

A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting

Simon Stegmaier[†] Magnus Strengert[†] Thomas Klein[†] Thomas Ertl[†]

Institute for Visualization and Interactive Systems
University of Stuttgart



Figure 1: Various volume renderings of a distance field derived from the Stanford Lucy data set, including translucency, transparent isosurfaces, refraction, and reflection.

Abstract

In this work we present a flexible framework for GPU-based volume rendering. The framework is based on a single pass volume raycasting approach and is easily extensible in terms of new shader functionality. We demonstrate the flexibility of our system by means of a number of high-quality standard and non-standard volume rendering techniques. Our implementation shows a promising performance in a number of benchmarks while producing images of higher accuracy than obtained by standard pre-integrated slice-based volume rendering.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Scientific volume visualization plays an important role in both academia and industry. Depending on the application, various visualization techniques are being employed. In order to obtain a productive system that is universally applicable—in medicine, geosciences, engineering, etc.—it is necessary to integrate these techniques into a single tool. Furthermore, the optimal volume rendering tool is supposed to be future-proof. Of course, this refers to technical ad-

vances, like e.g. in graphics hardware, to guarantee best performance also for increasingly large data sets; but this desire also refers to new methodologies, i.e. we expect the optimal tool to be very flexible and to allow for fast and painless integration of innovative new volume rendering techniques if they can help to support doctors in diagnosing illnesses or engineers in recognizing deterioration in critical mechanical parts.

At the moment, actual volume rendering implementations are almost exclusively based on slice-based methods where axis- or viewport-aligned textured slices are blended together to approximate the volume rendering integral. With-

[†] (stegmaier|strengert|klein|ertl)@vis.uni-stuttgart.de

out doubt, these slice-based approaches have their benefits. However, slice-based implementations are rasterization-limited and can be hardly optimized from an algorithmic point of view. Furthermore, when applying a perspective projection the integration step size will vary along viewing rays when using planar proxy geometries, leading to visible artifacts. Finally, they do not easily allow for an implementation of techniques with viewing rays changing direction as it does occur in refracting volumes. Slice-based techniques, therefore, fail to meet the criteria for an optimal volume visualization framework.

On the other hand, the advent of DirectX *Shader Model 3* and comparable OpenGL extensions has led to graphics processors providing an ideal platform for efficiently mapping raycasting-based volume rendering to hardware. This fragment-program-based raycasting does not suffer from any flexibility issues and, therefore, fulfills the second criterion of an optimal volume visualizing tool as it was defined above. Furthermore, hardware manufacturers urge to use any novel fragment program features, promising that this functionality will become very fast in future generations of graphics hardware. It can thus be assumed that GPU-based raycasting is also future-proof from a technical point of view and, accordingly, matches all criteria of an optimal volume visualization framework.

In this paper we describe our experiences in developing such a framework (Secs. 4 and 5) and a series of shaders for both standard and non-standard volume rendering techniques (Sec. 6), elaborating on all the idiosyncrasies involved in the implementation. A performance comparison between GPU-based raycasting and slice-based volume rendering and a qualitative comparison—discussing image quality and flexibility—will be given in Sec. 7 in order to demonstrate the claimed properties and to establish the presented approach as a worthwhile alternative to existing approaches.

2. Related Work

Most of the work in direct volume visualization in recent years has been focused on texture-based approaches. First introduced by Cullip and Neumann [CN93] and Cabral et. al [CCF94], the basic raycasting concept is realized by sampling the volume data using a stack of, typically, planar slices as a proxy geometry and approximating the evaluation of the volume rendering integral by blending the textured slices in front-to-back or back-to-front order in the framebuffer. This pixel-parallel processing of the viewing rays during rasterization of the proxy geometry exploits the unmatched bi- or trilinear interpolation capabilities of modern graphics hardware and is the primary reason for the unsurpassed speed and success of this method. Many enhancements to this simple approach have been proposed that exploit more advanced texture mapping capabilities of today's graphics hardware to increase the interactivity and ap-

plicability of the method, e.g. [RSEB*00, WE98]. The introduction of multidimensional transfer functions [KKH02] and pre-integrated volume rendering [EKE01] has greatly improved the quality of renderings that can be achieved. Also, some acceleration techniques proposed for the original raycasting approach, such as early ray termination and empty space skipping [LMK03], or hierarchical acceleration structures [BNS01, GWGS02, LHJ99] have been successfully adopted to texture-based direct volume rendering. Nevertheless, it is still much harder and requires considerably more effort to integrate such techniques into a slice-based volume renderer compared to the implementation in an obviously much more flexible software-based raycasting code.

Recently, GPU-based implementations of the raycasting algorithm for both structured and unstructured grids that do not rely on the volume slicing approach have been presented [KW03, RGWE03, WKME03]. Both approaches differ from our approach in that they perform multiple rendering passes in order to traverse the volume and have to store intermediate results in temporary buffers. An early technology demo that implements the basic functionality of a complete raycasting algorithm for regular volume data in a single fragment program that does not require multiple rendering passes has been recently shown by NVIDIA [NVIO4b].

3. Raycasting on the GPU

In the last years programmability of workstation and consumer level graphics hardware has evolved at an increasing pace. Driven by the steadily growing demands of the game industry, performance of modern graphics processors has exceeded the computational power of CPUs both in raw numbers and in their extraordinary rate of growth. Abandoning the simple fixed-function pipeline which was the characteristic feature of graphics processors only five years ago, today's GPUs have evolved into very sophisticated, highly programmable SIMD processing units. With the advent of graphics processors supporting the new features of DirectX Pixel Shader 2.x/3.0 [Mic04] and OpenGL NV_fragment_program2 [NVIO4a], namely dynamic looping and branching, GPUs are becoming more and more "general purpose processing units" comparable to the CPU.

```

Compute volume entry position
Compute ray of sight direction
While in volume
    Lookup data value at ray position
    Accumulate color and opacity
    Advance along ray

```

Figure 2: Pseudo code of a fragment-program-based volume raycaster.

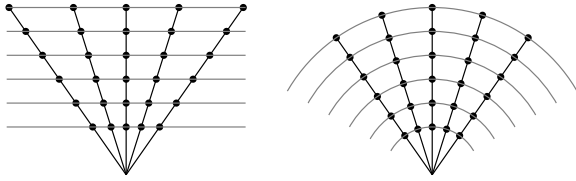


Figure 3: Illustration of sample distances in direct volume rendering approaches. In contrast to raycasting (right), the sampling distance is not constant in slice-based volume rendering (left).

The basic raycasting approach fits very well into the intrinsically parallel stream processing semantics of these new fragment processors. For each pixel of the final image a single ray is traced independently through the volume. Therefore, a fragment program implementation of this simple algorithm working on the fragments generated by rasterizing a polygon covering the screen space area of the volume's projected bounding box is sufficient to compute the correct result. The volume rendering integral for each pixel is then approximately evaluated by sampling the ray at a finite number of positions inside the volume. The contributions of those samples along the ray are accumulated to the overall chromaticity and opacity. By applying an appropriate optical model every desired kind of interaction between light and the volumetric object can be realized. Unfortunately, due to limited capabilities of graphics hardware most of the elaborate optical models already proposed a decade ago [Max95] have not or only with considerable effort and overhead been integrated into slice-based volume renderers. In contrast, it is often very easy to include them into a raycasting system [KPHE02, RC01].

Pseudo code for a simple fragment-program-based volume raycaster is shown in Fig. 2. First, the direction of the viewing ray for the respective fragment/pixel together with the entry point, i.e. the first intersection of the eye ray with the volume's bounding-box, has to be determined. Then in each step of the loop the actual data value for the current sample point is fetched from a 3D texture map and the already accumulated color and opacity values for the fragment are updated according to the chosen optical model or rendering style. In the simplest case, this could be a dependent texture look-up combined with alpha-blending for a basic volume rendering transfer function. Then, the sampling position is advanced along the ray by a specific step size. The loop is terminated either if the ray left the volume or if some other criteria—depending on the chosen optical model—is met, e.g. early ray termination due to high accumulated opacity or the first hit semantic if an opaque isosurface is encountered.

Until recently there has been no real branching and looping support available in fragment level shading programs. Therefore, previous solutions [KW03, RGWE03, WKME03]

that map this basic raycasting algorithm to programmable graphics hardware had to revert to multiple rendering passes and additional pixel tests, e.g. early depth test (z-culling) and occlusion queries, in order to simulate a dynamic, data-dependent number of loop iterations, e.g. terminating the loop if the ray has left the volume. In contrast, now it is possible to take advantage of the new dynamic flow control capabilities provided by the DirectX Pixel Shader 3.0 API and NVIDIA's `NV_fragment_program2` OpenGL extension to implement a single pass volume rendering solution. Currently, only NVIDIA's GeForce 6 series GPUs support the required new set of features, but support from the other major GPU manufacturers is expected in upcoming generations of their graphics processors.

How such a single pass volume rendering shader can actually be implemented is discussed in more technical detail in Sec. 5.

However, the fact that principal rasterization complexity is the same for both raycasting and conventional slice-based rendering poses the question about why a single pass volume raycasting should be superior to 3D texture-based slicing. First, it obviously eliminates the necessity for intermediate buffer reads and writes. Second, since basically only a single polygon has to be rendered in order to generate the necessary fragments, raycasting exhibits a very low geometry processing and fragment generation overhead. And, third, raycasting allows for adaptive step sizes (including early ray termination and empty space skipping), and by definition samples the volume at equal distances (Fig. 3), thereby avoiding artifacts [LHJ99]. Other optimizations based on the history along the sampling ray also benefit from the flexibility of our raycasting approach as will be demonstrated in Secs. 5 and 6.

Moreover, raycasting has a much higher accuracy than slice-based rendering since the entire algorithm is performed with full 32 bit floating point precision. In contrast, slice-based rendering suffers from the lack of high accuracy floating point blending and framebuffer support in today's graphics hardware. Although, it is possible to avoid framebuffer quantization and to emulate full precision floating point blending by rendering to a floating point texture target instead of to the framebuffer, this approach further increases the cost of buffer accesses.

4. Framework

The system for volume raycasting proposed in this paper consists of two major parts: a framework written in plain C and based on OpenGL and GLUT, and a set of shaders described in Sec. 6 that are flexibly loaded and modifiable at runtime. The system is portable between MS Windows and Linux.

Independently of the selected shader, the framework coordinates the two-step rendering process, initializes the tex-

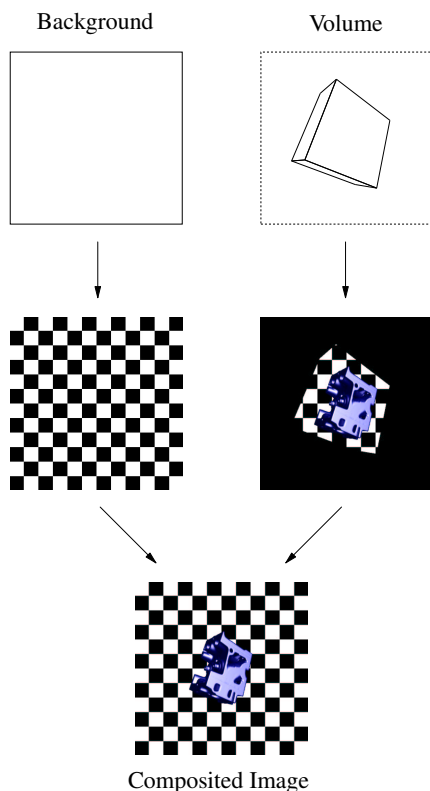


Figure 4: Two-step rendering process. The first part renders the background by drawing a viewport-filling quadrilateral (left), the second part adds the volume visualization and seamlessly overdraws certain parts of the background (right).

tures, and provides various functionality like event handling and transfer functions. Besides coordinating the rendering process, this is all standard functionality found also in any slice-based volume rendering application.

The volume data is expected in unsigned char format and stored in the alpha-component of a 3D RGBA texture. The gradient information that is needed by a number of different volume rendering techniques presented in Sec. 6 is being precomputed using central differences or a $3 \times 3 \times 3$ Sobel operator. Albeit precomputing the gradients poses a non-negligible texture memory consumption overhead as compared to on the fly computation in the respective fragment shader program, we chose to compute the gradients in advance since this allows for the application of filters and usually results in an increased rendering performance. All gradient components are quantized to 8 bit and stored in the RGB-components of the RGBA texture already holding the scalar volume data. Nevertheless, it is possible to compute gradients on the fly if more accurate gradients are required or larger data sets must be loaded.

For the description of the two-step rendering process we first define the following volume properties. Let S_c denote the number of slices in c -direction and D_c the slice distance in c -direction with $c \in \{x, y, z\}$ [†]. Then

$$\begin{aligned} e_c &= S_c \cdot D_c, \\ E_c &= (S_c \cdot D_c) / \max(\{e_x, e_y, e_z\}), \text{ and} \\ C_c &= E_c / 2 \end{aligned}$$

define the volume extents, normalized volume extents, and the volume center. For displaying the volume, we bind a user-selected volume shader (see Sec. 6) and as proxy geometry render an axis-aligned box translated by $(-C_x, -C_y, -C_z)^T$ with backface-culling enabled. One corner of the bounding box is located at the origin, the opposite corner at $(E_x, E_y, E_z)^T$. The box defines the normalized bounding box of the volume considering both the number of slices and their distances. For each bounding box vertex we define texture coordinates identical to the vertex positions. Each fragment can thus access the respective ray's entry point by just looking at its interpolated texture coordinates and—in combination with the camera position—easily compute the parametric ray of sight[‡].

A significant benefit of this approach is the inherent minimization of the number of rays. However, starting rays from the bounding box surface leads to the problem that the viewport is only partially being filled (see Fig. 4, right). For many shaders this poses no problem since, e.g. in direct volume rendering a uniform background is usually chosen. Fragments outside the bounding box can then be set by appropriately setting the clear color. Unfortunately, this does not work for patterned backgrounds which are, e.g., used to visualize refraction. In our system, the remaining fragments are thus processed in a separate step prior to rendering the volume.

Assuming the background pattern is stored in a texture, one way to accomplish this is to render a large, viewport-filling textured quadrilateral. This approach, however, easily leads to artifacts at the crossings to the background pixels generated in the raycasting step. Studying Fig. 4 reveals that the fragment program used to display the volume needs some functionality to render the background pattern anyway. Thus, a robust, artifact-free solution is obtained by re-using this section of the fragment program to assign color values to pixels outside the projected volume bounding box. We will elaborate on the fragment program in Sec. 5. In order to use the fragment program for rendering background pixels, the

[†] We assume the data to be defined on a regular or uniform grid.

[‡] The fragment attribute `fragment.position` stores the (x, y) window coordinates of the fragment center relative to the lower left corner of the window and fragment's z window coordinate and *not* the fragment's coordinates in object space; accessing the object space coordinates is, therefore, not possible without the apparently redundant setting of texture coordinates equal to vertex coordinates.

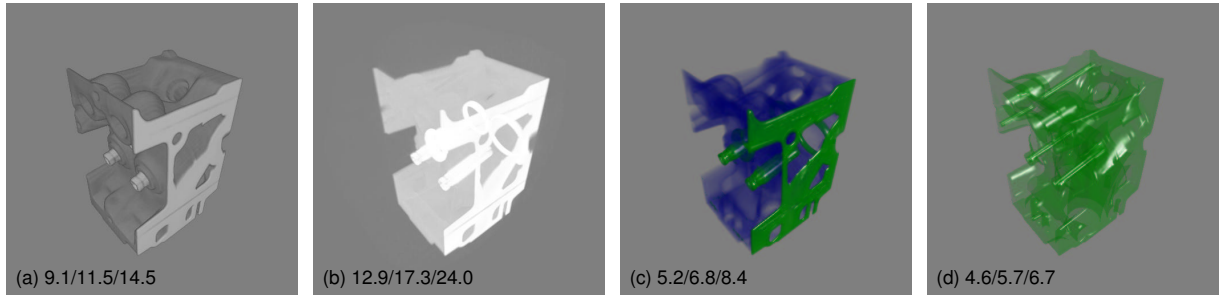


Figure 5: Illustration of different volume shaders for the Engine data set ($256 \times 256 \times 110$ voxels). From left to right: pre-integrated volume rendering, maximum-intensity projection, opaque isosurface combined with pre-integrated volume rendering, and semi-transparent isosurfaces. The minimum, average, and maximum framerates for a 512×512 viewport are given in the lower left corner.

framework again has to render some proxy geometry with texture coordinates set to appropriate object space coordinates. We derive this proxy geometry by unprojecting the four viewport corners using the inverse of the modelview-projection matrix used for rendering the volume bounding box and subsequently render a quadrilateral with the unprojected vertices defining both vertex and texture coordinates. By definition, the resulting quadrilateral is viewport-filling and assigns a valid entry point to each fragment.

Fig. 4 illustrates the final two-step rendering process with the background rendering on the left and the volume raycasting being shown on the right. The viewport is drawn in dashed lines, proxy geometries in solid lines.

5. Single-Pass Volume Shader

All volume shaders implemented for this work are based on a simple, single-pass raycaster consisting of the actual raycasting part sampling the volume and the background rendering part already addressed in discussing the framework. All shaders are based on NVIDIA's `NV_fragment_program2` extension [NV104a]. We will discuss the skeleton at the example of pre-integrated direct volume rendering (see Appendix for source code).

The shader starts with setting up the parametric ray equation. For this purpose, the camera position is first computed by reversing the translation dictated by the modelview matrix. Since the camera is initially located at the origin, this translates to a single instruction in the fragment program. Afterwards, the ray direction is computed. Using the ray's entry point given by the texture coordinates and the computed ray direction, it is now possible to sample the volume with a set of two nested `REP` loops, allowing for an overall number of 65,025 iterations. The actual volume sampling is then straightforward since we basically can now follow the software approach without sacrificing the parallel-processing power of modern GPUs. And this is what finally makes the difference in flexibility of the given approach.

In the implementation presented, the integration does not have to resort to low-accuracy framebuffer blending operations and can take advantage of highly accurate 32 bit floating point computations. Furthermore, while slice-based techniques require an overall of three texture look-ups per fragment to take advantage of pre-integration, the raycasting approach requires only two look-ups since the complete history along the sampling ray is available. However, there are still two subtle issues that have to be addressed.

First, when leaving the volume, the sample lying outside the volume must be projected back along the ray direction onto the respective bounding box back side and the volume rendering integral must be evaluated using the reduced distance. However, a memory-efficient 2D pre-integration table is only valid for a single fixed interval length and thus a 3D pre-integration table would be required, consuming an immense amount of memory. We therefore accept the error introduced by terminating the sampling just before the volume is left and neglect a correct boundary handling.

Second, as seen in Sec. 4, the normalized bounding box extents are given by the values E_c . Positions within this bounding box are thus invalid texture coordinates and cannot be directly used for accessing volume data without being scaled to the maximum texture coordinates $t_c = S_c/T_c$. Here $T_c = 2^n$, where n is the smallest integer such that $2^n \geq S_c$, denotes the actual volume texture size in direction c . The required scaling factors are then given by:

$$\begin{aligned} F_c &= \frac{t_c}{E_c} \\ &= \frac{S_c}{T_c} \cdot \frac{\max(\{e_x, e_y, e_z\})}{e_c} \\ &= \frac{\max(\{e_x, e_y, e_z\})}{T_c \cdot D_c}. \end{aligned}$$

The scaling factors F_c do only depend on the volume data set to be visualized and are computed once in the framework and passed to the shaders as program parameters. The scal-

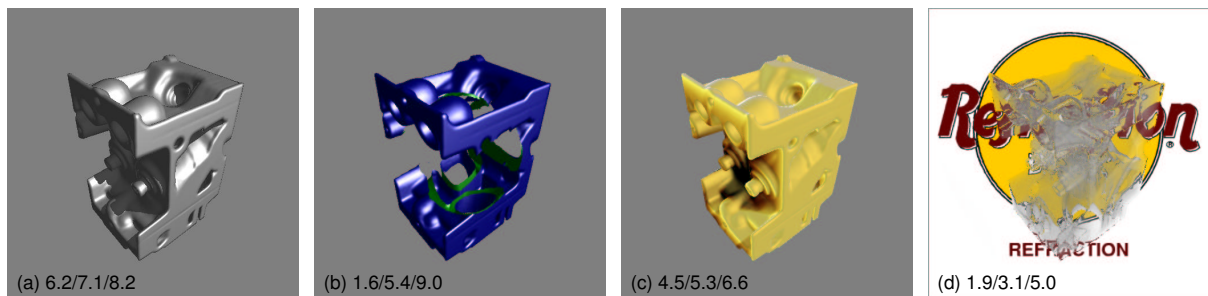


Figure 6: Other shaders implemented for this work. From left to right: Opaque isosurface with self-shadowing, opaque isosurface clipped by a simple spherical clip volume, translucent material, and a continuously refracting volume. The minimum, average, and maximum framerates for a 512×512 viewport are given in the lower left corner.

ing can then be either accomplished within the inner loop—which is very costly—or outside the loop by scaling the entry point and the direction vector. It should be noted that the latter must be done *after* normalization. The resulting raycasting then takes place in texture space and neither the ray direction nor the sampling position can be used any more for geometric computations in object space which are required for rendering the background.

To optimize the rendering performance, we implemented both early ray termination and an inside-test checking for whether the ray has left the volume and, accordingly, whether the sampling can be terminated. Both optimizations are based on a conditional BRK and, therefore, demand the usage of TXL—the equivalent of TEX requiring an explicit mipmap level—although no mipmaps are being used. Only the latter optimization proved advantageous. The former did not improve the overall rendering performance (Sec. 7) and is thus not shown in the given source code.

Once the raycasting step has determined the fragment’s base color and opacity, the fragment is blended with the background. This can be both a uniform color or a pattern defined by a 2D texture map. Therefore, a hypothetical background plane with its normal being the normalized negative viewing direction (initially $(0, 0, -1)^T$) is defined and the ray/plane intersection is computed analytically. To obtain a static background, the plane normal has to be transformed with the inverse modelview matrix MV (as does the camera position to match the view seen by the user when rendering the bounding box). Since in general the transformed normal is given by $\vec{n}' = (M^{-1})^T \cdot \vec{n}$, we obtain $\vec{n}' = ((MV^{-1})^{-1})^T = MV^T$. Furthermore, since the image of $(0, 0, 1)^T$ is stored in the 3rd column of the modelview matrix but column-access is not supported by ARB_fragment_program, we need to transpose the matrix again and can obtain the sought-after normal efficiently with a single instruction. The local coordinate system of the texture coordinates used for accessing the background texture is then similarly derived by the images of the unit vectors in x - and y -direction, respectively.

6. Volume Shader Examples

We have implemented a number of volume shaders to illustrate the flexibility of our framework and the selected raycasting approach. All shaders can be easily derived from the basic volume shader described in Sec. 5. A screenshot of the Engine data set rendered with this shader is shown in Fig. 5a.

A straightforward and trivial modification of the basic direct volume renderer is a maximum-intensity-projection shader (Fig. 5b). The standard isosurface functionality was implemented by searching along the ray for sign changes of the difference of the isovalue and the current and previous sample, respectively. The accuracy of this approach can be further improved by linearly interpolating between the sample points enclosing the isosurface once a sign change has been detected. This optimization basically comes at no costs but nevertheless has a dramatical effect on the resulting image quality (Fig. 7). The standard isosurface functionality has further been combined with direct volume rendering (Fig. 5c) and various lighting techniques.

Isosurfaces can also be made semi-transparent. Combined with standard surface lighting, this can be used to visualize both the exterior surfaces and structures obscured in opaque isosurface renderings (Fig. 5d).

Self-shadowing was implemented by just re-orienting the sampling ray towards the light source upon an isosurface hit and by subsequently checking for occluders (Fig. 6a). A similar technique was applied to implement image-based lighting. We have demonstrated this for highly reflective material with an implementation of sphere mapping. The latter is shown in Fig. 1 for the Lucy data set being illuminated by the Grace Cathedral lightprobe [Deb04].

Another useful extension is volume clipping. This has been implemented by deciding for each sampling point whether it lies inside or outside the clipping geometry defined by the isosurface of another arbitrary volume, e.g. a distance field generated from a polygonal mesh [WEE02]. This approach proved to be more flexible than using a binary

clipping volume. Fig. 6b shows an isosurface of the Engine data set clipped against the isosurface of a radial field.

Since none of the previous volume visualization techniques succeeds in modeling indirect light attenuation, we implemented a translucency shader that decreases the light intensity based on the distance light has to travel within the volume enclosed by the isosurface on its way to a surface point. Results of this shader are shown in Figs. 1 and 6c.

While all the shaders described heretofore can also be relatively easily implemented with slice-based volume rendering approaches, see e.g. [KPHE02], a slice-based renderer for volumetric data with continuous refraction requires considerable effort. On the other hand, implementing refraction is straightforward with the presented approach and easily integrates into the framework, requiring no modifications besides extending the transfer function by another dimension for storing the refraction coefficients. Applications of this shader are shown in Figs. 1 and 6d for the Stanford Lucy and Engine data sets.

7. Results and Discussion

In this section results regarding the performance and quality of the single pass volume raycasting technique are presented and advantages of our approach compared to standard slice-based volume rendering are discussed.

All performance measurements were conducted on a standard PC equipped with an NVIDIA GeForce 6800 GT based graphics accelerator card. Figs. 5 and 6 show examples of renderings of the $256 \times 256 \times 110$ Engine data set performed with the different shaders we described in Secs. 5 and 6. Of course, the rendering performance differs significantly for the different shaders. Although the basic structure of all shaders is identical, the amount of computation that has to be carried out and the efficiency of the optimization tech-

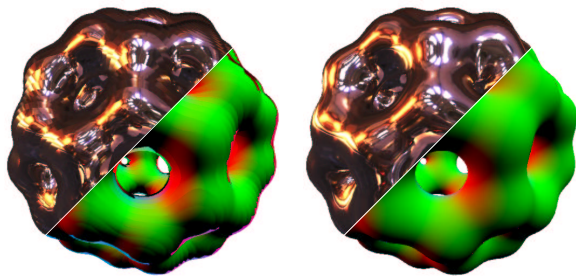


Figure 7: Isosurface renderings of the 128^3 Bucky Ball data set. The images were rendered with a sampling distance equivalent to 50 slices. For the right image accuracy was improved by linear interpolation. The upper parts show spheremap renderings using the Grace lightprobe, the lower parts show color codings of the corresponding surface normals.

Table 1: Comparison between texture-based volume rendering and the raycasting approach presented in this paper. The figures represent average framerates.

	Raycasting		Texture-based	
	512^2	1024^2	512^2	1024^2
Bucky Ball (128^3)	14.0	4.1	41.3	11.5
Engine ($256^2 \times 110$)	10.8	3.2	25.5	7.2
Head ($256^2 \times 225$)	7.2	2.2	16.6	4.6
Lucy ($256^2 \times 512$)	7.0	2.6	<i>n/a</i> *	<i>n/a</i> *

* data set could not be loaded

niques applied varies significantly. The minimum, average, and maximum framerates are given in the lower left corners of the respective images, measured while rotating the tilted volume around the y-axis. Thus, the given framerates account for differences due to changing viewing directions. Considering these framerates, it becomes apparent that volume clipping appears to be very slow regarding the low computational complexity of the required operations. We currently have no explanation for this severe performance breakdown which is accompanied by extensive view-dependent, mosaic-like artifacts. Since these artifacts are non-deterministic and even occur when moving the window we assume this to be a driver or hardware problem.

As already mentioned in Sec. 5 the performance gain expected by early ray termination is often foiled by the overhead posed by dynamic flow control instructions, i.e. BRKs, in the loops. Although the performance benefit that can be achieved by early ray termination is hard to quantify—it strongly depends on the structure of the data set, the chosen transfer function, the current view direction, and the sampling rate—we found that in most cases very little is gained. For semi-transparent volume rendering of the Engine data set shown in Fig. 5 we have measured everything from a 20% performance loss up to a 60% performance increase.

In order to relate our results to standard volume rendering techniques in terms of performance and accuracy, we compared our single pass volume raycasting solution with a slice-based volume renderer. The results are summarized in Tab. 1. For all measurements the step size was chosen such that the object space sampling distance was equal for both approaches and—for viewing directions aligned with the volume bounding box—reflected the actual number of slices contained in the data set. Obviously, the raycasting implementation is currently only approximately half as fast as the reference implementation. The primary reason for this is the current low performance of dynamic loops and branches which however can be expected to become considerably better with the next generation of graphics processors [Gre04]. The presented performance figures should, thus, only be considered preliminary.

8. Conclusion and Future Work

We have presented a framework for the hardware-accelerated visualization of volumetric data. Our system is based on a single-pass raycasting approach taking advantage of recent advances in programmable graphics hardware. Compared to standard slice-based volume rendering techniques, the system exhibits very high flexibility and allows for an easy integration of non-trivial volume rendering techniques which has been demonstrated for a number of examples. Since the blending is performed on the GPU, our raycasting approach is more accurate than implementations that have to rely on framebuffer blending and can even approach the accuracy of software solutions. This improved accuracy is currently traded for considerable performance penalties which, however, will hopefully be alleviated with the next generation of graphics cards.

The framework—including all shaders—can be found in source code on our web page at <http://www.vis.uni-stuttgart.de/eng/research/fields/current/spvolren>.

Acknowledgments

We would like to thank Klaus D. Engel, Siemens Corporate Research, for providing us with his pre-integrated slice-based volume renderer used for the performance comparison.

References

- [BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer* 17, 3 (2001), 185–197. 2
- [CCF94] CABRAL B., CAM N., FORAN J.: Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In *VVS '94: Proceedings of the 1994 Symposium on Volume Visualization* (1994), ACM Press, pp. 91–98. 2
- [CN93] CULLIP T. J., NEUMANN U.: *Accelerating Volume Reconstruction With 3D Texture Hardware*. Tech. Rep. TR93-027, University of North Carolina at Chapel Hill, 1993. 2
- [Deb04] DEBEVEC P.: Light Probe Image Gallery. <http://www.debevec.org/Probes>, 2004. 6
- [EKE01] ENGEL K., KRAUS M., ERTL T.: High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01* (2001), Annual Conference Series, Addison-Wesley Publishing Company, Inc., pp. 9–16. 2
- [Gre04] GREEN S.: GeForce 6 Series OpenGL Extensions, NVIDIA Technical Developer Relations. http://developer.nvidia.com/object/6800_leagues_presentations.html, 2004. London, United Kingdom. 7
- [GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization '02* (2002), IEEE Computer Society, pp. 53–60. 2
- [KKH02] KNISS J., KINDLMANN G., HANSEN C.: Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics* 8, 3 (2002), 270–285. 2
- [KPHE02] KNISS J., PREMOZE S., HANSEN C., EBERT D.: Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization '02* (2002), IEEE Computer Society, pp. 109–116. 3, 7
- [KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03* (2003), pp. 287–292. 2, 3
- [LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution Techniques for interactive Texture-based Volume Visualization. In *VIS '99: Proceedings of the Conference on Visualization '99* (1999), IEEE Computer Society Press, pp. 355–361. 2, 3
- [LMK03] LI W., MUELLER K., KAUFMAN A.: Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *Proceedings of IEEE Visualization '03* (2003), pp. 317–324. 2
- [Max95] MAX N.: Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics* 1, 2 (1995), 99–108. 3
- [Mic04] MICROSOFT CORPORATION: DirectX 9 SDK. <http://www.microsoft.com/directx>, 2004. 2
- [NVI04a] NVIDIA CORPORATION: NVIDIA OpenGL Extension Specifications for the CineFX 3.0 Architecture (NV4x). http://developer.nvidia.com/object/nvidia_opengl_specs.html, 2004. 2, 5
- [NVI04b] NVIDIA CORPORATION: NVIDIA SDK 8.0. http://developer.nvidia.com/object/sdk_home.html, 2004. 2
- [RC01] RODGMAN D., CHEN M.: Refraction in discrete raytracing. In *Volume Graphics 2001* (New York, 2001), Mueller K., Kaufman A., (Eds.), Springer. ISBN 3-211-83737-X. 3
- [RGWE03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T.: Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03* (2003), pp. 231–238. 2, 3
- [RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-textures and Multi-stage Rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2000), ACM Press, pp. 109–118. 2

[WE98] WESTERMANN R., ERTL T.: Efficiently using Graphics Hardware in Volume Rendering Applications. In *SIGGRAPH '98: Proceedings of the 25th annual Conference on Computer Graphics and interactive Techniques* (1998), ACM Press, pp. 169–177. 2

[WEE02] WEISKOPF D., ENGEL K., ERTL T.: Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization. In *Proceedings of IEEE Visualization '02* (2002), pp. 93–100. 6

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization '03* (2003), pp. 333–340. 2, 3

Appendix — Basic Volume Shader

```
# Compute the ray's entry point
MOV geomPos, fragment.texcoord[0];

# Compute the camera position
MOV camera,
    state.matrix.modelview.invtrans.row[3];

# Compute the ray direction
SUB geomDir, geomPos, camera;

# Normalize the direction (done manually instead
# of with NRM to improve accuracy)
DP3 geomDir.w, geomDir, geomDir;
RSQ geomDir.w, geomDir.w;
MUL geomDir, geomDir, geomDir.w;
MOV geomDir.w, 0.0;

# Account for slice distances and texture sizes
MUL dir, geomDir, scaleFactors;
MUL pos, geomPos, scaleFactors;
# w-component select mipmap level
MOV pos.w, 0.0;

# Initialize scalar value
# RGB = gradient, alpha = scalar value
TXL scalar, pos, texture[0], 3D;
MOV scalar.g, scalar.a;
MOV scalar.a, 0.0;

REP params2.g;
    REP params2.g;

        # Lookup new scalar value
        TXL tex, pos, texture[0], 3D;
        MOV scalar.r, tex.a;

        # Lookup color in pre-int texture
        TXL src, scalar, texture[1], 2D;

        # Perform blending
        SUB texblen.r, 1.0, dst.a;
```

```
MAD_SAT dst, src, texblen.r, dst;

# Move one step forward
MAD pos, dir, stepsize, pos;

# Terminate loop if outside volume
SGE temp1, pos, 0.0;
SLE temp2, pos, texMax;
DP3 temp1.r, temp1, temp2;
SEQC temp1.r, temp1.r, 3.0;
BRK (EQ.x);

# Save current scalar value
MOV scalar.g, scalar.r;

ENDREP;

BRK (EQ.x);

ENDREP;

# Compute the normal of the background plane
MOV normal, state.matrix.modelview.row[2];

# Compute ray parameter
SUB temp.rgb, center, geomPos;
DP3 temp.r, normal, temp;
DP3 temp.g, normal, geomDir;
DIV temp.r, temp.r, temp.g;

# Compute ray/plane intersection
MAD temp.rgb, temp.r, geomDir, geomPos;

# Compute the difference vector
SUB diffVec, temp, center;

# Compute the texture coordinates
DP3 temp.r, diffVec,
    state.matrix.modelview.row[0];
DP3 temp.g, diffVec,
    state.matrix.modelview.row[1];
MUL temp.rg, temp, params2.r;

# Center background image
ADD temp.rg, temp, .5;

# Look up the texel value
TEX temp.rgb, temp, texture[3], 2D;
MOV temp.a, 1.0;

# Blend the background pixel
SUB texblen, 1.0, dst.a;
MAD dst, temp, texblen.x, dst;

# Write destination color
MOV result.color, dst;

END
```