Accelerated Volume Ray-Casting using Texture Mapping

Rüdiger Westermann, Bernd Sevenich

Scientific Visualization & Imaging Group
University of Technology Aachen*

Abstract

Acceleration techniques for volume ray-casting are primarily based on pre-computed data structures that allow one to efficiently traverse empty or homogeneous regions. In order to display volume data that successively undergoes color lookups, however, the data structures have to be re-built continuously. In this paper we propose a technique that circumvents this drawback using hardware accelerated texture mapping. In a first rendering pass we employ graphics hardware to interactively determine for each ray where the material is hit. In a second pass ray-casting is performed, but ray traversal starts right in front of the previously determined regions. The algorithm enables interactive classification and it considerably accelerates the view dependent display of selected materials and surfaces from volume data. In contrast to other techniques that are solely based on texture mapping our approach requires less memory and accurately performs the composition of material contributions along the ray.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - color, shading and texture; I.3.8 [Computer Graphics]: Applications.

Additional Keywords: Volume Rendering, Ray-Casting, Texture Mapping, Visualization, Graphics Hardware.

1 Introduction and related work

The efficient generation of a visual representation of volumetric data sets has been studied extensively during the last decades. Once the fundamental equation describing the physics of light transport in participating media was identified as the key to volume rendering many practical solutions based on simplified models neglecting scattering and frequency effects or assuming homogeneous material were developed [1, 12, 15, 21, 32, 26, 31]. However, due to the huge number of volume elements to be processed and the numerical complexity of the operations to be performed interactive volume rendering is still a challenge to the computer graphics community.

Considerable effort has been spent on the development of acceleration techniques for indirect and direct volume rendering, and on the design and exploitation of dedicated hardware to achieve interactive frame rates. The majority of optimization strategies for direct volume rendering takes advantage of pyramidal data structures to directly encode empty or homogeneous regions and to effectively reduce the number of sample points based on some global error metric [19, 17, 5, 36, 9, 29]. Alternative techniques rely on fast cell traversal algorithms [11] and exploit optimized data layout strategies [3, 23] as well as auxiliary data structures to spatially localize the relevant features. In particular, the effectiveness of flat pyramids and proximity clouds to skip empty and homogeneous regions were demonstrated in [35, 8], bounding cells and geometries were used in [27, 28] to efficiently determine the first appearance of the structures to be displayed along the view rays, and in [16] the rendering process was accelerated considerably by run-length encoding of empty space.

Although these approaches differ significantly in terms of the underlying methodology and in the kind of structures they are able to display, they all rely on the classification of features in the data by means of any suitable algorithm to be performed in a pre-processing step. Then the pyramidal or auxiliary representation is built that allows for the efficient assembly of the material contribution along the rays of sight. Among others, popular classification schemes used in volume rendering applications include zero (iso-values) [20], first (gradients) [18] or higher order [6, 13] statistics of the scalar field.

The algorithm proposed in [18], for example, assigns color and opacity values to each voxel so as to enhance selected materials and boundary regions around them. The volumetric structures can now be encoded using any appropriate data structure that enables the rendering algorithm to efficiently skip non-classified material. In [27] the classification process was accompanied by a surface fitting step generating polygonal models that entirely enclose the classified structures. Prior to volume rendering an initial starting point for ray traversal can then be determined by intersecting the geometric representation.

Pyramidal or intermediate data structures as described, however, impose certain limitations on the volume rendering algorithm. In particular the data structures have to be re-built if the classification is going to be changed. On the other hand, interactive classification based on color lookup tables has been proven to be a very powerful mechanism to explore large-scale volumetric data sets. By means of continuous modifications of the assignment of scalar values to material characteristics arbitrary structures within the data can be enhanced, suppressed or shaded in a particular way. Unfortunately, interactivity can no longer be achieved if auxiliary data structures are used. In general this is due to the computational complexity of the process used to re-build the data structures whenever the classification is changed.

A potential alternative that overcomes the mentioned drawback is hardware accelerated volume rendering. Just recently, different approaches have been proposed that allow for interactive direct volume rendering by taking advantage of hardware assisted 2D and 3D texture mapping as well as dedicated volume graphics architectures [2, 33, 4, 30, 25, 10, 14, 24, 22].

Although impressive frame rates can be achieved, the most serious limitation imposed by texture based techniques is the additional

*University of Technology, Aachen, Seffenter Weg 23, 52056 Aachen, Germany, Phone:+49-(0)241-80-28920, Fax:+49-(0)241-8888-241, e-mail: westermann@sc.rwth-aachen.de

amount of memory necessary to store gradient information if lighting calculations have to be performed. Furthermore, OpenGL compositing as it is used on current graphics accelerators introduces artifacts due to separate interpolation of color and opacity and it generates less accurate results due to the limited precision of the internal pixel and texture formats. Dedicated volume graphics architectures, on the other hand, are not yet ready to enter the consumer market. This is due to the price that still makes these systems not affordable for the mass market, and due to the fact that an interface to standard graphics APIs like OpenGL is not yet available.

In this paper we outline a new approach for the direct volume rendering of high-resolution data that enables interactive classification on graphics units suitable for the consumer market. Rather than proposing a particular software or hardware solution we outline a hybrid approach that exploits the rasterization capability of graphics hardware but also takes advantage of CPU performance. The goal of our approach is twofold: to emphasize the impact of affordable state-of-the-art graphics hardware on current volume visualization techniques *and* to demonstrate how it can be used to accelerate volume ray-casting. In particular we will show that texture based volume rendering can be used to interactively classify scalar volume data by means of texture color tables, and we will demonstrate that the rendered image can be re-used to accelerate volume ray-casting which is implemented in software. Because in our work we are targeting the consumer market we restrict our attention to graphics units that are affordable and let us allow to exploit hardware features through standard APIs like OpenGL. In this respect we decided to implement our algorithm on the nVidia GeForce family GPU, which provides the functionality we need and which supports a set of additional features we are effectively taking advantage of.

The reminder of this paper is organized as follows. First, we review texture based volume rendering techniques and we demonstrate that 2D textures can be effectively used to interactively classify the sampled scalar field and to accurately resample the result. Next, we present our technique for accelerated volume ray-casting in which hardware supported graphics operations are paramount. We finally discuss implementation details and further improvements. We conclude the paper with a detailed discussion, and we show results and timings of our approach applied to reasonably sized data sets.

2 Texture based volume rendering

During the last couple of years volume rendering techniques that exploit hardware support for texture mapping have become a powerful tool to interactively display and thus analyze 3D scalar fields.

In [2, 33, 4] the general idea to interpret volume rendering as the resampling of a discrete 3D texture map on appropriately oriented geometries was first introduced. Using this approach the visual impression conveyed by semi-transparent media can be efficiently simulated on any graphics system supporting 3D texture interpolation and per-pixel blending operations.

Recently an alternative technique was proposed that solely relies on 2D texture mapping, and which exploits advanced per-fragment operations provided by the nVidia GeForce family GPUs for speed-up and lighting effects [25]. The voxel data is decomposed into three stacks of 2D textures as illustrated in Figure 1. For the current view, the object space axis that is most parallel to the viewing axis now determines the appropriate stack to be rendered.

Although today hardware support for 3D texture mapping is effectively available on a couple of low-cost graphics systems, e.g. Wildcat 4120, ATI Radeon and GeForce3, we will exclusively focus on 2D texture based techniques in this paper. We do so because of two reasons. First, a considerable loss in performance can be noticed when 3D texture mapping is employed. Compared to 2D texture mapping our experiments have shown a decrease in fill rate of up to 60%, which is mainly due to the highly irregular memory access pattern that can't be realized as efficient as random 2D texture access. Second, by using 2D texture maps our approach can be run without a considerable decrease in performance on any current graphics board.

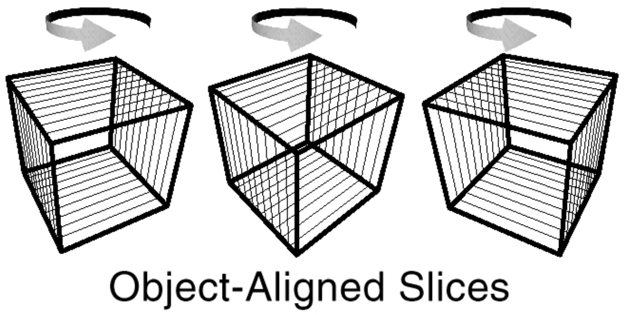

Figure 1: Three different stacks of textures need to be generated in order to perform volume rendering via 2D textures.

The design of our algorithm, on the other hand, allows for the integration of any texture based volume rendering technique as long as the view dependent resampling of the scalar field can be guaranteed. At the core of our approach we are independent of a particular texture representation, and thus we can use the algorithm without any modifications if this representation is changed.

Let us now outline some extensions to 2D texture based volume rendering by which an accurate resampling of the sampled scalar field can be achieved. The technique we propose won't be used to display the volumetric scalar field but to generate an intermediate image of the data in a first rendering pass. From the depth values of each pixel in this image the accurate position in object space of the structures to be displayed can be computed. This information is used further on to accelerate the final rendering pass. The intermediate image thus serves as a 3D stencil which drives the rendering process.

2.1 View independent 2D texture resampling

In contrast to volume ray-casting where the sample distance along the ray is independent of the current view this no longer holds for volume rendering via 2D textures (see Figure 2). For a certain pixel the distance between successive samples increases with respect to the angle between the most parallel object space axis and the viewing direction. As a consequence small features in the data might be missed during the texture based rendering pass, which is only performed to appropriately resample the scalar field but not to display the data. Thus it is not sufficient to simply correct the opacity of each slice with respect to varying sample distance, which obviously doesn't give us any additional information about the material distribution between successive sample points.

View independent sampling intervals can be realized by adding new texture slices appropriately positioned between the original ones as shown in the rightmost image of Figure 2. For each new view V the offset d_V is computed and new slices at positions $k \cdot d_V$ are generated and rendered.

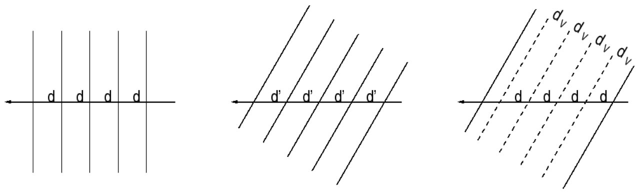

Figure 2: View dependent and view independent data resampling.

In [25] a technique was proposed to linearly interpolate between two textures in a single rendering pass. At the core of this algorithm multitextures and the nVidia texture combiners were employed to combine texture samples on a per-fragment basis. The results of two independent texturing operations that are performed in two separate texture units can be modulated and finally blended to get the fragments color.

In our implementation the interpolation factors needed to generate a new slice by linear interpolation between two original textures are issued as separate colors to be used in each texture unit. These colors modulate the multitexture samples before blending is performed. Figure 3 exemplifies the principal setup and the per-fragment operations which are employed.

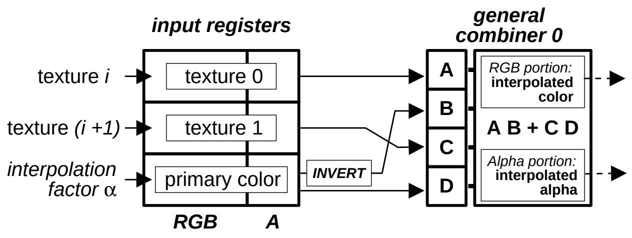

Figure 3: Texture combiner setup for 2D texture interpolation.

Using this approach a constant sampling distance can be ensured for parallel projections and arbitrary views. For perspective projections, however, the sampling distance increases towards the outer regions of the field of view. On the other hand, because the inter-slice distance can be decreased arbitrarily, it can be chosen so that an upper bound for the sampling intervals as they occur under the current view is never going to be exceeded. As a matter of fact the proposed technique is also suitable for the resampling of scalar fields that are displayed under perspective projections.

2.2 Material culling

One of the key features provided by many graphics architectures are texture color tables that allow for the direct manipulation of transfer functions used to perform the mapping from scalar values to RGBα values. Texture samples might be mapped to color and opacity values before or after texture interpolation. On the nVidia chip, for example, texture elements are first looked up in the table before they are going to be interpolated on a per-fragment basis during the texture operation.

By means of color tables meaningful mappings of material values to visual quantities can be found interactively by the user without that the texture has to be reloaded. Arbitrary assignments can be realized which allow one to color structures differently or to discard particular ranges of values. Although this classification scheme only takes into account the distribution of the scalar values and the visual representation of the displayed structures it has been proven to be a very effective and efficient mechanism to interactively explore large volumetric data sets. In particular this holds if direct visual feedback to the color table modification is possible, e.g. in volume rendering via texture maps and on dedicated volume graphics architectures.

As proposed in [30] texture color tables can effectively be utilized in texture based volume rendering to cull materials which should not be visualized. Therefore the color lookup is performed in such a way as to make these materials transparent, i.e. scalar values outside the selected range get assigned an alpha component equal to zero during the mapping. Now the generated fragments can be discarded by means of the OpenGL alpha test before they are going to be drawn into the color buffer. By enabling the appropriate comparison function only non-transparent fragments with an alpha value greater than zero will be accepted and rendered.

In Figure 4 some examples demonstrate the effects that can be achieved. The original data set was rendered using different texture color tables, which were properly initialized in order to cull certain ranges of values.

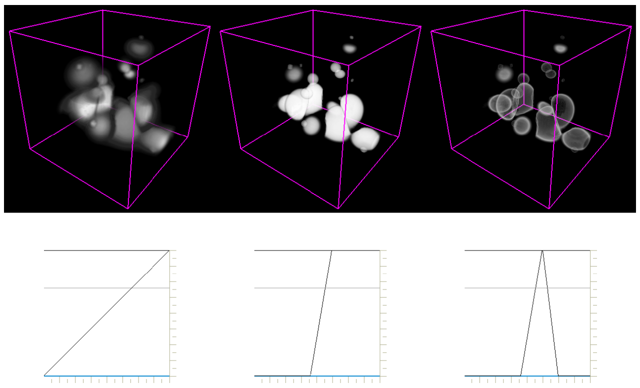

Figure 4: The data set on the left was rendered using different texture color tables.

In the next section we will demonstrate how to use the intermediate image generated via 2D texture maps to accelerate volume ray-casting which we implemented in software. Although our hybrid approach cannot be as efficient as the texture based volume rendering approach it introduces two major improvements which are of particular interest in practical applications. First, considerably less memory is needed for the rendering of illuminated structures. Second, image quality doesn't suffer from insufficient internal texture and pixel resolution and separate interpolation of color and opacity.

3 Hybrid volume rendering

In the approach proposed so far we employ hardware supported 2D texture mapping and OpenGL per-fragment operations to interactively classify arbitrary materials by means of texture color tables. Although this approach allows for quite impressive frame rates, the use of textures for direct volume rendering introduces severe drawbacks with regard to memory requirement and image quality.

- Gradient maps as introduced in [30] for the display of non-polygonal illuminated iso-surfaces lead to a significant increase in memory usage. In addition to the original data set an RGBα texture storing gradient components and sampled scalar values has to be generated. This overhead becomes even worse if 2D textures are employed and for each stack of textures an additional stack of gradient maps has to be created.

- On current low-cost graphics units the internal texture resolution is restricted to 8 bits per voxel. As a consequence scalar data samples are usually quantized and differences between structures showing similar scalar values might be lost. The same argument holds for the internal resolution of the color buffer.

- For the direct rendering of semi-transparent objects separate interpolation of pre-shaded color and opacity samples during the texturing operation introduces color bleeding artifacts as demonstrated in [34]. Although on our current target architecture this drawback can be circumvented by means of dependent textures [7], this implementation requires both an additional texture unit and a texture fetch operation.

- It is still rather cumbersome and often impossible to implement advanced lighting models, i.e. the Phong lighting model, by means of the extended OpenGL functionality available on the nVidia chip set. Quite often texture units have to be wasted for computing approximate solutions although preferably they should be kept free for other tasks. The use of vendor specific functionality like texture shaders, on the other hand, makes the code dependent on the current platform and prohibits portability.

In order to avoid the mentioned limitations we propose a hybrid volume rendering approach that combines hardware supported texture mapping and volume ray-casting. In the outlined scenario texture rendering is only used to determine the classified regions. From this point on ray-casting is performed in software on the original data set.

3.1 Space leaping

We proceed our description with a short summary of some of the core features of OpenGL that will be exploited to skip those regions that have been culled by means of texture color tables as proposed.

Upon passing the alpha test but before a fragment is going to be rendered into the color buffer the OpenGL depth test is applied. The depth buffer usually stores for each pixel the distance from the viewpoint to the closest object covering that pixel. The distance of incoming fragments is first compared to the current depth value, and based on the outcome of the comparison the fragment is discarded or rendered.

Since in volume rendering we are not only interested in those structures that are closest to the viewpoint the data is usually displayed with disabled depth test or in back-to-front order. Using the latter approach the depth buffer keeps track of the distance to the textured slice that was rendered last.

However, because the alpha test already discards fragments before the depth test is applied, values in the depth buffer now correspond to the closest structures that got assigned an opacity value greater than zero during texture lookup.

Hence for each pixel the object space coordinate of the closest structure can be obtained by retrieving the depth value and by transforming the screen space coordinate back into world space. From world space coordinates we can easily map into local object coordinates from which the parametric ray equation in object coordinates can be computed as well. In particular the following transformation is applied to each (x_s, x_s, z_s) screen space coordinate:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \mathbf{S}(\mathbf{PM})^{-1}\mathbf{V} \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}$$

In this equation (x, y, z, w) specify homogeneous world space coordinates, P and M are the projection matrix and the modelview matrix, respectively, V specifies the viewport transformation and S captures the final transformation that appropriately maps world space coordinates to local object coordinates. Here we assume that the object is axis aligned and centered around the origin. P, M and V are queried using OpenGL calls, and S is initialized as

$$\begin{pmatrix} (s_x - v_x)/s_x & 0 & 0 & s_x - v_x \\ 0 & (s_y - v_y)/s_y & 0 & s_y - v_y \\ 0 & 0 & (s_z - v_z)/s_z & s_z - v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The components of $\vec{s}$ and $\vec{v}$ are each equal to half the length of the volume and half the length of a voxel in the $x-, y-$ and $z-$direction.

Finally, after screen space coordinates have been transformed, for each ray we obtain the position on that ray where non-transparent media is hit for the first time. All material along the ray that has been culled using the alpha test can be skipped effectively by letting the ray start at that position.

The proposed technique can also be utilized to determine up to which position along each ray the traversal has to be performed. So far, if a ray enters material exhibiting low opacity and early-ray termination cannot be applied, this ray is always going to be traversed until it leaves the volume. However, by just slightly modifying the current procedure the position of the last sample point along the ray that is inside non-transparent material can be determined as well. Therefore the texture stack is rendered again in front-to-back order but the depth comparison function is reversed. Now those fragments farest away from the view point are retained and stored in the depth buffer. Although in this case the object has to be rendered twice via 2D textures and also the depth buffer has to be read for a second time, we can effectively save a considerable amount of time because the number of samples along each ray that need to be evaluated is significantly reduced.

3.2 Ray traversal

Once the first hit along each ray has been determined ray traversal starts right in front of that position. For each ray we also compute the distance t from that position to the entry point $\vec{e}$ of that ray into the volume. The first sample is then positioned at location $\vec{e} + (\lfloor d/s \rfloor \cdot s)\vec{d}$, where s specifies the unique sampling distance used during ray-traversal (see Figure 5) and d is the normalized ray direction. In this way we considerably reduce sampling patterns that arise if the first sample position is always located in one of the axis-aligned slices.

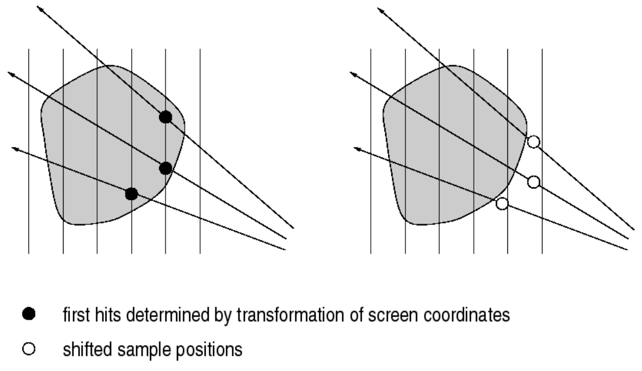
● first hits determined by transformation of screen coordinates
○ shifted sample positions

Figure 5: This image illustrates the shift towards the viewpoint of the first sample point.

For each sample along the ray the material value is reconstructed from the values at adjacent grid points by tri-linear interpolation. At this point it is important to employ the same procedure for the interpolation of alpha values as it is performed by the graphics hardware during texture lookup. As a matter of fact we interpolate pre-shaded scalar values during ray-casting in the current implementation. We also approximate the gradient with a tri-linear interpolation between discrete grid points. Therefore central differences are computed first to approximate the gradients at each grid point. The entire procedure is performed at each sample along the ray, which obviously increases the numerical complexity of our algorithm but allows us to avoid the storage of the gradient map.

To prevent color interpolation artifacts, opacity-weighted color samples are composited along the ray in front-to-back order as follows:

$$C_{new} = (1 - \alpha_{front})\tilde{C}_{back} + \tilde{C}_{front}$$

$$\alpha_{new} = (1 - \alpha_{front})\alpha_{back} + \alpha_{front}$$

According to the notation used in [34], $\tilde{C}$ specifies opacity-weighted colors that are generated by first multiplying sample colors with their opacities before interpolation is performed. Ray traversal finally stops once an opacity threshold is reached and further contributions along the ray become negligible.

We should mention here that in general the step size used during ray traversal doesn't have to be equal to the distance between consecutive texture slices rendered in the first pass. Whereas the latter one determines the accuracy by which the first hit with non-transparent material will be detected, according to the sampling theorem the former one is chosen to be equal to half of the voxel size in our examples.

4 Results

A pleasant feature of the presented approach is that the intermediate image generated by the texture based rendering pass already provides an overall impression of the volumetric object. We take this observation into account in that we exclusively display the result of the first rendering pass during motion and whenever the texture color table is modified. In both situations the ray-casting module is simply skipped (see Figure 6), which allows for interactive classification and navigation before the final rendering pass is initiated.

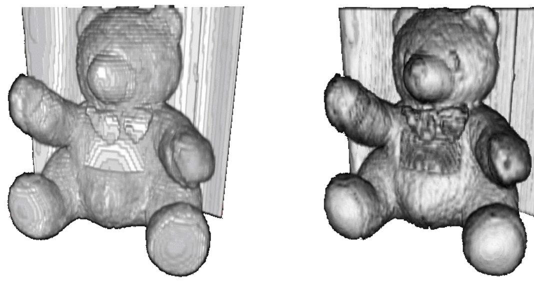

Figure 6: Both images show the same data set; once as it is displayed during updates and once as it is finally rendered.

The appropriate choice of the transfer function has a very particular importance in our approach. In the first rendering pass the transfer function is used for interactive classification that allows one to efficiently discard particular regions in the data. On the other hand, the transfer function is also used to shade the material, but now it is applied during ray-casting. For example, steep edges in the transfer function yield opaque, MC-like iso-surfaces, whereas slowly rising transitions result in smooth silhouettes exhibiting a certain thickness. As a consequence we need two separate transfer functions in general: one that only affects the mapping of 8 bit texture values to alpha values during texture lookup, and one that specifies the final mapping of the original scalar values to color values during ray-casting. For example, if the original data is represented with more than 8 Bits per voxel we can only perform a less accurate classification on the quantized data. The accurate result is then generated in the second pass where we can deal with arbitrary resolutions. In all our current examples, however, the data was given at 8 bits per sample and the same transfer function was used for culling and shading.

All our results were computed on a Pentium III processor running at 733 MHz and a GeForce2 graphics unit with 32 MB local memory. Our experiments were run on three data sets: (a) a human head CT-scan (128^3), (b) the well known engine block (256^2x128) and (c) an aneurysm MRI-scan (256^3). Below we will give exact timings for the relevant parts of our algorithm. In particular we measured the time required to render the data sets via 2D textures (*TexRnd*), to read the depth values necessary to compute screen space coordinates for each pixel (*ZbOps*) and to perform ray-casting (*RayCast*) until an opacity cutoff of 95% was reached. In the first column of Table 1 we also give the average number of samples per ray (#*Smp*) for which the interpolation of color and/or gradients between discrete grid points was performed. In all our experiments the image size was 512x512.

Table 1: Timings (seconds) for different parts of our algorithm. The corresponding images are shown in the color plate below (see Figure 7).

	#Smp	TexRnd	ZbOps	RayCast	Total
(a)	1/6.5	0.05	0.11	0.67/3.90	0.83/4.06
(b)	1/10.2	0.08	0.11	0.52/5.41	0.71/5.60
(c)	1/7.2	0.26	0.11	0.47/3.21	0.84/3.58

Each data set was rendered using two different rendering modes. In the first image column a transfer function was issued so as to discard all fragments with an alpha value less than a specific threshold. Each ray was then traversed until it hit a cell in which the data range was containing the threshold. Only then the intersection of the ray with the corresponding iso-surface passing through that cell was calculated as proposed in [23]. The sample was finally shaded using the approximated gradient at the intersection point and ray traversal was stopped.

Smooth transitions from transparent to opaque material are shown in the second image column. In all images shading was performed and performance speed-up was mainly achieved due to early-ray termination. In the last row the benefits the proposed algorithm can be clearly recognized. Although the data set includes rather fuzzy structures which would make the construction of a pyramidal data structure a rather cumbersome task, the complexity of the vascular structures doesn't affect the performance of our approach. Even if the vessels had low transparency and early ray termination could not be applied, for most of the rays only a small number of samples would have to be computed because the method described in section 3.1 would give us the exact position of the first and the last intersection point with any of the tube like structures.

Because the time needed to access the depth buffer and to read the depth values into main memory only depends on the image resolution it remains constant throughout all of our experiments. Although non-optimized depth buffer access on our current target architecture results in rather poor performance we can still achieve multiple frames per second for the first rendering pass via 2D textures. In particular this allows for interactive classification by means of texture color tables without the need to re-built any auxiliary data structures. This is a considerable improvement over other techniques, for example the one proposed in [28], where it took roughly 10 seconds to prepare the data structure necessary to render a 256^2x124 data set.

Concerning the performance of our ray-casting module we should mention here that gradients are always approximated on-the-fly by central differences and tri-linear interpolation. Opacity-weighted color samples are computed from pre-shaded texture samples, but no further optimization strategies are used. The time required to transform screen space coordinates into local object coordinates is included in *RayCast*.

5 Conclusion

In this paper we have emphasized a novel approach to accelerate the display of 3D scalar fields sampled on Cartesian grids. The major contribution here is that we effectively take advantage of hardware assisted volume rendering to generate an intermediate image that is used as a basis for volume ray-casting to be performed in software. Interpreting the intermediate image allows us to immediately determine those positions where the rays of sight enter non-transparent material for the first time. In this way empty space between the observer and the material can be skipped efficiently.

In particular we have shown that our approach enables interactive classification via texture color tables without the need to modify or to re-built any auxiliary data structures. This makes the method superior to others in case that the classification is going to be changed frequently.

However, even if no such changes occur and auxiliary data structures could be employed to accelerate the ray-casting procedure we still expect our method to be at least as efficient as any alternative as long as only a few samples per ray have to be computed until early ray termination applies. This is due to the fact that the overhead that is introduced by our method for determining the first intersection points is extremely small and can hardly be beaten by any other approach. On the other hand, if expansive homogeneous structures with high transparency are contained in the data our method fails and a pyramidal approach will give much better results.

In contrast to other techniques we can select any desired accuracy by which the data is going to be resampled and consequently by which the first hits are going to be determined without that the internal texture representation has to be changed. But even more importantly, the complexity of the first rendering pass does not depend on the complexity of the structures contained in the data. Thus rendering time and memory requirement remain constant independent of the current classification.

Our method is easy to implement and can be put on top of any texture based volume rendering technique without that the core implementation needs to be modified. However, a drawback compared to common acceleration techniques is the additional amount of memory we need to generate the texture representation. On the other hand, because we entirely avoid the use of gradient maps considerably less texture memory is needed in contrast to texture based volume rendering techniques. Furthermore we can take advantage of all the benefits that are inherent to a software based ray-caster, i.e. opacity-weighted color interpolation, arbitrary lighting models, early-ray termination and accurate data resampling.

We are currently trying to further improve our approach with respect to the following issues. On the GeForce3 we will take advantage of hardware accelerated 3D texture mapping and we will investigate the speed-up due to optimized memory transfer of compressed depth values. In addition we will accelerate the ray-casting module in two stages. First, we will exploit memory coherences by re-organizing the data into smaller tiles that entirely fit into cache lines (see [23]). Second, we will use the SSE instruction set to optimize numerical computations that have to be performed during ray traversal, i.e. tri-linear interpolation. Overall we expect an increase in performance of about a factor of 3 to 4 by means of these extensions.

6 Acknowledgement

In particular, the authors would like to thank C. Reszk-Salama for providing the volume rendering tool in which our approach was integrated, and for the generous use of Figure 3 and the Teddybear data set.

References

[1] J. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics, Proc. SIGGRAPH '82*, 16(3):21–30, July 1982.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, pages 91–98, 1994.

[3] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings IEEE Visualization '97*, pages 235–244, 1997.

[4] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[5] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *ACM Workshop on Volume Visualization '92*, pages 91–98, 1992.

[6] B. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[7] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, 2001.

[8] J. Freund and K Sloan. Accelerated volume rendering using homogeneous region encoding. In *Proceedings IEEE Visualization '97*, pages 191–197, 1997.

[9] M. Gross, L. Lippert, A. Dreger, and R. Koch. A new method to approximate the volume rendering equation using wavelets and piecewise polynomials. *Computers and Graphics*, 19(1), 1995.

[10] T. Günther, C. Poliwoda, C. Reinhart, R. Hesser, R Männer, H.-P. Meinzer, and H.-J. Baur. Virim: A massively parallel processor for real-time volume visualization in medicine. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, pages 705–710, 1995.

[11] Amanatides J. and Woo A. A fast voxel traversal algorithm for ray tracing. In *EUROGRAPHICS '87*, pages 202–210, 1987.

[12] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *ACM Computer Graphics, Proc. SIGGRAPH '84*, 18(3):165–174, July 1984.

[13] G. Kindlemann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings ACM Symposium on Volume Visualization '98*, pages 79–87, 1998.

[14] G. Knittel and W. Strasser. Vizard-visualization accelerator for realtime display. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, pages 139–146, 1997.

[15] W. Krüger. The application of transport theory to the visualization of 3-d scalar data fields. In *Proceedings IEEE Visualization '90*, pages 273–280, 1990.

[16] P. Lacroute and M Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. *Computer Graphics, Proc. SIGGRAPH '94*, 28(4):451–458, 1994.

[17] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM Computer Graphics, Proc. SIGGRAPH '93*, 25(4):285–288, July 1991.

[18] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, March 1988.

[19] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[20] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (SIGGRAPH 87 Proceedings)*, pages 163–169, 1987.

[21] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *Proceedings ACM Workshop on Volume Visualization 91*, pages 27–33, 1991.

[22] M. Meißner, U. Kanus, and W. Straßer. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, pages 61–68, 1998.

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

[24] H. Pfister, J. Hardenberg, J. Knittel, H. Lauer, and L. Seiler. The volume-pro real-time ray-casting system. In *Computer Graphics (SIGGRAPH 99 Proceedings)*, pages 251–260, 1999.

[25] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and Ertl. T. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, pages 109–119, 2000.

[26] P.A. Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics*, 22(4):51–58, August 1988.

[27] L.M. Sobierajski and R.S. Avila. Hardware acceleration for volumetric ray tracing. In *Proceedings IEEE Visualization '95*, pages 27–34, 1995.

[28] M. Wan, A. Kaufman, and S. Bryson. High-performance presence-accelerated ray casting. In *Proceedings IEEE Visualization '99*, pages 379–389, 1999.

[29] R. Westermann and T. Ertl. A multiscale approach to integrated volume segmentation and rendering. *Computers Graphics Forum (EUROGRAPHICS '97)*, 16(3):96–107, 1997.

[30] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 291–294, 1998.

[31] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, August 1990.

[32] P. Williams and N. Max. A volume density optical model. In *Proceedings ACM Workshop on Volume Visualization 92*, pages 61–69, 1992.

[33] O. Wilson, A. Van Geldern, and J. Wilhelms. Direct volume rendering via 3d textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

[34] C. Wittenbrink, T. Malzbender, and M. Goss. Opacity-weighted color interpolation for volume sampling. In *Proc. ACM Symposium on Volume Visualization '98*, pages 135–143, 1998.

[35] R. Yagel. Shell accelerated volume rendering of transparent regions. *The Visual Computer*, pages 53–61, 1994.

[36] R. Yagel and Z. Shi. Accelerated Volume Animation by Space-Leaping. In *Proceedings IEEE Visualization '93*, pages 62–69, 1993.

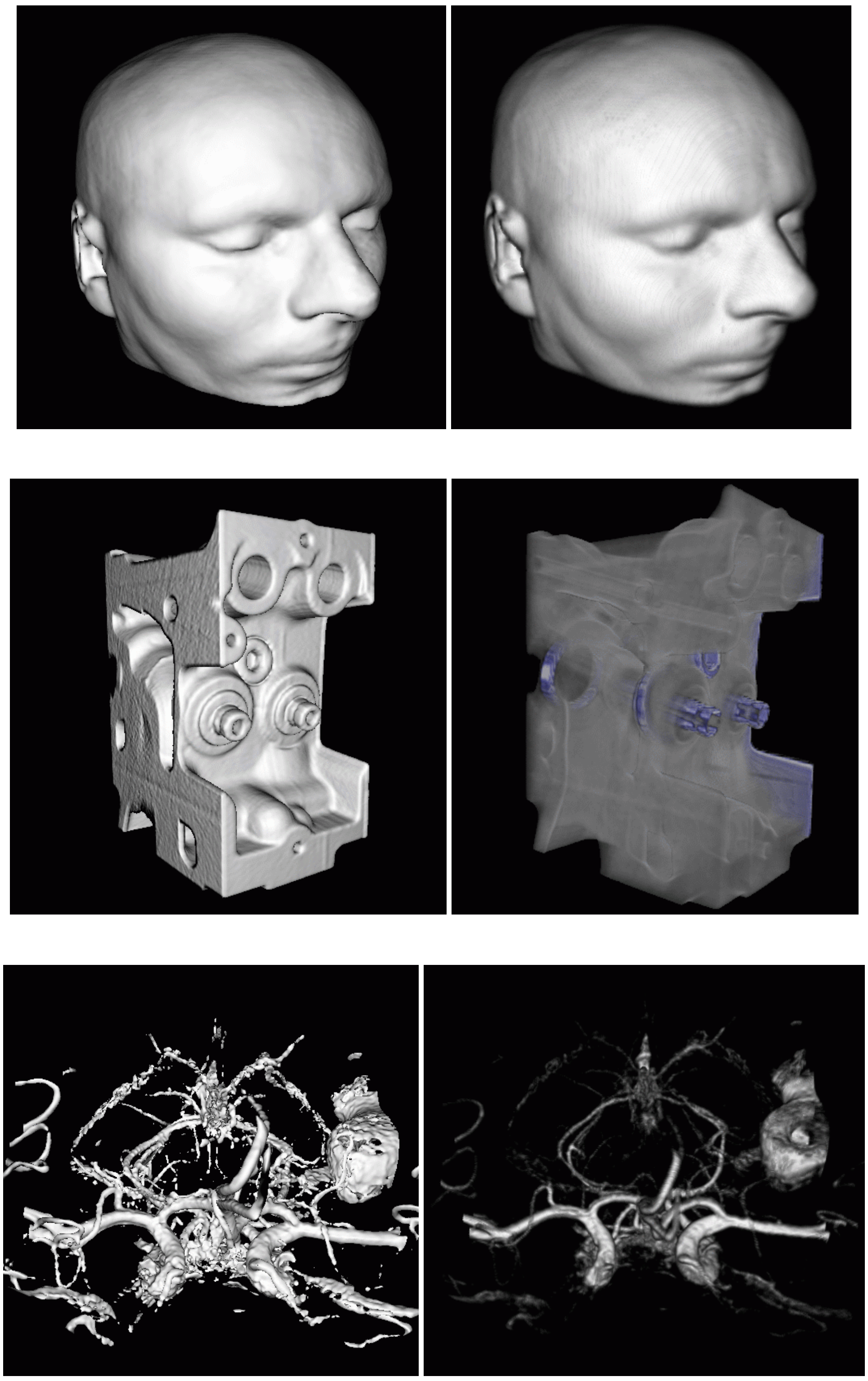

Figure 7: *Volume rendering of example data sets using using different transfer functions.*