

Efficient Raycasting of View-Dependent Piecewise Constant Representations of Volumetric Data

Aryaman Gupta*
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG, Dresden

Ulrik Günther
CASUS, Görlitz
Center for Systems Biology Dresden
MPI-CBG, Dresden

Pietro Incardona
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG, Dresden

Guido Reina
Visualization Research
Center, University of
Stuttgart

Steffen Frey
University of Groningen

Stefan Gumhold
Technische Universität Dresden

Ivo F. Sbalzarini†
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG, Dresden

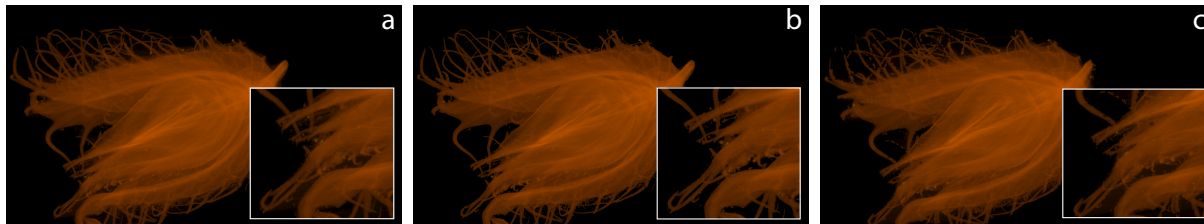


Figure 1: (a) Rendering of a 1280x720 view-dependent piecewise constant representation, at 30° rotation from the viewpoint of generation, of the Beechnut dataset (1024x1024x1546 uint16) using the proposed method at 47 fps on an Nvidia GeForce RTX 3090. (b) Reference volume rendering at 14 fps with SSIM between (a) and (b) of 0.982. (c) Accelerated preview rendering of (a) at 60 fps using the proposed method with SSIM of 0.90 w.r.t. (a).

ABSTRACT

We present an efficient raycasting-based rendering algorithm for view-dependent piecewise constant representations of volumetric data. Our algorithm leverages the properties of perspective projection to simplify intersections of rays with the view-dependent frustums that form part of these representations. It also leverages spatial homogeneity in the underlying volume data to minimize memory accesses. We further introduce techniques for skipping empty-space and for dynamic subsampling for accelerated approximate renderings at controlled frame rates. Benchmarks show that responsive frame rates can be achieved close to the viewpoint of generation for HD display resolutions, while providing high-fidelity approximate renderings of Gigabyte-sized volumes.

Index Terms: Human-centered computing—Visualization—Visualization theory, concepts and paradigms Human-centered computing—Visualization—Visualization techniques

1 INTRODUCTION

View-dependent, piecewise constant representations of volumetric data decompose the volume rendering integral into segments that store composited color and opacity. Rendering such a representation involves compositing these segments, which is much less expensive than performing the full integration [7], and produces high-fidelity approximations for camera viewpoints near the viewpoint from which the representation was generated [5, 9]. These representations are much smaller than the original volume data and can be generated and streamed efficiently [4, 5], providing an attractive solution for interactive remote rendering. However, available

rendering techniques for visualizing view-dependent piecewise constant representations do not scale to display and volume resolutions common today.

In this work, we present an efficient raycasting-based rendering method that is designed to scale to large volumes and high-resolution (full-HD) displays. At the core of our raycasting algorithm is a simplified way of computing intersections of rays with segments by computing them in clip space, as well as minimizing memory accesses by leveraging spatial homogeneity in the data. We additionally show how empty regions can be efficiently skipped in our raycasting algorithm, and we propose a technique to subsample a VDI to maintain a desired frame rate. We benchmark our algorithm on multiple datasets and with different levels of compression, and package our implementation as an extension of an existing open-source visualization library *omitted for blind review*.

2 BACKGROUND AND RELATED WORK

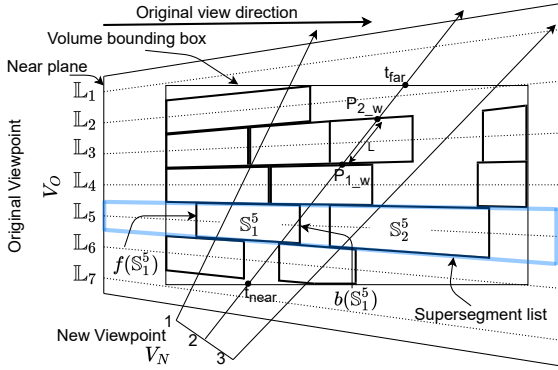
Frey et al. [5] introduced the Volumetric Depth Image (VDI) as a compressed, view-dependent piecewise constant representation of volumetric data, and we adopt the nomenclature introduced therein. Fig. 2a illustrates the structure of a VDI. Each ray cast into the volume from the original viewpoint (V_O) decomposes the rendering integral into so-called *supersegments*. Each ray i generates a list \mathbb{L}_i of supersegments \mathbb{S}_j^i that store the distance of their front and the back faces from the near-plane, $f(\mathbb{S}_j^i)$ and $b(\mathbb{S}_j^i)$, respectively. Each \mathbb{S}_j^i contains pre-classified accumulated color and opacity between $f(\mathbb{S}_j^i)$ and $b(\mathbb{S}_j^i)$, potentially determined with global lighting techniques, such as ambient occlusion. Fully transparent regions in the volume are not included in supersegments.

Given perspective projection during VDI generation, \mathbb{L} evenly subdivides the space spanned by the respective view frustum. As such, all \mathbb{L}_i and \mathbb{S}_j^i are irregular pyramidal frustums themselves.

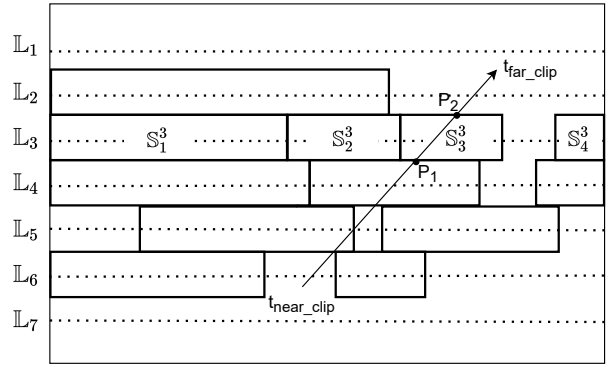
VDI generation techniques [2, 5, 9] predefine a maximum number of supersegments per-list \mathbb{N}_S , which determines the size of the VDI

*e-mail: aryaman.gupta@tu-dresden.de

†e-mail: ivo.sbalzarini@tu-dresden.de



(a) The structure of the Volumetric Depth Image [5] and the rendering calculations that are performed in world space. \mathbb{S} and \mathbb{L} form pyramidal frustums in world space. The start and end points for ray marching are determined by intersecting the viewport and the volume bounding box. Once S_j^i intersection points are determined in clip space, they are converted back to world space to determine the length of intersection.



(b) Raycasting the VDI in clip space of V_O for determining \mathbb{S} intersections. The start and end points of the ray are converted to clip space, and the ray steps through \mathbb{L} , determining \mathbb{S} intersections. The intersection points are converted back to world space to determine intersection length.

Figure 2: The structure of the VDI and the coordinate system transformation involved in the rendering.

along with the viewport dimensions, i.e., the number of lists, $N_{\mathbb{L}}$. Typically, the number of lists is much greater than the number of per-list supersegments, i.e., $N_{\mathbb{L}} \gg N_{\mathbb{S}}$. Different criteria have been used to determine supersegment lengths. Brady et al. [2] generate equal-sized supersegments in each ray. Lochmann et al. [9] divide the accumulated transmittance along the ray equally among the supersegments, while Frey et al. [5] use homogeneity within the supersegments as the criterion, and we follow this approach in our implementation. Note that our proposed method to render VDIs is agnostic to the method with which the VDI is generated.

Rendering a VDI representing a volume data set requires integrating over the S_j^i instead of the original voxels. Several methods have been proposed for this. Brady et al. [2] rely on equal-sized supersegments in each list for their alpha-blending-based rendering approach, and therefore cannot generalize to supersegments with arbitrary lengths. Frey et al. [5] propose an object-space approach that creates frustum geometry for each supersegment S_j^i . The supersegment lists \mathbb{L}_i are sorted for the new viewpoint and rendered using alpha-blending. The opacity contribution from S_j^i is based on the intersection length with a ray from the new viewpoint. This approach, however, requires creating six triangles to represent the front faces of each individual S_j^i , leading to an excessive total number for high-resolution VDIs.

Ray-based techniques do not create any geometry and therefore scale better to higher-resolution VDIs. Being image-based techniques, they also allow for early ray termination and can better leverage the anisotropy of the VDI, as rays can march quickly along the lists. In the raycasting method by Lochmann et al. [9], a ray from V_N is projected onto V_O , rasterized, and traversed using DDA (digital differential analyzer) to determine the lists intersected. For each intersected list, it then computes intersections with the pyramidal frustum-shaped supersegments. Our method, on the other hand, carries out all S_j^i intersections in the clip space of V_O where all S_j^i and \mathbb{L}_i are cuboids Fig. 2b, thereby enabling the use of voxel stepping [1] to traverse \mathbb{L} , and simplifying the calculations required to determine intersections with \mathbb{S} . Additionally, we leverage spatial homogeneity across lists to minimize memory accesses (Alg. 1), propose a method to skip empty regions along rays, and a method for subsampling the ray for preview rendering.

3 VDI RENDERING BY RAYCASTING

For each pixel in the final rendered image, we cast a ray into the VDI. The ray passes through supersegment lists \mathbb{L} , searches for supersegments \mathbb{S} within them, calculates the intersection length with each intersected \mathbb{S} , thereby adjusting the color contribution obtained from the supersegment, which it accumulates using the over operator [10]. The process is illustrated in Fig. 2. The start and end points for the ray marching are determined by the intersection of the viewport that was used to create the VDI, and the bounding box of the volume in the scene. In Fig. 2, for example, the ray marching begins at t_{near} and ends at t_{far} .

To simplify the procedure of traversing through \mathbb{L} and determining intersection lengths with \mathbb{S} , the calculations are carried out in the perspective-deformed clip space of V_O . While in world space \mathbb{S} and \mathbb{L} form pyramidal frustums, in perspective clip space, they are transformed into cuboids (Fig. 2b). In clip space, we therefore have a regular 2D grid of \mathbb{L} , each of which is a cuboid. The rendering ray then traverses this grid using the fast voxel traversal algorithm by Amanatides and Woo [1]. The traversal not only determines which \mathbb{L} are intersected by the ray, but also returns the intersection points with a given \mathbb{L}_i , which are then used to search for S_j^i within \mathbb{L}_i .

For each intersected \mathbb{L} , we need to find the \mathbb{S} that cover the region between the entry and exit points of the ray. Once the first of these S_j^i is determined (if any), finding the next is trivial, since the \mathbb{S} within \mathbb{L} are implicitly sorted by their position. The algorithm simply needs to check the next or preceding index, depending on the relative direction of the ray within \mathbb{L} , which is evaluated by the sign of the scalar product between the ray and the original ray vector.

To determine the first intersected S_j^i in \mathbb{L}_i , we use information from the intersections in the previously intersected \mathbb{L}_k . For the very first \mathbb{L} intersected by the ray, no such information exists, and therefore a binary search is done over the \mathbb{S} in the first \mathbb{L} . For every subsequent \mathbb{L}_i , the index p of the last S_p^k found in the previous \mathbb{L}_k is used as an initial guess. If no \mathbb{S} was found, p is the index of the nearest \mathbb{S} from the exit point in \mathbb{L}_k . If S_p^i is not the first supersegment intersected in the \mathbb{L}_i , the adjacent index is checked depending on which side of the ray S_p^i lay. If no S_j^i is found still, then a binary search is carried out among the relevant \mathbb{S} indices. Algorithm 1 describes this process in detail. In Fig. 2b, for example, when the ray enters \mathbb{L}_3 , it first tests for intersection with S_2^3 , which was the last index intersected in \mathbb{L}_4 . Since S_2^3 lies behind the entry point, S_3^3

is tested next which is found to intersect the ray.

Our \mathbb{S} search routine (see Alg. 1) leverages the spatial smoothness in the volume data on which the VDI was created: since neighbouring \mathbb{L} are created from rays that pass nearby in the data, they are likely to contain similar sized \mathbb{S} . The algorithm is aimed at minimizing costly memory accesses, and provides the method with good scaling properties with respect to $N_{\mathbb{S}}$ (Table 1).

Algorithm 1 Find the first supersegment on a ray entering a list \mathbb{L}_i

Input: (i_{entry}, i_{exit}) : entry and exit intercepts of the ray in \mathbb{L}_i , p : the index of the last intersected \mathbb{S} in the previous \mathbb{L}_k

Output: index j of the first \mathbb{S} intersected in \mathbb{L}_i , -1 if no such \mathbb{S}

Where: $BinS(i_p)$ is a binary search function returning an index j such that $b(\mathbb{S}_j^i) < i_p < b(\mathbb{S}_{j+1}^i)$

```

1: if  $p$  is -1 then
2:    $BinS(i_{entry})$ 
3: else
4:   if does not exist  $\mathbb{S}_p^i$  or  $b(\mathbb{S}_p^i) \geq i_{entry}$  then
5:     if  $b(\mathbb{S}_{p-1}^i) < i_{entry}$  then
6:       if exists  $\mathbb{S}_p^i$  then  $j = p$  else  $j = BinS(i_{entry})$ 
7:     else
8:       if  $b(\mathbb{S}_{p-2}^i) < i_{entry}$  then
9:         if exists  $b(\mathbb{S}_{p-1}^i)$  then  $j = p - 1$  else  $j =$ 
            $BinS(i_{entry})$ 
10:    end if
11:    end if
12:  else
13:    if  $b(\mathbb{S}_{p+1}^i) \geq i_{entry}$  then  $j = p + 1$  else  $j = BinS(i_{entry})$ 
14:  end if
15: end if
16: if  $f(\mathbb{S}_j^i) > i_{exit}$  then
17:   return  $j$ 
18: else
19:   return -1
20: end if

```

4 EMPTY-SPACE SKIPPING

The proposed image-order raycasting technique has a number of advantages over object-order techniques, as outlined in Sect. 2, but suffers from the lack of inherent support for skipping empty regions, which are common in most datasets and transfer functions. Since the performance of the proposed method heavily depends on the number of memory accesses along a ray, we require an acceleration data structure built on top of the VDI to skip empty regions.

The VDI does contain the information required to skip empty space implicitly within a \mathbb{L} : consecutive \mathbb{S} are sorted, and contain front and back depths indicating their positions along \mathbb{L} . However, this information cannot be queried based on the current ray position when moving from one \mathbb{L} to the next. In Fig. 3, Ray 1 must sample each \mathbb{L}_i at least once even in empty areas, to determine that no \mathbb{S}_j^i in those \mathbb{L}_i is intersected.

We therefore use a grid data structure that informs whether the region covered by a grid cell is empty or not. When the ray lands in a cell that is empty, it can immediately jump to the other end of the cell. Ray 2 in Fig. 3, for example, thus skips several memory accesses in empty regions. Of course, the grid data structure itself must be accessed once to identify an empty cell.

The grid cells are created such that they have a constant depth extent in view space. In clip space, this corresponds to cells that are larger near the near plane, and smaller near the far plane, due to the non-linear conversion. Each cell spans an equal number of \mathbb{L} along both dimensions of the viewing plane. The depth of a cell in view

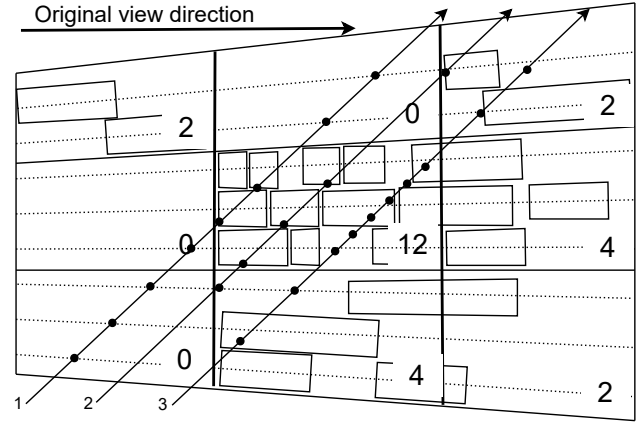


Figure 3: The grid data structure used for empty-space skipping and for preview rendering. Numbers in lower right corners are the values stored in each cell, which indicate the number of supersegments intersecting that cell. The rays illustrate different traversal strategies: Ray 1 is the base raycasting algorithm, ray 2 skips empty cells, and ray 3 subsamples the VDI for preview rendering. Black dots indicate points at which the rays query the VDI to search for a supersegment in a list.

space is much larger than its width or height, in keeping with the anisotropic compression in the VDI: there are far fewer \mathbb{S}_j^i along \mathbb{L}_i than there are \mathbb{L}_i in the viewing plane.

The 3D grid data structure could be extended to hierarchical structures that provide better empty-space skipping performance, such as an octree [8] or sparse-leap [6]. Although, for empty space skipping, it would be sufficient to simply store a 1 in a grid cell to indicate that it is not empty, we store instead the total number of \mathbb{S} contained in the region covered by the cell, as shown in Fig. 3. Grids that span multiple cells are considered as belonging to each one. The generation of the grid data structure can be integrated into the generation of the VDI, with each \mathbb{S}_j^i generated triggering an atomic add on the appropriate grid cell. The total number of \mathbb{S} in a cell is required for preview rendering when the viewpoint changes significantly, as explained in the next section.

5 DYNAMIC SUBSAMPLING FOR PREVIEW RENDERING

The rendering performance of the proposed raycasting algorithm depends on the number of memory accesses made by the rays as they traverse the VDI. In the original view direction (V_O), and nearby, high frame rates can be achieved as the VDI is compressed along V_O . When the viewpoint changes significantly, rendering performance reduces, as a larger component of the ray is along one of the view plane dimensions. In a real-time remote rendering application, a new VDI could be generated when the viewpoint changes significantly, which, once generated, would provide fast and high quality rendering near the new viewpoint. While the new VDI is generated and streamed, however, the rendering of the old VDI may need to maintain rendering performance, trading off quality, to provide preview rendering. This is especially important in applications that require a frame rate guarantee, such as Virtual Reality.

We achieve this by sampling a dynamically determined number of points along the ray when the frame rate falls below the desired value. No \mathbb{S} intersections are calculated. As the ray marches, it simply queries which \mathbb{S}_j^i a given sample point lies within, if any, and obtains the color from \mathbb{S}_j^i . Length-based opacity correction is applied based on the distance (in non-empty cells) from the previous sample point, thereby making the assumption that the emittance and

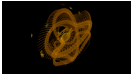
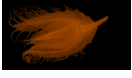
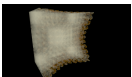
Dataset	N_L	Volume		VDI, N_S15						VDI, N_S30					
		f, 5°	f, 30°	5°			30°			5°			30°		
				f	f_e	s	f	f_e	s	f	f_e	s	f	f_e	s
 Kingsnake 1024x1024x795 8bit	1280x720	31	30	231	277	0.982	63	92	0.966	207	235	0.984	58	97	0.978
	1920x1080	17	17	90	126	0.985	20	32	0.964	80	110	0.987	19	36	0.980
 Beechnut 1024x1024x1546 16bit	1280x720	18	17	166	239	0.99	47	68	0.982	143	175	0.992	40	59	0.988
	1920x1080	16	14	68	100	0.991	15	24	0.982	57	74	0.993	13	21	0.988
 Richtmyer- Meshkov [3] 2048x2048x1920 8bit	1280x720	9	7	267	302	0.948	83	108	0.970	218	226	0.97	71	87	0.983
	1920x1080	6	4	106	121	0.958	28	39	0.976	84	91	0.976	24	32	0.986

Table 1: Benchmarking the proposed rendering algorithm, including the empty-space skipping technique, for different datasets, at different VDI resolutions, and at different degrees of movement around V_O . Columns titled 'f' represent frame rates without empty space skipping, f_e is the frame rate with empty space skipping, and s is the similarity metric SSIM. Inset images show baseline volume rendering at V_O .

transmittance stored in S_j^i remain constant over the jump distance. Since S_j^i intersections do not need to be calculated, the subsampling takes place directly in world space, which also allows for the samples to be taken at regular intervals along the ray. Each sample point determines which L_i it lies within, based on its x- and y-coordinates in clip space, and then uses Alg. 1 to determine the S_j^i along L_i .

The total number of samples to be taken along the ray is determined based on the render time of the last 5 frames. If the average frame rate differs from the desired value by more than a user-defined threshold, a rough estimate of the cost per sample is calculated as the average frame time / number of samples, and the number of samples is accordingly adjusted.

The ray only samples the VDI in non-empty cells of the grid acceleration data structure. The total number of samples is divided among the non-empty cells intersected by the ray, with the sampling frequency in a cell directly proportional to the number of S contained in the region covered by that cell, which is stored in the grid (Sect. 4). Ray 3 in Fig. 3, for example, samples the grid with 12 S more finely than the ones with 4 and 2 S .

6 IMPLEMENTATION AND EVALUATION

Our raycasting algorithm is implemented using Vulkan compute shaders. It is available open-source, implemented as an extension of the *omitted for blind review* visualization library. All benchmarks were carried out on a workstation with an Nvidia GeForce RTX 3090, running Ubuntu 20.04. For technical reasons, volume datasets greater than 2 GB in size needed to be split into slices before loading for volume rendering. VDIs of a given viewport resolution N_L were always rendered at N_L display resolution.

We evaluate our method on a VDIs of different sizes, generated on various volume datasets. Results are reported in Table 1. VDIs were generated at V_O , chosen such that the data filled the viewport. Image similarity was compared using the Structural Similarity (SSIM) metric [11], where higher values are better, 1.0 representing identical images. Baseline volume-rendering performed simple over-operator based accumulation, with a 1D transfer function, and no advanced lighting.

The performance of the rendering algorithm was found to scale well with N_S . Despite double the number of S generated by non-empty L in the N_S30 configuration, frame rates drop by $\sim 20\%$ in most cases. This is in contrast to object-space approaches [5] which scale directly with both N_S and N_L . On the other hand, frame rates fell sharply when using the full-HD resolution, a consequence of the fact that 4x rays were to be cast, with each ray stepping through a

Target fps	40 fps	50 fps	60 fps	70 fps	80 fps
SSIM	0.9059	0.9025	0.8978	0.8917	0.8836

Table 2: SSIM similarity between full VDI rendering and images produced by dynamic subsampling for a target fps on a 1920x1080 VDI of the Beechnut dataset with N_S15 at $V_N=30^\circ$.

higher N_L . Rendering quality was found to be higher when using N_S30 , with the difference more pronounced at the 30° deviated viewpoint. An anomaly in this regard was the Richtmyer-Meshkov dataset [3], where the higher approximation quality at 30° was attributed to the choice of V_O which presented more detail at the 5° viewpoint. Empty-space skipping was found to perform well, providing speed ups in each case, and $>20\%$ in most cases. The observed heavy reliance of performance on V_N is expected due to the anisotropic view-dependent compression in the VDI.

The dynamic subsampling approach for preview rendering was evaluated on the Beechnut dataset, at the 1920x1080 resolution with N_S15 , at 30° around V_O . The full-resolution VDI rendering in this configuration yielded 24 fps with empty-space skipping (Table 1). Table 2 reports the SSIM between the image produced by the proposed subsampling method for a target frame-rate, and the reference full-resolution VDI rendering.

7 CONCLUSIONS

We presented an efficient raycasting algorithm for view-dependent piecewise constant representations of volumetric data, such as the Volumetric Depth Image (VDI) [5]. We used raycasting with super-segment intersection computed in clip space, efficient empty-space skipping, and minimized memory accesses by exploiting spatial smoothness in the data. This resulted in a VDI rendering algorithm suited for generating high-resolution images of large volume data at responsive frame rates.

We presented benchmarks on different volume datasets and for different VDI compression ratios, rendering images at HD and Full HD resolution. In all cases, the frame rates achieved by our approach were significantly higher than those achieved by volume rendering. We observed >200 fps for small (5°) differences between the viewpoint from which the VDI was generated and the rendering viewpoint. For large (30°) viewpoint differences, frame rates remained >20 fps when empty-space skipping was used.

We further introduced dynamic subsampling for accelerated approximate rendering of VDIs. This enabled achieving desired frame rates at the expense of rendering quality, which we quantified. We

anticipate that the proposed approach may be particularly useful in interactive remote rendering applications.

ACKNOWLEDGMENTS

This work was supported by the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) Dresden/Leipzig, funded by the Federal Ministry of Education and Research (BMBF, Bundesministerium für Bildung und Forschung). This work was partially funded by the Center for Advanced Systems Understanding (CASUS), financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament. We acknowledge the Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zurich (UZH) for the acquisition of the μ CT Beechnut dataset. We acknowledge the University of Texas High-Resolution X-ray CT Facility (UTCT) for the acquisition of the Kingsnake dataset, under NSF grant IIS-9874781.

REFERENCES

- [1] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *EG 1987-Technical Papers*. Eurographics Association, 1987. doi: 10.2312/egtp.19871000
- [2] M. Brady, K. Jung, H. Nguyen, and T. Nguyen. Two-phase perspective ray casting for interactive volume navigation. In *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, pp. 183–189, 1997. doi: 10.1109/VISUAL.1997.663878
- [3] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-dimensional simulation of a richtmyer–meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002. doi: 10.1063/1.1504452
- [4] O. Fernandes, S. Frey, F. Sadlo, and T. Ertl. Space-time volumetric depth images for in-situ visualization. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 59–65, 2014. doi: 10.1109/LDAV.2014.7013205
- [5] S. Frey, F. Sadlo, and T. Ertl. Explorable volumetric depth images from raycasting. In *2013 XXVI Conference on Graphics, Patterns and Images*, pp. 123–130, 2013. doi: 10.1109/SIBGRAP.2013.26
- [6] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. Sparseleap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974–983, 2018. doi: 10.1109/TVCG.2017.2744238
- [7] A. Kaufman and K. Mueller. *Overview of Volume Rendering*, vol. 7, pp. 127–XI. 12 2005. doi: 10.1016/B978-012387582-2/50009-5
- [8] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash. Octree rasterization: Accelerating high-quality out-of-core gpu volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(10):1732–1745, 2013. doi: 10.1109/TVCG.2012.151
- [9] G. Lochmann, B. Reinert, A. Buchacher, and T. Ritschel. Real-time Novel-view Synthesis for Volume Rendering Using a Piecewise-analytic Representation. In M. Hullin, M. Stamminger, and T. Weinkauff, eds., *Vision, Modeling 'I&' Visualization*. The Eurographics Association, 2016. doi: 10.2312/vmv.20161346
- [10] T. Porter and T. Duff. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '84*, p. 253–259. Association for Computing Machinery, New York, NY, USA, 1984. doi: 10.1145/800031.808606
- [11] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861