STAG: Smart Tools and Applications in Graphics (2021)
P. Frosini, D. Giorgi, S. Melzi, and E. Rodolà (Editors)

Remote Volume Rendering with a Decoupled, Ray-Traced Display Phase

Stefan Zellmann[†1,2]

[1]University of Cologne, Department of Computer Science
[2]Bonn-Rhein-Sieg University of Applied Sciences, Institute of Visual Computing

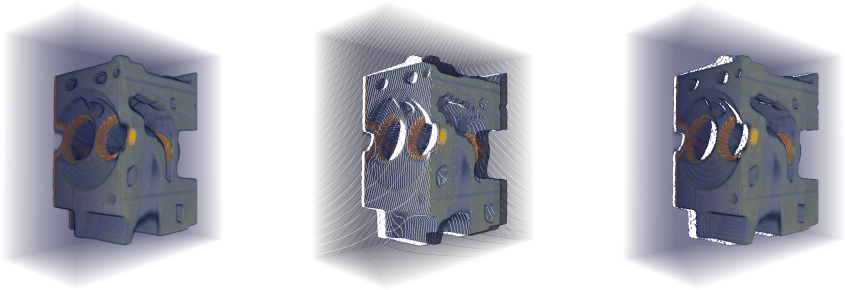

Figure 1: *Remote volume rendering where 2.5D depth images from outdated frames are reprojected to hide latency. Left: reference image. Middle: 2.5D reprojection with OpenGL point primitives. Right: object space reprojection using ray tracing. The ray tracing-based technique is less susceptible to reprojection artifacts as the footprint of the reprojected points depends on the distance to the viewer.*

Abstract

We propose an image warping-based remote rendering technique for volumes that decouples the rendering and display phases. For that we build on prior work where we sample the volume on the client using ray casting and reconstruct z-values based on heuristics. Color and depth buffers are then sent to the client, which reuses this depth image as a stand-in for subsequent frames by warping it to reflect the current camera position and orientation until new data was received from the server. The extension we propose in this work represents the depth pixels as spheres and ray traces them on the client side. In contrast to the reference method, this representation adapts the footprint of the depth pixels to the distance to the camera origin, which is more effective at hiding warping artifacts, particularly when applied to volumetric data sets.

CCS Concepts

• ***Human-centered computing*** → ***Visualization techniques; Scientific visualization;*** • ***Computing methodologies*** → ***Ray tracing; Graphics processors;***

1. Introduction

Remote rendering is an important technique to overcome the typical bandwidth limitations in in-situ scenarios, or when accessing graphics workstations over LAN or WAN using thin clients. Remote rendering algorithms can be classified by the type of data—image pixels, proxy geometry, etc.—that is sent over the network, and by the amount of post-processing that needs to be done on the client, with the spectrum ranging from *send-image* over *send-geometry* to *send-data* approaches [BCH12]. According to this classification, send-image implementations execute the full rendering pipeline on the remote server or workstation, while the client is responsible only for display.

We present a remote rendering technique based on prior work by Zellmann et al. [ZAL12] that decouples the rendering and display phases. By that, latency introduced by the network or the rendering algorithm itself can be hidden and the user interface remains responsive. This is an important property for certain use cases, e.g. for virtual reality applications with head tracking, and can help to improve the overall user experience.

With real-time ray tracing nowadays being widely available even on consumer hardware, we present and evaluate a simple improvement to the algorithm by Zellmann et al. that does not render images directly, but interprets depth pixels (color + depth) as object

† zellmann@uni-koeln.de

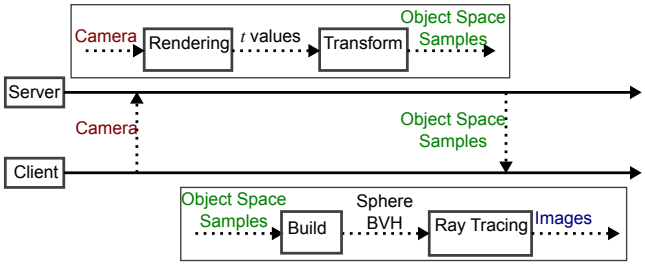

Figure 2: *Overview of the client/server remote rendering architecture. After receiving camera information, the server uses ray marching to generate primary ray parameter values t_i for each pixel and transforms those into object space samples. The client receives those object space samples, reinterprets them as colored, semi-transparent spheres, builds a bounding volume hierarchy (BVH) and ray traces them using a simple first-hit ray tracing query and with no shading other than the constant sample color.*

space splats and renders them as spheres in a ray tracer. This enhancement can help to conceal reprojection artifacts and is specifically helpful for remote *volume* rendering, which the algorithm was originally designed for. On top of that, as a byproduct, we have an acceleration data structure available over the depth images that is rebuilt per frame and that can for example be used to perform neighbor queries.

2. Background

We present a remote volume rendering technique that broadly falls into the category of *send-image* approaches [BCH12], where fully rendered images are sent over the network. Our technique builds on prior work by Zellmann et al. [ZAL12]. In that work, the authors decoupled the rendering and display phases by presenting final images at a different rate than delivered by the server. On the server, the authors render 2.5D images (color + depth). They proposed a set of heuristics to generate a depth buffer during volume rendering, which is generally challenging in the presence of alpha transparency.

On the client, a vertex buffer object and a color texture are used to render the depth buffer as point primitives. When the camera position or orientation changes, the current buffer is warped according to the new transformation; the pixel buffer is updated as soon as the server sends an updated image. When the rendering phase on the server and the display phase on the client operate at exactly the same rate, this will result in the client always displaying the correct image. At different rates, the client will display pixels from outdated frames that hence appear warped. This effect is more exaggerated as the delay between the two phases increases. On the other hand, assuming that an image can be displayed faster than it can be rendered, the user interaction appears smooth because user input can be processed at high rates.

3. Related Work

Send-image remote rendering is a popular approach that has for example been proposed by Stegmaier et al. [SME02]. Visualization tools like ParaView [AGL05] support client / server rendering modes that exchange images. Dedicated remote rendering tools like VirtualGL [vir] allow the user to use accelerated graphics over a broadband network via send-image remote rendering. We refer the reader to the text book by Bethel et al. [BCH12] and the survey article by Shi and Hsu [SH15] for an introduction to and a good general overview of the various remote rendering techniques.

The idea to use image warping in low bandwidth scenarios is relatively old and was e.g. proposed by Bao et al. [BG03]. Research has focused on augmenting send-image approaches with additional data like depth buffers [ZAL12], image layers [LRBR16], or light fields [MBGM20]. The work by Shi et al. [SNC12] has focused on image warping techniques using depth images targeting mobile devices. The paper by Pajak et al. [PHE*11] has explored compression techniques based on spatio-temporal upsampling on the client that also includes the use of depth buffers sent over the network.

The work by Schied et al. [SKW*17, SPD18] on spatio-temporal variance-guided filtering is also related to our approach, as it is based on rendering with outdated image samples. Their technique is however based on sample accumulation and on extrapolating samples into the future using motion vectors, whereas our approach, in comparison, predicts the present image samples based on past image data. While the motion vectors with Schied's algorithm are explicit, it can be argued that the technique by Zellmann et al. [ZAL12] *implicitly* stores a per-pixel motion vector, and while Schied's technique interpolates the motion values, Zellmann et al.'s algorithm exhibits holes between samples.

There are several approaches that are orthogonal to our technique, such as the approach by Marton et al. [MAG19] who perform remote volume rendering of time-varying data sets and combine that with dedicated, compressed representations extracted on the server depending on the capabilities of the client. Alternative approaches that are well suited for rendering of sparse volume representations are for example multi-fragment and depth peeling methods [VVP20].

We propose to augment the remote rendering algorithm by Zellmann et al. [ZAL12] by switching from image reprojection to real-time ray tracing on the client computer. Instead of generating 2.5D data on the remote server, we generate object space samples storing the final composited color from volume ray marching. On the client, we transform those samples to a point cloud that we render using first-hit ray tracing. Qualitatively, this approach is similar to splatting [Wes90], which could be used as an alternative object order strategy.

4. Method

A challenge of the rasterization-based warping technique used by Zellmann et al. [ZAL12] is that the point primitives' size is fixed to a certain number of pixels. This can cause artifacts that can be avoided when the 2.5D data set is represented with solid objects. In the latter case, solids that are closer to the viewer will cover more pixels. Effectively, this can be regarded as splatting, where the footprint of the splats decreases with increasing distance to the viewer. While rendering of geometrically complex objects like tessellated spheres with OpenGL is prohibitive w.r.t. memory consumption, with real-time ray tracing and arbitrary user geometry it is a viable

option to render the 2.5D geometry as spheres that are represented with real quadrics. With this extension, we hope to reduce the impact of the artifacts encountered with rasterization-based 2.5D image warping—especially in the presence of volumetric data and semi-transparent pixels.

4.1. Object Space Samples

On the server side, we render the volume using a ray marcher that employs one of the heuristics presented by Zellmann et al. to estimate which depth value represents the volume best. We march rays $r = o + \vec{d}t$ with origin o, direction vector $\vec{d}$ and ray parameter t. When a representative depth according to one of the heuristics was found, that depth is associated with a certain value for $t = t_i$. Zellmann et al. first transform that value to a position in object space $p = o + \vec{d}t_i$ and then *reproject* the resulting point to obtain a position in OpenGL window coordinates by applying the viewing, camera, and viewport transforms.

In contrast to that, we directly send object space samples to the client that are comprised of the position vectors p and a footprint radius. We quantize the object space coordinate and pack it, together with the radius, into 64 bits. The ray marcher fills a screen-sized buffer with object space samples and associated volume-rendered colors. We currently just set the footprint radius to half the size of a voxel's diagonal when the ray hit the volume and integrated a color with non-zero opacity. If the integrated color has zero opacity or the ray did not hit the volume, we set the radius to zero. We use compaction to send only those object space samples to the client that have non-zero radius.

On the client, when we receive a buffer with object space samples and colors, we reinterpret the object space samples as semi-transparent spheres, build a bounding volume hierarchy from those using the LBVH algorithm [LGS*09, ZHL19], and render them as a point cloud using ray tracing. We chose LBVH for its fast rebuild times as compared to for example kd-trees [ZSL18, ZSL19]. We could also have used an OptiX BVH to make use of RT cores, which has recently been shown to be effective for volumetric ray casting [WZM21], but by using a software implementation we acknowledge that the client-side algorithm does not necessarily require a high-end GPU with latest ray tracing extensions—in fact, a typical scenario where we often productively employ our algorithm is with thin clients such as laptop PCs. We illustrate the overall process including the client and server-side communication in Fig. 2.

We also experimented with multi-hit ray tracing [ZHL17], but as the results regarding image quality were mixed, we ultimately decided to use this simple splatting approach that simply colorizes each sphere according to its designated color. We still decided to use semi-transparent spheres, which in the case of multi-hit ray tracing would have been over-composited. Instead, we use the alpha value to blend with the background color, which we found visually more appealing than rendering opaque spheres.

4.2. Implementation

We implemented the framework described above using NVIDIA CUDA and network communication using the C++ Boost Asio library. We use a standard ray marching volume renderer that writes out colors and t values to off-screen buffers. When the client sends an updated camera, the server performs volume rendering, fills the off-screen buffers and sends them to the client. Rendering, camera motion processing and communication are handled by separate threads to make the display/presentation phase asynchronous. After rendering, the server transforms and quantizes the t values and sends both position and color buffers to the client. While 16-bit quantization provided good results, we note that this parameter can be left as a user option to favor either bandwidth or rendering quality. The camera that the buffers were produced for are also sent along as meta data so the client does not need to keep track of it. The client, after receiving the buffers, builds an LBVH over the point cloud that is subsequently used to accelerate rendering. An open source version of the implementation can be found online: `https://github.com/szellmann/warpvr`.

Compaction (view-independent)					23 ms
LBVH Construction (view-independent)					11 ms
	View 1	View 2	View 3	View 4	View 5
FPS	292	276	283	229	408

Table 1: *Performance results for rendering the five views from Fig. 3 with our ray tracing technique. We render with a viewport of* 1024×1024 *pixels. The point cloud after compaction is comprised of* $623K$ *object space samples.*

5. Results and Discussion

For a qualitative comparison, we implemented the reference method by Zellmann et al. using OpenGL point rendering. Fig. 3 shows this qualitative comparison. In the example a sequence of warped frames is obtained using the gradient heuristic. For a performance evaluation we report compaction rates on the server (currently using a serial implementation), LBVH construction time, as well as average rendering performance for the five views from Fig. 3 in Table 1. For the benchmarks, we used an NVIDIA RTX 2080 GPU. We find this simple extension to the original algorithm by Zellmann et al. to be effective. As can be seen from Fig. 3, the visual quality when rendering the 2.5D point cloud as object space splats instead of OpenGL points with a fixed size in pixels improves dramatically. We also experimented with setting a variable point size in OpenGL mode but found this setting to be hard to control as it biases the rendered results in that the opacity of the composited point sprites increases with an increase in fill-rate. Besides, depth compositing for point primitives with OpenGL is not performed per fragment but per vertex.

The overall pipeline is currently bound by compaction performance on the server, which would however be easy to fix, e.g., by using the `remove_if` standard algorithm from the C++ / GPGPU library `thrust`. We deliberately do not report network-related performance as that would relate to the available bandwidth and latency of a specific network connection. We currently send 32 bits per color and 64 bits per object space sample (position and radius). The 1024×1024 pixel images in Fig. 3, after compaction, consist of $623K$ individual object space samples.

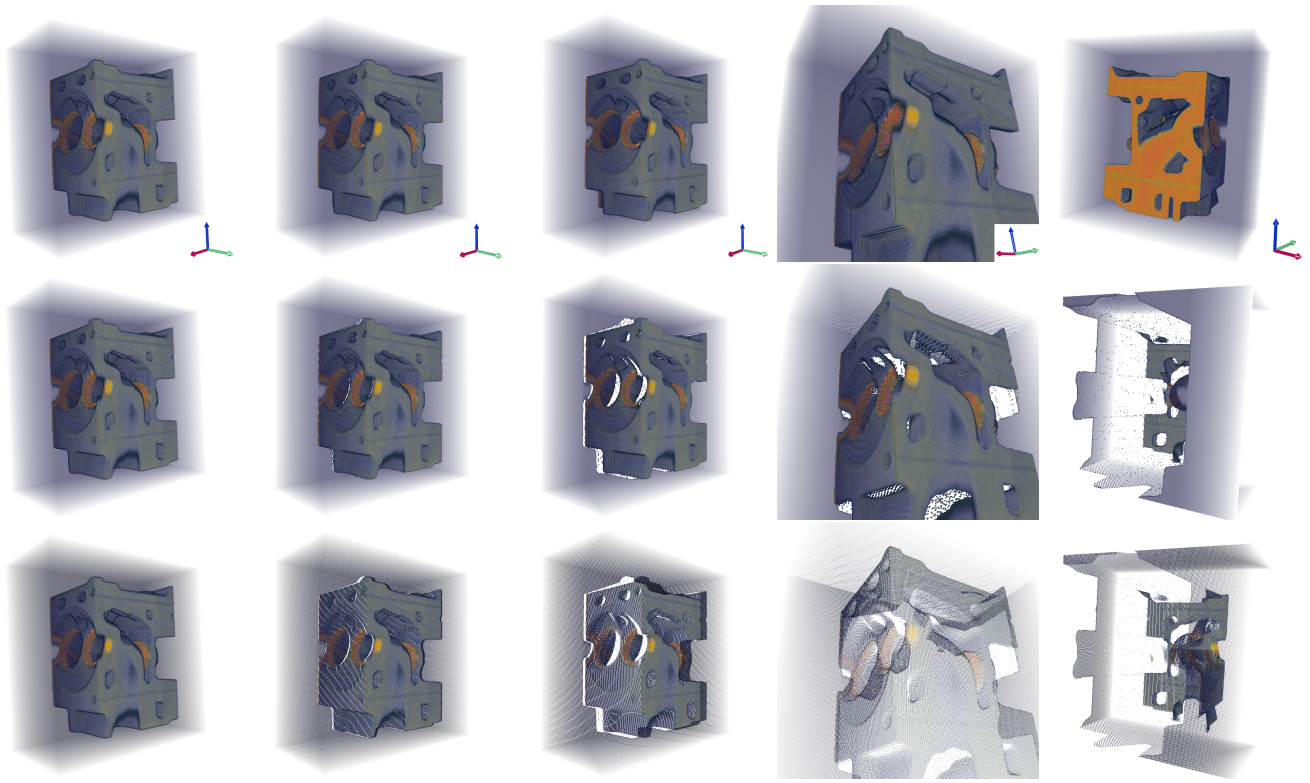

Figure 3: *Comparison between OpenGL reprojection with rasterized points, and our ray tracing technique rendering object space samples as spheres. Top row: a sequence of successive frames obtained using volume rendering, with the camera position gradually changing to positions that show the volume from different viewing angles. Middle row: the same sequence of frames. The left image is subsequently warped as a 2.5D point cloud to the viewing positions from the top row using our ray tracing technique. Bottom row: the sequence of frames is warped in the same manner, but with the OpenGL point rasterization technique by Zellmann et al. [ZAL12]. Rendering artifacts especially in regions that appear volumetric and transparent are much more exaggerated even when the camera is only slightly moved.*

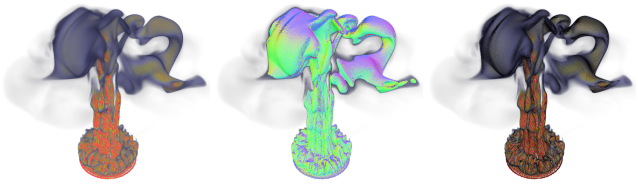

Figure 4: *Client-side normal reconstruction. Left: heptane gas data set without shading. Middle: reconstructed surface normals. Right: normals are used for shading. One advantage of using ray tracing with a BVH instead of splatting with rasterization and frustum culling is the* availability *of the BVH, which in this case is also used to perform k-nearest neighbor queries and plane fitting.*

5.1. Advantages over Object Order Techniques

Alternatives to client-side ray tracing that potentially address the rendering artifacts in a similar way would be splatting using imposters, or geometry shaders to expand the points into more complex shapes. One compelling advantage of using ray tracing instead of those object order techniques is the fact that we have a *per frame* BVH that was built over the client-side point cloud available that can be used to perform neighbor queries. A typical scenario for this is surface normal reconstruction from the point cloud on the client. We demonstrate this in Fig. 4 where we use the BVH to perform (truncated) k-nearest neighbor (kNN) queries in a local neighborhood around the primary intersection position to find nearby points. We then fit planes to these using a simple heuristic based on computing the covariance matrix and its determinant. This ultimately provides us with normal vectors that, for illustrative purposes, are used for shading. In a more complex scenario, if the volume/transfer function combination favors near isosurfaces, kNN queries could for example be used for surface reconstruction from the sparse point cloud using similar methods [BTS*14]. As neighbor queries can also be mapped to ray tracing hardware [ZWW20], the approach also lends itself to an implementation with RT Cores.

6. Conclusion and Future Work

We presented a simple yet effective extension to the algorithm by Zellmann et al. [ZAL12] that improves the rendering artifacts that this remote volume rendering technique otherwise suffers from by replacing 2.5D point warping with real-time ray tracing. This potentially comes at moderate additional costs regarding memory

bandwidth as we have to store object space coordinates, although we in turn can benefit from compaction on the server side.

Switching to a ray tracing pipeline presents us with a wealth of possibilities that we intend to explore in the future and that this work lays the groundwork for. Possible extensions would be sample accumulation across a couple of frames, or varying the radius of the object space samples depending on the uncertainty associated with the depth value: pixels for which we are uncertain where to place them along the viewing ray could be smeared out across some interval whose length is proportional to the uncertainty.

Certain angles have not been explored due to the short paper format. We deliberately have not concentrated on network communication and compression of object space samples in this paper, which is important for high throughput. Future work could also be concerned with a thorough evaluation of the client side technique. While we customarily use the algorithm in productive settings on thin clients and note that it is well suited for that, a more rigorous evaluation on various, possibly mobile, architectures would certainly be of interest and is left as future work.

References

[AGL05] AHRENS J. P., GEVECI B., LAW C. C. W.: Paraview: An end-user tool for large-data visualization. In *The Visualization Handbook*. 2005.

[BCH12] BETHEL W. E., CHILDS H., HANSEN C.: *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, Nov. 2012.

[BG03] BAO P., GOURLAY D.: Superview 3d image warping for visibility gap errors. *IEEE Transactions on Consumer Electronics 49*, 1 (2003), 177–182.

[BTS*14] BERGER M., TAGLIASACCHI A., SEVERSKY L. M., ALLIEZ P., LEVINE J. A., SHARF A., SILVA C. T.: State of the Art in Surface Reconstruction from Point Clouds. In *Eurographics 2014 - State of the Art Reports* (2014), Lefebvre S., Spagnuolo M., (Eds.), The Eurographics Association. doi:10.2312/egst.20141040.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2009.01377.x.

[LRBR16] LOCHMANN G., REINERT B., BUCHACHER A., RITSCHEL T.: Real-time novel-view synthesis for volume rendering using a piecewise-analytic representation. In *Proceedings of the Conference on Vision, Modeling and Visualization* (Goslar, DEU, 2016), VMV '16, Eurographics Association, p. 85–92.

[MAG19] MARTON F., AGUS M., GOBBETTI E.: A framework for gpu-accelerated exploration of massive time-varying rectilinear scalar volumes. *Computer Graphics Forum 38*, 3 (2019), 53–66. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13671, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13671, doi:https://doi.org/10.1111/cgf.13671.

[MBGM20] MARTIN S. K., BRUTON S., GANTER D., MANZKE M.: Synthesising light field volume visualisations using image warping in real-time. In *Computer Vision, Imaging and Computer Graphics Theory and Applications* (Cham, 2020), Cláudio A. P., Bouatouch K., Chessa M., Paljic A., Kerren A., Hurter C., Tremeau A., Farinella G. M., (Eds.), Springer International Publishing, pp. 30–47.

[PHE*11] PAJAK D., HERZOG R., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming. *Computer Graphics Forum* (Mar. 2011).

[SH15] SHI S., HSU C.-H.: A survey of interactive remote rendering systems. *ACM Comput. Surv. 47*, 4 (May 2015), 57:1–57:29. URL: http://doi.acm.org/10.1145/2719921, doi:10.1145/2719921.

[SKW*17] SCHIED C., KAPLANYAN A., WYMAN C., PATNEY A., CHAITANYA C. R. A., BURGESS J., LIU S., DACHSBACHER C., LEFOHN A., SALVI M.: Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, Association for Computing Machinery. URL: https://doi.org/10.1145/3105762.3105770, doi:10.1145/3105762.3105770.

[SME02] STEGMAIER S., MAGALLÓN M., ERTL T.: A generic solution for hardware-accelerated remote visualization. In *Proceedings of the Symposium on Data Visualisation 2002* (Goslar, DEU, 2002), VISSYM '02, Eurographics Association, p. 87–ff.

[SNC12] SHI S., NAHRSTEDT K., CAMPBELL R.: A real-time remote rendering system for interactive mobile graphics. *ACM Trans. Multimedia Comput. Commun. Appl. 8*, 3s (Oct. 2012). URL: https://doi.org/10.1145/2348816.2348825, doi:10.1145/2348816.2348825.

[SPD18] SCHIED C., PETERS C., DACHSBACHER C.: Gradient estimation for real-time adaptive temporal filtering. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 2 (Aug. 2018). URL: https://doi.org/10.1145/3233301, doi:10.1145/3233301.

[vir] A brief introduction to VirtualGL. https://virtualgl.org/About/Introduction. Accessed: 2020-05-20.

[VVP20] VASILAKIS A. A., VARDIS K., PAPAIOANNOU G.: A survey of multifragment rendering. *Computer Graphics Forum 39*, 2 (2020), 623–642. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14019, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14019, doi:https://doi.org/10.1111/cgf.14019.

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (Sept. 1990), ACM.

[WZM21] WALD I., ZELLMANN S., MORRICAL N.: Faster RTX-Accelerated Empty Space Skipping using Triangulated Active Region Boundary Geometry. In *Eurographics Symposium on Parallel Graphics and Visualization* (2021), Larsen M., Sadlo F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20211042.

[ZAL12] ZELLMANN S., AUMÜLLER M., LANG U.: Image-Based Remote Real-Time Volume Rendering - Decoupling Rendering from View Point Updates. In *Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference* (12 -15 Aug. 2012), ASME.

[ZHL17] ZELLMANN S., HOEVELS M., LANG U.: Ray traced volume clipping using multi-hit BVH traversal. In *Proceedings of Visualization and Data Analysis (VDA)* (2017), IS&T.

[ZHL19] ZELLMANN S., HELLMANN M., LANG U.: A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the 12th IEEE Pacific Visualization Symposium* (2019), IEEE.

[ZSL18] ZELLMANN S., SCHULZE J. P., LANG U.: Rapid k-d tree construction for sparse volume data. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), Childs H., Cucchietti F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20181097.

[ZSL19] ZELLMANN S., SCHULZE J. P., LANG U.: Binned k-d tree construction for sparse volume data on multi-core and GPU systems. *IEEE Transactions on Visualization and Computer Graphics* (2019), 1–1. doi:10.1109/TVCG.2019.2938957.

[ZWW20] ZELLMANN S., WEIER M., WALD I.: Accelerating force-directed graph drawing with RT cores. In *2020 IEEE Visualization Conference (VIS)* (2020), pp. 96–100. doi:10.1109/VIS47514.2020.00026.