An Efficient Approach for Emphasizing Regions of Interest in Ray-Casting based Volume Rendering

T. Ropinski, F. Steinicke, K. Hinrichs

Institut für Informatik, Westfälische Wilhelms-Universität Münster

Email: {ropinski, fsteini, khh}@math.uni-muenster.de

Abstract

We propose a volume visualization algorithm, which allows the combination of different visual appearances within one volume dataset to emphasize certain regions of interest or to eliminate extraneous information. For example, isosurface rendering may be combined with direct volume rendering to visualize surface structures of certain regions within a volume dataset while maintaining volumetric information for the remaining parts of the dataset. To achieve such a combination we use a lens volume, i.e., a 3D shape having an arbitrary convex geometry, which defines a region within the dataset to which a different visual appearance is applied during rendering. The proposed algorithm extends ray-casting approaches and exploits the features of current graphics hardware and thus enables rendering at high frame rates even on commodity hardware.

Keywords: volume rendering, interactive volume visualization, region of interest

1 Introduction

Volume rendering has become an important field of interactive 3D computer graphics. It supports professionals from different domains when exploring volume datasets, e.g., medical or seismic data. Since the advent of programmable graphics hardware many hardware-accelerated volume rendering algorithms have been developed, which allow rendering at interactive frame rates on commodity hardware. Thus expensive graphics workstations are no longer required to enable interactive volume visualization. Therefore many volume rendering applications for interactive exploration of volume datasets are now available. To further assist the user during volume exploration special visualization techniques are needed which can be applied intuitively and still enable interactive rendering on commodity graphics hardware.

In this paper we introduce a novel visualization algorithm for interactive volume visualization which is based upon the ray-casting approach introduced in [1]. Our algorithm allows to visualize certain regions of a volume dataset using different visual appearances by preserving interactive frame rates. Therefore an arbitrary convex 3D shape serves as a lens volume with a special visual appearance assigned to it. All parts of the volume dataset intersecting the lens volume are rendered using this visual appearance (see Figure 1). Because our algorithm is fully accelerated by current graphics hardware the shape of the lens, its position and orientation as well as the visual appearance can be modified interactively. Furthermore, in contrast to the clipping volume approach presented in [2] our algorithm is applicable to the very efficient GPU-based ray-casting approach introduced in [1] and needs considerably less hardware resources in terms of per-fragment operations during rendering. With this algorithm, regions of interest can be highlighted, cut out or used for isosurface extraction without affecting the appearance of the rest of the volume dataset. Thus the user can visually intrude into the datasets and extract as much information as possible without loosing contextual information, since the parts outside the lens volume are rendered using the regular visual appearance. The presented approach can be classified as a ray-casting technique which exploits early ray-termination as well as empty space skipping and thus achieves interactive frame rates.

The next section discusses related work and briefly introduces GPU-based volume ray-casting. In Section 3 the idea of the proposed rendering algorithm is outlined and some details are discussed. Section 4 describes how the proposed concepts can be implemented using OpenGL by exploiting per-fragment operations. To demonstrate the performance of our algorithm Section 5 gives frame rate measurements. The paper concludes in Section 6.

2 Related Work

Since dedicated volume rendering graphics hardware, e.g., the *VolumePro* real-time ray-casting system ([3]) is very expensive compared to commodity graphics boards, numerous algorithms have been developed

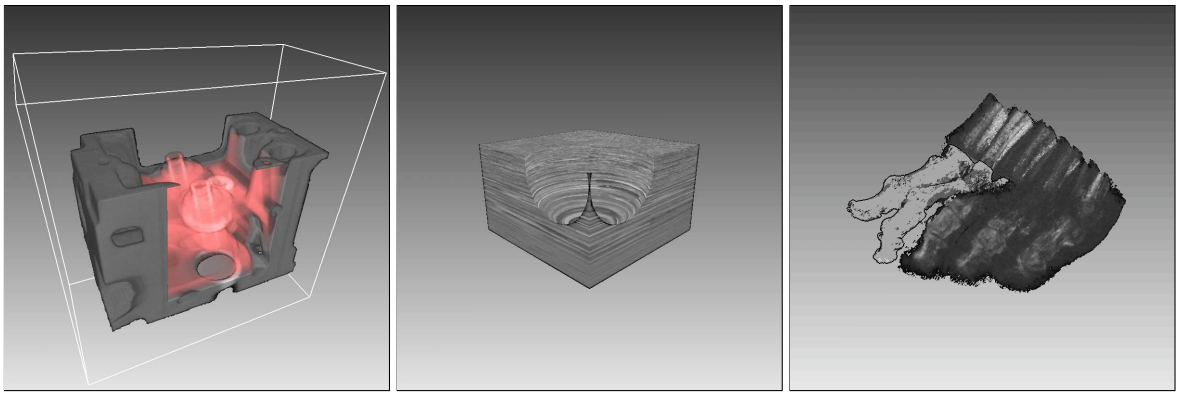

Figure 1: An engine dataset with a cuboid transparency lens (left). A seismic volume dataset with a spherical clipping lens applied (middle). Isosurface rendering applied to an x-ray scan of a human foot (right).

in the past few years with the intention to allow high-quality volume rendering on commodity graphics hardware at interactive frame rates ([1, 4]). A common image-based volume rendering technique is based on ray-casting. With ray-casting a pixel's color is calculated by casting rays originating from the eye position through the pixel's position and the volume dataset. The final pixel color is determined by blending the colors at the samples encountered on the ray. Image quality of image-based rendering approaches is proportional to the sampling rate of the volume dataset. In texture-based approaches this sampling rate is given by the number of polygons used for the proxy geometry, while in ray-casting it is the number of steps needed for processing each ray.

Since in ray-casting a higher sampling rate, which results in better image quality, does not require to send any more proxy geometry down the rendering pipeline it can be implemented very efficiently on current graphics hardware. The algorithm presented in this paper has been developed for volume ray-casting techniques. The underlying GPU-based volume ray-casting technique, in the following called *GPU-based ray-casting*, has been introduced by Krüger and Westermann in 2003 ([1]). It allows very efficient volume rendering on commodity graphics hardware by casting rays through the volume dataset, which is represented as a 3D texture. For each ray the entry and exit points at the bounding box of the volume dataset are encoded as RGB color values which specify the corresponding volume texture coordinates. A major benefit of this GPU-based volume ray-casting algorithm is that it supports early ray termination as well as empty space skipping to further accelerate rendering without affecting the quality of the final image.

In 2003 Weiskopf et al. ([2]) have introduced two interactive clipping techniques for volume visualization: *depth-based clipping* and clipping against a volumetric clipping object. With these techniques it is possible to define 3D clipping shapes against which a volume dataset is clipped. Besides the fact, that our technique allows to apply a different visual appearance inside the lens volume, our approach has the following two advantages. Depth-based clipping, which is based on performing an additional depth test, is not suitable for GPU-based ray-casting since due to invariance the lens's surface depth values cannot be compared to a sample's depth value on the ray. Furthermore depth-based clipping as well as volumetric clipping lead to an additional texture fetch for each sample to determine whether the lens is intersected. This additional texture fetch has the drawback that it has to be performed for each sample, which leads to lower frame rates.

In 2004 Viola et al. ([5]) have proposed a visualization algorithm, which allows highlighting regions of interest in volume datasets. In contrast to our algorithm, this technique does not allow interactive frame rates.

3 Using different Visual Appearances with Ray-Casting based Volume Rendering

To apply a different visual appearance inside the lens volume, the voxels contributing to a pixel in image space need to be distinguished whether they are inside or outside the lens volume. There are two different approaches to perform this determination. The voxels can be distinguished either on the fly during rendering or before rendering the volume dataset. We have decided to use the second approach which results in a better performance because only voxels within the currently rendered region have to be processed during rendering, and therefore a lower number of per-fragment operations is needed when traversing the volume dataset.

Considering a ray cast through a volume dataset which is intersected by a convex lens volume. The ray is split into three different sections if it intersects the lens: There is one section in front of the lens, one inside the

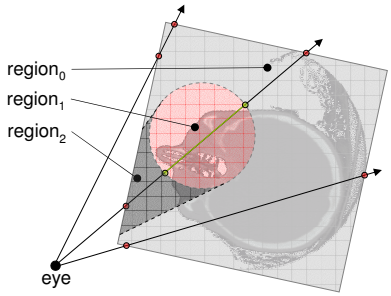

Figure 2: Scheme showing additional entry and exit points at the lens volume.

lens and one behind the lens (see Figure 2). Therefore in our algorithm we distinguish three view-dependent regions:

- $region_0$: voxels behind and next to the lens volume,
- $region_1$: voxels inside the lens volume, and
- $region_2$: voxels in front of the lens volume.

The algorithm renders these view-dependent regions in three separate rendering passes starting with $region_0$. The number of samples used during rendering of each region is determined based on the length of the current section of the ray. Thus, regardless if a lens is applied or the dataset is rendered using regular volume rendering, the number of samples on each ray having the same length is not affected. Furthermore, the number of per-fragment operations executed for each sample is not increased when using our approach (see Section 4).

As mentioned above, we determine the view-dependent regions of the volume dataset before accessing the dataset itself, i.e, only the lens geometry and the bounding box of the volume dataset are considered during this computation. Since in general the shape of the lens is very simple with respect to the number of faces, this step can be performed very fast. Our algorithm has been developed with the goal to be applicable to GPU-based ray-casting; therefore a mechanism is needed, which permits to identify the three regions in an appropriate way. The techniques proposed by Weiskopf et al. [2] are not sufficient for our approach due to three reasons. First, to achieve a better performance we want to separate each ray into the needed sections before rendering. The two techniques introduced by Weiskopf et al. perform this separation during rendering by either accessing a depth texture or a 3D texture storing the lens volume. The second and more important reason is that the depth-based volume clipping technique changes the depth value of the current fragment. Due to the change of the depth value the early depth test performed by current graphics hardware is automatically disabled. Since exploiting the early depth test is one major benefit of GPU-based volume ray-casting, altering the depth value of the processed fragments leads to a significant loss of performance. Third, depth-based clipping cannot be combined with GPU-based volume ray-casting because of numeric issues. In GPU-based ray-casting the position of each sample on the ray is calculated in volume texture coordinates. In depth-based volume clipping the lens volume is defined by its depth information in image space. Since these values are given in different coordinate systems with a different precision, comparison may involve accuracy errors.

Therefore, in order to distinguish between the regions we need a different mechanism to calculate additional entry and exit points on the surface of the lens volume for those rays which intersect the lens volume. In particular this mechanism should support the determination of these intersection points in volume texture coordinates to eliminate problems caused by numeric issues. For a convex lens volume at most two additional intersection points per ray are required, which can be stored in two additional textures. In contrast to regular GPU-based ray-casting, where the entry and exit points are determined by rendering the front resp. back faces of the volume's bounding box, in our case more complex surfaces have to be handled, because the lens volume can be defined by an arbitrary convex shape. Image-based CSG rendering has itself proven to be a very efficient concept to determine these entry and exit points. In contrast to regular CSG rendering techniques we have to take care of the needed color, i.e., we want the colors to encode the volume texture coordinates needed during volume traversal. Thus storing the needed entry and exit points in the textures is ensured when using appropriate CSG rendering techniques (see Section 4). For the spherical lens volume applied to the seismic dataset shown in Figure 1, these textures encoding the intersection points as RGB colors are shown in Figure 3. The left figure shows the entry points for rays cast through $region_0$, i.e., behind and next to the lens. The right figure shows the exit points of the rays cast through $region_2$, i.e., in front of the lens volume.

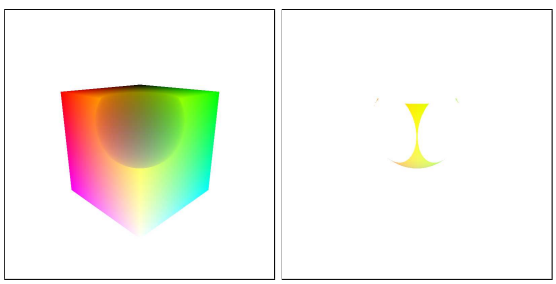

Figure 3: Textures storing intersection points at the transition between $region_1$ and $region_0$ (left) resp. $region_2$ and $region_1$ (right).

In the following paragraphs we are going to explain the concepts required to compute the additional entry and exit points. The implementation is discussed in the following Section 4. To render the $region_0$ consisting of voxels behind and next to the lens volume, in most cases the exit points of the cast rays are given by the complete back face of the bounding box. Only for those rays where the back face of the lens lies entirely behind the back face of the bounding box, there are no voxels behind the lens. In this case $region_0$ contains only voxels next to the lens, and therefore the corresponding exit points are only given by those fragments of the volume's bounding box back face, which lie next to the lens. While the intersection points on the back face of the volume's bounding box can be computed quite easily the entry points for $region_0$ require a little more effort. Since $region_0$ contains all voxels behind and next to the lens volume, each entry point for this region is either given by the ray's intersection point with the back face of the lens volume or by the intersection point with the front face of the volume's bounding box. In cases where a particular ray intersects both, the lens's back face as well as the bounding box's front face, the intersection point farther away from the viewer is used for further processing. In this step we have to ensure that only those parts of the lens are taken into account, which intersect the volume's bounding box. This is important because only these parts affect the volume dataset, and to efficiently perform volume rendering later on, the parts of the lens outside the volume's bounding box should be ignored. Because the computation of the exit points of $region_2$ is similar to that of the entry points of $region_0$, we proceed with $region_2$. $region_2$ consists of all voxels which lie in front of the lens volume. Thus, the exit points for $region_2$ lie on the front face of the lens volume, while the entry points are given by the volume's bounding box front face. In this section we describe only the computations needed to determine the exit points for $region_2$, since the entry points can be determined in a straightforward way as in GPU-based volume ray-casting. Again, to exclude parts lying outside the volume's bounding box from rendering, we consider only those intersection points with the lens's front face, which lie inside the volume's bounding box. Hence by considering all intersection points with the lens's front face which lie inside the volume's bounding box, we obtain the exit points for $region_2$.

In this paragraph the computation of the entry and exit points needed to render the parts inside the lens volume is explained. Since during calculation of the entry points for $region_0$ and the exit points for $region_2$ only those parts of the lens are considered which lie inside the volume's bounding box, those pairs of entry and exit points would not define a closed lens surface. Hence we need another approach to determine the entry and exit points for $region_1$. Although, again we only want to consider those parts of the lens volume which are inside the volume's bounding box, we need a special mechanism to generate a closed surface from the considered entry and exit points, i.e., for each entry point of $region_1$ exists a corresponding exit point. Because only the parts intersecting both the lens as well as the volume's bounding box will be considered, determination of the ray's intersection points with $(lens\ volume \cap bounding\ box)$ is required. These can be computed quite easily by using a standard CSG rendering technique, for example the SCS algorithm ([6]). The resulting surfaces with an appropriate color encoding representing the ray's intersection points in volume texture coordinates are shown in Figure 4.

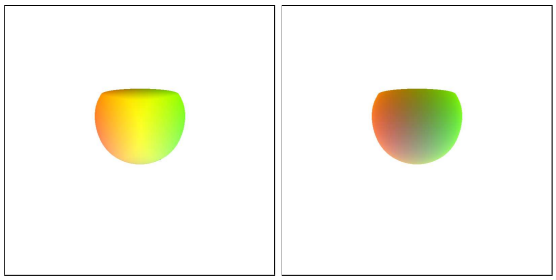

Figure 4: Lens's front face (left) and lens's back face (right) with color encoded volume texture coordinates used to render $region_2$. Only those fragments of the lens, which are inside the bounding box of the volume dataset, contribute to these images.

In the preceding paragraphs we have described how the intersection points for each ray that are required by our algorithm can be obtained. Thus each ray cast through the volume is specified by either two or four intersection points, depending on whether it intersects the lens volume or not. With this knowledge it is quite easy to render $region_0$, $region_1$ and $region_2$ in a back-to-front order using GPU-based volume ray-casting.

4 A Hardware-Accelerated Implementation using OpenGL

As mentioned above, our algorithm for applying different visual appearances to volume datasets is based on the GPU-based ray-casting volume rendering algorithm proposed in [1]. Hence an appropriate mechanism is needed to determine the additionally required entry and exit points introduced by the lens volume. We use 2D textures to store the entry and exit points. This section points out implementation details about how to determine the needed intersection points for each ray and how to perform the volume rendering later on by exploiting the features provided by current graphics hardware. Compared to GPU-based volume ray-casting no additional per-fragment operations have to be performed when rendering the volume dataset in the main rendering pass.

Before the volume rendering can be performed in the main rendering pass, we need to determine the entry and exit points for each ray. Since a ray can have at most four intersection points, i.e., with the front and back face of the volume's bounding box as well as with the front and the back face of the lens, four images storing RGB color coded intersection points in volume texture coordinates are needed. We are going to refer to the images storing the entry resp. exit points at the the volume's bounding box's front resp. back face as $bbox_{front}$ resp. $bbox_{back}$. The images, which store the transitions between $region_0$ and $region_1$ resp. $region_1$ and $region_2$ are denoted as $transition_0$ resp. $transition_1$. Furthermore, the entry and exit points for $region_1$, i.e., the lens volume, are needed and represented by $lens_{front}$ and $lens_{back}$. While the generation of $bbox_{front}$ and $bbox_{back}$ can be performed in a straightforward way by using an appropriate culling technique, the generation of the other images is more complex and will be explained in the following paragraphs.

To render the images denoted as $transition_0$, $transition_1$, $lens_{front}$ and $lens_{back}$ a second independently configurable depth test is needed in addition to the standard OpenGL depth test. To simulate this second depth test we use a concept similar to the *depth peeling* technique introduced in [7]. Within depth peeling a depth texture of type `GL_DEPTH_COMPONENT` is used as a secondary read-only depth buffer, and the shadow test is used to simulate the second depth test operating on the additional depth buffer. Since our implementation exploits the features provided by programmable graphics hardware, we can easily perform this second depth test within a fragment program. Therefore, we access appropriate depth textures containing the required depth information, and we perform the depth test by comparing the current fragment's z-coordinate to the depth value stored in the depth texture at the corresponding position. Dependent on the result of this comparison the fragment may be discarded, i.e., the second depth test fails. The additional depth test performed in a fragment program has a different behavior than the standard depth test. In our algorithm we rely on the fact that a fragment, which is discarded within a fragment program, is not processed any further by the rendering pipeline. In contrast when a fragment fails the standard depth test, it is possible to define an appropriate stencil operation to modify the stencil buffer.

To obtain the information stored in $transition_0$ we do not change the standard OpenGL depth test which is set to *GL_LESS* and render the back face of the lens as well as the front face of the volume's bounding box. Since as aforementioned the effect of the lens is restricted to the volume's bounding box, we need to consider the parts of the lens intersecting the bounding box only. Furthermore, we have to address the cases, where the back face of the lens lies behind the bounding box, i.e., at this pixel position $region_0$ contains no voxels behind the lens. In our implementation we simply use the alpha channel to encode that there is no intersection point at the current position. Thus we set the alpha value to 0.0 in $transition_0$ at every pixel position where the lens's back face is behind the volume's bounding box back face. To fulfill these demands we use a second depth test in combination with the stencil test. We initialize the stencil buffer with 0's and replace the stencil value with 1's where the volume's bounding box's back face would be rendered by also writing the depth values of the volume's bounding box back face to the depth buffer. Then the stencil function is set to render only those pixels where the stencil value equals 1, and the lens's back face is rendered to the color buffer. During this rendering an additional depth test, which is set to *GL_GREATER* and which performs on a depth texture containing the depth information of the volume's bounding box's front face, is used to render only those parts of the lens which intersect the volume's bounding box. To prepare the subsequent rendering of the volume's bounding box's front face it is important to configure the stencil operation to replace the current stencil value by 0 if the stencil test passes regardless of the result of the standard depth test. Thus only if the current fragment lies in front of the volume's bounding box no stencil buffer update occurs, since the fragment is discarded within the fragment program and is not processed any further by the rendering pipeline. So far we have generated a color buffer containing the parts of the back face of the lens which intersect the volume's bounding box and a stencil buffer containing 0's at these positions as well as at the positions the lens's back face lies behind the volume's bounding box's back face; and 1's at the remaining positions. Finally, we render the front of the volume's bounding box where the stencil buffer contains 1's to obtain $transition_0$. To obtain $transition_1$ we simply render the fragments of the lens's front face, which lie inside the volume's bounding box. Therefore, we render the back face of the volume's bounding box to the depth buffer and set the standard depth test to *GL_GREATER*. Then we render the front face of the lens volume and configure the additional depth test to compare the current fragment's z-coordinate with *GL_LESS* to the corresponding depth value of the volume's bounding box's front face.

The images denoted as $lens_{back}$ and $lens_{front}$ can be obtained by applying a standard image-based CSG rendering technique, for instance the one described in [6]. As explained in Section 3 it is important that, while only the parts of the lens intersecting the volume's bounding box are considered, for each ray intersecting the lens volume $lens_{back}$ and $lens_{front}$ contain a pair of intersection points. Therefore the

Table 1: Frame rates measured in frames per second which are achieved when rendering various volume datasets with different viewport sizes.

viewport size	512^2		1024^2	
	no lens	lens applied	no lens	lens applied
seismic dataset (256^3)	60.0	58.3	53.0	49.1
skull dataset (256^3)	30.1	24.0	19.0	14.7
foot dataset (256^3)	29.7	25.3	14.8	9.7

CSG expression ($lens\ volume \cap bounding\ box$) is evaluated to generate the image shown in Figure 4 (left). The image shown in Figure 4 (right) is obtained in a very similar way, where the role of the back and front faces of the lens volume is exchanged.

5 Performance Analysis

For measuring the performance of the proposed algorithm for arbitrary convex lens shapes some datasets have been rendered by applying our technique. The engine the seismic dataset as well as the foot dataset are shown in Figure 1. During the measurements the datasets have been rendered both with and without applying a lens. To show the effect of changing the final image resolution the viewport size has been altered as well. The results are shown in Table 1. For the performance tests an Intel Pentium 4, 3GHz system, running Windows XP Professional, with 1 GB RAM and a *n*Vidia GeForce FX 6800 based graphics board equipped with 256 MB RAM has been used.

Table 1 shows that interactive frame rates are remained when applying a lens to the dataset. Furthermore the table shows that the frame rate drops when a larger viewport is used, since GPU-based ray-casting makes extensive use of per-fragment operations.

6 Conclusion

In this paper we have proposed a novel visualization algorithm for volume datasets. The algorithm exploits different image-based state-of-the-art rendering techniques to allow the combination of different visual appearances within a single volume dataset. Thus when applying the presented visualization techniques, the user is able to define an arbitrary convex shape which serves as a lens. Because the proposed algorithm facilitates GPU-based volume ray-casting which supports early ray-termination as well as empty space skipping it allows volume visualization at interactive frame rates even on commodity graphics hardware.

References

[1] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Proceedings IEEE Visualization 2003*, 2003.

[2] D. Weiskopf, K. Engel, and T. Ertl, "Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading," *Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 298–312, 2003.

[3] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The VolumePro Real-Time Ray-Casting System," in *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 251–260, ACM Press/Addison-Wesley Publishing Co., 1999.

[4] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," in *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pp. 91–98, ACM Press, 1994.

[5] I. Viola, A. Kanitsar, and M. E. Groller, "Importance-Driven Volume Rendering," in *VIS '04: Proceedings of the conference on Visualization '04*, (Washington, DC, USA), pp. 139–146, IEEE Computer Society, 2004.

[6] N. Stewart, G. Leach, and S. John, "Improved CSG Rendering Using Overlap Graph Subtraction Sequences," in *International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE 2003)*, pp. 47–53, 2003.

[7] C. Everitt, "Interactive Order-Independent Transparency," tech. rep., *n*Vidia Corporation, 2002.