2023 IEEE 16th Pacific Visualization Symposium (PacificVis) | 979-8-3503-2124-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/PacificVis56936.2023.00014

Efficient Raycasting of Volumetric Depth Images for Remote Visualization of Large Volumes at High Frame Rates

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

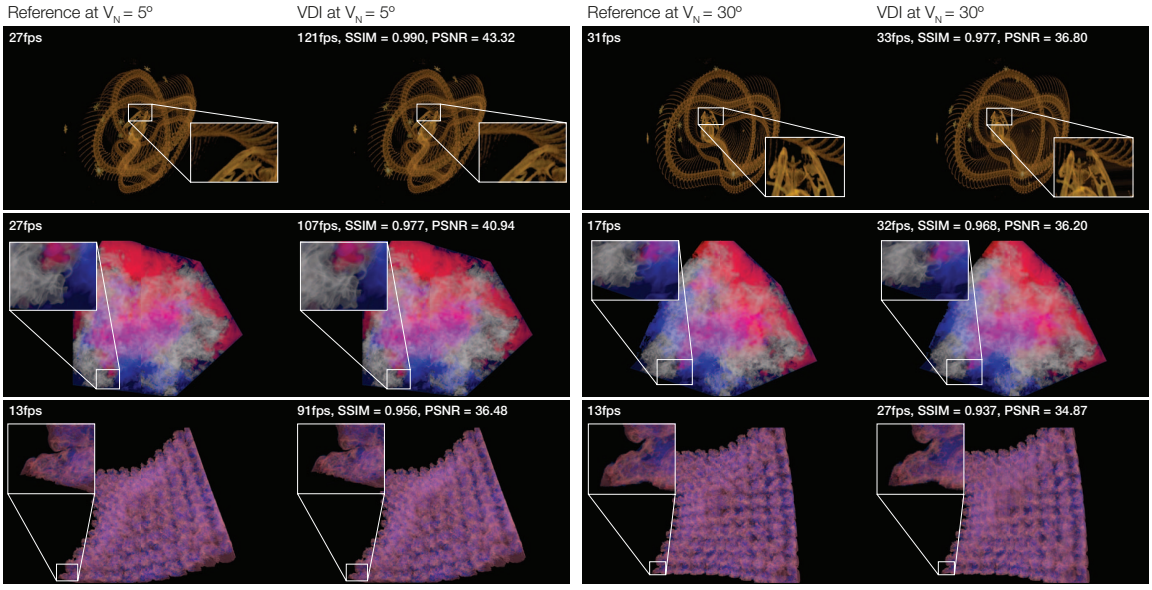


Figure 1: Visual comparison of VDI rendering quality with direct volume rendering (DVR) for the Kingsnake (top), Rayleigh-Taylor (middle), and Richtmyer-Meshkov (bottom) datasets. All frame rates are reported without empty-space skipping for comparability. Image quality metrics are computed w.r.t. the DVR image at the same angle.

Abstract

We present an efficient raycasting algorithm for rendering Volumetric Depth Images (VDIs), and we show how it can be used in a remote visualization setting with VDIs generated and streamed from a remote server. VDIs are compact view-dependent volume representations that enable interactive visualization of large volumes at high frame rates by decoupling viewpoint changes from expensive rendering calculations. However, current rendering approaches for VDIs struggle with achieving interactive frame rates at high image resolutions. Here, we exploit the properties of perspective projection to simplify intersections of rays with the view-dependent frustums in a VDI and leverage spatial smoothness in the volume data to minimize memory accesses. Benchmarks show that responsive frame rates can be achieved close to the viewpoint of generation for HD display resolutions, providing high-fidelity approximate renderings of Gigabyte-sized volumes. We also propose a method to subsample the VDI for preview rendering, maintaining high frame rates even for large viewpoint deviations. We provide our implementation as an extension of an established open-source visualization library.

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

1 Introduction

Interactive direct volume rendering is commonly used in the exploration and analysis of three-dimensional volume data. Rendering at high, consistent frame rates is crucial for enabling interactive viewpoint changes and zooming, which are important for gaining depth perception and spatial understanding. As scientific simulations and experimental devices generate larger data, however, it is increasingly challenging to achieve high, consistent volume rendering frame rates. In remote visualization applications, such as live *in situ* visualization of numerical simulations, fluent user interaction is potentially also hindered by network latency to the server.

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

2165-8773/23/$31.00 ©2023 IEEE
DOI 10.1109/PacificVis56936.2023.00014

View-dependent, piecewise constant representations of volumetric data, also known as Volumetric Depth Images (VDIs) [10], provide a potential solution by decoupling expensive rendering from interactive viewpoint changes. These representations decompose the volume rendering integral into segments that store composited color and opacity. Rendering such a representation involves compositing these segments, which is less expensive than performing the full integration [18] and produces high-fidelity approximations for camera viewpoints near the viewpoint from which the representation was generated [10, 21]. Additionally, VDIs are more compact than the original volume data and can be generated and streamed efficiently [9, 10]. This provides an attractive potential solution for interactive remote rendering. However, the large number of segments and their shape (pyramidal frustums when using perspective projection) makes VDIs challenging to render efficiently. Existing VDI rendering methods are thus unable to maintain interactive frame rates for high-definition (HD) displays.

Here, we present an efficient raycasting-based VDI rendering method designed to scale to large volumes and high-resolution (full HD) displays. At the core of our raycasting algorithm is a simplified way of intersecting rays with segments by computing them in Normalized Device Coordinate (NDC) space, as well as minimizing memory accesses by exploiting spatial homogeneity in the data. We additionally show how empty regions can be skipped in VDI raycasting. In comparison to the previous state-of-the-art object-order rendering algorithm for VDIs [10], we report frame rates that are an order of magnitude higher near the viewpoint of generation, while maintaining the same rendering quality with respect to ground-truth volume rendering. In comparison to a previously proposed raycasting approach [21], our method further reduces calculations and memory accesses.

Generating a VDI requires a dataset and transfer-function dependent sensitivity parameter to control the partitioning of rays into segments. In previous works, this parameter needed to be tuned manually, further hampering interactivity. We here instead propose a technique to automatically optimize the VDI generation parameter for given constraints. Importantly, this enables the parameter to be individually tuned for each ray, generating effectively content-adaptive VDIs that provide better-quality visualizations.

We particularly target a use case in which VDIs are generated at a remote server from the user's most recent viewpoint and streamed to a visualization client. A VDI at the client enables local renderings of user interactions until an updated VDI becomes available from the server. To account for the fact that the rendering frame rate decreases with increasing deviation in camera viewpoint—due to the anisotropic shape of the VDI—, we propose a technique to adaptively adjust the sampling along a ray so as to maintain a set frame rate.

In summary, we contribute an efficient raycasting method for VDIs that outperforms the current state of the art [10, 21]. We also suggest a downsampling method for preview rendering at set frame rates and a method to automatically adjust the VDI generation parameter individually for each ray, removing the need for manual tuning. We benchmark the proposed algorithms on several datasets and provide an open-source implementation as part of the visualization library *scenery* [14].

2 Related Work and Background

Recent work in remote rendering has leveraged hardware-accelerated video encoding [13, 20], client-side reprojections [3, 27], and motion prediction [12] to achieve high frame rates and low interaction latency. However, for direct volume rendering of large data, the primary bottleneck remains the rendering time itself. We review the literature in explorable image representations (Sect. 2.1) and deep-learning novel view synthesis (Sect. 2.2), which both provide potential solutions. Then, we provide some background about the specific explorable image representation we use here, the VDI [10] (Sect. 2.3).

2.1 Explorable Image Representations

Several explorable image representations have been proposed in the literature, often with the goal of decoupling rendering from interaction in remote-rendering applications. Shade et al. [26] introduced the view-dependent Layered Depth Image (LDI), storing multiple pixels along each line of sight. This allows for deferred rendering, but is limited to surface and geometry data. Stone et al. [29] rendered and streamed omni-directional stereoscopic images of molecular dynamics simulations from remote compute clusters, using local reprojections at frame rates suitable for Virtual Reality (VR). However, omni-directional stereoscopic images require warping to prevent distortions [28], which depends on depth information and therefore cannot be applied to volume data. For reprojecting volume data, Zellmann et al. [37, 38] transmitted a single depth layer along with the color buffer from the rendering server and provided a number of heuristics to generate the depth buffer. While the use of a single depth value per pixel minimizes message sizes, it only yields low-quality reprojections with visible holes where rays do not intersect the depth layer. VDIs were introduced by Frey et al. [10] and store a piecewise constant discretization of the volume rendering integral with no gaps or holes. They have been shown to produce higher-quality renderings [10].

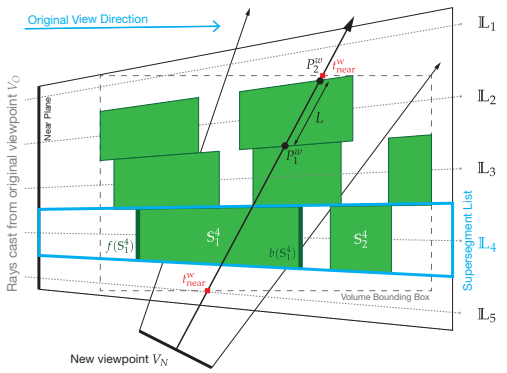
Tikhonova et al. [30, 31] proposed compact view-dependent representations that support interactive transfer function changes. Recently, Rapp et al. [25] modeled scalar densities along each ray in the Fourier domain, generating more compact representations of the volume while still supporting interactive transfer function changes. The focus of these works, however, is on interactive transfer function changes. This differs from the works mentioned in the first paragraph, which all consider a given transfer function, but focus on rendering speed from novel viewpoints. While Rapp et al. [25] do support viewpoint changes, they require slow bilinear interpolation of Lagrange multipliers.

In another fundamentally different approach, Ahrens et al. [1] proposed *Cinema* for *post hoc* explorative visualization of numerical simulations from a database of images generated *in situ* using different visualization parameters and camera viewpoints. All parameters and viewpoints, however, must be specified in advance, and the database becomes large if many viewpoints are considered.

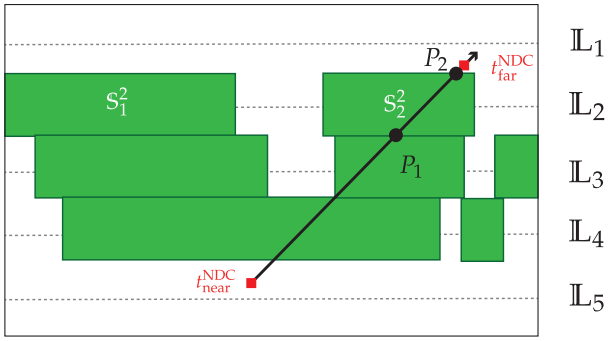
2.2 Deep Learning for Novel View Synthesis

More generally, explorable image representations can be seen as approaches to novel view synthesis, i.e., to the problem of using "images" of a scene to generate an image from a new viewpoint. In recent years, this problem has also been addressed using deep learning techniques. Mildenhall et al. [23], for example, proposed the NeRF (Neural Radiation Fields) representation. Their neural network encodes a continuous volume using weights approximating pre-classified RGBα values at any point in space. More recent works explored implicit neural representations of large volume data, achieving high compression ratios [16, 22, 34]. Unlike NeRF, these methods do not approximate classified values (i.e., with the transfer function applied), but directly predict the data at the query point.

Importantly, implicit neural representations can be rendered from new viewpoints by raycasting. Collecting samples by neural inference, however, is slow [22, 23]. Higher frame rates are achieved by efficiently distributing samples along the ray [24], varying the step size using an acceleration data structure [35], or by sampling a discretized grid [11] or octree [36]. Large, dense regions in volumes, however, still require many samples to be taken, limiting the frame rate. The present VDI approach is complementary, as it could, in fact, be used to cache a neural representation for efficient rendering by VDI raycasting at high frame rates.



(a) **World space**. The structure of a Volumetric Depth Image [10] 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 $\mathbb{S}_j^i$ intersection points are determined in NDC space, they are converted back to world space to determine the length of intersection.



(b) **NDC space**. Raycasting the VDI in NDC space of V_O for determining $\mathbb{S}$ intersections. The start and end points of the ray are converted to NDC 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 a Volumetric Depth Image (VDI) and the coordinate transformations involved in VDI rendering.

2.3 Volumetric Depth Image and Rendering

A VDI is a compressed, view-dependent piecewise constant explorable representation of volumetric data. We recall it here using the original notation by Frey et al. [10]. 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 distances between the near-plane and their front and back faces, $f(\mathbb{S}_j^i)$ and $b(\mathbb{S}_j^i)$, respectively. Each $\mathbb{S}_j^i$ contains classified accumulated color and opacity between $f(\mathbb{S}_j^i)$ and $b(\mathbb{S}_j^i)$. Fully transparent regions in the volume are not included in supersegments.

Given perspective projection during VDI generation, $\mathbb{L}$ subdivide the space spanned by the perspective view frustum. As such, all $\mathbb{L}_i$ and $\mathbb{S}_j^i$ are irregular pyramidal frustums. During VDI generation, the number of supersegments in each list, $\mathbb{N}_\mathbb{S}$, is limited to a pre-set maximum [5,10,21]. This, together with the viewport resolution, i.e., the number of lists $\mathbb{N}_\mathbb{L}$, determines the size of the VDI. Typically, the number of lists is much larger than the number of supersegments per list, i.e., $\mathbb{N}_\mathbb{L} \gg \mathbb{N}_\mathbb{S}$.

Different criteria have been proposed to determine supersegment lengths. Brady et al. [5] generate equal sized supersegments along each ray. This, however, composites supersegments over potentially highly heterogeneous samples, hampering the quality of rendering from a new viewpoint V_N. Lochmann et al. [21] therefore divide the accumulated opacity along a ray equally among the supersegments. This, however, does not account for potentially varying color values within supersegments. Frey et al. [10] generated accurate VDIs using homogeneity as the criterion for supersegment lengths. Samples along the ray are accumulated into a supersegment until they differ from the supersegment by more than a user-defined threshold γ, in which case a new supersegment is started. Here, we extend this method by automatically determining γ separately for each ray, eliminating manual tuning and further improving quality.

Rendering a VDI requires integrating over the $\mathbb{S}_j^i$ instead of the original data voxels. Several methods have been proposed for this. Brady et al. [5] rely on equal-sized supersegments in each list for their alpha-blending rendering. This, however, cannot generalize to supersegments of arbitrary lengths. Frey et al. [10] thus proposed an object-space approach that creates a frustum geometry for each supersegment $\mathbb{S}_j^i$. The supersegment lists $\mathbb{L}_i$ are then sorted from the new viewpoint and rendered using alpha blending. The opacity contribution from $\mathbb{S}_j^i$ is based on the intersection length with a ray from the new viewpoint. This approach, however, requires creating six triangles for each $\mathbb{S}_j^i$, becoming prohibitive for large VDIs.

Dataset	Dimensions	Datatype
Engine	$256 \times 256 \times 256$	uint8
Kingsnake	$1024 \times 1024 \times 795$	uint8
Rayleigh-Taylor [7]	$1024 \times 1024 \times 1024$	uint16
Richtmyer-Meshkov [6]	$2048 \times 2048 \times 1920$	uint8

Table 1: Datasets used to evaluate the presented algorithms.

Ray-based techniques do not explicitly create any geometry and therefore scale better to large VDIs. 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. [21], a ray from V_N is projected onto V_O, rasterized, and traversed using a DDA (digital differential analyzer) to determine the intersected lists. It then computes intersections with the pyramidal frustums of the supersegments.

The present raycasting method instead computes $\mathbb{S}_j^i$ intersections in the Normalized Device Coordinate (NDC) space of V_O, where all $\mathbb{S}_j^i$ and $\mathbb{L}_i$ are cuboids (Fig. 2b). This enables the use of voxel stepping [2] to traverse $\mathbb{L}$ and simplifies the computation of the intersections. Additionally, we exploit the spatial smoothness across lists to reduce memory accesses, and we propose methods to skip empty regions and to sub-sample rays for preview rendering.

Throughout this manuscript, we rationalize our design decisions in benchmarks using the four volume datasets listed in Table 1. The Engine dataset is a CT scan of two cylinders of an engine block. The Kingsnake dataset is an X-ray CT scan of an egg of the *Lampropeltis getula* snake species. The Rayleigh-Taylor dataset is the density field at a single time step of a computer simulation of the fluid instability of same name [7]. The Richtmyer-Meshkov dataset is the entropy field at a single simulation time step of the so-named instability [6]. Unless otherwise stated, measurements were done on a workstation with an Nvidia GeForce RTX 3090 GPU and an AMD Ryzen Threadripper 3990x 64-core CPU running Ubuntu 20.04. Volume raycasting, both for generating VDIs and for direct volume rendering (DVR), used an emission-absorption illumination model. Due to a buffer size limitation in the runtime system, datasets larger than 2 GB were distributed across multiple buffers for DVR.

Dataset	$\mathbb{N}_\mathbb{L}$=1280 × 720	$\mathbb{N}_\mathbb{L}$=1920 × 1080
Kingsnake	0.21 / 0.23 / 0.25	0.40 / 0.42 / 0.46
Rayleigh-Taylor	0.43 / 0.40 / 0.55	0.73 / 0.63 / 0.80
Richtmyer-Meshkov	0.75 / 0.82 / 0.82	1.55 / 1.36 / 1.36

Table 2: Wall-clock times in seconds to generate a single VDI with $\mathbb{N}_\mathbb{S}$= 15 / 20 / 30 for the datasets from Table 1. The camera is rotated about the data in 10° increments, and means over 30 camera positions are reported. See Fig. 1 for transfer functions.

3 Ray-Adaptive Generation of Supersegments

In order to generate accurate VDIs, Frey et al. [10] proposed a homogeneity criterion τ for supersegment termination:

$$\tau : \gamma > ||C(\mathbb{S})\alpha(\mathbb{S}) - C'\alpha'||_2, \quad (1)$$

where C' and α' are the color and the length-adjusted opacity of the next sample. In words, a sample along the ray is merged into the current $\mathbb{S}$ unless it differs from the pre-multiplied color of $\mathbb{S}$ by more than a user-defined sensitivity parameter γ, in which case a new $\mathbb{S}$ is started. This criterion generates homogeneous $\mathbb{S}$, but the sensitivity parameter γ is constant across rays and must be carefully tuned manually for a given dataset and transfer function so as to prevent visual artefacts. Too high values of γ generate insufficient supersegments to represent the data. Too low values exhaust the supersegment budget $\mathbb{N}_\mathbb{S}$, potentially causing "smearing" artefacts as the last $\mathbb{S}$ must contain all remaining data. We address this issue by proposing a method to automatically determine a suitable γ independently for each ray, while guaranteeing a maximum of $\mathbb{N}_\mathbb{S}$ supersegments per ray.

Leveraging the fact that the number of supersegments produced decreases monotonically with increasing γ, we perform bisection search between the highest and lowest possible values of γ until a value is found that generates $\mathbb{N}_\mathbb{S}$ supersegments, i.e., we optimize the resolution along rays without violating the constraint $\mathbb{N}_\mathbb{S}$ (see Supplement for pseudo-code of the algorithm). Since the distance metric in Equation 1 is an L_2 distance between pre-multiplied color vectors with 3 elements each, the highest possible γ value is $\sqrt{3}$ and the lowest is 0. Each iteration of bisection search samples the volume along the ray to determine the number of supersegments generated for the current γ. Since many iterations may be required to determine a γ that generates exactly $\mathbb{N}_\mathbb{S}$ supersegments, a tolerance of up to δ fewer supersegments than $\mathbb{N}_\mathbb{S}$ is permitted, but never more than $\mathbb{N}_\mathbb{S}$ as this would cause "smearing" artefacts. We empirically find a δ of 15% of $\mathbb{N}_\mathbb{S}$ to provide a good trade-off between performance and quality. To eliminate rays that pass through empty or homogeneous regions, we initialize γ to a small positive value, here 10^{-5}. If the first iteration then generates fewer supersegments than $\mathbb{N}_\mathbb{S}$, the samples along that ray are homogeneous, and that ray can immediately terminate, freeing computational resources for other rays. Measured VDI generation times for different datasets and resolutions are given in Table 2. In some cases, generation times are lower for larger $\mathbb{N}_\mathbb{S}$, which is because γ search converged faster.

While bisection search for γ requires each ray to pass through the volume multiple times, it generates more accurate VDIs with each ray adaptively finding a near-optimal γ, instead of using one manually-tuned value across all rays. Importantly, automatic γ determination enables remote VDI generation without user intervention and streaming to a display client.

4 VDI Rendering by Raycasting

For each pixel in the viewport, we cast a ray into the VDI. The ray passes through supersegment lists $\mathbb{L}$, searches for supersegments $\mathbb{S}$ within them, and calculates the intersection length with each $\mathbb{S}$. It uses this to adjust the color contribution from each supersegment before accumulating them by alpha compositing.

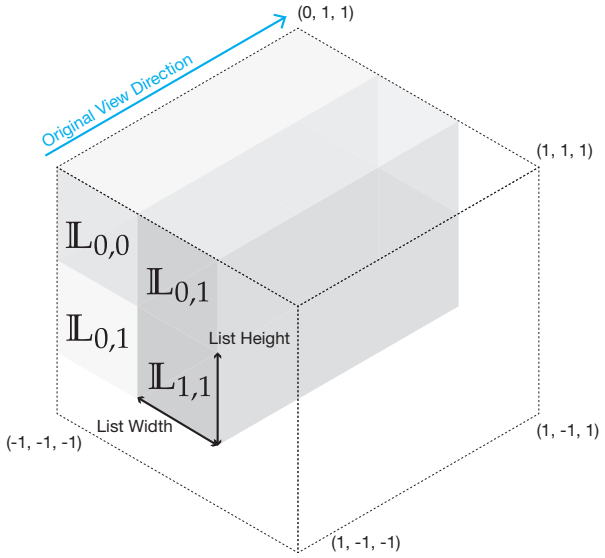


Figure 3: The supersegment lists $\mathbb{L}$ form a regular 2D grid of cuboids in the NDC space of V_O. In the `OpenGL` convention used in this figure, NDC range from -1 to 1 along all axes. The width and height of each cell (List Width and List Height) are therefore $2/w$ and $2/h$, respectively, where w and h are the x- and y-resolutions of the viewport used for generating the VDI.

4.1 Ray Traversal Through Lists

The traversal of a ray is illustrated in Fig. 2. The start and end points for the ray marching are determined by the intersection of ray with 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 traversing through $\mathbb{L}$ and determining intersection points with $\mathbb{S}$, the calculations are done in the perspective-deformed Normalized Device Coordinate (NDC) space of V_O. In world space, $\mathbb{S}$ and $\mathbb{L}$ form pyramidal frustums. In perspective NDC space, they are cuboids (Fig. 2b), with $\mathbb{L}$ forming a regular 2D grid as illustrated in Fig. 3. A ray then traverses this grid using the fast voxel traversal algorithm by Amanatides and Woo [2]. With only two floating-point and two integer additions, and one floating-point and one integer comparison per iteration, the $\mathbb{L}$ intersected next is determined, along with the intersection points with a given $\mathbb{L}_i$, which are then used to search for $\mathbb{S}^i_j$ within $\mathbb{L}_i$.

4.2 Supersegment Search with Spatial Smoothness

For each intersected $\mathbb{L}$, we find the $\mathbb{S}$ that cover the region between the entry and exit points of the ray. Once the first of these $\mathbb{S}$ is determined (if any), the next is found by adjacency search, since the $\mathbb{S}$ within $\mathbb{L}$ are sorted by their position. The algorithm only needs to check the next or preceding index, depending on the direction of the ray, which is defined by the sign of the scalar product between the ray and the original ray vector in world space.

The procedure to determine the first intersected supersegment in a list is detailed in Alg. 1. The algorithm minimizes memory accesses by leveraging spatial smoothness in the volume data: neighboring $\mathbb{L}$ are created from rays that pass nearby in the data, and they are thus likely to have similar sized $\mathbb{S}$. The index p of the last supersegment $\mathbb{S}^h_p$ intersected in the previous list $\mathbb{L}_h$ is therefore used as an initial

Algorithm 1 Find the first supersegment intersected in $\mathbb{L}_i$

```
if L_i is the first list intersected then
    return binary search from 0 to N_S
end if
p ← index of last supersegment intersected in previous list
if S^i_p is the first intersected supersegment in L_i then
    return p
else
    if ray position is behind S^i_p then
        return binary search between 0 and p − 1
    else
        return binary search between p + 1 and N_S
    end if
end if
```

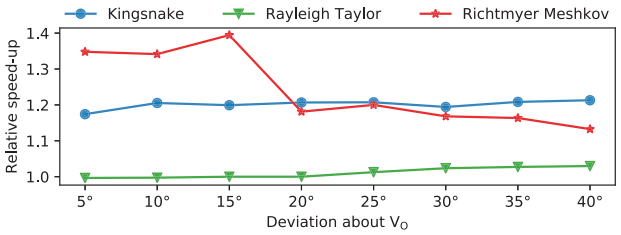


Figure 4: Speed-up in VDI rendering frame rates when using Alg. 1 compared to, in each case, the best-performing among linear or binary search for the first $\mathbb{S}$. A VDI generated from V_O is rendered at rotating viewpoints for three different datasets (symbols).

guess for the first $\mathbb{S}$ in the current list $\mathbb{L}_i$. If no $\mathbb{S}$ was intersected in $\mathbb{L}_h$, p is the index of the closest $\mathbb{S}$ from the previous exit point. In Fig. 2b, for example, when the ray enters $\mathbb{L}_2$, it first tests for intersection with $\mathbb{S}^2_2$, since 2 was the last index intersected in $\mathbb{L}_3$. The algorithm is further analyzed in the Supplement.

Fig. 4 reports the relative speed-up in VDI rendering frame rates when using Alg. 1 compared to the respective best of binary or linear search for the first supersegment in a list. Baseline binary search is initialized with the middle $\mathbb{S}$ in the $\mathbb{L}$, and linear search with the first $\mathbb{S}$. In all cases, subsequent $\mathbb{S}$ are found by adjacency search; the only difference is in how the first $\mathbb{S}$ is found. Alg. 1 yields up to 40% better frame rates.

For each intersected supersegment, the opacity accumulated by the ray needs to be adjusted by the intersection length of the ray with the supersegment, as [8]:

$$\widetilde{\alpha} = 1 - (1 - \alpha)^l \tag{2}$$

where $\widetilde{\alpha}$ is the adjusted opacity, α is the opacity stored in the supersegment, and l is the intersection length. Intersections are computed in NDC space, but adjusting opacity requires the intersection length l in world space. This needs two additional matrix-vector multiplications to convert both intersection points to world space. Raycasting in the NDC space of V_O, in comparison to a world (or view) space technique, such as the one proposed by Lochmann et al. [21], therefore optimizes intersections with supersegment lists, but has a higher cost for each supersegment intersected. Fig. 5 plots the number of supersegment lists $\mathbb{L}$ and supersegments $\mathbb{S}$ intersected during VDI rendering at various viewpoint deviations around V_O. The number of $\mathbb{L}$ traversed is larger than the number of $\mathbb{S}$ intersected, since for the majority of lists no supersegments are found. While the difference is particularly stark for the sparse Kingsnake dataset plotted here, this is true for more dense datasets, too. The difference between the number of $\mathbb{S}$ and $\mathbb{L}$ intersected also grows with viewpoint deviation due to the anisotropic shape of the VDI. Early ray termination limits

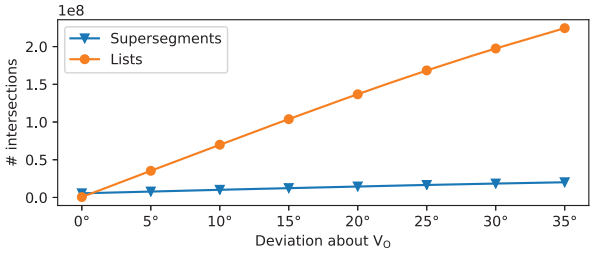


Figure 5: The total number of supersegment lists $\mathbb{L}$ and supersegments $\mathbb{S}$ (symbols) intersected by all rays for different V_N around V_O for an $\mathbb{N}_\mathbb{L}$=1920 × 1080, $\mathbb{N}_\mathbb{S}$=20 VDI of the Kingsnake dataset.

the number of $\mathbb{S}$ intersected, while there is no limit on the number of $\mathbb{L}$ intersected. The present strategy thus optimizes the costliest part of the rendering, which is the traversal of $\mathbb{L}$.

4.3 Empty-Space Skipping

The performance of the present method depends on the number of memory accesses along a ray. We optimize this by an acceleration data structure on top of the VDI to skip empty regions. The VDI implicitly contains the information required to skip empty space within a $\mathbb{L}$, since $\mathbb{S}$ are sorted by front and back depths defining their position in $\mathbb{L}$. However, this cannot be queried based on the ray position when moving from one $\mathbb{L}$ to the next. Consider the example in Fig. 6, where Ray A must sample each $\mathbb{L}_i$ at least once, even in empty areas, to determine that no $\mathbb{S}^i_j$ in those $\mathbb{L}_i$ is intersected.

We therefore supplement the VDI with a grid acceleration data structure that stores the number of $\mathbb{S}$ overlapping with each grid cell (black numbers in Fig. 6). This count, rather than just a binary indicator, is needed for preview rendering as explained in the next section. When a ray hits a cell storing a 0, it jumps to the other end of that cell. This way, Ray B skips several $\mathbb{L}$ in the empty regions covered by the first and third cell it traverses. Querying the grid data structure requires one memory access per cell.

The grid cells have a constant depth extent in view space. In NDC space, this corresponds to cells that are larger toward the near plane, and smaller toward the far plane. Each cell spans an equal number of $\mathbb{L}$ along both dimensions of the viewing plane. The depth of a cell in view space is larger than its width or height, due to the anisotropic nature of a VDI (there are fewer $\mathbb{S}^i_j$ in $\mathbb{L}_i$ than there are $\mathbb{L}_i$).

The 3D regular grid could be replaced by hierarchical grids that provide better empty-space skipping performance, such as an octree [19] or SparseLeap [15]. However, we chose the current structure since it can be created during VDI generation with each $\mathbb{S}^i_j$ generated triggering an atomic add on the appropriate grid cell.

4.4 Dynamic Subsampling for Preview Rendering

The performance of the proposed raycasting algorithm depends on the number of memory accesses and calculations made by the rays as they traverse the VDI. Around the original view direction (V_O), high frame rates can be achieved as the VDI is compressed along V_O. For larger viewpoint deviations, rendering performance reduces, as larger portions of the rays go along one of the view plane dimensions. As illustrated in Fig. 5, the number of intersected supersegment lists $\mathbb{L}$ increases with increasing viewpoint deviation. Each $\mathbb{L}$ intersected in a non-empty cell requires memory accesses to search for $\mathbb{S}$.

In the proposed remote visualization application, new VDIs are generated when the user's viewpoint changes. If large deviations from V_O occur at the display client faster than a new VDI can be generated and streamed, however, the old VDI is used to bridge the

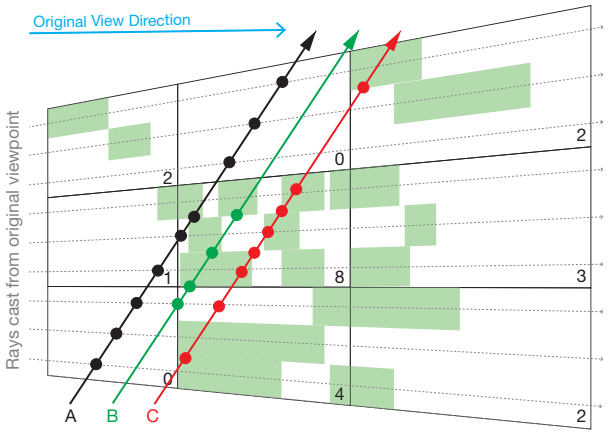


Figure 6: The grid data structure used for empty-space skipping and for preview rendering. Numbers in lower right corners are the values stored in each cell, which indicate the number of supersegments intersecting that cell. The rays of different color illustrate traversal strategies: Ray A is the base raycasting algorithm, ray B skips empty cells, and ray C subsamples the VDI for preview rendering. Dots indicate points at which the rays query the VDI to search for a supersegment in a list.

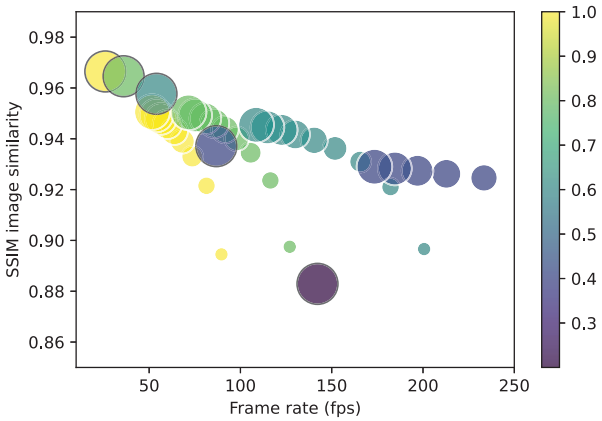


Figure 7: Performance (fps) and quality (SSIM w.r.t. DVR) of adaptively sub-sampling VDI rendering along image dimensions (D_I) and along the ray (D_R). Color is used to represent D_I, smaller circle radii indicate smaller D_R. Circles with black outlines use full-resolution rendering along the ray. Display resolution is always 1920×1080. Images rendered with D_I<1.0 are upsampled for display using bilinear interpolation.

time, trading off quality. In order to maintain frame rates, this uses dynamic sub-sampling for preview rendering.

One way to achieve preview rendering would be to decrease the number of rays cast, followed by upsampling to the desired display resolution. Another way is to sub-sample along rays. We analyze the influence of both on VDI rendering performance and quality. Quality is measured using the SSIM [32] w.r.t. fully resolved DVR. Higher SSIM indicate better rendering quality, with 1.0 indicating identical images. Fig. 7 shows the results for a $\mathbb{N}_\mathbb{L}$=1920 × 1080, $\mathbb{N}_\mathbb{S}$=20 VDI generated on the Richtmyer-Meshkov dataset, rendered at 30° from V_O. The VDI is the same in all cases.

The largest circles of each color, highlighted with a black outline, indicate the quality and performance obtained at various levels of downsampling (D_I) in image space, i.e., by decreasing number of rays. D_I is indicated by color, with 1.0 indicating a full-resolution rendering. At D_I=1.0 (yellow), the VDI renders at 45 fps. Frame rates increase when decreasing D_I, but only slowly. To achieve a frame rate of ≈150 fps for example requires D_I=0.2, which generates an image with SSIM=0.88. This is significantly worse than the SSIM of 0.97 for D_I=1.0.

We therefore propose to additionally sub-sample the rendering along the view ray, limiting the number of memory accesses and allowing higher values of D_I for the same set frame rate. In order to sample the VDI according to its information content, we use the acceleration data structure (Sect. 4.3) to distribute samples along the ray proportional to the number of $\mathbb{S}$ in each grid cell. The samples are then uniformly thinned by a factor D_R. Rendering is further simplified by not calculating $\mathbb{S}$ intersections. As the ray marches, it simply queries which $\mathbb{S}^i_j$ a given sample point lies within, if any, and obtains the color from $\mathbb{S}^i_j$. Length-based opacity correction (Eq. 2) is approximated using the distance from the previous sample. Since $\mathbb{S}^i_j$ intersections are not computed, sampling is done in world space. The $\mathbb{L}_i$ of a sample is found from its x- and y-coordinates in NDC space. The $\mathbb{S}^i_j$ in $\mathbb{L}_i$ is found using the algorithm from Sect. 4.2.

The number of samples within each cell of the acceleration grid is found by multiplying the sampling rate D_R with the intersection length of the ray with the cell and with the number of $\mathbb{S}$ in the cell, which is stored in the grid (Sect. 4.3). Samples are then placed at regular intervals along the ray within each cell. This amounts to adaptive sampling, as empty cells are not sampled and regions covered by more $\mathbb{S}$ are sampled more finely. Ray C in Fig. 6 shows an example.

Adaptively sub-sampling the VDI enables rendering higher-quality images with larger D_I for the same frame rate, as shown in Fig. 7. Circles with white outline represent adaptive sub-sampling with smaller values of D_R shown by smaller radii. The same ≈150 fps can now be achieved with significantly higher SSIM of 0.94 with D_I=0.6 in image space and D_R=0.14 along the rays.

Sub-sampling the VDI for preview rendering requires choosing D_I and D_R to obtain good image quality for a given target frame rate. We analyzed several datasets at multiple view configurations (see Supplement for details) and, while the results are qualitatively similar to Fig. 7, the values at which a set of parameters becomes better than another one vary widely. The present remote visualization application (Sect. 5) therefore uses a dynamic PI (Proportional-Integral) controller to modulate D_I, while allowing the user to choose the target frame rate and D_R.

5 Interactive Remote Visualization using VDIs

Using the described methods to adaptively generate VDIs and to render them efficiently, we present the design of an application that enables remote visualization of large data volumes at high frame rates. VDIs are generated on a server and streamed to a display client where they are rendered. Compared to remote DVR and streaming of rendered images, the proposed approach affords better interactive responsiveness, since the VDI is rendered locally on the client, bypassing network latency for small viewpoint changes.

Each time the user changes the camera viewpoint on the display client, the camera pose is communicated to the server. As soon as the server finishes generating a VDI, it reads the most recent camera pose from the network. If the pose is different from the one of the previous VDI, or if the data themselves changed, a new VDI is generated. The data can change, e.g., if the server provides live *in situ* visualization of a simulation. The VDI can optionally be compressed before transmission to the client. Compression and

Dataset	VDI resolution with $\mathbb{N}_\mathbb{S} = 20$	V_N		
		10°	20°	40°
Engine	512×512	28	28	27
		1174	**428**	**291**
	1024×1024	9	9	9
		454	**80**	**60**
Kingsnake	512×512	31	31	30
		359	**298**	**234**
	1024×1024	11	10	10
		441	**50**	**44**
Rayleigh-Taylor	512×512	20	18	18
		710	**589**	**401**
	1024×1024	5	5	5
		226	**167**	**124**

Table 3: Comparing the performance (in fps) of our approach (bold) for rendering VDIs with the rasterization-based technique from [10].

streaming of the VDI are done asynchronously, interleaved with the generation of the next VDI, using separate CPU threads. Streaming of VDIs and communication of viewpoint changes is implemented using the networking library ZeroMQ (see zeromq.org).

At the display client, receiving, decompressing, and uploading the VDI to the GPU are done asynchronously with VDI rendering on the GPU. Double buffering is used on the GPU to ensure that the previous VDI can be rendered while a new one is uploaded. The client always receives only the most recent VDI generated by the server. Near the viewpoint from which the VDI was generated (V_O), full-resolution VDI rendering is performed, maintaining high image quality at high frame rates. If frame rates drop below a user-set minimum at larger deviations from V_O, rendering transparently switches to adaptive sub-sampling for preview rendering until the new VDI is received from the server. At that moment, full-resolution rendering resumes. Note, however, that rendering of the new VDI need not begin from the viewpoint of generation, as the user may already have changed the camera position meanwhile. Future extensions could explore techniques that predict the user's camera movements to speculatively generate a VDI from the predicted viewpoint.

We implemented this remote visualization application in the open-source visualization library *scenery* [14]. The source code is available under the BSD license at github.com/scenerygraphics/scenery. The implementation uses the high-performance Vulkan graphics API, enabling it to run across GPUs from different manufacturers. Both rendering and generation of VDIs are implemented using compute shaders. For work distribution in the compute shaders, a local workgroup size of 16×16 is used, i.e., the screen space is divided into 2D blocks of that size. Each ray within a block corresponds to a thread on the GPU and a single pixel on screen. Each VDI consists of two floating-point textures: one for storing color and opacity of supersegments (type `RGBA32F`) and one for the depth of the supersegments (type `R32F`), with both front and back depth stored within.

6 Evaluation

We evaluate our implementation of the present algorithms on the real-world datasets from Table 1. For evaluation, VDIs were generated from one or more viewpoints (V_O) and rendered at one or more new viewpoints V_N. The V_N were rotations of the camera about the dataset center with the camera always facing the dataset center. Unless otherwise stated, results are reported as frame rates, i.e., the inverse of the frame time at a given camera pose.

6.1 Comparison with other VDI Rendering Techniques

We first compare the present raycasting-based VDI rendering with the rasterization-based method of Frey et al. [10]. To ensure that both methods render identical VDIs, the VDIs were generated using the implementation of Frey et al. and imported into our software for rendering. We also disabled empty-space skipping (Sect. 4.3) in our implementation, since the code by Frey et al. does not have it. Table 3 reports the results. We could only use the three smaller datasets, and smaller viewport resolutions, for this comparison, due to technical limitations in the implementation by Frey et al.

In all cases, the present method outperforms the rasterization-based approach, often by more than an order of magnitude. For the implementation by Frey at al., performance is similar to that reported in the original 2013 paper, indicating that their technique of sorting supersegment lists $\mathbb{L}$ in front-to-back order for a given rendering viewpoint V_N, followed by α-blending, does not substantially benefit from recent advances in GPUs.

The performance of Frey et al. [10] remains similar at all tested V_N. The present raycasting method, however, is faster at smaller viewpoint deviations. This is because raycasting can take advantage of the anisotropic view-dependent shape of the VDI, where rays can march quickly along supersegment lists, leading to higher frame rates near V_O. Another advantage of the raycasting approach can be observed for the dense Rayleigh-Taylor dataset, where early ray termination provides significant speed-ups not possible with object-space approaches like that of Frey et al. The Kingsnake dataset is challenging when disabling empty-space skipping, because it contains large empty regions. Still, the present method outperforms Frey et al., even without empty-space skipping.

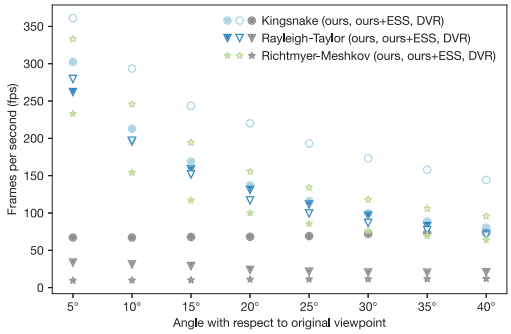
The present raycasting approach is also more consistent in its memory requirement, since no geometry needs to be generated. The memory required by the code of Frey et al. [10] depends heavily on the dataset. For the dense Rayleigh-Taylor dataset at 1024×1024 viewport resolution, Frey et al. generate 3473 MB of geometry and for the sparse Kingsnake dataset only 349 MB. Our code has a constant memory requirement of 480 MB across all datasets for 1024×1024 viewport resolution. In addition to providing faster rendering performance, our raycasting approach does not require creation of geometry which is a one-time per-VDI cost for Frey et al. The rendering quality produced by the two methods is the same, and therefore not compared.

We were unable to perform empirical performance comparisons with the raycasting technique proposed by Lochmann et al. [21] for a similar data structure, since their code is not publicly available. As explained in Sect. 4, however, our technique of traversing the VDI in the NDC space of V_O reduces the number of calculations compared to the strategy described by Lochmann et al., where traversal is performed in view space requiring intersections with pyramidal frustums (confirmed by original authors in personal communication). While Lochmann et al. do not provide details on how supersegments are searched for within supersegment lists, our optimized search strategy leveraging spatial homogeneity provides speed-ups of up to 40% over binary search (Fig. 4). Lochmann et al. also did not use empty-space skipping, likely losing further performance (Fig. 8).

6.2 Comparison with Direct Volume Rendering

Next, we compare the present approach with DVR. VDIs are generated from four different V_O, placed around the dataset at 90° rotations, rendered at different deviations V_N about each V_O, and compared with DVR from the same viewpoint. Fig. 8 plots the mean frame rates over the four V_O for two different viewport resolutions, $\mathbb{N}_\mathbb{L}$=1280×720 (standard HD) and $\mathbb{N}_\mathbb{L}$=1920×1080 (full HD). VDI frame rates are reported with and without empty-space skipping (ESS) for comparability with DVR, which does not use empty-space skipping.

At small V_N=5°, VDI rendering achieves significant speed-ups over DVR in the range of $4.5\ldots24.5\times$ for standard HD and $3.5\ldots7.5\times$ for full HD across datasets. VDI raycasting frame rates decrease for increasing V_N, as rays have to do more work due to the



(a) Rendering frame rates for 1280×720, $\mathbb{N}_S$=20, VDIs.

(b) Rendering frame rates for 1920×1080, $\mathbb{N}_S$=20, VDIs.

Figure 8: VDI rendering frame rates without ("ours", blue/green filled symbols) and with empty-space skipping ("ours+ESS", blue/green open symbols) for different datasets, compared with direct volume rendering ("DVR", gray symbols) from the same viewpoint.

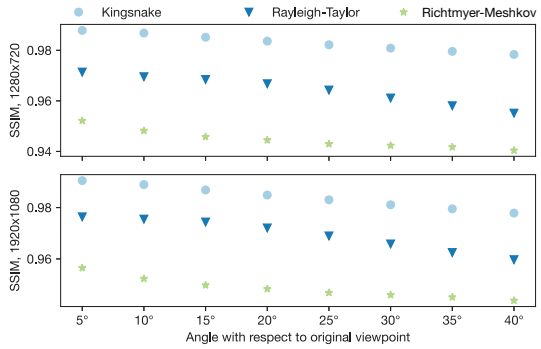


Figure 9: SSIM image similarity between VDI rendering and DVR from the same viewpoint for 1280×720 (top) and 1920×1080 (bottom) with $\mathbb{N}_S$=20 for different datasets (symbols).

anisotropic view-dependent shape of the VDI. However, they remain higher than DVR at all V_N except for the full HD rendering of the Kingsnake beyond 30°. The lower speed-ups for full HD resolution compared to standard HD are expected, since the sizes of the VDI and the rendering viewport both increase. Empty-space skipping mostly increased VDI rendering frame rates, particularly for the sparse Kingsnake and Richtmyer-Meshkov datasets, but occasionally reduced them slightly for the dense Rayleigh-Taylor dataset.

We also compared the quality of the images generated by VDI rendering with those from DVR. Fig. 9 provides the results in terms of the SSIM (Structural Similarity Index Measure) [32], where higher values are better and 1.0 corresponds to identical images. Fig. 1 provides visual comparisons (see Supplement for full resolution images). Like frame rates, SSIM values are also higher for smaller V_N, as view rays are better aligned with the rays that generated the VDI. The reduction in rendering quality, however, is minor even at high V_N of up to 40°. Similar results are observed when using the PSNR quality metric (see Supplement).

VDI rendering quality can further be increased by increasing $\mathbb{N}_S$ at the cost of larger VDI size and reduced frame rates. Rendering frame rates, however, were found to reduce sub-linearly with increasing $\mathbb{N}_S$, falling to between $0.61\times$ and $0.89\times$ when doubling $\mathbb{N}_S$.

Videos can be found in the Supplementary material showing interactive visualization sessions using both DVR and VDI rendering on all datasets.

6.3 Comparison with Remote Volume Visualization

Finally, we compare the present VDI-based remote visualization system with remote DVR using NVENC video encoding for image streaming. For network streaming, our implementation compresses VDIs using the lossless LZ4 algorithm, which we found to provide the best trade-off between speed and compression in our benchmarks, in comparison with zstd and Snappy. LZ4 yields compressed VDIs of ≈100 MiB for standard HD (1280×720) resolution and ≈225 MiB for full HD (1920×1080). The corresponding uncompressed VDI sizes are 422 MiB and 950 MiB, respectively. In addition, the empty-space skipping data structure (2.5 MiB for full HD) and the metadata about V_O (≈200 Bytes) are also transmitted.

To evaluate the performance of the present system in its entirety, we compare with existing remote visualization functionality in *scenery* [14], which offers video streaming with hardware-accelerated encoding and decoding using NVENC and CUVID, respectively. The volume data reside on a server with Nvidia GeForce RTX 3090 GPU, where DVR and VDI generation take place. The display client is a standard office workstation with an AMD Radeon RX 5700XT GPU, connected via 1 Gbit/s Ethernet across rooms.

All camera viewpoint changes are applied synchronously at the client in order to ensure frame-to-frame comparability. Overall frame times are measured to estimate the "motion-to-photon" latency, i.e., the time between the user making a movement and the movement being fully reflected on the display. For the VDI, this is the rendering frame time of the VDI at the client. For remote DVR, it is the rendering frame time plus the streaming time, which includes the time to send the new camera pose to the server, encode the rendered frame, stream, and decode the frame at the client.

A camera path is pre-recorded, consisting of four phases that evaluate different modes of interactive visualization: Phases 1 (frames 0-500) and 4 (frames 1500-2000) show steady camera movements for data exploration; Phase 2 (frames 500-1000) consists of fast movements for navigation, and in Phase 3 (frames 100-1500) the camera zooms in on a point of interest and then out again. Screencasts of the interactive sessions with DVR and VDI rendering are provided in the Supplement. Fig. 10 reports the performance measurements for full HD resolution of the Richtmyer-Meshkov dataset.

Streaming time was found to add only a marginal overhead on DVR timings in our setup. While we were unable to measure decoding timings on the AMD Radeon RX 5700XT due to incompatibility with *scenery*'s CUVID decoding, on the RTX 3090 we found decoding times < 0.5 ms per frame. NVENC encoding added a consistent overhead of approximately 5 ms. Fig. 10 also reports the frame numbers at which new VDIs become available for rendering, providing an estimate of the "VDI latency", which includes VDI generation,

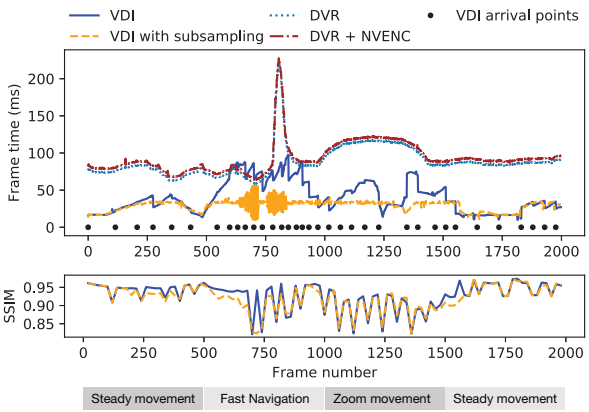


Figure 10: Top panel: end-to-end frame times for remote visualization of the Richtmyer-Meshkov dataset in full HD resolution using remote DVR (on an RTX 3090) and VDI rendering (on a Radeon RX 5700XT) over an interactive session of 2000 frames (see video in Supplement). Bottom panel: VDI rendering and VDI subsampling quality at every 20th frame as SSIM w.r.t. DVR images.

transmission, compression and decompression, as well as GPU upload. Note that this latency is hidden from the user through double buffering (Sect. 5).

During the steady camera movements of Phases 1 and 4, new VDIs that arrive from the server are rendered at not-too-distant viewpoints from V_O. This results in smooth interactive frame rates with frame-times significantly shorter than those achieved by remote DVR, while maintaining high rendering quality. VDI frame times are also significantly shorter than those of remote DVR during the zooming Phase 3. As the camera zooms back out towards the end of Phase 3, missing regions are visible in the VDI rendering. This is because the VDI generated from the zoomed-in viewpoint is still rendered, representing only data within its viewport. This leads to the drop in SSIM towards the end of Phase 3, until the new VDI for the zoomed-out viewpoint arrives.

The fast navigation in Phase 2 is challenging for VDI rendering, as large deviations from V_O occur. This leads to increasing VDI frame times, eventually becoming comparable to those of DVR in this phase. Note, however, that VDI rendering runs on an AMD Radeon RX 5700XT, which is significantly less powerful (0.24× computational throughput, 0.44× peak memory bandwidth) than the RTX 3090 used for DVR.

We also evaluate the adaptive preview rendering strategy of Sect. 4.4 and demonstrate how the PI controller dynamically adjusts D_I to maintain a target frame rate. Adaptive sampling along the ray is manually activated during Phase 2 with D_R=0.3. The PI controller is able to maintain a steady frame rate of 25 fps through most of the session, though jerkiness was observed during Phase 2.

7 Conclusions

We have presented an efficient raycasting-based rendering method for VDIs and its use in remote visualization of large volume data. At its core is an efficient way of intersecting supersegment lists $\mathbb{L}$ and supersegments $\mathbb{S}$ by ray marching in the NDC space of V_O, where all $\mathbb{L}$ are transformed from irregular pyramidal frustums to cuboids. These can then be traversed by voxel stepping [2]. We presented an efficient method for finding $\mathbb{S}$ within $\mathbb{L}$ by leveraging spatial smoothness in the data. This increases frame rates by up to 40% over the binary search baseline. We also presented a method for skipping empty space during raycasting.

The presented method significantly outperforms the rasterization-based VDI rendering by Frey et al. [10] (Table 3), which we found to not scale well to modern GPUs. Our raycasting approach also has a few inherent advantages: it does not require the creation of geometry, benefits from early ray termination, and better leverages the anisotropy of the VDI near V_O. While we were unable to present a direct comparison with the raycasting approach by Lochmann et al. [21] due to unavailability of their code, we argue that our $\mathbb{L}$ traversal, $\mathbb{S}$ search, and empty-space skipping are likely to result in better performance. We also found that our VDI rendering frame rates are significantly higher than DVR (Fig. 8) close to V_O, while providing high-quality approximations (Fig. 9).

Finally, we have shown how the present method can be used in a remote visualization application. In this context, we have proposed an extension to the VDI generation technique of Frey et al. [10], where γ values are adaptively determined for each ray, generating more accurate VDIs without manual parameter tuning. To perform preview rendering at large viewpoint deviations before the next VDI arrives, we proposed an adaptive sub-sampling along the ray, which we have shown to combine well with image-space sub-sampling to yield consistently high frame rates. Overall, the entire remote visualization application sustained higher frame rates than remote DVR (Fig. 10).

A current limitation of our implementation is that the adaptive subsampling along the rays needs to be activated and tuned manually, while subsampling in image space is dynamically controlled by a PI controller. Future extensions could relax this limitation by exploring Multi Input Multi Output (MIMO) controllers. A limitation of the VDI data structure itself is that while it can model non-directional lighting, such as local ambient occlusion [17], directional effects, such as specular lighting, cannot be realized because the VDI stores classified color and opacity.

We see the presented rendering approach to VDIs as a step toward remote visualization of large volumes. The frame rates achieved by the present implementation are consistently higher than those of DVR and of other VDI rendering approaches. Further improvements, such as foveated rendering [4, 33], can potentially increase them even further in the future.

Acknowledgments

This work was supported by the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) Dresden/Leipzig. 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 thank the University of Texas High-Resolution X-ray CT Facility (UTCT) for the Kingsnake dataset.

References

[1] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 424–434. IEEE Press, Piscataway, NJ, USA, 2014. doi: 10.1109/SC.2014.40

[2] 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

[3] M. Aumüller. Hybrid Remote Visualization in Immersive Virtual Environments with Vistle. In H. Childs and S. Frey, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2019. doi: 10.2312/pgv.20191113

[4] D. Bauer, Q. Wu, and K.-L. Ma. Foveolnet: Fast volume rendering using foveated deep neural networks. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–11, 2022. doi: 10.1109/TVCG.2022.3209498

[5] 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

[6] 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

[7] A. W. Cook, W. Cabot, and P. L. Miller. The mixing transition in Rayleigh-Taylor instability. *Journal of Fluid Mechanics*, 511:333–362, 2004. doi: 10.1017/S0022112004009681

[8] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, p. 29–es. Association for Computing Machinery, New York, NY, USA, 2004. doi: 10.1145/1103900.1103929

[9] 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

[10] 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/SIBGRAPI.2013.26

[11] S. J. Garbin, M. Kowalski, M. Johnson, J. Shotton, and J. Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 14346–14355, October 2021.

[12] S. Gül, D. Podborski, T. Buchholz, T. Schierl, and C. Hellge. Low-latency cloud-based volumetric video streaming using head motion prediction. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '20, p. 27–33. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3386290.3396933

[13] S. Gül, D. Podborski, J. Son, G. S. Bhullar, T. Buchholz, T. Schierl, and C. Hellge. Cloud rendering-based volumetric video streaming system for mixed reality services. In *Proceedings of the 11th ACM Multimedia Systems Conference*, MMSys '20, p. 357–360. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3339825.3393583

[14] U. Günther, T. Pietzsch, A. Gupta, K. I. Harrington, P. Tomancak, S. Gumhold, and I. F. Sbalzarini. scenery: Flexible virtual reality visualization on the Java VM. In *2019 IEEE Visualization Conference (VIS)*, pp. 1–5, 2019. doi: 10.1109/VISUAL.2019.8933605

[15] 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

[16] J. Han and C. Wang. Coordnet: Data generation and visualization generation for time-varying volumes via a coordinate-based neural network. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–12, 2022. doi: 10.1109/TVCG.2022.3197203

[17] F. Hernell, P. Ljung, and A. Ynnerman. Local ambient occlusion in direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):548–559, 2010. doi: 10.1109/TVCG.2009.45

[18] A. Kaufman and K. Mueller. 7 - overview of volume rendering. In C. D. Hansen and C. R. Johnson, eds., *Visualization Handbook*, pp. 127–174. Butterworth-Heinemann, Burlington, 2005. doi: 10.1016/B978-012387582-2/50009-5

[19] 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

[20] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser. Cutting the cord: Designing a high-quality untethered VR system with low latency remote rendering. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, p. 68–80. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3210240.3210313

[21] 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. Weinkauf, eds., *Vision, Modeling & Visualization*. The Eurographics Association, 2016. doi: 10.2312/vmv.20161346

[22] Y. Lu, K. Jiang, J. A. Levine, and M. Berger. Compressive neural representations of volumetric scalar fields. *Computer Graphics Forum*, 40(3):135–146, 2021. doi: 10.1111/cgf.14295

[23] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM*, 65(1):99–106, dec 2021. doi: 10.1145/3503250

[24] T. Müller, A. Evans, C. Schied, and A. Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. doi: 10.1145/3528223.3530127

[25] T. Rapp, C. Peters, and C. Dachsbacher. Image-based visualization of large volumetric data using moments. *IEEE Transactions on Visualization and Computer Graphics*, 28(6):2314–2325, 2022. doi: 10.1109/TVCG.2022.3165346

[26] J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered Depth Images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, p. 231–242. Association for Computing Machinery, New York, NY, USA, 1998. doi: 10.1145/280814.280882

[27] S. Shi, V. Gupta, M. Hwang, and R. Jana. Mobile VR on edge cloud: A latency-driven design. In *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19, p. 222–231. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3304109.3306217

[28] A. Simon, R. Smith, and R. Pawlicki. Omnistereo for panoramic virtual environment display systems. In *IEEE Virtual Reality 2004*, pp. 67–279, 2004. doi: 10.1109/VR.2004.1310057

[29] J. E. Stone, W. R. Sherman, and K. Schulten. Immersive molecular visualization with omnidirectional stereoscopic ray tracing and remote rendering. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1048–1057, 2016. doi: 10.1109/IPDPSW.2016.121

[30] A. Tikhonova, C. D. Correa, and K.-L. Ma. Explorable images for visualizing volume data. In *2010 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 177–184, 2010. doi: 10.1109/PACIFICVIS.2010.5429595

[31] A. Tikhonova, C. D. Correa, and K.-L. Ma. An exploratory technique for coherent visualization of time-varying volume data. *Computer Graphics Forum*, 29(3):783–792, 2010. doi: 10.1111/j.1467-8659.2009.01690.x

[32] 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

[33] A. Waschk and J. Krüger. FAVR - accelerating direct volume rendering for Virtual Reality systems. In *2020 IEEE Visualization Conference (VIS)*, pp. 106–110, 2020. doi: 10.1109/VIS47514.2020.00028

[34] S. Weiss, P. Hermüller, and R. Westermann. Fast neural representations for direct volume rendering. *Computer Graphics Forum*, 41(6):196–211, 2022. doi: 10.1111/cgf.14578

[35] Q. Wu, D. Bauer, M. J. Doyle, and K.-L. Ma. Instant neural representation for interactive volume rendering, 2022. doi: 10.48550/ARXIV.2207.11620

[36] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021.

[37] S. Zellmann. Remote Volume Rendering with a Decoupled, Ray-Traced Display Phase. In P. Frosini, D. Giorgi, S. Melzi, and E. Rodolà, eds., *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. The Eurographics Association, 2021. doi: 10.2312/stag.20211479

[38] S. Zellmann, M. Aumüller, and U. Lang. Image-Based Remote Real-Time Volume Rendering: Decoupling Rendering From View Point Updates. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 1385–1394, 08 2012. doi: 10.1115/DETC2012-70811