RTVR – a flexible Java library for interactive volume rendering

Lukas Mroz and Helwig Hauser†

VRVis Research Center; Vienna, Austria;
http://www.vrvis.at/

Abstract

This paper presents several distinguishing design features of RTVR – a Java-based library for interactive volume rendering. We describe, how the careful design of data structures, in our case based on voxel enumeration, and an intelligent use of lookup tables enable interactive volume rendering even on low-end PC hardware. By assigning voxels to distinct objects within the volume and by using an individual setup and combination of lookup tables for each object, object-aware rendering can be performed: different transfer functions, shading models and compositing modes (MIP, DVR) can be mixed within a single scene, while still providing rendering results in real-time. While providing frame rates similar to volume visualization using 3D consumer hardware, RTVR provides much more flexibility and extensibility due to it's pure software nature. Furthermore, due to an efficient intermediate data representation, RTVR can be used to provide volume viewing facilities over low-bandwidth networks, with almost full control over rendering and visualization mapping parameters (clipping, shading, transfer function) for the user. This paper also addresses specific problems which arise by the use of Java for interactive Visualization.

Key words: interactive volume visualization, Internet-based visualization, Java

1. Introduction

Volume visualization has proven to be a valuable tool for exploration, analysis and presentation of data from numerous fields of application like medicine, geo sciences, or mathematics for example. 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. Also, communicating the results of a 3D visualization to the viewer is easier, if the viewer is able to manipulate the visualization output to a certain degree. On the other hand, high-quality rendering, which usually can not be performed interactively, is often required for the presentation of data.

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 library. It unifies several techniques for interactive rendering[12, 13, 7, 3] and efficient data transmission[11] into a flexible framework which can be used to provide volume visualization on PC-hardware. In this paper we escribe some of the distinguishing design issues of RTVR which are responsible for its excellent efficiency with respect to real-time volume rendering.

In contrast to established volume visualization toolkits like VolVis[1] or VTK[14], 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 a pre-filtering (decimation) of the data and an internal data structure which is well-suited for fast rendering[12, 13]. The data structure itself and also the rendering method which is used by RTVR – a fast shear-warp[9] based approach – are not suited to perform high-quality rendering. On the other hand, z-buffer output from the method can be used to accelerate high-quality ray-casting based rendering.

A memory efficient way of handling volumetric data makes RTVR well-suited for remote rendering of volumetric data over low-bandwidth networks[11], either for interac-

† mroz@vrvis.at, hauser@vrvis.at

tive presentation of data which has been generated offline, or within a split client/server approach for online visualization. The challenge of interactively presenting images of volumetric data, which also can be manipulated interactively, over networks on standard desktop-hardware has already been addressed by several other approaches. The simplest way to display objects contained within volumetric data sets is to extract a polygonal surface representation of the object and to render a sufficiently simplified version of the model at the client (via a VRML browser, for example). Although current consumer 3D hardware is already quite powerful, it is still not possible to render highly detailed models from real-life data sets at interactive frame rates. To overcome this problem, Engel and others[6] 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[5]. 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[15]. A multi-dimensional array of images is rendered and stored in 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 (inbetween pre-computed views). Furthermore, the size of even small-scale movies already becomes a limiting factor for viewing over low-bandwidth networks.

The approach implemented by the RTVR library is located inbetween the methods discussed above. The amount of data which actually is transmitted to the client for visualization is very low (about the size of several images), especially in comparison to the Quicktime VR approach. 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 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.

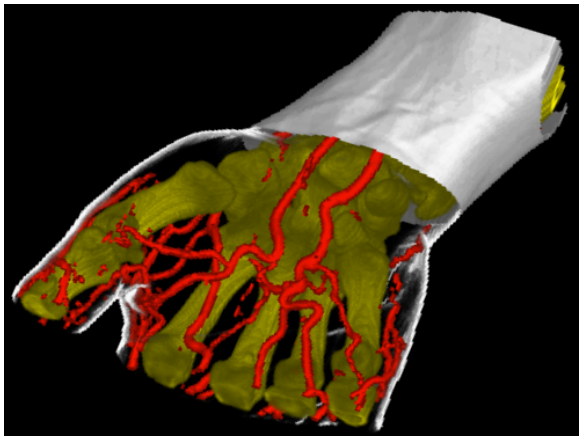


Figure 1: *Visualization of a human hand created with RTVR: surface rendering of vessels, combined with direct volume rendering of bones, and a surface rendering of the skin. The skin is clipped into two parts, one shaded using Phong shading, the other one using a non-photorealistic rendering model which emphasizes contours.*

In the following, visualization capabilities and the internal structure of the RTVR library are presented. Section 2 gives a short overview over the rendering features and visualization techniques which are implemented by RTVR. Section 3 presents RTVS's internal data structure, its handling of user interactions, and the rendering algorithms used. Timings for typical application scenarios are given in section 4, followed by the presentation of sample applications, which are based on the RTVR library (section 5).

2. 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 usually 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 our approach.

RTVR interprets the input volume as being composed of objects, like bones, vessels, and other tissue making up a data-set of a human hand (Fig. 1). The necessary segmenta-

tion 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[4, 3]. In addition to the shading model, the way in which voxels are blended to the image plane can be defined in a object-aware way. Most volume rendering packages only allow to render a whole data set using either the usual opacity blended compositing[8], or surface rendering[10, 13], or maximum intensity projection (MIP)[12]. In RTVR, compositing modes can be selected on a per-object basis and combined with another inter-object compositing mode (two-level volume rendering[7]). 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 belongs to other time (parameter) steps is 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 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.

3. 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 (Fig. 2), voxels which actually are relevant for the user-defined visualization are extracted and stored (object by object) within a special 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[10], voxels with a low gradient magnitude do not noticeably contribute to an image and therefore can be omitted.

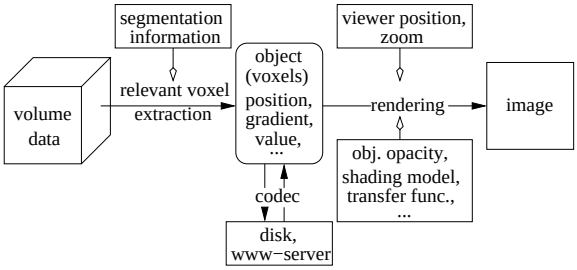


Figure 2: *Volume data flow within RTVR: first, voxels with actually contribute to the visualization are extracted, then this representation of volumetric objects is used for fast and flexible rendering.*

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[11] and storing it on disk together with the currently used visualization and rendering parameters.

For rendering, RTVR uses a fast, shear-warp based algorithm, 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-carthesian 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 lookup tables to obtain and modulate color and opacity values in a way which is defined by the selected rendering mode. The lookup tables are

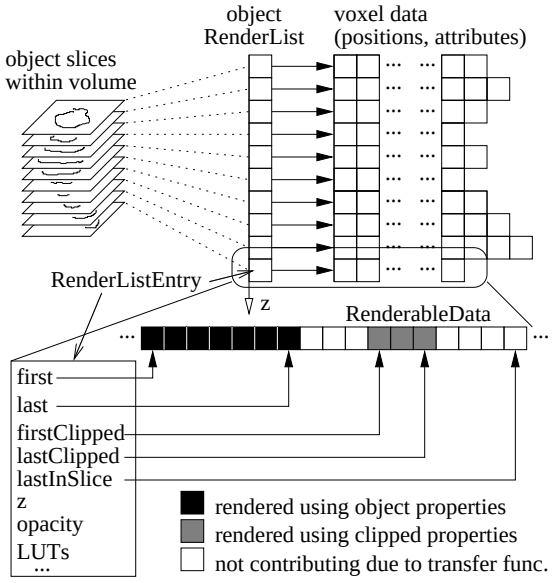


Figure 3: *Volumetric object representation: voxels which are relevant for rendering an object are extracted slice by slice from the volume and stored into* `RenderList`*s.*

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.

3.1. `RenderList` as Data Representation

During the object extraction process, the volume is scanned slice by slice, producing for each object within the volume a so called `RenderList` which is an array of `RenderListEntry` objects, each one containing the object's voxels within a slice (Fig. 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 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 processing, 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 256^2 slices. 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 per-

formed 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 `RenderLists` 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[2]. 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.

3.2. The RTVR Scene Graph

After extraction, the `RenderLists` and attribute data of volumetric objects are encapsulated into `VolumeObjects` and added to a common scene graph for rendering (Fig. 4). A common task of all types of nodes within the scene graph is to deliver up-to-date `RenderLists` which represent the

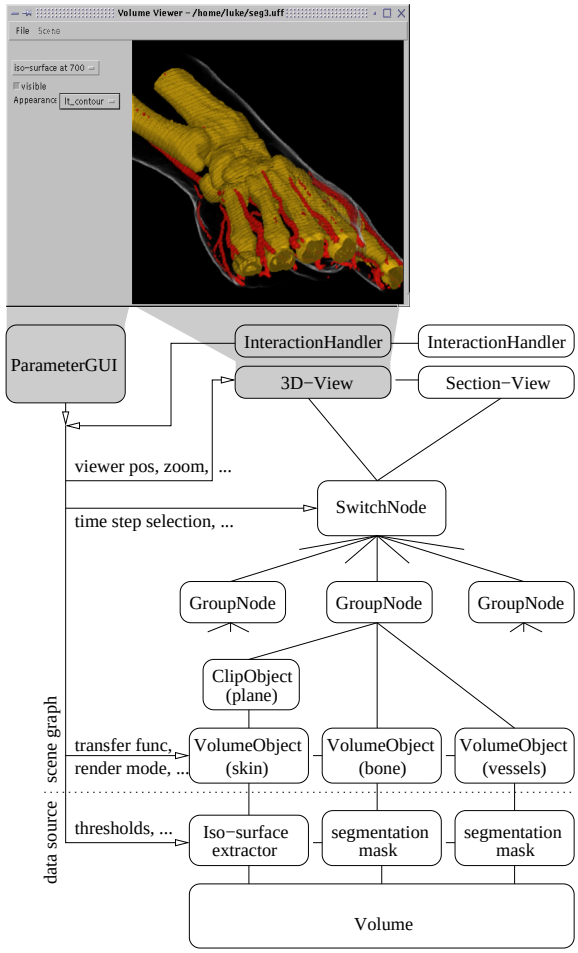


Figure 4: *RTVR scene graph and user interaction handling*

content of their subgraphs. In the following a short overview over 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 `RenderListEntrys`. In addition to providing a simple way of handling multiple objects, the main purpose of the `GroupNode` is to merge and sort the `RenderLists` 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. `SwitchNodes` allow to 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-step volumes.

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.

3.3. 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 practice, 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 keys and dragging. Transfer function contrast (color and alpha), 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 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.

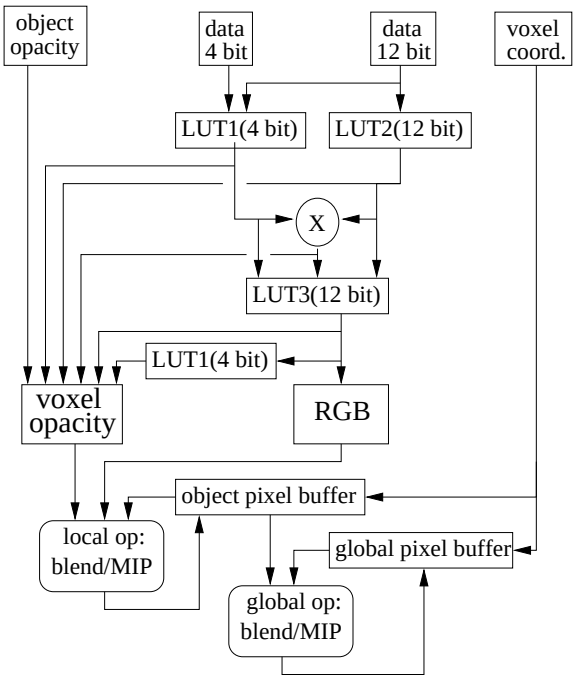


Figure 5: *Voxel rendering within RTVR: LUT1 can be used either for color or opacity computations. For color computations, it can be indexed by the 4 bit attribute or by the upper bits of the 12 bit attribute. LUT2 is addressed by the 12 bit attribute. LUT3 is addressed using the output of LUT1 or LUT2 (which may be an identity mapping), or a combination of both.*

3.4. Rendering

To achieve interactive rendering rates even on standard desktop hardware, a 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 a previously published version of this fast algorithm[13], RTVR includes an extended version, which provides more flexibility for mapping voxel attributes to color and opacity. Three look-up tables (typically 1x4 bit, 2x12 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 Fig. 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 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 (color plate a) and a non-photorealistic shading table (color plate b) which enhances the contour of an object[4]. 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 (Fig. 5), and indexed by the 12 bit data channel which contains the gradient vector. The output of the lookup 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 lookup into LUT2, 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 combinations 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 color plate a). 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.
- *Gradient magnitude weighted opacity*: if voxels in areas with high gradient magnitude are rendered 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 color plate b. If the result of the LUT2 lookup is also used as voxel opacity, the object becomes almost entirely transparent – except for the contours which remain opaque (Fig. 1, clipped skin).
- *Colored contour outlines*: contours are colored differently from the remaining (Phong shaded) parts of the object (similar to method presented by Ebert and Rheingans[4]). 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 (direct volume rendering, DVR). The action to be performed for compositing inbetween objects can be defined independently from the object compositing modes (two-level volume rendering[7], color plate c). 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 (Fig. 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 away, clipped data is rendered using a different set of attributes. Separate values can be set for clipped object opacity, rendering mode (LUT con-

figuration) 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 (color plate d). By using for example 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 (Fig. 1).

To obtain highest possible 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 inbetween 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[12]. 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.

3.5. Data Optimization

Depending on the compositing method in use, on the content of lookup 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 function mapping) do not contribute to the result. If opacity-weighted blending 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 Fig. 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.

4. 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[13], MIP[12] and two-level rendering approach[7] used by RTVR have been published in previous work. 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 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 1. Skin, bones, and vessels are represented by their surface voxels. The rendering is carried out using MIP, DVR, a grayscale 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 color plate d, with bone, skin and vessels represented as surfaces. The data set in row 3 is similar to the one depicted in color plate f. The basin is represented by it's surface voxels, the chaotic attractor is a truely 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 lookup 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 a 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

data set	*size*	*scene voxels*	*MIP*	*DVR*	*DVR/mono*	*mixed MIP/DVR*
hand & vessels	256^2x124	380k	80ms	135ms	80ms	170ms
head & vessels	256^2x158	640k	100ms	185ms	110ms	290ms
attractor & basin	256^3	1M	130ms	250ms	140ms	333ms

Table 1: *Timings for various data sets and rendering modes*

is used by Netscape browsers (Version <= 4.7.4) on Linux – more than ten times slower than the timings in table 1.

5. Sample Applications of RTVR

The RTVR library has been successfully used to provide volume visualization functionality within two projects. The first application is a volume viewer which can be used for fast volume data exploration and analysis in the field of medical data. Visualizations created within the viewer (extracted objects and visualization parameters) can be stored using a compact representation (typically just a few hundred kilobytes[11]) for later interactive viewing or for publication on the Internet. An applet version of the viewer which provides the same 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/`.

A second application which makes use of the capabilities of RTVR is a visualization and analysis system for 3D dynamical systems (discrete maps)[2]. The application is used to analyze and visualize structures and events within the phase-space of the systems. For this application, objects of interest are attractors (often complex and chaotic), their basins of attraction (i.e., the set of all system states which are attracted by them) and surfaces which separate regions with different properties (color plate e). Events (bifurcations) are often caused by contacts between structures as some parameter of the dynamical system is changed. The process of visualization is split into two parts. A volumetric representation of the structures within phase space is computed offline (256^3 volumes) and stored in a space-efficient form. 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 is extremely useful for large sequences of volumes) which provides application specific functionality, like the shooting of trajectories by pointing with the mouse at the start position. To ease the detection of contacts between objects, distance information can be mapped to voxel color, as shown in color plate f. 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.

6. Conclusions

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, while providing 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.

7. Acknowledgements

This project has been financed within the Kplus program of the Austrian federal government. The medical data sets depicted in this paper are property of Tiani Medgraph, Vienna, Austria. The authors want to thank Meister E. Gröller, Csébfalvi Balázs, and the team of Tiani Medgraph.

References

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

2. G.-I. Bischi, L. Mroz, and H. Hauser. Studying basin bifurcations in nonlinear triopoly games by using 3D visualization. *Accepted for publication in the Journal of Nonlinear Analysis. Also available as a technical report from the authors*.

3. 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.

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

5. 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.

6. K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proceedings IEEE Visualization*, 1999.

7. 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.

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

9. P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorisation of the viewing transform. In *Proceedings of ACM SIGGRAPH '94*, pages 451–459, 1994.

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

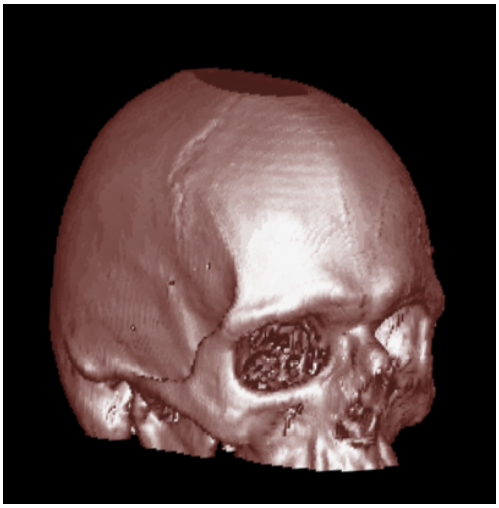
11. L. Mroz, H. Hauser, and E. Gröller. Space efficient boundary representation of volumetric objects. Technical Report TR-VRVis-2000-006 at the VRVis Research Center, Vienna, `http://www.vrvis.at/`, 2000.

12. 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*, pages 135–144, 1999.

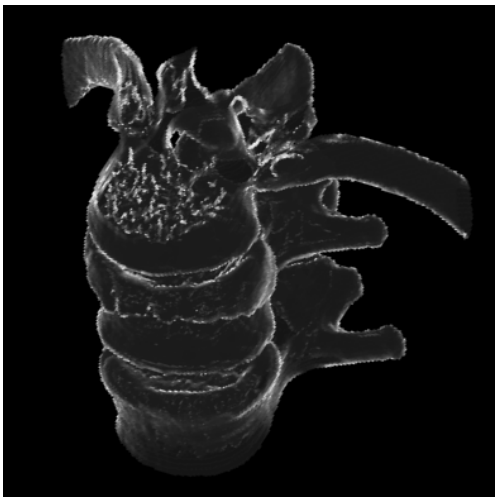
13. L. Mroz, R. Wegenkittl, and E. Gröller. Mastering interactive surface rendering for java-based diagnostic applications. In *Proceedings IEEE Visualization 2000*, pages 437–440, 2000.

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

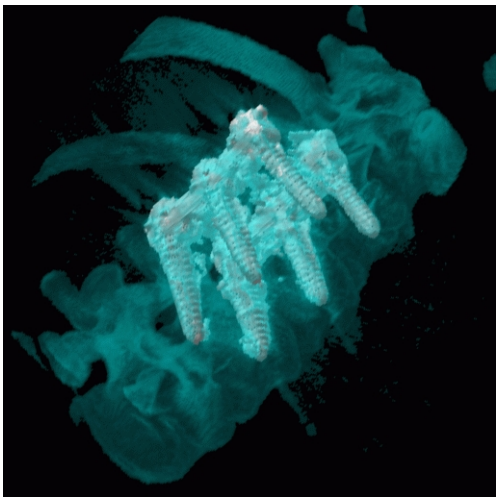
15. R. Schubert, B. Pflesser, A. Pommert, K. Preiesmeyer, M. Riemer, Th. Schiemann, U. Tiede, P. Steiner, and H. Höhne. Interactive volume visualization using intelligent movies. In *Proceedings Medicine Meets Virtual Reality*, 1999.



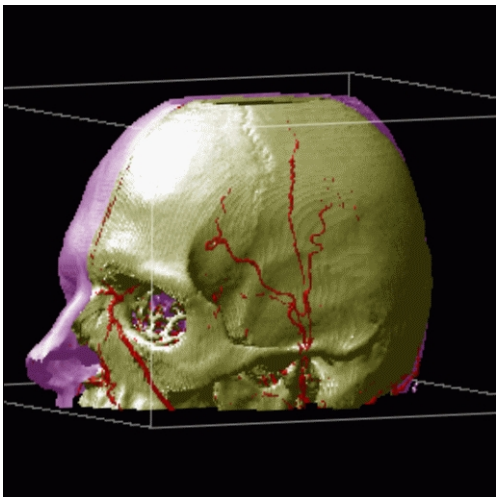
a) surface representation of a skull



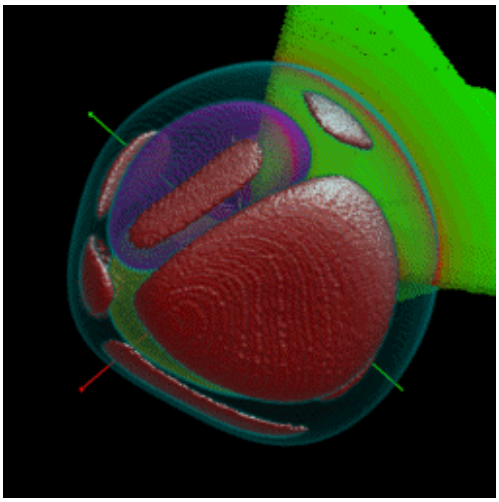
b) contour enhanced vertebrae



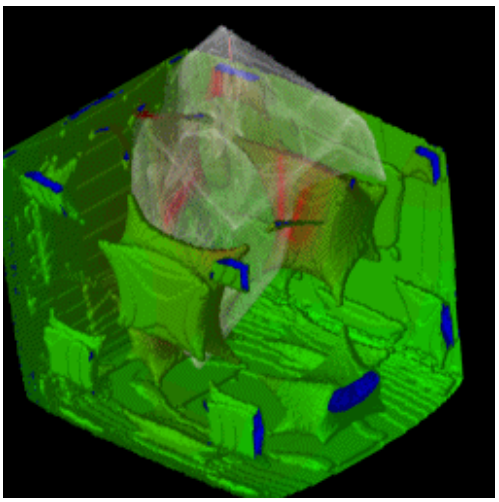
c) MIP for bone, DVR for screws



d) object aware clipping of skin surface



e) basins of attraction and a critical surface



f) contact bifurcation - one out of a sequence of 40 volumes