D slices only is an extremely difficult task, a pure 3D visualization is also not well-suited for this purpose, as the critical surfaces are folded in a complex way in the area of interest (figure 7.5, sections). The capabilities of RTVR allow to efficiently create a meaningful visualization which depicts the location of the contact (figure 7.7), by depicting only objects which are involved in the contact: one basin, one sheet of the $\widehat{CS}$, and the boundaries of one of the Z regions. Additionally, the distance of voxels of the basin boundary to the $\widehat{CS}$ is mapped to color, and a clipping plane is applied to reveal insight into the region of crossing.



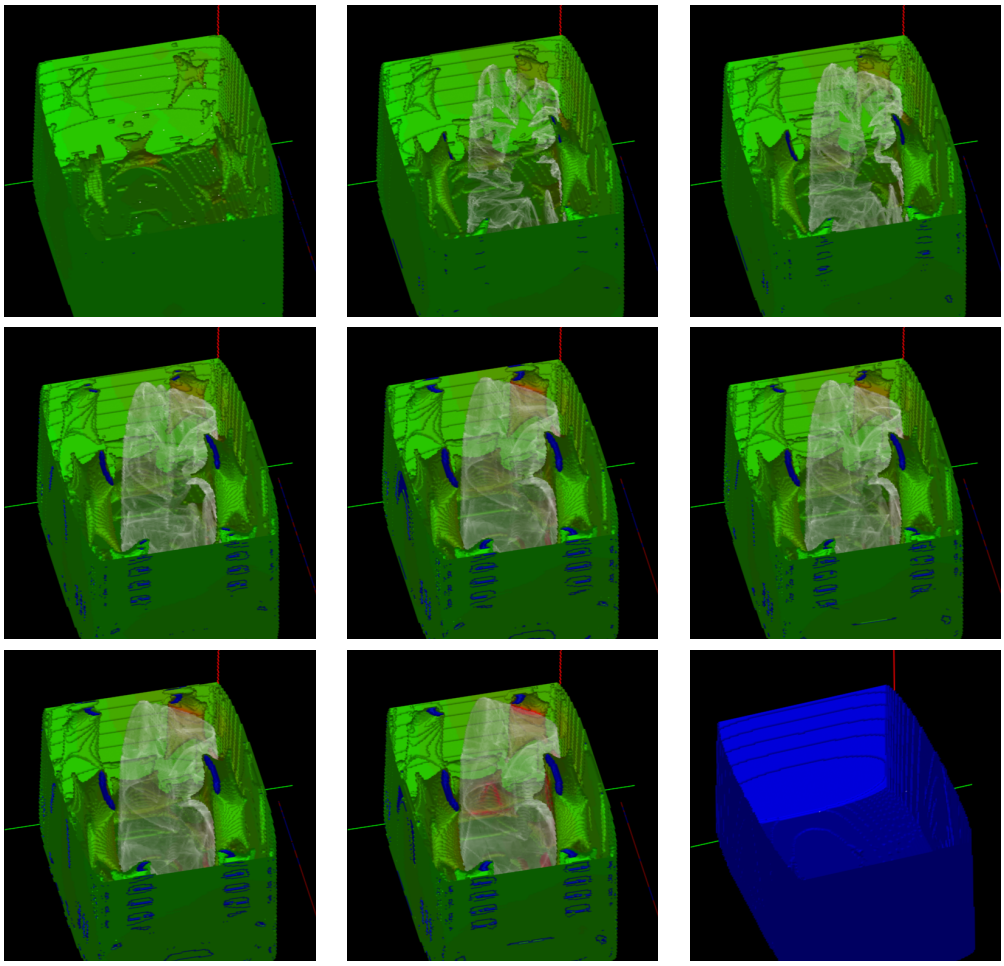
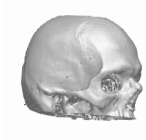


Figure 7.4: Visualization of a contact bifurcation: as the bifurcation parameter is changed, the attractor approaches the boundary of it's basin (green). After the contact, which happens at the shared boundary to the blue basin, the green basin merges with the blue basin (all the states previously converging to the chaotic attractor now converge to the point attractor of the blue basin). The chaotic attractor disappears, only the point attractor of the blue basin remains. The distance between attractor and boundary is color-coded on the boundary. Red parts indicate proximity of the attractor, and thus regions which probably are responsible for the contact. The blue spots on the outer boundary of the green basin are aliasing artefacts due to the discretization of phase-space. The blue basin is extremely close (below the resolution of the sampled volume) to the outer boundary of the green basin.



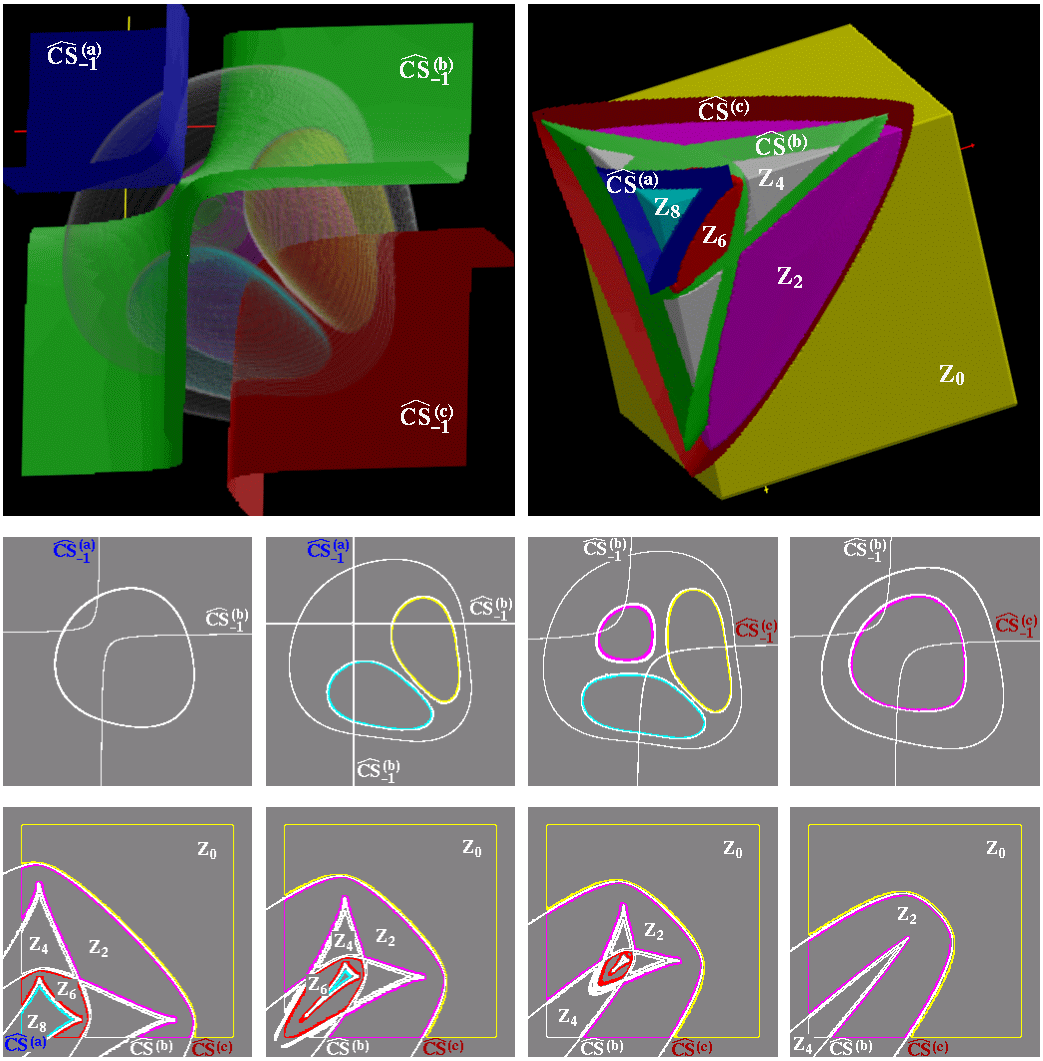
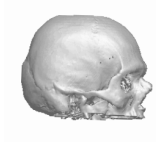


Figure 7.5: $\widehat{CS}_{-1}$ (top left image, three sheets $\widehat{CS}_{-1}^{(a)}$, $\widehat{CS}_{-1}^{(b)}$, and $\widehat{CS}_{-1}^{(c)}$, depicted together with basins of attraction) and $\widehat{CS}$ (top right image, three sheets $\widehat{CS}^{(a)}$, $CS^{(b)}$, and $\widehat{CS}^{(c)}$, depicted together with separated zones Z_0, Z_2, Z_4, Z_6, and Z_8) visualized in 3D – the images below the top row show planar intersections at different (increasing) depth values for both 3D illustrations.



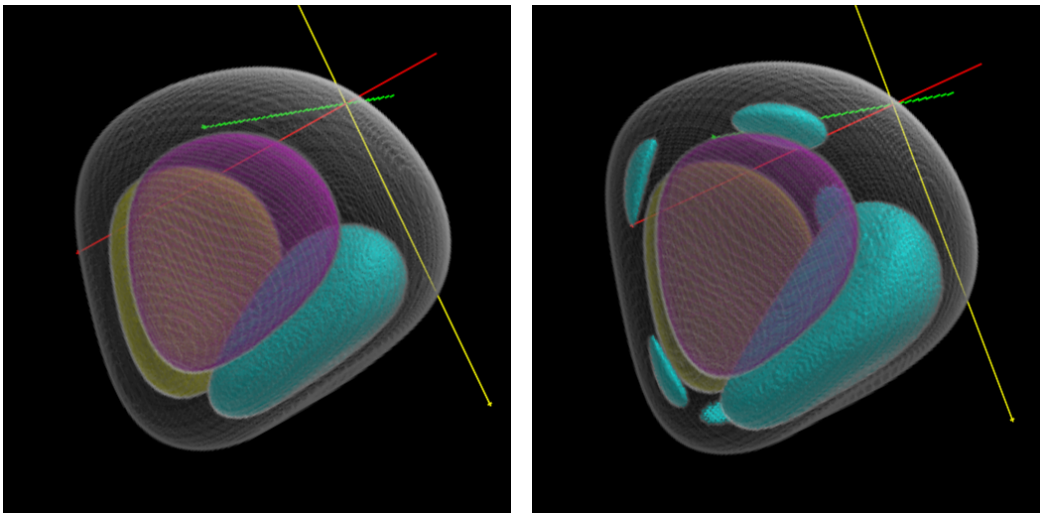


Figure 7.6: Basins of attraction (four attractors except ∞) visualized before and after the contact bifurcation – the creation of disjoint parts of one basin (depicted in cyan) is clearly visible.

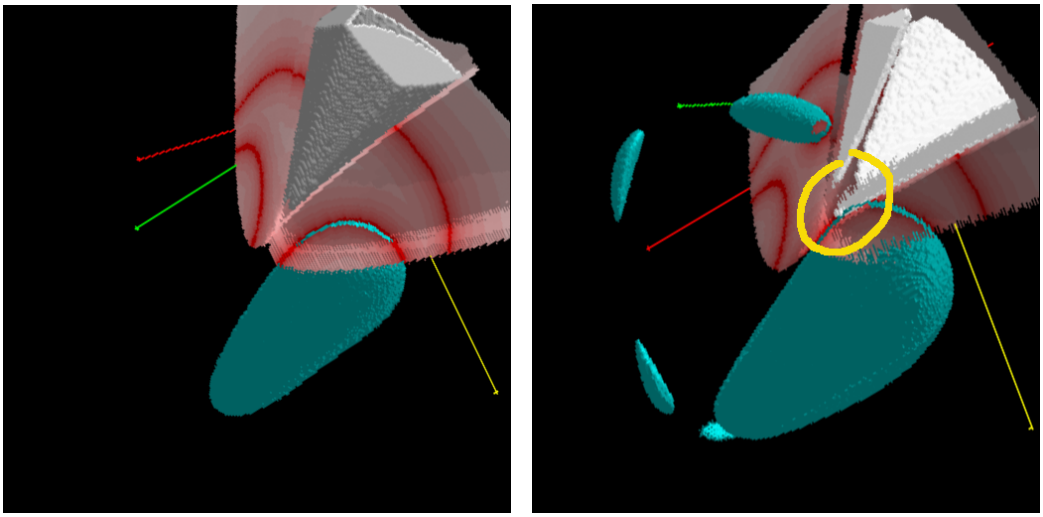
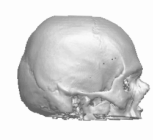


Figure 7.7: The basin of attraction which causes the contact bifurcation is visualized in 3D together with responsible parts of the $\widehat{CS}$.



Chapter 8

Summary

Interactivity is crucial for efficient exploration and analysis of volume data. Complex data sets require careful and frequent tuning of visualization parameters to obtain meaningful visualization results. The specification of a proper transfer function [23, 27, 29, 34], i.e., the assignment of optical properties to data values within the volume is a complex task which benefits greatly from immediate visual feedback by interactive rendering of the volume. The main obstacle for interactive volume rendering is simply the amount of data to be processed for generating an image from a volumetric data set. Typical volume sizes in medicine range from 256^3 voxels for MR data to $512^2 \times 2000$ voxels for data acquired with recent multi-detector CT scanners. For a straight-forward approach, this would mean shading and compositing 16–500 million voxels for each single image – a tough task even for multi processor hardware. Simple straight forward implementations of volume rendering are only competitive in terms of performance if directly implemented in hardware – like the VolumePro (vp500) volume rendering board from Real Time Visualization [48].

Within this work, a novel, purely software-based solution to interactive rendering of volumetric data is presented, which is able to deliver interactive frame rates even on low-end hardware. The approach is also well-suited for use in networked environments due to a compact data representation. Several distinguishing features make the presented method a fast and flexible solution to interactive, software-based volume rendering for low-end hardware:

- **preprocessing**: during a preprocessing step, voxels which potentially contribute to a visualization result are identified.
- **voxel enumeration**: possibly contributing voxels are extracted from the volume and stored in a derived data structure, which is basically a list of individual voxels.

- **compact representation**: the extracted voxels are well-suited for efficient compression and can be transmitted over a network for visualization at a remote computer at very low cost.
- **voxel ordering**: the extracted voxels can be ordered in a way which is optimized for rendering using specific compositing modes and visualization parameter settings.
- **fast rendering**: A fast shear/warp-based rendering [28] is used to project the extracted voxels.
- **object awareness**: if segmentation information is available, extracted voxels can be assigned to individual objects. Visualization parameters can be defined on a per-object basis, allowing to individually adjust opacity and color transfer functions, shading models, and even compositing modes, without much impact on rendering performance.

The voxel extraction approach can be seen as a hybrid approach between direct volume rendering, which directly operates on the original volume data, and approaches like marching cubes [33], which derive a polygonal representation of objects within the volume for rendering. On one hand, only a secondary data representation which represents the volume is used for rendering – the list of potentially contributing voxels. On the other hand, the voxel data within this data structure is just a space-efficient storage representation for a sparsely populated volume.

8.1 Preprocessing

Given a set of visualization parameters, the goal of the preprocessing step is to classify voxels of the volume into voxels which possibly contribute to an image and voxels which do not contribute to an image. The classification criteria depend on the chosen opacity transfer function, the desired compositing method, and the degree of freedom which should be provided for further manipulation of the transfer function. Generally speaking, the more voxels are classified as irrelevant, the fewer data has to be processed during projection, and the faster the rendering becomes.

Different rendering methods and data characteristics require different extraction strategies:

- **Object surfaces** (iso-surfaces): To obtain a representation for the iso-surface, the volume is scanned for transitions between voxels within and

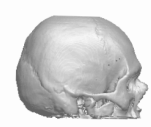
outside an object. If a voxel is located within the object (data value greater or equal than the threshold for an iso-surface) and if it has at least one 26-connected neighbor outside the object, it belongs to the object's surface and is considered to be relevant. The relevant voxels identified during the scan correspond to a 6-connected surface within the volume at the specified threshold value or object boundary. The 6-connectedness of the surface voxels is required to ensure that no holes appear during (shear/warp-based) rendering due to displacement of voxels within successive slices. Just a few percent of all voxels within a volume usually belong to a meaningful object surface.

- **Maximum Intensity Projection (MIP)**: Most approaches to optimize the performance of MIP rendering aim at excluding voxels from the traversal and rendering process, which contain less-important information like low-valued background noise. In fact, in addition to this low-importance data, there is usually a remarkable amount of voxels which never contribute to a MIP image. A voxel V does never contribute to a MIP (and can be discarded from rendering) if all possible viewing rays through the voxel hit another voxel W with $d(W) \geq d(V)$, either before or after passing through V, where $d(V)$ is the data value at voxel V. This fact can be exploited when original voxel values are used for rendering using nearest neighbor interpolation, as it is done within the presented approach. Several elimination strategies can be applied to identify such voxels, ranging from examining just the direct neighbors of each voxel, to tracking of voxel influences into distant parts of the volume. By also considering viewing direction into the elimination process, up to 75% of all voxels can be discarded.

- **Gradient Magnitude Modulation**: If a transfer function is used which modulates voxel opacity according to gradient magnitude, gradient magnitude can be used as an elimination criterion.

8.2 Data Representation

All voxels which have been classified as relevant are extracted from the volume and stored in a secondary data structure. The structure is simply a list, or array, containing attribute information for the extracted voxels. Each entry corresponds to one extracted voxel, and holds the voxel's position, data value, gradient information and/or other attributes. If different objects are distinguished within the volume, separate lists are created for the voxels of each object. The voxels are sorted by one of the attributes (for example, data value, or one of the coordinates), voxels



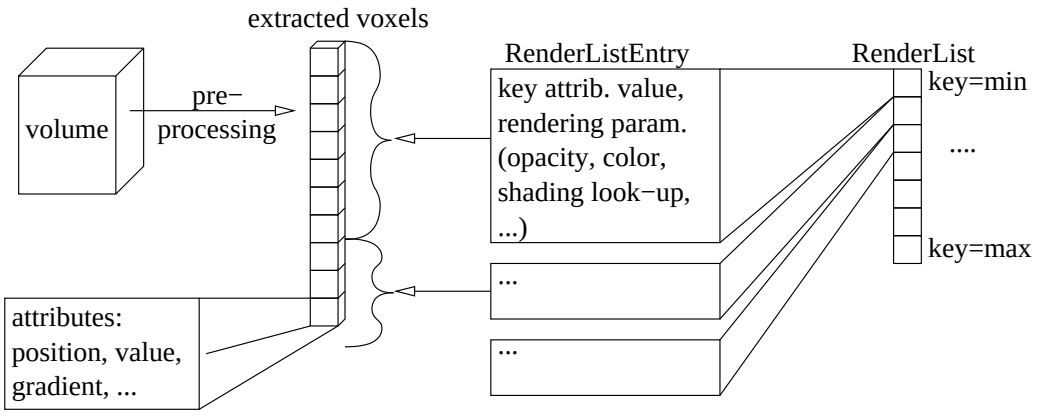


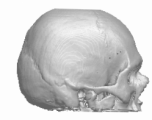
Figure 8.1: Data structure for efficient volume rendering. During preprocessing, potentially relevant voxels are extracted from the volume and stored into a list (an array). For each voxel, the position, and a set of attributes are stored. The voxels are ordered by one of the attributes, the "key" attribute. This might be, for example, data value, or one of the coordinates. Groups of voxels with an identical key value are joint into `RenderListEntry`s. All `RenderListEntry`s sorted by the key value form a `RenderList`.

within the list can be grouped into blocks, so-called `RenderListEntry`s, with the same value of a "key" attribute (figure 8.1). To save memory, the key attribute has to be stored just once for all voxels of the group. All `RenderListEntry`s, sorted by the value of the key attribute form a `RenderList`.

For rendering, only information contained within the list is used. Different compositing methods require a specific ordering of the voxels:

- For **DVR**, a consistent front-to-back or back-to-front order is required. Voxels are sorted according to depth (or the axis most parallel to the viewing direction, for shear/warp-based rendering).

- For **MIP**, the spatial ordering of voxels is not relevant. By sorting voxels by value several important advantages are gained. The voxels can be splatted in the order of ascending data values as the array is traversed for rendering. Therefore *comparing* the value of the actual voxel with the screen content is *not necessary* at all. Instead of directly mapping data values to a linear ramp of gray values for viewing ($d_{min} \Rightarrow g_{min}$, $d_{max} \Rightarrow g_{max}$), medical data sets are usually viewed using a window to improve contrast and to focus on certain details. The window is defined by a center value c and a width w which maps all data values below $c - w/2$ to black, all data values above $c + w/2$ to white, and the data in between to a gray ramp. As realistic



window functions (such as used by doctors) map significant portions of the data to black, the corresponding parts of the voxel array can be skipped during rendering at almost zero cost.

- For **non-photorealistic volume rendering** which enhances object contours, opacity of a voxel may depend on the direction of the gradient vector. By grouping voxels according to gradient direction, entire groups of transparent voxels can be skipped.

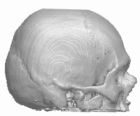
8.3 Rendering

The fastest software-based rendering method is shear/warp projection. Voxels from the `RenderList` can be efficiently projected onto the base plane using look-up tables for determining projected positions. To maximize performance, nearest-neighbor interpolation is used during projection of voxels. The calculation of the contribution depends on the compositing mode used. If optical properties of voxels are stored at the corresponding `RenderListEntry`s (figure 8.1), it is easily possible to assign different parameters to each group of voxels. In practice, all `RenderListEntry`s which store voxels of the same object will have identical parameters, like color and opacity transfer functions.

Projection is performed, by merging the `RenderList`s of all distinct objects, and sequentially processing all `RenderListEntry`s of the joint `RenderList` which represents the volume. Within each `RenderListEntry` all voxels are also projected sequentially. The strictly sequential order of voxel projection which does not depend on the viewing direction leads to an excellent cache-efficiency of this approach.

If rendering parameters are changed, some of the extracted voxels may become irrelevant for the visualization. An effective way to skip them during rendering without much effort, is to reorder voxels belonging to each `RenderListEntry` in a way, that irrelevant voxels are moved to the end of the group. This allows to render the block of relevant voxels at the beginning of the group, and to efficiently skip the remaining, irrelevant ones. Clipping or removal of parts of the volume can be done in a similar way. If voxels which are clipped and become invisible are moved to the end of each `RenderListEntry`'s voxels, they can be skipped in the same way as irrelevant voxels.

For fast evaluation of lighting models, an approach based on look-up tables can be used [19]. For this purpose, the gradient vector is quantized to 12–16 bit, and stored with the voxels as an attribute. The compact representation of the gradient vector is used as an index into a look-up table which stores precomputed shading



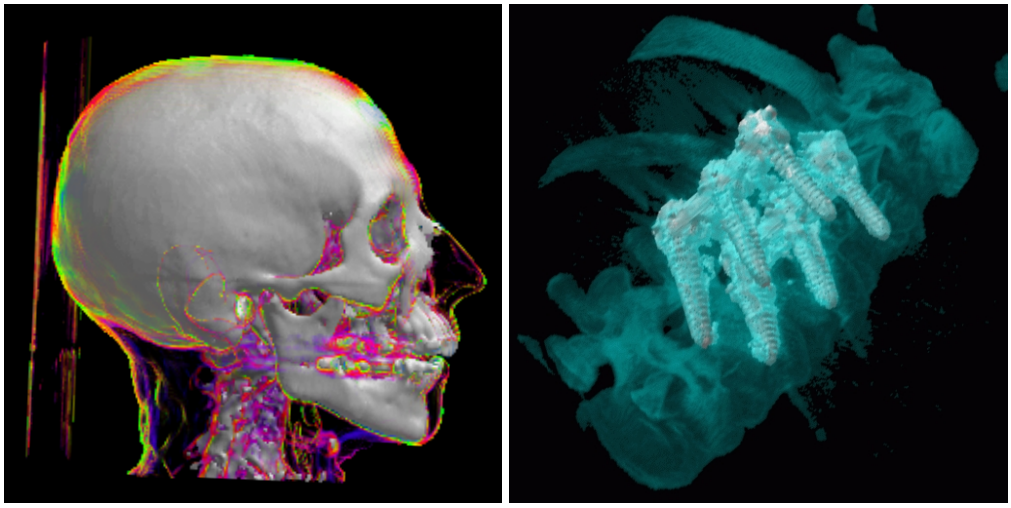
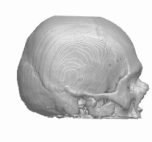


Figure 8.2: a) mixing different lighting models: Phong shading for the bone, non-photorealistic contour enhancement for the skin. b) two-level volume rendering: MIP for the bones, surface rendering of the screws

information. The content of the look-up table has to be recomputed whenever a factor which influences lighting is modified, for example, a light source is moved, or the volume is rotated. Due to the small size of the table (4096–65536 entries, depending on the gradient quantization), various shading models can be applied on a per-object basis at interactive frame rates. Phong shading [49] and non-photorealistic shading models [10, 11] can be easily mixed within a single image (figure 8.2a).

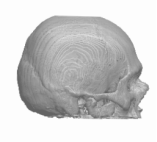
Using the `RenderList`-based volume representation, not only shading models can be easily assigned on a per-object basis, also the object-wise assignment of compositing modes can be implemented efficiently. This allows to render each object using the compositing mode which fits its structure best (figure 8.2b). For implementing this feature (two-level volume rendering [22]) based on the `RenderList` structure, two sets of buffers are used for the base plane image. An object buffer is used for performing rendering within an object, while a global buffer is used to perform inter-object rendering. In addition to intermediate pixel values, each pixel of the object buffer additionally stores a unique ID for the currently front-most object. If a voxel is projected onto the intermediate image, it's ID is compared with the stored ID in the object buffer. If both IDs match, the value in the object buffer is updated using an operation which corresponds to the local rendering mode of the object (maximum selection or blending of the voxel value with the buffer content). If the ID of the voxel differs from the ID of the pixel in the buffer, the viewing ray though this pixel must have entered a new object.



The content of the object buffer pixel has to be combined with the corresponding global buffer pixel using an operation which depends on the global rendering strategy (MIP or DVR). Afterwards the object buffer pixel is initialized according to the voxel of the new object and the new local rendering mode. After all voxels have been projected, the contribution of the front-most segment at each pixel has to be included by performing an additional scan of the buffers and merging the segment values left in the local buffer into the global buffer.

8.4 Space-Efficient Representation

The `RenderList` structure for storing extracted voxels is well-suited for further compression. Exploiting spatial coherence (especially in the case of extracted surface voxels), voxel positions, gradient information, and data values can be compressed to a few bit per voxel. This compact representation can be used for efficient transmission of extracted data for visualization over low-bandwidth networks, like the Internet.



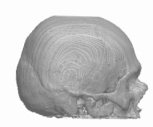
Conclusions

The techniques presented in this work allow to visualize volumetric data at highly interactive frame rates even on standard desktop hardware. The key to high rendering performance is based on the observation, that in many applications of volume visualization the rendering of a small percentage of the volume's data is sufficient to depict the desired structures. The approach presented in this work –scanning for voxels which contribute to the visualization considering the chosen visualization parameters, and the extraction of relevant voxels into a secondary data structure – allows efficient skipping of non-contributing parts of the volume during rendering at zero cost. Depending on the chosen rendering methods and visualization scenario, this allows to get rid of up to 99% of the volume data without sacrificing spatial accuracy of the visualized objects. Although it is possible to perform high-quality, cell-based rendering using a similar data structure containing volume cells, the data structure is primarily designed to perform really fast rendering of single voxels, for example using a fast shear/warp-based projection. Clearly, the approach is a trade off between fast rendering and high quality of the images, as are also other interactive volume visualization techniques.

The explicit association of extracted voxels to segmented objects within the volume allows an extremely efficient handling of rendering and visualization parameters on a per-object basis. Without impact on rendering performance, color and opacity transfer functions, shading modes, and even compositing methods can be changed individually for each object. The flexibility of the approach in terms of the manipulation of visualization parameters, makes it well suited for data exploration and analysis.

The compact representation of the volume by the extracted voxels is well-suited for effective compression. It can be utilized to provide volume visualization capabilities over slow networks, like the Internet.

The combination of flexibility in terms of visualization parameters and high interactivity even on low-end hardware, makes the presented techniques unique.



Acknowledgements

This work has been funded by:

- the FWF (`http://www.fwf.ac.at/`) as part of project P-12811/INF ("BandViz", `http://bandviz.cg.tuwien.ac.at/`)
- the V^{is}M^{ed} project (`http://www.vismed.at`) which is supported by *Tiani Medgraph*, Brunn/Geb., Austria, `http://www.tiani.com/`, and the FFF (`http://www.fff.co.at/`), Austria.
- the Kplus research program by the Austrian government as part of the basic research on visualization (`http://www.vrvis.at/vis/`) in the VRVis Research Center, Vienna, Austria.

The data sets depicted in figures 4.8a, 4.11, and 4.17 are available from the United Medical and Dental Schools (UMDS) Image Processing Group in London `http://www-ipg.umds.ac.uk/archive/heads.html`.

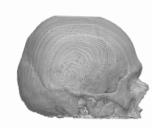
The head depicted in figure 4.24 is obtained from the Chapel Hill Volume Rendering Test Dataset which is provided courtesy of North Carolina Memorial Hospital.

The remaining medical data sets are courtesy of Tiani Medgraph, Brunn/Geb., Austria, `http://www.tiani.com/` and their clinical partners.

The author wants to thank Meister Eduard Gröller, Helwig Löffelmann (66%), Helwig Hauser (34%) (he (or they?) happened to change name a year ago). The people at the Institute of Computer Graphics (and Algorithms) and the VRVis, i.e., Balázs, Jiri, Anna, Andreas, Helmut, Markus, Thomas, Rainer (for discussions and beer), and the team at Tiani Medgraph. Gian-Italo Bischi, for the highly interesting cooperation on dynamical systems.

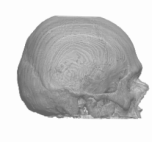
Karolina, for tolerating long nights of paper-writing. My parents, for buying me a Commodore 64 a long time ago.

If you feel that I forgot to mention you, drop me a message. I will mention you in my next life.



Further Resources

- Interactive surface rendering:
  `http://www.cg.tuwien.ac.at/research/vis/vismed/ssd/`
- Real-Time Maximum Intensity Projection (applet):
  `http://www.cg.tuwien.ac.at/research/vis/vismed/RT-MIP/`
- Cell-based high-quality MIP (applet):
  `http://www.cg.tuwien.ac.at/research/vis/vismed/CMIP/`
- Home of the BandViz project:
  `http://www.cg.tuwien.ac.at/research/vis/bandviz/`
- Home of the VisMed project:
  `http://www.cg.tuwien.ac.at/research/vis/vismed/`
- Visualization of Non-Inverible 3D Maps (basins, attractors, bifurcations):
  `http://bandviz.cg.tuwien.ac.at/basinviz/`
- Compressed boundary representation, sample visualizations (applet):
  `http://chicken.cg.tuwien.ac.at/basinviz/compression/`
- RTVR and related work:
  `http://www.vrvis.at/vis/research/rtvr/`
- Non-photorealistic volume rendering:
  `http://www.vrvis.at/vis/research/npvr/`
- Institute of Computer Graphics and Algorithms,
  Vienna University of Technology: `http://www.cg.tuwien.ac.at/`
- VRVis Research Center: `http://www.vrvis.at/`
- Tiani Medgraph: `http://www.tiani.com/`
- Gian-Italo Bischi: `http://www.econ.uniurb.it/bischi/bischiweb.htm`



Bibliography

[1] Ch. S. Ang, D. C. Martin, and M. D. Doyle. Integrated control of distributed volume visualization through the world-wide-web. In *Proceedings IEEE Visualization '94*, pages 13–20, 1994.

[2] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified volume visualization system. In *Proceedings IEEE Visualization '94*, pages 31–39, 1994.

[3] C.L. Bajaj, V. Pascucci, and D.R. Schikore. The contour system. In *Proceedings IEEE Visualization '97*, pages 167–174, 1997.

[4] G.-I. Bischi, L. Mroz, and H. Hauser. Studying basin bifurcations in nonlinear triopoly games by using 3D visualization. *Accepted for publication in Journal of Nonlinear Analysis*, 2001.

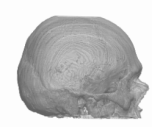
[5] W. Cai and G. Sakas. Maximum intensity projection using splatting in sheared object space. In *Proceedings EUROGRAPHICS '98*, pages C113–C124, 1998.

[6] G. Celniker, I. Chakravarty, and J. Moorman. Visualization and modeling of geophysical data. In *Proceedings IEEE Visualization '93*, pages 362–365, 1993.

[7] T. Chiueh, C. Yang, T. He, H. Pfister, and A. Kaufman. Integrated volume compression and visualization. In *Proceedings IEEE Visualization '97*, pages 329–336, 1997.

[8] D. Cohen and Z. Sheffer. Proximity clouds – an acceleration technique for 3D grid traversal. *The Visual Computer*, 10(11):27–38, 1994.

[9] B. Csebfalvi, A. König, and E. Gröller. Fast maximum intensity projection using binary shear-warp factorization. In *Proceedings of the 7-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media'99, WSCG '99*, pages 47–54, 1999.



[10] B. Csebfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. Technical Report TR-VRVis-2001-002 at the VRVis Research Center, Vienna, `http://www.vrvis.at/`, 2001.

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

[12] D. S. Ebert, R. Yagel, J. Scott, and Y. Kurzion. Volume rendering methods for computational fluid dynamics visualization. In *Visualization'94*, pages 232–239, October 1994.

[13] K. K. Udupa (Editor) and G. T. Herman. *3D Imaging in Medicine*. CRC Press, 1999.

[14] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl. Combining local and remote visualization techniques for interactive volume rendering in medical applications. In *Proceedings IEEE Visualization 2000*, pages 449–452, 2000.

[15] K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proceedings IEEE Visualization '99*, pages 139–146, 1999.

[16] B. H. Mc Cormick et al. Visualization in scientific computing. *ACM Computer Graphics*, 21(6), 1987.

[17] J. Fowler and R. Yagel. Lossless compression of volume data. In *Proceedings IEEE Volume Visualization Symposium '94*, pages 43–50, 1994.

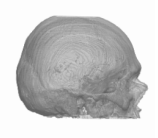
[18] J. Gailly and M. Adler. `gzip`. URL: `http://www.gzip.org`.

[19] A. S. Glassner. Normal coding. *Graphics Gems*, pages 257–264, 1990.

[20] C. Grebogi, E. Ott, and J. A. Jorke. Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics. *Science*, 238:256–261, 1987.

[21] R. B. Haber and D. A. McNabb. *Visualization Idioms: A conceptual Model for scientific visualization systems, Visualization in Scientific Computing*, pages 74–93. 1996.

[22] H. Hauser, L. Mroz, G.-I. Bischi, and E. Gröller. Two-level volume rendering – fusing MIP and DVR. In *Proceedings IEEE Visualization 2000*, pages 211–218, 2000.



[23] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of transfer functions with stochastic search techniques. In *Proceedings of IEEE Visualization '96*, pages 227–234, 1996.

[24] W. Heidrich, M. McCool, and J. Stevens. Interactive maximum projection volume rendering. In *Proceedings of IEEE Visualization '95*, pages 11–18, 1995.

[25] M. Jern. Information drill-down using web tools. In *Proceedings of the 8th EUROGRAPHICS Workshop on Visualization in Scientific Computing*, pages 1–12, 1997.

[26] J. T. Kajiya. Ray tracing volume densities. In *Proceedings of ACM SIGGRAPH '84*, pages 165–174, 1984.

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

[28] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Proceedings of ACM SIGGRAPH '94*, pages 451–458, 1984.

[29] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, 1988.

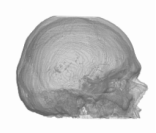
[30] M. Levoy. Volume rendering using the Fourier projection-slice theorem. In *Proceedings of Graphics Interface '92*, pages 61–69, 1992.

[31] Ch. Lindenbeck and H. Ulmer. Geology meets virtual reality: VRML visualization server applications. In *Proceedings of the 6-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media'98, WSCG '98*, Plzen, Czech Republic, 1998.

[32] L. Lippert, M. Gross, and C. Kurmann. Compression domain volume rendering for distributed environments. In *Proceedings of EUROGRAPHICS '97*, pages C95–C107, 1997.

[33] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of ACM SIGGRAPH '87*, pages 163–189, 1987.

[34] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber.



Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of ACM SIGGRAPH '97*, pages 389–400, 1997.

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

[36] M. Meissner, J. Huang, D. Bartz, K. Müller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings IEEE Volume Visualization Symposium 2000*, pages 81–90, 2000.

[37] C. K. Michaels and M. J. Bailey. VizWiz: A Java applet for interactive 3D scientific visualization on the web. In *Proceedings Visualization '97*, pages 261–267, 1997.

[38] C. Mira, L. Gardini, A. Barugola, and J. C. Cathala. *Chaotic Dynamics in Two-Dimensional Noninvertible Maps*. World Scientific, Singapore, 1996.

[39] L. Mroz and H. Hauser. RTVR – a flexible Java library for interactive volume rendering. Technical Report TR-VRVis-2001-003 at the VRVis Research Center, Vienna, `http://www.vrvis.at/`, 2001.

[40] L. Mroz and H. Hauser. Space-efficient boundary representation of volumetric objects. Accepted for publication in Data Visualization 2001, Proceedings of the Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization, 2001.

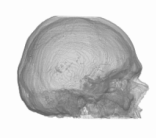
[41] L. Mroz, H. Hauser, and E. Gröller. Interactive high-quality maximum intensity projection. In *Proceedings of EUROGRAPHICS 2000*, pages C341–C350, 2000.

[42] L. Mroz, A. König, and E. Gröller. Real-time maximum intensity projection. In *Data Visualization '99, Proceedings of the Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, 1999.

[43] L. Mroz, A. König, and E. Gröller. Maximum intensity projection at warp speed. *Computers & Graphics*, 24(3):343–352, June 2000.

[44] L. Mroz, R. Wegenkittl, and E. Gröller. Mastering interactive surface rendering for Java-based diagnostic applications. In *Proceedings IEEE Visualization 2000*, pages 437–440, 2000.

[45] L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4D linear regression. In *Proceedings of EUROGRAPHICS 2000*, pages C351–C357, 2000.



[46] P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *Proceedings IEEE Visualization '93*, pages 11–18, 1993.

[47] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings IEEE Visualization '98*, pages 233–238, 1998.

[48] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volume pro real-time ray-casting system. In *Proceedings of ACM SIGGRAPH '99*, pages 251–260, 1999.

[49] B.-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[50] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop 2000*, 2000.

[51] G. Sakas, M. Grimm, and A. Savopoulos. Optimized maximum intensity projection. In *Proceedings of 5th EUROGRAPHICS Workshop on Rendering Techniques*, pages 55–63, Dublin, Ireland, 1995.

[52] Y. Sato, N. Shiraga, S. Nakajima, S. Tamura, and R. Kikinis. LMIP: Local maximum intensity projection – a new rendering method for vascular visualization. *Journal of Computer Assisted Tomography*, 22(6), 1998.

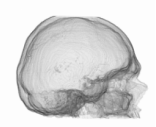
[53] W. Schroeder, K. Martin, and W. Lorensen. The visualization toolkit. In *An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1996.

[54] R. Schubert, B. Pflesser, A. Pommert, K. Preiesmeyer, M. Riemer, Th. Schiemann, U. Tiede, P. Steiner, and H. Höhne. Interactive volume visualization using intelligent movies. In *Proceedings Medicine Meets Virtual Reality*, 1999.

[55] M. Sramek. Fast surface rendering from raster data by voxel traversal using chessboard distance. In *Proceedings IEEE Visualization '94*, pages 188–195, 1994.

[56] J. Trapp and H.-G. Pagendarm. A prototype for a WWW-based visualization service. In *Proceedings of the 8th EUROGRAPHICS Workshop on Visualization in Scientific Computing*, pages 23–30, 1997.

[57] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.



[58] M. Wan, A. Kaufmann, and S. Bryson. High performance presence-accelerated ray casting. In *Proceedings of IEEE Visualization 1999*, pages 379–386, 1999.

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

[60] L. Westover. Interactive volume rendering. In *1989 Chapel Hill Volume Visualization Workshop*, pages 9–16, 1989.

[61] J. D. Wood, K. W. Brodlie, and H. Wright. Visualization over the world wide web and its application to environmental data. In *Proceedings IEEE Visualization '96*, pages 81–86, 1996.

[62] WWW. Experimental volume visualization environment . URL: http://www.inf.ethz.ch/personal/lippert/EVOLVE/.

[63] WWW. Java exploration tool for dynamical systems . URL:http://www.cg.tuwien.ac.at/research/vis/dynsys/frolic/.

[64] WWW. The Java virtual wind tunnel . URL:http://raphael.mit.edu/Java/.

[65] WWW. Volumepro net. URL:http://www.rtviz.com/products/vpnet.html.

[66] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray-casting using 3D distance transforms. In *Visualization in Biomedical Computing 1992*, volume 1808 of *Proceedings SPIE*, pages 324–335, 1992.

[67] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Techniques for speeding up high-quality perspective maximum intensity projection. *Pattern Recognition Letters*, 15:507–517, 1994.