High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware

Markus Hadwiger Christoph Berger Helwig Hauser *

VRVis Research Center, Austria

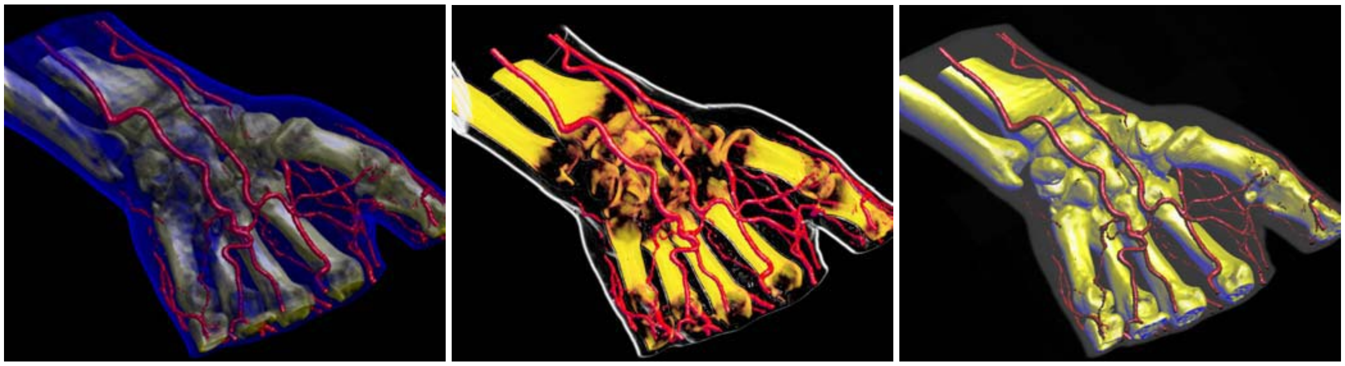


Figure 1: Segmented hand data set (256x128x256) with three objects: skin, blood vessels, and bone. Two-level volume rendering integrates different transfer functions, rendering and compositing modes: (left) all objects rendered with shaded DVR; the skin partially obscures the bone; (center) skin rendered with non-photorealistic contour rendering and MIP compositing, bones rendered with DVR, vessels with tone shading; (right) skin rendered with MIP, bones with tone shading, and vessels with shaded iso-surfacing; the skin merely provides context.

Abstract

One of the most important goals in volume rendering is to be able to visually separate and selectively enable specific objects of interest contained in a single volumetric data set, which can be approached by using explicit segmentation information. We show how segmented data sets can be rendered interactively on current consumer graphics hardware with high image quality and pixel-resolution filtering of object boundaries. In order to enhance object perception, we employ different levels of object distinction. First, each object can be assigned an individual transfer function, multiple of which can be applied in a single rendering pass. Second, different rendering modes such as direct volume rendering, iso-surfacing, and non-photorealistic techniques can be selected for each object. A minimal number of rendering passes is achieved by processing sets of objects that share the same rendering mode in a single pass. Third, local compositing modes such as alpha blending and MIP can be selected for each object in addition to a single global mode, thus enabling high-quality two-level volume rendering on GPUs.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: volume rendering, segmentation, non-photorealistic rendering, consumer graphics hardware

*mailto:{Hadwiger|Berger|Hauser}@VRVis.at, http://www.VRVis.at/vis/

IEEE Visualization 2003,
October 19-24, 2003, Seattle, Washington, USA
0-7803-8120-3/03/$17.00 ©2003 IEEE

1 Introduction

In many volume rendering methods, all voxels contained in a volumetric data set are treated in an identical manner, i.e., without using any a priori information that specifies object membership on a per-voxel basis. In that case, visual distinction of objects is usually achieved by either using multiple semi-transparent iso-surfaces or, more commonly, with direct volume rendering and an appropriate transfer function. In the latter case, multi-dimensional transfer functions [Kindlmann and Durkin 1998; Kniss et al. 2001] have proven to be especially powerful in facilitating the perception of different objects. In recent years, non-photorealistic volume rendering approaches [Ebert and Rheingans 2000; Csebfalvi et al. 2001; Lu et al. 2002] have also been used successfully for improving the perception of distinct objects embedded in a single volume.

However, it is also often the case that a single rendering method or transfer function does not suffice in order to distinguish multiple objects of interest according to a user's specific needs. A very powerful approach to tackling this problem is to create explicit object membership information via segmentation [Udupa and Herman 1999], which usually yields one binary segmentation mask for each object of interest, or an object ID for each of the volume's voxels.

Unfortunately, integrating segmentation information and multiple rendering modes with different sets of parameters into a fast high-quality volume renderer is not a trivial problem, especially in the case of consumer hardware volume rendering, which tends to only be fast when all or most voxels can be treated identically. On such hardware, one would also like to use a single segmentation mask volume in order to use a minimal amount of texture memory. Graphics hardware cannot easily interpolate between voxels belonging to different objects, however, and using the segmentation mask without filtering gives rise to artifacts. Thus, one of the major obstacles in such a scenario is filtering object boundaries in order to attain high quality in conjunction with consistent fragment assignment and without introducing non-existent object IDs.

In this paper, we show how segmented volumetric data sets can be rendered efficiently and with high quality on current consumer graphics hardware. The segmentation information for object distinction can be used at multiple levels of sophistication, and we describe how all of these different possibilities can be integrated into a single coherent hardware volume rendering framework.

First, different objects can be rendered with the same rendering technique (e.g., DVR), but with different transfer functions. Separate per-object transfer functions can be applied in a single rendering pass even when object boundaries are filtered during rendering. On an ATI Radeon 9700, up to eight transfer functions can be folded into a single rendering pass with linear boundary filtering. If boundaries are only point-sampled, e.g., during interaction, an arbitrary number of transfer functions can be used in a single pass. However, the number of transfer functions with boundary filtering in a single pass is no conceptual limitation and increases trivially on architectures that allow more instructions in the fragment shader.

Second, different objects can be rendered using different hardware fragment shaders. This allows easy integration of methods as diverse as non-photorealistic and direct volume rendering, for instance. Although each distinct fragment shader requires a separate rendering pass, multiple objects using the same fragment shader with different rendering parameters can effectively be combined into a single pass. When multiple passes cannot be avoided, the cost of individual passes is reduced drastically by executing expensive fragment shaders only for those fragments active in a given pass. These two properties allow highly interactive rendering of segmented data sets, since even for data sets with many objects usually only a couple of different rendering modes are employed. We have implemented direct volume rendering with post-classification, pre-integrated classification [Engel et al. 2001], different shading modes, non-polygonal iso-surfaces, and maximum intensity projection. See figures 1 and 2 for example images. In addition to non-photorealistic contour enhancement [Csebfalvi et al. 2001] (figure 1, center; figure 2, skull), we have also used a volumetric adaptation of tone shading [Gooch et al. 1998] (figure 1, right), which improves depth perception in contrast to standard shading.

Finally, different objects can also be rendered with different compositing modes, e.g., alpha blending and maximum intensity projection (MIP), for their contribution to a given pixel. These per-object compositing modes are object-local and can be specified independently for each object. The individual contributions of different objects to a single pixel can be combined via a separate global compositing mode. This two-level approach to object compositing [Hauser et al. 2001] has proven to be very useful in order to improve perception of individual objects.

In summary, the major novel contributions of this paper are:

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

- An efficient method for mapping a single object ID volume to and from a domain where filtering produces correct results even when three or more objects are present in the volume. The method is based on simple 1D texture lookups and able to map and filter blocks of four objects simultaneously.

- An efficient object-order algorithm based on simple depth and stencil buffer operations that achieves correct compositing of objects with different per-object compositing modes and an additional global compositing mode. The result is conceptually identical to being able to switch compositing modes for any given group of samples along the ray for any given pixel.

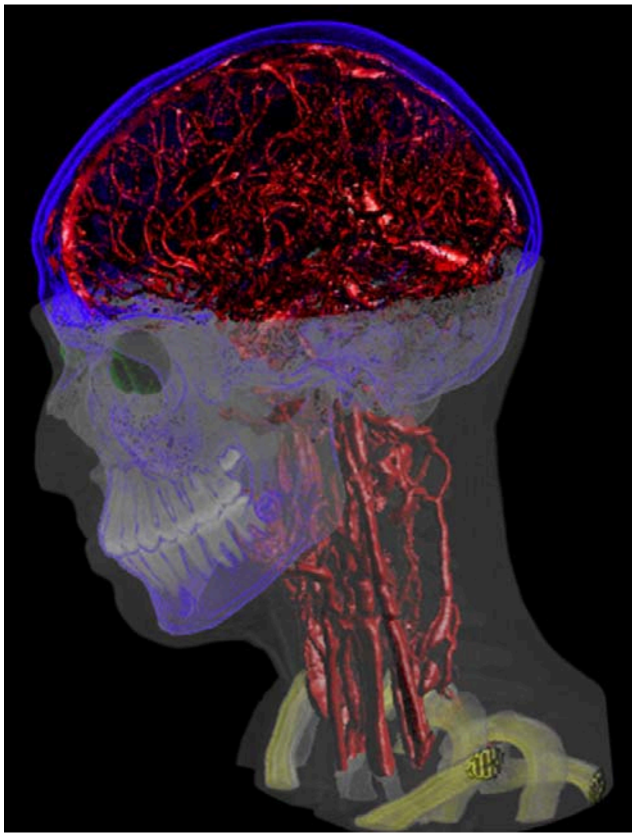


Figure 2: Segmented head and neck data set (256x256x333) with six different enabled objects. The skin and teeth are rendered as MIP with different intensity ramps, the blood vessels and eyes are rendered as shaded DVR, the skull uses contour rendering, and the vertebrae use a gradient magnitude-weighted transfer function with shaded DVR. A clipping plane has been applied to the skin object.

Related work

The framework presented in this paper is based on re-sampling and rendering a volume via a stack of textured slices that are blended on top of each other in either back-to-front or front-to-back order. For this purpose, either view-aligned slices through 3D textures [Cullip and Neumann 1993; Cabral et al. 1994; Westermann and Ertl 1998], or object-aligned slices with 2D textures can be used. In the latter case, intermediate slices can also be interpolated on-the-fly during rendering in order to attain tri-linear interpolation [Rezk-Salama et al. 2000]. The number of slices necessary for high-quality results can be reduced drastically by considering two adjacent slices as constituting a single slab and compensating for non-linear transfer function changes within each slab via a lookup into a pre-integrated transfer function table [Engel et al. 2001]. Multi-dimensional transfer functions [Kindlmann and Durkin 1998] are a very important tool for distinguishing different objects contained in a volume, especially when combined with an intuitive and interactive interface for specifying them [Kniss et al. 2001]. In recent years, there also has been a remarkable interest in non-photorealistic rendering techniques such as tone shading [Gooch et al. 1998] that are increasingly being applied to volumes [Ebert and Rheingans 2000; Csebfalvi et al. 2001; Lu et al. 2002]. If no segmentation information is present, multiple rendering passes with one transfer function each and non-photorealistic shading can be used in order to enhance perception of individual objects [Lum and Ma 2002].

The idea of using two conceptual levels for compositing volumetric objects has first been described in the context of a fast soft-

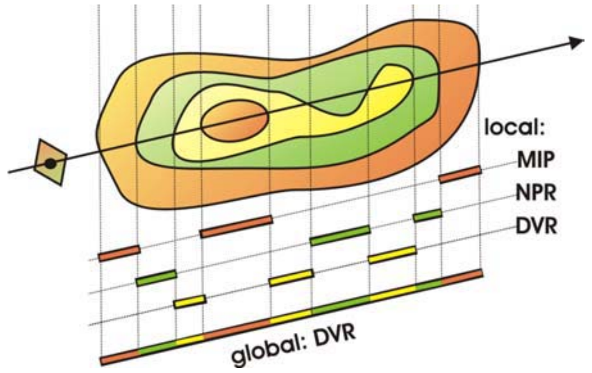


Figure 3: A single ray corresponding to a given image pixel is allowed to pierce objects that use their own object-local compositing mode. The contributions of different objects along a ray are combined with a single global compositing mode. Rendering a segmented data set with these two conceptual levels of compositing (local and global) is known as *two-level volume rendering* .

ware two-level volume renderer [Hauser et al. 2001], and we refer to this earlier work for detailed suggestions on when and how different rendering and compositing modes together with appropriately specified transfer functions can facilitate object perception.

The topic of image and volume segmentation is a huge area on its own [Udupa and Herman 1999], and we simply treat the segmentation information as additional a priori input data that are already available for a given data set. In order to achieve high rendering quality, it is necessary to distinguish individual objects with subvoxel precision [Tiede et al. 1998], i.e., what we refer to as pixel-resolution boundary filtering. Even linear filtering of segmentation data is not directly possible on graphics hardware when more than two objects have been segmented, since object IDs cannot be interpolated directly. Ultimately, rendering segmented data sets can be viewed as being composed of multiple individual volumetric clipping problems. Recent work has shown how to achieve high-quality clipping in graphics hardware [Weiskopf et al. 2003], which can also be combined with pre-integrated classification by adjusting the lookup into the pre-integration table accordingly [Röttger et al. 2003]. However, it is not trivial to apply clipping approaches to the rendering of segmented data as soon as the volume contains more than two objects and high quality results and a minimal number of rendering passes are desired. Excluding individual fragments from processing by an expensive fragment shader via the early z-test is also crucial in the context of GPU-based ray casting in order to be able to terminate rays individually [Krüger and Westermann 2003].

2 Rendering segmented data sets

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

In order to render a segmented data set, we determine object membership of individual fragments by filtering object boundaries in the hardware fragment shader (section 3). Object membership determines which transfer function, rendering, and compositing modes should be used for a given fragment. We render the volume in a number of rendering passes that is basically independent of the number of contained objects. It most of all depends on the required number of different hardware configurations that cannot be changed during a single pass, i.e., the fragment shader and compositing mode. Objects that can share a given configuration can be rendered in a single pass. This also extends to the application of multiple per-object transfer functions (section 4) and thus the actual number of rendering passes is usually much lower than the number of objects or transfer functions. It depends on several major factors:

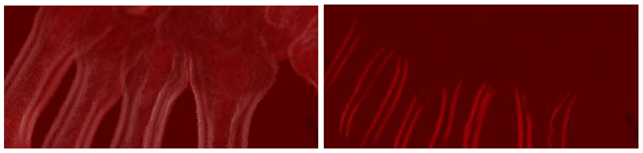


Figure 4: Detecting changes in compositing mode for each individual sample along a ray can be done exactly using two rendering buffers (left), or approximately using only a single buffer (right).

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

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

Transfer functions. Much more often than the basic rendering mode, a change of the transfer function is required. For instance, all objects rendered with DVR usually have their own individual transfer functions. In order to avoid an excessive number of rendering passes due to simple transfer function changes, we apply multiple transfer functions to different objects in a single rendering pass while still retaining adequate filtering quality (section 4).

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

Different compositing modes per object also imply that the (conceptual) ray corresponding to a single pixel must be able to combine the contribution of these different modes (figure 3). Especially in the context of texture-based hardware volume rendering, where no actual rays exist and we want to obtain the same result with an object-order approach instead, we have to use special care when compositing. In order to ensure correct compositing, we are using two render buffers and track the current compositing mode for each pixel. Whenever the compositing mode changes for a given pixel, the already composited part is transferred from the *local compositing buffer* into the *global compositing buffer*. Section 5 shows that this can actually be done very efficiently without explicitly considering individual pixels, while still achieving the same compositing behavior as a ray-oriented image-order approach, which is crucial for achieving high quality. For faster rendering we allow falling back to single-buffer compositing during interaction (figure 4).

2.1 Basic rendering loop

We will now outline the basic rendering loop that we are using for each frame. Table 1 gives a high-level overview.

Although the user is dealing with individual objects, we automatically collect all objects that can be processed in the same rendering pass into an *object set* at the beginning of each frame. For each object set, we generate an *object set membership texture*, which is a 1D lookup table that determines the objects belonging to the set. In order to further distinguish different transfer functions in a single object set, we also generate 1D *transfer function assignment textures*. Both of these types of textures are shown in figure 5 and described in sections 2.3, 3, and 4. After this setup, the entire slice stack is rendered. Each slice must be rendered for every object set containing an object that intersects the slice, which is determined in a pre-process. If there is more than a single object set for the current slice, we optionally render all object set IDs of the slice into the depth buffer before rendering any actual slice data. This enables us to exploit the early z-test during all subsequent passes for each object set, see below. For performance reasons, we never use object ID filtering in this pass, which allows only conservative fragment culling via the depth test. Exact fragment rejection is done in the fragment shader. Before a slice can be rendered for any object set, the fragment shader and compositing mode corresponding to this set must be activated. Using the two types of textures mentioned above, the fragment shader filters boundaries, rejects fragments not corresponding to the current pass, and applies the correct transfer function. In order to attain two compositing levels, slices are rendered into a local buffer, as already outlined above. Before rendering the current slice, those pixels where the local compositing mode differs from the previous slice are transferred from the local into the global buffer using the global compositing mode. After this transfer, the transferred pixels are cleared in the local buffer to ensure correct local compositing for subsequent pixels. In the case when only a single compositing buffer is used for approximate compositing, the local to global buffer transfer and clear are not executed.

2.2 Conservative fragment culling via early z-test

On current graphics hardware, it is possible to avoid execution of the fragment shader for fragments where the depth test fails as long as the shader does not modify the depth value of the fragment. This early z-test is crucial to improving performance when multiple rendering passes have to be performed for each slice. If the current slice's object set IDs have been written into the depth buffer before, see above, we conservatively reject fragments not belonging to the current object set even before the corresponding fragment shader is started. In order to do this, we use a depth test of `GL_EQUAL` and configure the vertex shader to generate a constant depth value for each fragment that exactly matches the current object set ID.

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

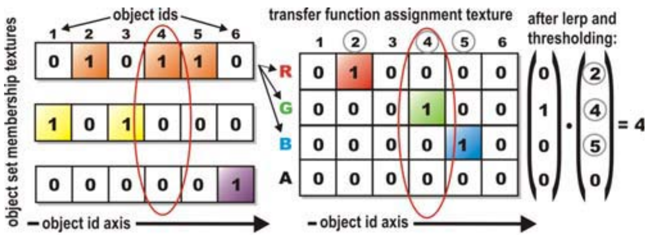
Table 1: The basic rendering loop that we are using. Object set membership can change every time an object's rendering or compositing mode is changed, or an object is enabled or disabled.

2.3 Fragment shader operations

Most of the work in volume renderers for consumer graphics hardware is done in the fragment shader, i.e., at the granularity of individual fragments and, ultimately, pixels. In contrast to approaches using lookup tables, i.e., paletted textures, we are performing all shading operations procedurally in the fragment shader. Section 6 contains details about the actual volume shading models we are using. However, we are most of all interested in the operations that are required for rendering segmented data. The two basic operations in the fragment shader with respect to the segmentation mask are fragment rejection and per-fragment application of transfer functions:

Fragment rejection. Fragments corresponding to object IDs that cannot be rendered in the current rendering pass, e.g., because they need a different fragment shader or compositing mode, have to be rejected. They, in turn, will be rendered in another pass, which uses an appropriately adjusted rejection comparison. For fragment rejection, we do not compare object IDs individually, but use 1D lookup textures that contain a binary membership status for each object (figure 5, left). All objects that can be rendered in the same pass belong to the same object set, and the corresponding object set membership texture contains ones at exactly those texture coordinates corresponding to the IDs of these objects, and zeros everywhere else. The re-generation of these textures at the beginning of each frame, which is negligible in terms of performance, also makes turning individual objects on and off trivial. Exactly one object set membership texture is active for a given rendering pass and makes the task of fragment rejection trivial if the object ID volume is point-sampled. When object IDs are filtered, it is also crucial to map individual IDs to zero or one before actually filtering them. Details are given in section 3, but basically we are using object set membership textures to do a binary classification of input IDs to the filter, and interpolate after this mapping. The result can then be mapped back to zero or one for fragment rejection.

Per-fragment transfer function application. Since we apply different transfer functions to multiple objects in a single rendering pass, the transfer function must be applied to individual fragments based on their density value and corresponding object ID. Instead of sampling multiple one-dimensional transfer function textures, we sample a single global two-dimensional transfer function texture (figure 6). This texture is not only shared between all objects of an object set, but also between all object sets. It is indexed with one texture coordinate corresponding to the object ID, the other one to the actual density. Because we would like to filter linearly along the axis of the actual transfer function, but use point-sampling along the axis of object IDs, we store each transfer function twice at adjacent locations in order to guarantee point-sampling for IDs, while we are using linear interpolation for the entire texture. We have applied



object ids
object set membership textures
object id axis
transfer function assignment texture
after lerp and thresholding:
R
G
B
A
= 4

Figure 5: Object set membership textures (left; three 1D intensity textures for three sets containing three, two, and one object, respectively) contain a binary membership status for each object in a set that can be used for filtering object IDs and culling fragments. Transfer function assignment textures (right; one 1D RGBA texture for distinction of four transfer functions) are used to filter four object boundaries simultaneously and determine the corresponding transfer function via a simple dot product.

this scheme only to 1D transfer functions, but general 2D transfer functions could also be implemented via 3D textures of just a few layers in depth, i.e., the number of different transfer functions.

We are using an extended version of the pixel-resolution filter that we employ for fragment rejection in order to determine which of multiple transfer functions in the same rendering pass a fragment should actually use. Basically, the fragment shader uses multiple RGBA transfer function assignment textures (figure 5, right) for both determining the transfer function and rejecting fragments, instead of a single object set membership texture with only a single color channel. Each one of these textures allows filtering the object ID volume with respect to four object boundaries simultaneously. A single lookup yields binary membership classification of a fragment with respect to four objects. The resulting RGBA membership vectors can then be interpolated directly. The main operation for mapping back the result to an object ID is a simple dot product with a constant vector of object IDs. If the result is the non-existent object ID of zero, the fragment needs to be rejected. The details are described in section 4. This concept can be extended trivially to objects sharing transfer functions by using transfer function IDs instead of object IDs. The following two sections will now describe filtering of object boundaries at sub-voxel precision in more detail.

3 Pixel-resolution boundaries

One of the most crucial parts of rendering segmented volumes with high quality is that the object boundaries must be calculated during rendering at the pixel resolution of the output image, instead of the voxel resolution of the segmentation volume. Figure 7 (left) shows that simply point-sampling the object ID texture leads to object boundaries that are easily discernible as individual voxels. That is, simply retrieving the object ID for a given fragment from the segmentation volume is trivial, but causes artifacts. Instead, the object ID must be determined via filtering for each fragment individually, thus achieving pixel-resolution boundaries.

Unfortunately, filtering of object boundaries cannot be done directly using the hardware-native linear interpolation, since direct interpolation of numerical object IDs leads to incorrectly interpolated intermediate values when more than two different objects are present. When filtering object IDs, a threshold value s_t must be chosen that determines which object a given fragment belongs to, which is essentially an iso-surfacing problem. However, this cannot be done if three or more objects are contained in the volume, which is illustrated in the top row of figure 8. In that case, it is not possible to choose a single s_t for the entire volume. The crucial observation to make in order to solve this problem is that the segmentation volume must be filtered as a successive series of binary volumes in order to achieve proper filtering [Tiede et al. 1998], which is shown in the second row of figure 8. Mapping all object IDs of the current object set to 1.0 and all other IDs to 0.0 allows using a global threshold value s_t of 0.5. We of course do not want to store these binary volumes explicitly, but perform this mapping on-the-fly in the fragment shader by indexing the *object set membership texture* that is active in the current rendering pass. Filtering

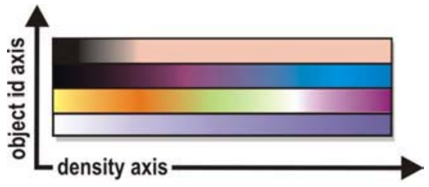


Figure 6: Instead of multiple one-dimensional transfer functions for different objects, we are using a single global two-dimensional transfer function texture. After determining the object ID for the current fragment via filtering, the fragment shader appropriately samples this texture with $(density, object\_id)$ texture coordinates.

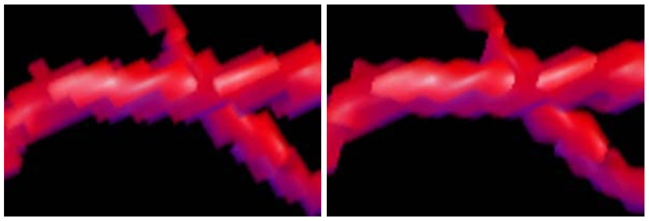


Figure 7: Object boundaries with voxel resolution (left) vs. object boundaries determined per-fragment with linear filtering (right).

in the other passes simply uses an alternate binary mapping, i.e., other object set membership textures. One problem with respect to a hardware implementation of this approach is that texture filtering happens before the sampled values can be altered in the fragment shader. Therefore, we perform filtering of object IDs directly in the fragment shader. Note that our approach could in part also be implemented using texture palettes and hardware-native linear interpolation, with the restriction that not more than four transfer functions can be applied in a single rendering pass (section 4). However, we have chosen to perform all filtering in the fragment shader in order to create a coherent framework with a potentially unlimited number of transfer functions in a single rendering pass and prepare for the possible use of cubic boundary filtering in the future.

After filtering yields values in the range $[0.0, 1.0]$, we once again come to a binary decision whether a given fragment belongs to the current object set by comparing with a threshold value of 0.5 and rejecting fragments with an interpolated value below this threshold (figure 8, third row). Actual rejection of fragments is done using the `KIL` instruction of the hardware fragment shader.

Linear boundary filtering. For object-aligned volume slices, bi-linear interpolation is done by setting the hardware filtering mode for the object ID texture to nearest-neighbor and sampling it four times with offsets of whole texels in order to get access to the four ID values needed for interpolation. Before actual interpolation takes place, the four object IDs are individually mapped to 0.0 or 1.0, respectively, using the current object set membership texture. We perform the actual interpolation using a variant of texture-based filtering [Hadwiger et al. 2001], which proved to be both faster and use fewer instructions than using `LRP` instructions. With this approach, bi-linear weight calculation and interpolation can be reduced to just one texture fetch and one dot product. When intermediate slices are interpolated on-the-fly [Rezk-Salama et al. 2000], or view-aligned slices are used, eight instead of four input IDs have to be used in order to perform tri-linear interpolation.

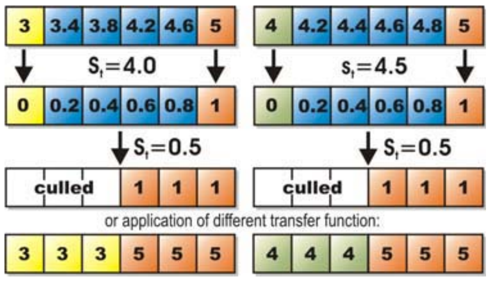


Figure 8: Each fragment must be assigned an exactly defined object ID after filtering. Here, IDs 3, 4, and 5 are interpolated, yielding the values shown in blue. Top row: choosing a single threshold value s_t that works everywhere is not possible for three or more objects. Second row: object IDs must be converted to 0.0 or 1.0 in the fragment shader before interpolation, which allows using a global s_t of 0.5. After thresholding, fragments can be culled accordingly (third row; see section 3), or mapped back to an object ID in order to apply the corresponding transfer function (fourth row; see section 4).

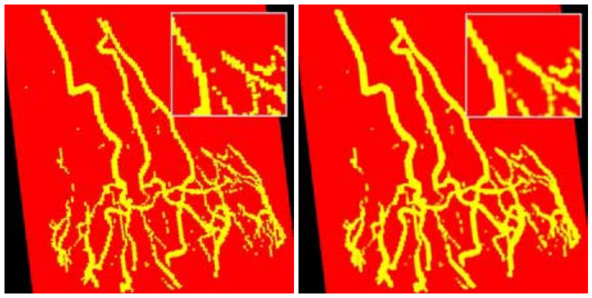


Figure 9: Selecting the transfer function on a per-fragment basis. In the left image, point-sampling of the object ID volume has been used, whereas in the right image procedural linear interpolation in the fragment shader achieves results of much better quality.

Combination with pre-integration. The combination of pre-integration [Engel et al. 2001] and high-quality clipping has been described recently [Röttger et al. 2003]. Since our filtering method effectively reduces the segmentation problem to a clipping problem on-the-fly, we are using the same approach after we have mapped object IDs to 0.0 or 1.0, respectively. In this case, the interpolated binary values must be used for adjusting the pre-integration lookup.

4 Multiple per-object transfer functions in a single rendering pass

In addition to simply determining whether a given fragment belongs to a currently active object or not, which has been described in the previous section, this filtering approach can be extended to the application of multiple transfer functions in a single rendering pass without sacrificing filtering quality. Figure 9 shows the difference in quality for two objects with different transfer functions (one entirely red, the other entirely yellow for illustration purposes).

In this case, we perform several almost identical filtering steps in the fragment shader, where each of these steps simultaneously filters the object boundaries of four different objects. After the fragment's object ID has been determined via filtering, it can be used to access the global transfer function table as described in section 2.3 and illustrated in figure 6. For multiple simultaneous transfer functions, we do not use object set membership textures but the similar extended concept of *transfer function assignment textures*, which is illustrated in the right image of figure 5. Each of these textures can be used for filtering the object ID volume with respect to four different object IDs at the same time by using the four channels of an RGBA texture in order to perform four simultaneous binary classification operations. In order to create these textures, each object set membership texture is converted into $\lceil \#objects/4 \rceil$ transfer function assignment textures, where $\#objects$ denotes the number of objects with different transfer functions in a given object set. All values of 1.0 corresponding to the first transfer function are stored into the red channel of this texture, those corresponding to the second transfer function into the green channel, and so on.

In the fragment shader, bi-linear interpolation must index this texture at four different locations given by the object IDs of the four input values to interpolate. This classifies the four input object IDs with respect to four objects with just four 1D texture sampling operations. A single linear interpolation step yields the linear interpolation of these four object classifications, which can then be compared against a threshold of $(0.5, 0.5, 0.5, 0.5)$, also requiring only a single operation for four objects. Interpolation and thresholding yields a vector with at most one component of 1.0, the other components set to 0.0. In order for this to be true, we require that interpolated and thresholded repeated binary classifications never overlap, which is not guaranteed for all types of filter kernels. In the case of bi-linear or tri-linear interpolation, however, overlaps can never occur [Tiede et al. 1998]. The final step that has to be performed is mapping the binary classification to the desired object ID. We do this via a single dot product with a vector containing the four object IDs corresponding to the four channels of the transfer function assignment texture (figure 5, right). By calculating this dot product, we multiply exactly the object ID that should be assigned to the final fragment by 1.0. The other object IDs are multiplied by 0.0 and thus do not change the result. If the result of the dot product is 0.0, the fragment does not belong to any of the objects under consideration and can be culled. Note that exactly for this reason, we do not use object IDs of zero. For the application of more than four transfer functions in a single rendering pass, the steps outlined above can be executed multiple times in the fragment shader. The results of the individual dot products are simply summed up, once again yielding the ID of the object that the current fragment belongs to. Note that the calculation of filter weights is only required once, irrespective of the number of simultaneous transfer functions, which is also true for sampling the original object ID textures.

Equation 1 gives the major fragment shader resource requirements of our filtering and binary classification approach for the case of bi-linear interpolation with LRP instructions:

$$4\textbf{TEX\_2D} + 4\left\lceil \frac{\#objects}{4} \right\rceil \textbf{TEX\_1D} + 3\left\lceil \frac{\#objects}{4} \right\rceil \textbf{LRP}, \quad (1)$$

in addition to one dot product and one thresholding operation (e.g., DP4 and SGE instructions, respectively) for every $\lceil \#objects/4 \rceil$ transfer functions evaluated in a single pass. Similarly to the alternative linear interpolation using texture-based filtering that we have outlined in section 3, procedural weight calculation and the LRP instructions can once again also be substituted by texture fetches and a few cheaper ALU instructions. On the Radeon 9700, we are currently able to combine high-quality shading with up to eight transfer functions in the same fragment shader, i.e., we are using up to two transfer function assignment textures in a single rendering pass.

5 Separation of compositing modes

The final component of our framework with respect to the separation of different objects is the possibility to use individual object-local compositing modes, as well as a single global compositing mode. The local compositing modes that can currently be selected are alpha blending (e.g., for DVR or tone shading), maximum intensity projection (e.g., for MIP or contour enhancement), and iso-surface rendering. Global compositing can either be done by alpha blending, MIP, or a simple add of all contributions.

Although the basic concept is best explained using an image-order approach, i.e., individual rays (figure 3), in the context of texture-based volume rendering we have to implement it in object-order. As described in section 2, we are using two separate rendering buffers, a local and a global compositing buffer, respectively. Actual volume slices are only rendered into the local buffer, using

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

Table 2: Detecting for all pixels simultaneously where the compositing mode changes from one slice to the next, and transferring those pixels from the local into the global compositing buffer.

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

In order to do this, we are using the depth buffer of both the local and the global compositing buffer to track the current local compositing mode of each pixel, and the stencil buffer to selectively enable pixels where the mode changes from one slice to the next. Before actually rendering a slice (see table 1), we render IDs corresponding to the local compositing mode into both the local and the global buffer's depth buffer. During these passes, the stencil buffer is set to one where the ID already stored in the depth buffer (from previous passes) differs from the ID that is currently being rendered. This gives us both an updated ID image in the depth buffer, and a stencil buffer that identifies exactly those pixels where a change in compositing mode has been detected. We then render the image of the local buffer into the global buffer. Due to the stencil test, pixels will only be rendered where the compositing mode has actually changed. Table 2 gives pseudo code for what is happening in the global buffer. Clearing the just transferred pixels in the local buffer works almost identically. The only difference is that in this case we do not render the image of another buffer, but simply a quad with all pixels set to zero. Due to the stencil test, pixels will only be cleared where the compositing mode has actually changed.

Note that all these additional rendering passes are much faster than the passes actually rendering and shading volume slices. They are independent of the number of objects and use extremely simple fragment shaders. However, the buffer/context switching overhead is quite noticeable, and thus correct separation of compositing modes can be turned off during interaction. Figure 4 shows a comparison between approximate and correct compositing with one and two compositing buffers, respectively. Performance numbers can be found in table 3. When only a single buffer is used, the compositing mode is simply switched according to each new fragment without avoiding interference with the previous contents of the frame buffer. The visual difference depends highly on the combination of compositing modes and spatial locations of objects. The example in figure 4 uses MIP and DVR compositing in order to highlight the potential differences. However, using approximate compositing is very useful for faster rendering, and often exhibits little or no loss in quality. Also, it is possible to get an almost seamless performance/quality trade-off between the two, by performing the buffer transfer only every n slices instead of every slice.

6 Rendering modes, performance

This section provides details on the actual rendering modes we are supporting, as well as some performance figures. We can use object-aligned slices with 2D textures, possibly with slice interpolation [Rezk-Salama et al. 2000], view-aligned slices with 3D textures, and slab instead of slice rendering for pre-integration [Engel et al. 2001]. Gradients can be pre-computed either via central differencing or a 3x3x3 Sobel operator, and are stored into a RGB texture in normalized form for sampling by the fragment shader.

Direct volume rendering. We have implemented both post-classification and pre-integrated classification [Engel et al. 2001]. Both of these modes can either be unshaded or shaded, optionally weighted with gradient magnitude [Levoy 1988].

Iso-surfacing. We support rendering of non-polygonal shaded iso-surfaces via the OpenGL alpha test [Westermann and Ertl 1998] and pre-integrated iso-surfaces [Engel et al. 2001], respectively.

Maximum intensity projection. The maximum intensity of all fragments corresponding to a given pixel can be retained in the frame buffer by using the `GL_MAX` compositing mode. Prior to this, the volume density is mapped through a monochrome transfer function and afterward multiplied by a constant color.

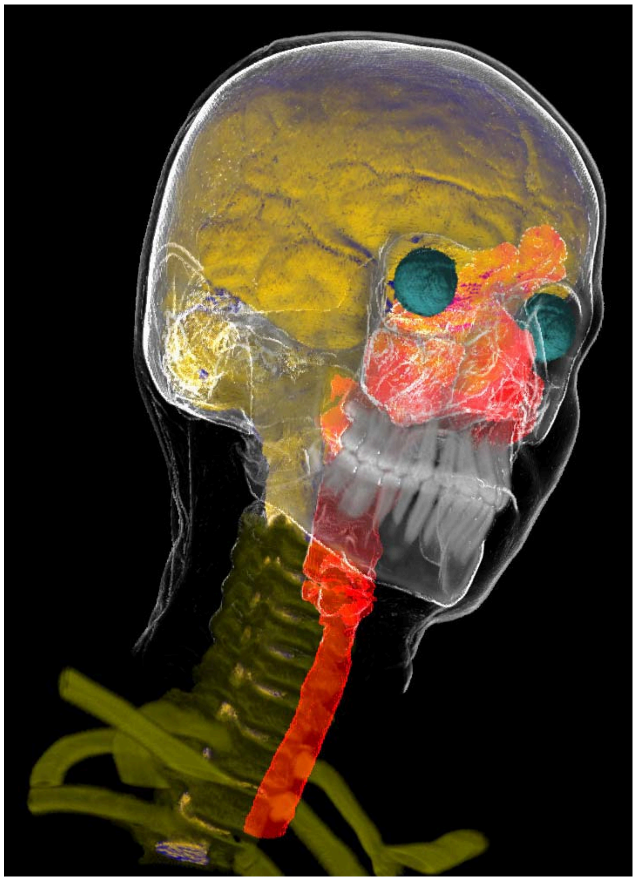


Figure 10: Segmented head and neck data set (256x256x333) with eight different enabled objects – brain: tone shading; skin: contour enhancement with clipping plane; eyes and spine: shaded DVR; skull, teeth, and vertebrae: unshaded DVR; trachea: MIP.

Contour enhancement. As one of two non-photorealistic rendering modes, we have adopted a contour shading model [Csebfalvi et al. 2001]. The intensity of a fragment is determined procedurally in the fragment shader by evaluating equation 2:

$$I = g\big(|\nabla|\big) \cdot \big(1 - |V \cdot \nabla|\big)^8, \tag{2}$$

where V is the viewing vector, ∇ denotes the gradient of a given voxel, and $g()$ is a windowing function for the gradient magnitude. We specify $g()$ through the usual transfer function interface, where the alpha component is the weighting factor for the view-dependent part, and the RGB components are simply neglected. The fragment intensity can be multiplied by a constant contour color; fragment alpha is set equal to I. The compositing mode for contours is MIP.

Tone shading. In order to enhance depth perception, we are also using tone shading [Gooch et al. 1998] adapted to volumes:

$$I = \left(\frac{1 + L\cdot\nabla}{2}\right)k_a + \left(1 - \frac{1 + L\cdot\nabla}{2}\right)k_b, \tag{3}$$

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

$$k_a = k_{cool} + \alpha k_t \tag{4}$$
$$k_b = k_{warm} + \beta k_t \tag{5}$$

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

Table 3: Performance on ATI Radeon 9700; 512x512 viewport; 256x128x256 data set; three and eight enabled objects, respectively. Numbers are in frames per second. Compositing is done with either one or two buffers, respectively. The *multi* column with early z-testing turned off is only shown for comparison purposes.

Performance. Actual rendering performance depends on a lot of different factors, so table 3 shows only some example figures. In order to concentrate on performance of rendering segmented data, all rates have been measured with unshaded DVR. Slices were object-aligned; objects were rendered all in a single pass (*single*) or in one pass per object (*multi+ztest*). Compositing performance is independent of the rendering mode, i.e., can also be measured with DVR for all objects. Frame rates in parentheses are with linear boundary filtering enabled, other rates are for point-sampling during interaction. Note that in the unfiltered case with a single rendering pass for all objects, the performance is independent of the number of objects. If more complex fragment shaders than unshaded DVR are used, the relative performance speed-up of *multi+ztest* versus *multi* increases further toward *single* performance, i.e., the additional overhead of writing object set IDs into the depth buffer becomes negligible.

7 Conclusions and future work

We have shown how segmented volumes can be rendered at interactive rates with high quality on current consumer graphics hardware such as the ATI Radeon 9700. The segmentation mask is filtered on-the-fly in the fragment shader, which provides greater flexibility and facilitates using higher-order filtering in the future. In general, we are expecting a move toward programmable filtering via procedural or texture-based filter kernels for a lot of applications in the near future. This has just become possible on the most recent architectures, but as instruction count limits in the fragment shader rise or are even removed, these approaches are rapidly becoming feasible. All the algorithms we have presented are meant to take the possibilities of future hardware into account. The current restriction to eight simultaneous transfer functions with object ID filtering is solely due to the instruction count limit of our target hardware, and increases trivially with more instructions. In general, we ensure a minimal number of rendering passes by only falling back to individual passes for changes of the hardware configuration where an on-the-fly adaptation is currently impossible, i.e., the fragment shader and compositing mode. When fragment shader adaptation on a per-pixel basis becomes possible, our framework will require only minor changes. In the future, we would like to incorporate cubic boundary filtering [Hadwiger et al. 2001], and extend the combination with pre-integrated classification [Röttger et al. 2003] for application of multiple pre-integrated transfer functions in a single rendering pass. In addition or as an alternative to the early z-test, texture hulls [Li and Kaufman 2002] could be used to minimize the number of fragments that are rendered without any contribution.

Acknowledgments

We would like to thank Christof Rezk-Salama, Thomas Theußl, Klaus Engel, Matej Mlejnek, André Neubauer, Lukas Mroz, Ivan Viola, Torsten Möller, Joe Kniss, and Rüdiger Westermann for much-appreciated contributions and discussions, and Michael Doggett from ATI for providing a Radeon 9700. The data sets we have used are courtesy of Tiani Medgraph. This work has been done in the basic research on visualization at the VRVis Research Center, which is funded by the Austrian Kplus project.

References

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, 91–98.

CSEBFALVI, B., MROZ, L., HAUSER, H., KÖNIG, A., AND GRÖLLER, M. E. 2001. Fast visualization of object contours by non-photorealistic volume rendering. In *Proceedings of EUROGRAPHICS 2001*, 452–460.

CULLIP, T. J., AND NEUMANN, U. 1993. Accelerating volume reconstruction with 3D texture mapping hardware. Tech. Rep. TR93-027, UNC, Chapel Hill.

EBERT, D., AND RHEINGANS, P. 2000. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of IEEE Visualization 2000*, 195–202.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Graphics Hardware 2001*, 9–16.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of SIGGRAPH '98*, 447–452.

HADWIGER, M., THEUSSL, T., HAUSER, H., AND GRÖLLER, E. 2001. Hardware-accelerated high-quality filtering on PC hardware. In *Proceedings of Vision, Modeling, and Visualization 2001*, 105–112.

HAUSER, H., MROZ, L., BISCHI, G.-I., AND GRÖLLER, E. 2001. Two-level volume rendering. *IEEE Transactions on Visualization and Computer Graphics 7*, 3, 242–252.

KINDLMANN, G., AND DURKIN, J. 1998. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of IEEE Volume Visualization '98*, 79–86.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2001. Multi-dimensional transfer functions for interactive volume rendering. In *Proceedings of IEEE Visualization 2001*, 255–262.

KRÜGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization '03*.

LACROUTE, P., AND LEVOY, M. 1994. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH '94*, 451–458.

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

LI, W., AND KAUFMAN, A. 2002. Accelerating volume rendering with texture hulls. In *Proceedings of IEEE VolVis 2002*, 115–122.

LUM, E. B., AND MA, K.-L. 2002. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of NPAR 2002*.

LU, A., MORRIS, C., EBERT, D., RHEINGANS, P., AND HANSEN, C. 2002. Non-photorealistic volume rendering using stippling techniques. In *Proceedings of IEEE Visualization 2002*, 211–218.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of Graphics Hardware 2000*.

RÖTTGER, S., GUTHE, S., WEISKOPF, D., ERTL, T., AND STRASSER, W. 2003. Smart hardware-accelerated volume rendering. In *Proceedings of VisSym 2003*, 231–238.

TIEDE, U., SCHIEMANN, T., AND HÖHNE, K. H. 1998. High quality rendering of attributed volume data. In *Proceedings of IEEE Visualization '98*, 255–262.

UDUPA, K. K., AND HERMAN, G. T. 1999. *3D Imaging in Medicine*. CRC Press.

WEISKOPF, D., ENGEL, K., AND ERTL, T. 2003. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 298–312.

WESTERMANN, R., AND ERTL, T. 1998. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, 169–178.