IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

Accelerating Volume Raycasting using Occlusion Frustums

Jörg Mensmann   Timo Ropinski   Klaus Hinrichs

Department of Computer Science, University of Münster, Germany

Abstract

GPU-based volume raycasting allows to produce high quality renderings on current graphics hardware. The use of such raycasters is on the rise due to their inherent flexibility as well as the advances in hardware performance and functionality. Although recent raycasting systems achieve interactive frame rates on high-end graphics hardware, further improved performance would enable more complex rendering techniques, e. g., advanced illumination models. In this paper we introduce a novel approach to empty space leaping in order to reduce the number of costly volume texture fetches during ray traversal. We generate an optimized proxy geometry for raycasting which is based on occlusion frustums obtained from previous frames. Our technique does not rely on any preprocessing, introduces no image artifacts, and—in contrast to previous point-based methods—works also for non-continuous view changes. Besides the technical realization and the performance results, we also discuss the potential problems of ray coherence in relation to our approach and restrictions in current GPU architectures. The presented technique has been implemented using fragment and geometry shaders and can be integrated easily into existing raycasting systems.

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

GPU-based volume raycasting is on the rise. It allows to generate high quality volume renderings on high-end graphics cards at interactive frame rates. Due to its flexibility, raycasting can be extended towards the integration of advanced rendering effects. However, usually these rendering effects consume valuable rendering time. Therefore, in order to make the integration of additional effects possible without loosing interactivity, it is important to get optimal rendering performance from GPU-based raycasting.

The goal of this paper is to increase raycasting performance by ignoring empty voxels which do not contribute to the final image. This empty space leaping is realized by exploiting the geometry processing capabilities of programmable graphics hardware which are normally not utilized during raycasting. The vertex and fragment processing units of earlier graphics hardware were independent, resulting in the vertex units being mostly idle when raycasting is implemented within a fragment shader. More recent GPUs switched to a unified architecture where processing units are dynamically assigned to process either vertices or fragments, and thus ensure better utilization of the available computing resources. Nonetheless, raycasting still does not make use of the geometry processing capabilities of the hardware. Therefore we have investigated how a more complex proxy geometry can be used for supporting the fragment shader, to save costly calculations.

Data exploration is an important application for volume rendering. For this task it is especially important for the user to be able to change all rendering parameters interactively. Changes in some parameters like the transfer function can have global effects on the empty space, depending on which source data values are mapped to zero opacity. These parameter changes can therefore invalidate any prior knowledge about the distribution of empty space and nullify any data structures which rely on this information. Hence, optimization techniques that rely on empty space information and require expensive preprocessing to adapt the underlying data structures to changes in the empty space are unsuitable for interactive data exploration. We introduce *occlusion frustums* to improve the rendering performance of GPU-based raycasting, which require no preprocessing as they are constantly regenerated. Using occlusion frustums for space leaping introduces

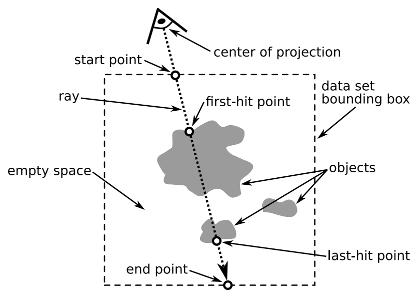


Figure 1: *Casting a ray through a volume data set.*

no rendering artifacts and also works for non-continuous viewpoint changes.

2. Related work

GPU-based raycasting was introduced by Röttger et al. [RGW*03] and enhanced by Krüger and Westermann [KW03]. The flexible single-pass technique is increasingly popular and will very likely replace slice-rendering as the de facto standard for volume rendering. It uses a proxy geometry most often resembling the data set bounding box to specify ray parameters, as shown in Figure 1.

Many acceleration techniques have been proposed for raycasting, most trying to reduce the large number of sampling operations in the volumetric data. Avila et al. [ASK92] introduced polygon assisted raycasting (PARC) which approximates the volume object by a polygon mesh and restricts raycasting to those parts of the rays lying inside the geometry. This was implemented by Leung et al. [LNM06] using Marching Cubes for extracting the object surface. Similarly, Westermann and Sevenich [WS01] utilize hardware-based texture slicing to speed up software-based raycasting. They render the data set with a slice-based approach and use the resulting depth image to get an optimal ray setup for the raycasting performed in a second rendering pass.

A coarser but also faster approximation of the volume object can be generated by partitioning the volume into uniform blocks and not rendering those consisting only of empty voxels, for example, shown by Hadwiger et al [HSS*05] and Scharsach et al. [SHN*06]. Li et al. [LMK03] use adaptively partitioned subvolumes to add empty space skipping to slice rendering, grouping similar voxels into subvolumes. These object-order techniques can adapt to limited changes in the transfer function by storing minimum and maximum voxel intensities for each block. But for larger changes these methods generally require to rebuild the data structure, for which every voxel in the data set has to be considered. This makes these techniques rather unsuitable for data exploration where the opacity mapping is changed frequently. Also the more complex approaches are often difficult—if not impossible—to adapt efficiently to the GPU programming paradigm and into existing rendering frameworks.

The idea of skipping empty voxels around a volume object by exploiting temporal and spatial coherence between consecutive frames was introduced by Gudmundsson and Randén [GR90] for parallel projection and later generalized by Yagel and Shi [YS93] under the name *space leaping*. They approximate optimal ray start points by extracting first-hit points from the depth image of previous frames and reprojecting them to the current view using point splatting. Due to discretization of screen-space positions to integer pixel locations, some pixels will not be covered by the reprojection and therefore a *hole filling* is required, triggering a full raycasting for such pixels. The reprojection is only possible for small view changes, for larger changes a full raycasting of virtually all pixels is necessary.

Several extensions of the reprojection approach have been presented. Yoon et al. [YDKN97] transform rays instead of points to accelerate isosurface rendering. Wan et al. [WSK02] presented a cell-based reprojection scheme which they combined with a spatial data structure based on distance fields for hole filling. Besides high memory requirements for the data structure, their technique will fail to detect suddenly appearing objects when large viewpoint changes are performed. Instead of frame-to-frame coherence, Lakare and Kaufman [LK04] exploit ray coherence by casting *detector rays* to get empty space information for multiple adjacent rays at the same time. While independent of the transfer function, their technique can only give accurate results when single voxels are projected over multiple pixels on the screen, as it is the case in virtual endoscopy applications where the camera is placed close to the object surface.

All of the previously described space leaping methods that make use of temporal and spatial coherence were implemented on the CPU. Many of them cannot be directly ported to the GPU, and are therefore not useful in a GPU-based raycasting system. Only few solutions were presented that directly use graphics hardware for acceleration. Klein et al. [KSSE05] implemented Yagel and Shi's space leaping with point reprojection on programmable graphics hardware using vertex shaders. For larger view changes artifacts are described as "unavoidable" with their technique. The reprojection technique was also applied to time-varying data by Grau and Tost [GT07].

3. Impact of hardware restrictions on raycasting

Although the highly parallel architecture of modern graphics processors makes volume raycasting usable for interactive applications, it must be kept in mind that this architecture has some restrictions. One hardware restriction significant for raycasting is branch coherence. Fragment processors on modern GPUs process fragments in groups rather than individually, and the fragment with the most time-consuming calculations limits the progress of the entire group, as the group can only finish when all of its fragments are completed. Houston [Hou07] demonstrated this for different GPUs by

distribution	fps	overhead
constant	251.3	–
linear	249.2	0.8%
random	149.7	67.9%

Table 1: *Influence of ray length distribution on casting 512^2 rays with an average ray length of 128 samples.*

data set	block size		
	4^2	8^2	16^2
head	+2.3%	+4.9%	+10.2%
engine	+4.7%	+10.3%	+19.9%
aneurysm	+3.4%	+7.1%	+13.8%

Table 2: *Overhead resulting from block grouping.*

distributing pixels that cause "slow" or "fast" calculations either randomly or in groups on the screen. Coherence regions with the same calculation time for 4×4 to 16×16 pixels give nearly optimal results, depending on the actual graphics hardware, while a random distribution results in the worst performance.

For GPU-based raycasting, calculation time for each fragment is mainly influenced by the ray length as this controls the number of texture fetches. Empty space leaping and early ray termination can disturb coherence between calculations for adjacent rays, as they modify the initially equal ray lengths depending on the volume data. To examine this matter, we have implemented a simplified GPU raycaster which can set the lengths of the rays to be either constant, linearly increasing, or randomly distributed over the screen. The sum of all ray lengths (and therefore the total number of texture fetches) is the same in either case. As shown in Table 1, there is no significant performance difference between constant and linear increasing lengths, but a random distribution of lengths increases rendering time by about 68%.

However, experimental results show that ray length distribution is not that random for non-synthetic data. This can be quantified by comparing the original unmodified volume raycasting to one where we group adjacent rays in blocks, and apply the longest ray length within each block to all rays in the block. Implementing this scheme in a full volume raycaster, we do not directly consider the lengths of the rays but instead count the number of texture fetches which also covers gradient calculation with six additional texture fetches for each non-empty voxel. Since the computation time for a block is determined by its longest ray, the shorter rays in the block result in idling fragment units. Table 2 shows the overhead introduced by the block grouping, i. e., the total idle time, for semi-transparent, dense, and sparse data sets. This overhead ranges from 2% to 20% compared to when no block grouping is applied. This is much less than what would be expected for a purely random distribution. Visualizing ray lengths as in Figure 2 shows that they are distributed quite uniformly, so we expect the penalty to pay for a block having to wait for the calculation of its longest ray to be comparatively small.

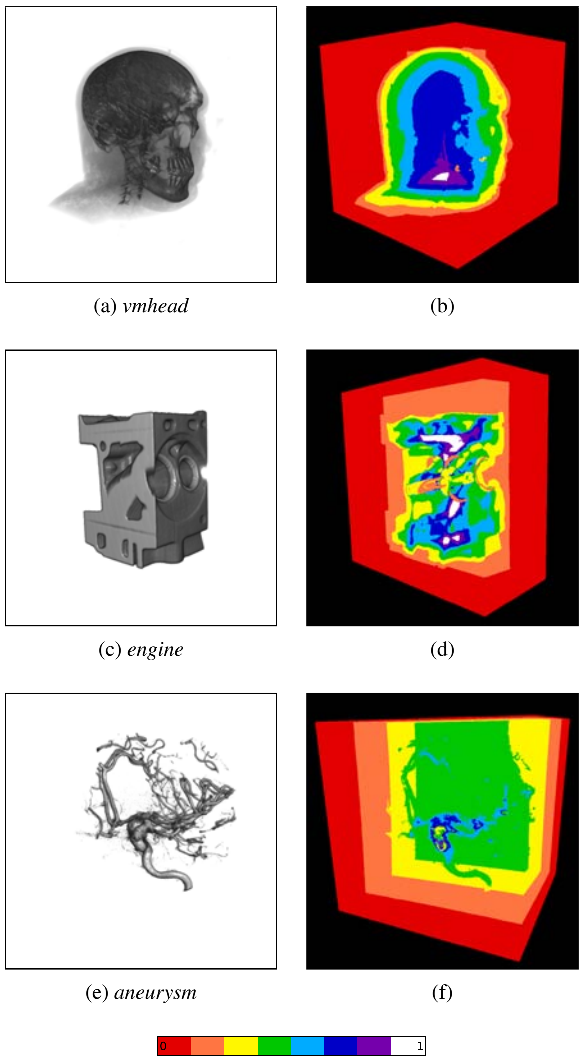


Figure 2: *Analyzing the distribution of ray calculation costs. The number of texture fetches per ray (including gradient calculations) is normalized and quantized for each data set.*

Thus we could show that although in theory branch coherence could be a potential problem for raycasting optimizations, the effects are most visible with synthetic worst-case data, while with real-world data the resulting ray lengths are much more coherent, which will not be changed dramatically by optimizations.

4. Optimizing the proxy geometry for space leaping

GPU-based raycasting usually utilizes a cube as its proxy geometry for generating the ray start and end points, illustrated in Figure 1. As the colors on the cube surface encode the exact position in space (compare Figure 3a), any other geometry can be used instead, as long as it encloses all relevant voxels.

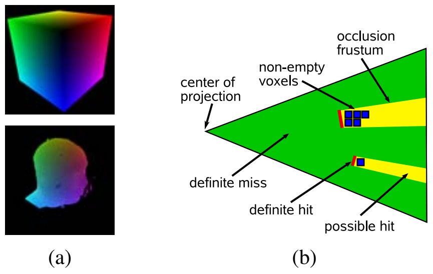


Figure 3: *(a) Proxy geometry and first-hit point image for the vmhead dataset. (b) Visibility information retrieved from a first-hit image, used for constructing the occlusion frustums.*

A straight-forward approach to minimize the sampling of empty voxels would therefore be to enclose all non-empty voxels inside a closely-fitted proxy geometry, as proposed with the PARC algorithm [ASK92]. The complexity of this geometry is data-dependent, and some kind of simplification would be needed to prevent the generation of excessively complex geometries. While giving optimal results with regard to preventing unnecessary sampling, generating such a geometry would be quite costly and, even worse, it would become invalid as soon as the transfer function is changed, a common operation for data exploration. Hence, an on-the-fly process with fast adaptation to changed viewing parameters would be preferred, even when giving slightly less optimal results. Therefore we refrain from any preprocessing and choose to do a less costly proxy geometry construction for each frame, using information obtained during the rendering of the previous frame.

4.1. Occlusion frustums as proxy geometry

The concept of occlusion volumes is well-known for geometric visibility calculation and occlusion culling, see Schaufler et al. [SDDS00]. Objects directly visible from the camera are considered as occluders that cast an *occlusion volume* into the scene, similar to casting a shadow. All objects inside this occlusion volume are then known to be invisible from the camera and can be removed for occlusion culling. Our approach is based on visibility information retrieved from a *first-hit image* (Figure 3a). In this image, each pixel corresponds to a ray, and the pixel color specifies the position of the first non-empty voxel on this ray. The alpha channel is used to mark rays that only hit empty voxels. When we consider these first-hit voxels as occluders, we can extract regions where no non-empty voxels are located. These are marked as *definite miss* in Figure 3b.

To prevent the raycaster from sampling in regions which are known to contain only empty voxels, we can now build a proxy geometry consisting of the first-hit voxels and their occlusion volume. Not an exact geometry is necessary but a simplification is sufficient as long as it contains all possibly non-empty voxels. For each first-hit voxel a quadrilateral

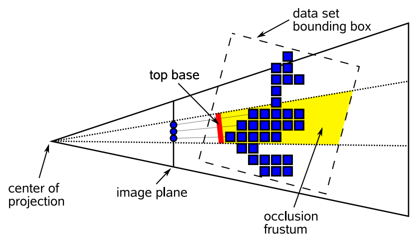


Figure 4: *Construction of a two-dimensional occlusion frustum with a block size of three.*

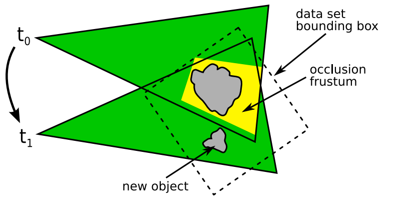


Figure 5: *New incoming object problem: The new object is outside the occlusion frustum constructed at t_0 and therefore not considered for raycasting at t_1.*

pyramidal *occlusion frustum* is constructed, with its top base located at the voxel position. The occlusion frustums are constructed by extruding the quad forming the top base along the projectors extending from the center of projection through the vertices of the quad (illustrated in Figure 4). The depth extent of the frustum is chosen large enough so that its bottom base lies outside the data set's bounding box. The union of all these constructed occlusion frustums gives the complete occlusion volume. Unlike point-based space leaping, no reprojection is necessary to adapt the generated proxy geometry to changed viewing parameters. As the geometry is created in object space, it is sufficient to apply the desired view transformation and simply render the geometry from the new view point.

Special consideration is necessary for objects which are initially located outside the view frustum. As the generation of occlusion frustums is image-based, they can only contain voxels lying inside the view frustum. Consequently, when the view is changed so that previously hidden objects become visible, they will be skipped by raycasting as they do not lie within the proxy geometry that consists of all occlusion frustums (see Figure 5). To handle this case, the proxy geometry must be enlarged to enclose all regions outside the view frustum where non-empty voxels may be located. The additional geometry can be generated by subtracting the view frustum of the previous frame from the initial cube proxy geometry, which is guaranteed to contain all voxels.

4.2. Clipping the occlusion volume

The occlusion frustums can be compared to shadow volumes where the silhouette of occluders is often extruded into infinity. This can not be directly copied in our case, since additional information is carried by the geometry. All proxy geometry has to be placed inside a bounding box of $[0,1]^2$ because of the way ray parameters are encoded as colors. Those are clamped to $[0,1]$ by the graphics hardware and therefore geometry reaching outside this unit cube would lead to a wrong ray direction. Additionally the frustum must be closed from all sides to give correct results for all possible view directions. Therefore the initial frustum has to be clipped against the data set bounding box to construct the final frustum.

With this approach, only the empty space between the ray start and the first-hit point is considered. Especially for sparse data sets, however, much more empty space can remain between the first-hit point and the ray end. The empty space *inside* an object can get too complex to be handled efficiently, but the empty space *behind* it is much simpler. Instead of first-hit points, now information about *last*-hit points would be necessary, i. e., the position of the last non-empty voxel between first-hit point and ray end. The last-hit image could be used to construct the back of the occlusion frustums, instead of building them by clipping against the bounding box. Though initially reasonable, we have not implemented this extension for two reasons: First, it can not be combined with early ray termination, as this might stop the ray traversal before reaching the last-hit point and therefore not provide the necessary last-hit point information. Second, while about doubling the costs for creating the occlusion frustum, for the typical use case of a slowly rotating object the amount of traversed empty voxels which could be saved with this extension is relatively small.

5. GPU implementation

As described before, the starting point of our approach is the first-hit image. It can be generated in the raycasting fragment shader by detecting the first-hit voxel and writing its position into an additional rendering buffer. By exploiting multiple render targets, this is done during normal ray traversal without the need for a second pass, so the first-hit image is extracted with minimal overhead. The steps described in the following subsections are inserted into the volume rendering pipeline just before the actual raycasting operation, where normally a cube proxy geometry is rendered. The only additional change is instructing the raycaster to also output the first-hit image, as described above. Thus, the proposed optimization can be easily integrated into existing volume raycasting frameworks that use the Krüger-Westermann approach [KW03].

5.1. Analyzing first-hit points

Instead of generating an occlusion frustum for every pixel in the first-hit image, which would result in a prohibitively large amount of geometry for high viewport resolutions, we group adjacent pixels as square *occlusion blocks*. We then analyze all voxels corresponding to pixels in each block to find the voxel with minimum distance to the view point. This voxel's position is used for constructing a frustum that encloses all non-empty voxels hit by rays associated with the occlusion block. The proxy geometry is enlarged by this simplification, which leads to sampling of some empty voxels, and thus reduces efficiency of space leaping. But as discussed in Section 3, the hardware processes fragments block-wise, and the slowest fragment limits the processing speed for all fragments in a block. Hence, the block simplification suits the hardware limitations, and an unsimplified solution would not result in significantly better performance results. The simplification can be implemented efficiently as a fragment shader and results in the *occlusion block texture*.

5.2. Generating occlusion frustums

The generation of the occlusion frustum geometry is especially suitable for implementation with the recently introduced geometry shaders. They allow to generate new graphics primitives from input primitives sent by previous stages of the graphics pipeline. While they can theoretically generate arbitrary amounts of output primitives from one input primitive, current implementations require to specify the maximum number of output primitives in advance and are most efficient when this number is not too large.

In our algorithm, an occlusion frustum which consists of twelve triangles has to be generated for each non-empty occlusion block. These input blocks can easily be modeled as point primitives, with the x- and y-coordinates set to the texture coordinates of the corresponding texel in the occlusion block texture. These points are sent to the geometry shader which either outputs the clipped frustum or zero triangles, depending on whether the block corresponding to the input point is empty or not. In the geometry shader the frustum is constructed and clipped against the data set's bounding box. To ensure that it contains all relevant voxels, it is slightly enlarged and moved towards the camera. The resulting geometry is rendered using a fragment shader that assigns the vertex position as fragment color, while the z-buffer handles overlapping frustums.

For supporting older hardware, the algorithm can also be adapted to vertex shaders since the maximum number of generated vertices for each occlusion block is known beforehand. However, this might result in a significant performance penalty, as the texture fetch will have to be made per-vertex instead of per-frustum. For empty occlusion blocks a degenerate geometry with all vertices of the corresponding frustum set to zero would be built, effectively removing it from the output.

Finally, we add the bounding box cube subtracted by the view frustum of the previous frame for detecting appearing objects (see Subsection 4.1). The resulting geometry can be used for ray setup, by rendering the front faces to get ray start points, while the ray end points are still retrieved by rendering the back faces of the data set bounding box. When the empty space information is invalidated by changing rendering parameters like the transfer function, a single frame has to be rendered using the data set bounding box as its proxy geometry, while subsequent frames can again use the occlusion frustums.

6. Results and discussion

The presented optimization techniques were implemented using OpenGL/GLSL and integrated into the Voreen volume rendering framework. All tests were conducted with an Intel Core 2 Duo E6300 CPU and an NVIDIA GeForce 8800 GT graphics board with 512 MB of onboard memory. We have tested our algorithm with different sparse and dense data sets, with the results shown in Table 4 and Figure 6. Rendering was performed using on-the-fly gradient calculation, Phong lighting, and early ray termination. The objects were constantly rotated and visualized using direct volume rendering. Occlusion frustum optimization was applied with a block size of 4×4, a compromise between accuracy and complexity of the generated proxy geometry. In our tests we did not observe artifacts caused by the optimization. As it is an image-based technique, undersampling could be problematic, since voxels missed due to undersampling might lead to incorrectly identifying parts of the volume as empty and permanently removing them from the proxy geometry. In practice this poses no problem because of the constant refreshing of the occlusion frustum geometry. Also the occlusion blocks lower the chance of this happening, as all voxels in a block would have to be missed due to undersampling to remove the associated frustum.

As expected, the greatest speed-ups were found with large but sparse data sets like *vertebra*. For dense objects with little empty space like *vmhead (skin)*, hardly any optimization is possible. The amount of empty space depends on the transfer function, and so does the speed-up, as shown with the *hand* and *vmhead* data set, where different transfer functions for showing skin and bone structures are applied. Since the optimization introduces an additional overhead, our approach can only result in a significant speed-up if the costs of sampling empty voxels are higher than those for generating the occlusion frustums. Fortunately, optimization is most useful for high-resolution data sets and high-quality rendering, where the costs for sampling empty voxels will also be high. As the runtime of our approach is mainly dependent on the resulting 2D image resolution, not 3D data set resolution, it will produce good results in this case. The costs for optimization increase with the viewport size, but the amount of saved sampling operations increases proportionally. Therefore the technique scales well even to the quite large viewport size of 1024^2, for some data sets even producing greater speed-ups than for 512^2.

Besides with rotations, we also tested the performance when the point of view is moved to a random position after each frame. While reprojection methods cannot give valid results in this case, our approach can even then produce an acceleration compared to the cube proxy geometry, as indicated in Table 3. Random viewpoint changes destroy frame-to-frame-coherence and therefore reduce the optimization efficiency. For some data sets like *engine* the optimization overhead then gets larger than the costs of sampling empty voxels, leading to a performance decrease. But nonetheless a correct image is rendered in any case, and an optimization is still reached for some data sets. Comparing our results with those of similar previous methods, Klein et al. [KSSE05] report a speed-up of up to 1.7 for semi-transparent volume rendering with small view changes. Our results on similar data sets are comparable if not better than their semi-transparent rendering, possibly because no hole-filling is required. Hence, our technique is competitive with point-based reprojection, with the additional benefit that it allows also large viewpoint changes without introducing artifacts.

data set	cube	opt.	s
vertebra	16.8	22.9	1.36
vmhead (bone)	19.1	19.5	1.02
engine	25.6	19.8	0.77

Table 3: *Frame rates and speed-up factor for random viewpoint selection on a 512^2 viewport.*

7. Conclusion

After examining architectural limitations of graphics hardware with respect to raycasting optimization, we have presented a purely GPU-based method for accelerating volume raycasting using empty space leaping. By exploiting the temporal coherence between consecutive frames we achieve a speed-up of up to a factor of two. Artifacts could be caused by undersampling, but as the created geometry is constantly refreshed, they will be removed immediately. As it requires no preprocessing, the technique is also suitable for data exploration applications. Best results are expected when large parts of the data set are removed by the transfer function, a typical operation for data exploration. Compared to point-based reprojection methods, our approach results in a proxy geometry which can be used for any point of view, though optimal space leaping is expected for small view changes. Holes can never appear, consequently no hole filling is necessary which would trigger full raycasting for unknown regions and reduce optimization efficiency. The technique can be easily integrated into an existing GPU-based raycasting infrastructure, as it requires only minimal changes within the raycaster and has no external dependencies. Hence, our approach could be suitable as a general acceleration technique to speed up volume rendering and thereby make more complex visualizations possible. As future work we might examine how to do incremental optimization of the proxy geometry by refining the previous occlusion frustums after each view change, instead of recreating them from scratch. This would further

		viewport 512^2				viewport 1024^2			
data set	size	cube	opt.	s	triangles	cube	opt.	s	triangles
vertebra	512^3	19.1	43.1	2.26	33,876	7.6	18.9	2.49	167,904
aneurysm	256^3	29.7	55.1	1.86	47,688	8.5	15.6	1.84	182,208
hand (skin)	256^3	19.4	25.1	1.29	65,772	5.6	7.1	1.27	250,056
hand (bone)	256^3	25.7	48.8	1.90	53,676	7.6	12.9	1.70	217,524
backpack	$512^2 \times 373$	24.0	44.6	1.86	31,608	8.4	16.8	2.00	128,724
vmhead (skin)	256^3	36.9	39.1	1.06	81,576	4.0	3.9	0.98	327,624
vmhead (bone)	256^3	27.0	36.7	1.36	54,888	7.1	9.2	1.30	216,060
engine	$256^2 \times 128$	24.2	27.7	1.14	47,664	7.4	8.3	1.12	189,672
engine (interior)	$256^2 \times 128$	37.5	64.6	1.72	24,300	11.3	21.9	1.94	93,132
stagbeetle	$416^2 \times 247$	8.5	15.3	1.80	37,464	3.1	6.6	2.13	135,108

Table 4: *Results for different data sets, with average frame rates for a full rotation of the object using a cube proxy geometry and our occlusion frustum optimization, resulting speed-up factor s, and triangle count of the optimized proxy geometry.*

improve optimization, but might also increase the geometry calculation costs.

Acknowledgments

The authors wish to thank the anonymous reviewers for their helpful comments. Additional implementation work was done by Stefan Diepenbrock. This work was partly supported by grants from Deutsche Forschungsgemeinschaft, SFB 656 MoBil Münster (project Z1). The presented concepts have been integrated into the Voreen volume rendering engine (http://www.voreen.org).

References

[ASK92] AVILA R., SOBIERAJSKI L., KAUFMAN A.: Towards a comprehensive volume visualization system. In *Proc. of IEEE Visualization* (1992), pp. 13–20.

[GR90] GUDMUNDSSON B., RANDÉN M.: Incremental generation of projections of CT-volumes. In *Proc. of the First Conference on Visualization in Biomedical Computing* (1990), pp. 27–34.

[GT07] GRAU S., TOST D.: Frame-to-frame coherent GPU ray-casting for time-varying volume data. In *VMV 2007: Proc. of the Vision, Modeling, and Visualization Conference* (2007), pp. 61–70.

[Hou07] HOUSTON M.: Understanding GPUs through benchmarking. In *SIGGRAPH '07 courses* (2007).

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proc. of Eurographics* (2005), pp. 303–312.

[KSSE05] KLEIN T., STRENGERT M., STEGMAIER S., ERTL T.: Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Proc. of IEEE Visualization* (2005), pp. 223–230.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proc. of IEEE Visualization* (2003), pp. 287–292.

[LK04] LAKARE S., KAUFMAN A.: Light weight space leaping using ray coherence. In *Proc. of IEEE Visualization* (2004), pp. 19–26.

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proc. of IEEE Visualization* (2003), pp. 317–324.

[LNM06] LEUNG W., NEOPHYTOU N., MUELLER K.: SIMD-aware ray-casting. In *Proc. of Volume Graphics* (2006), pp. 59–62.

[RGW*03] RÖTTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *VISSYM '03: Proc. of the Symposium on Data Visualization* (2003), pp. 231–238.

[SDDS00] SCHAUFLER G., DORSEY J., DECORET X., SILLION F. X.: Conservative volumetric visibility with occluder fusion. In *Proc. of SIGGRAPH* (2000), pp. 229–238.

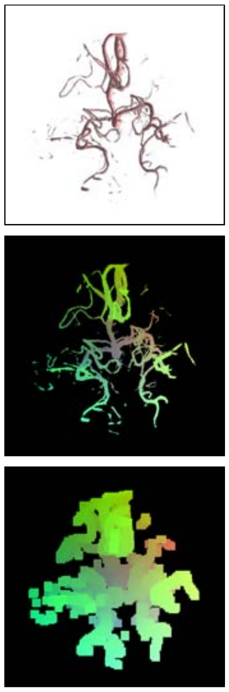
[SHN*06] SCHARSACH H., HADWIGER M., NEUBAUER A., WOLFSBERGER S., BÜHLER K.: Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *EUROVIS – Eurographics/IEEE VGTC Symposium on Visualization* (2006), pp. 315–322.

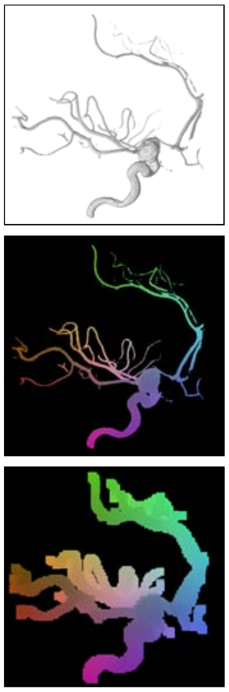
[WS01] WESTERMANN R., SEVENICH B.: Accelerated volume ray-casting using texture mapping. In *Proc. of IEEE Visualization* (2001), pp. 271–278.

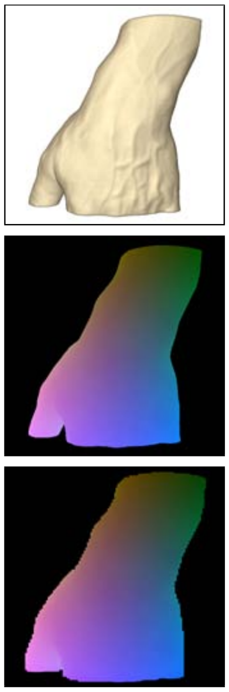
[WSK02] WAN M., SADIQ A., KAUFMAN A.: Fast and reliable space leaping for interactive volume rendering. In *Proc. of IEEE Visualization* (2002), pp. 195–202.

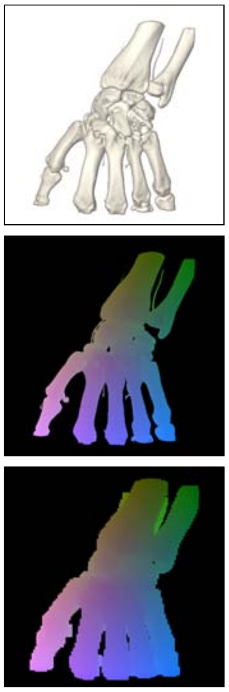
[YDKN97] YOON I., DEMERS J., KIM T., NEUMANN U.: Accelerating volume visualization by exploiting temporal coherence. In *Proc. of IEEE Visualization* (1997), pp. 21–24.

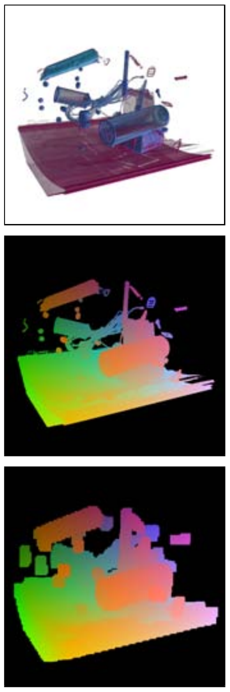
[YS93] YAGEL R., SHI Z.: Accelerating volume animation by space-leaping. In *Proc. of IEEE Visualization* (1993), pp. 62–69.

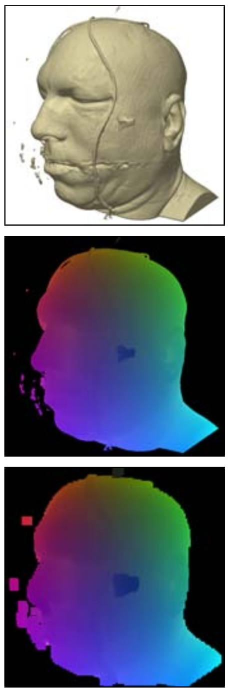

(a) *vertebra*

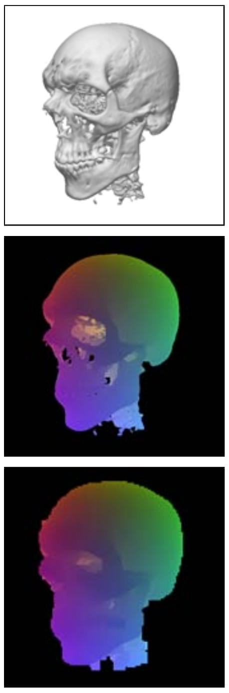

(b) *aneurysm*

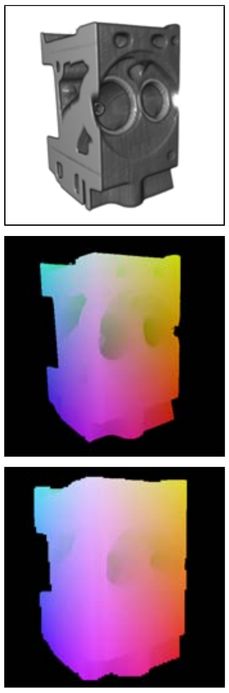

(c) *hand (skin)*

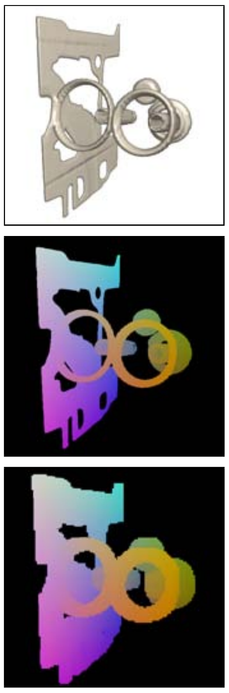

(d) *hand (bone)*


(e) *backpack*


(f) *vmhead (skin)*


(g) *vmhead (bone)*


(h) *engine*


(i) *engine (interior)*

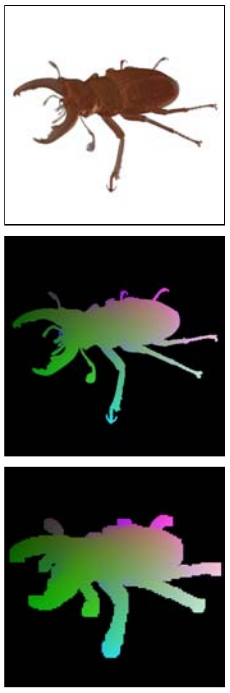

(j) *stagbeetle*

Figure 6: *Results of applying our space leaping technique to different dense and sparse data sets. Shown are the resulting final image, first-hit image, and the constructed occlusion frustum geometry which is an approximation of the first-hit image.*