A Voxel-based Rendering Pipeline for Large 3D Line Sets

Mathias Kanzler, Marc Rautenhaus, and Rüdiger Westermann

Abstract—We present a voxel-based rendering pipeline for large 3D line sets that employs GPU ray-casting to achieve scalable rendering including transparency and global illumination effects. Even for opaque lines we demonstrate superior rendering performance compared to GPU rasterization of lines, and when transparency is used we can interactively render amounts of lines that are infeasible to be rendered via rasterization. We propose a direction-preserving encoding of lines into a regular voxel grid, along with the quantization of directions using face-to-face connectivity in this grid. On the regular grid structure, parallel GPU ray-casting is used to determine visible fragments in correct visibility order. To enable interactive rendering of global illumination effects like low-frequency shadows and ambient occlusions, illumination simulation is performed during ray-casting on a level-of-detail (LoD) line representation that considers the number of lines and their lengths per voxel. In this way we can render effects which are very difficult to render via GPU rasterization. A detailed performance and quality evaluation compares our approach to rasterization-based rendering of lines.

Index Terms—Ray-casting, large 3D line sets, transparency, global illumination

✦

1 INTRODUCTION

3D line sets can be rendered efficiently via the rasterization-based rendering pipeline on GPUs, for instance, by constructing tubes around each line on-the-fly in a geometry shader. For the rendering of very large 3D line sets, however, GPU rasterization introduces the following limitations: a) When using transparency, which requires the rendered fragments to be blended in correct visibility order, all fragments need to be stored in a per-pixel linked list [1] on the GPU and sorted with respect to the current camera position. This approach works efficient for moderate sized line sets, yet for large sets the sorting operation significantly slows down performance, and in some cases the lists can even exceed the GPU's memory (see Fig. 17). b) Operations requiring spatial adjacency queries cannot be embedded efficiently into rasterization-based rendering. For instance, shadow simulation or ambient occlusion calculations to enhance the spatial perception of rendered lines. Even though hard shadows of point lights can be simulated via shadow mapping, this requires a second rendering pass using the full set of geometry, and the resulting high-frequency illumination variations are rather disturbing from a perceptual point of view (see Fig. 12).

An option to overcome these limitations is a rendering technique that does not build upon the order-independent projection of primitives, but is ray-guided and can efficiently traverse the line set along an arbitrary direction. For the rendering of polygon models, GPU voxel ray-casting has been established as a powerful alternative to rasterization. Voxel ray-casting employs a voxel-based object representation in combination with a regular sampling grid that can be traversed efficiently on the GPU [2], [3]. This approach is particular useful because it can generate the sample points along a ray in an order-dependent way, and provides the

ability to perform adaptive LoD selection. Furthermore, the regular sampling grid gives rise to efficient search operations, which have been used to simulate global illumination effects via ray-guided visibility computations [4], [5].

Mimicking voxel ray-casting for line primitives, however, is challenging, since no voxel representation of lines is known that can effectively encode spatial occupancy *and* direction. A solid voxelization of line primitives—or tubes generated from them—into a regular sampling grid [6] is not feasible due to the following reasons: Firstly, for the line sets we consider, too many voxels must be generated to differentiate between individual lines. Secondly, upon voxelization, changes of rendering parameters like line width and line removal require exhaustive processing. Thirdly, already at moderate zoom factors the lines will appear blocky. Finally, since voxels do not encode any directional information, direction-guided shading and filtering as well as direction-preserving LoD construction becomes infeasible.

1.1 Contribution

In this work we propose an alternative rendering technique for large 3D line sets on the GPU, which builds upon the concept of voxel ray-casting to overcome some of the limitations of rasterization-based line rendering. We present a new voxel model for 3D lines and demonstrate its use for GPU line rendering including transparency and ambient occlusion effects. We further show that the voxel model can be used in combination with rasterization-based rendering, by performing the simulation of volumetric effects on this model and letting the rendered fragments access the computed results.

Our specific contributions are:

- A novel voxel-based representation of lines, consisting of a macro voxel grid, and a per-voxel quantization structure using voxel face-to-face connectivity.

M. Kanzler, M. Rautenhaus and R. Westermann are with the Computer Graphics & Visualization Group, Technische Universität München, Garching, Germany.
E-mail: {mathias.kanzler, marc.rautenhaus, westermann}@tum.de

- A LoD line representation that considers per voxel the number of lines, their orientations, and lengths to estimate an average line density and direction at ever coarser resolutions.
- An implementation of GPU ray-casting on the voxel-based representation, including the computation of line, respectively tube intersections on the voxel level, to efficiently simulate transparency as well as local and global illumination effects.

We see our rendering approach as an alternative to existing rendering techniques for 3D line sets when there are many lines introducing massive overdraw, and when transparency or global illumination effects are used to enhance the visual perception (see Fig. 19). In this case, our method can significantly accelerate the rendering process, and it can even be used when the memory requirements of rasterization-based approaches exceed what is available on current GPUs. The possibility to construct a LoD line representation enables trading flexibly between quality and speed, and efficiently performing search operations required for simulating advanced illumination effects.

This paper is organized as follows: After reviewing work that is related to ours, in Sec. 3 we introduce the particular voxel-model underlying our approach and discuss the model generation process. Here we show images of voxelized line data using classical GPU line rendering to emphasize the effects of different voxelization parameters on the quality of the resulting representation. In Sec. 4 we describe the efficient realization of voxel ray-casting on the voxelized line representation, and we demonstrate the simulation of advanced rendering effects like transparency and global illumination using voxel-based ray-casting. Sec. 5 provides a thorough evaluation of our approach regarding quality, speed, and memory requirement, and we demonstrate its benefits and limitations with data sets from different applications. We conclude our work with some ideas on future extensions and applications.

2 RELATED WORK

Our approach is related to established techniques in visualization and rendering, namely line-based rendering of flow fields and voxel-based ray-tracing.

Line-based rendering of flow fields

Today, integral curves in 3D vector fields are usually visualized via the rasterization-based rendering pipeline on GPUs, either as illuminated line primitives [7], [8], or as ribbons or tubes which are constructed around each line on-the-fly in the GPU's shader units [9]. For dense sets of opaque lines, illustrative rendering [10], [11] has been proposed to keep separate lines perceptually visible. Other techniques abstract from single primitives and show flow features via line density projections [12], [13]. Automatic view-dependent selection of transparency has been introduced by Günther et al. [14], to selectively fade out lines in those regions where they occlude more important ones. A user study on perceptual limits of transparency-based line rendering for flow visualization has been conducted by Mishchenko and Crawfis [15]. They also suggest a number

of specific usages of transparency to effectively avoid visual clutter and high levels of occlusions.

When transparent lines are rendered, generated line fragments need to be blended in correct visibility order. On the GPU, this can be realized by using either depth peeling [16] or per-pixel fragment lists [1], [17]. Depth peeling does not require storing and sorting fragments, yet it requires rendering the data set as many times as the maximum depth complexity of the scene, i.e., the maximum number of fragments falling into the same pixel. For the data sets we consider, where along the majority of view rays the depth complexity is in the order of many hundred or even thousand, the resulting increase in rendering times is not acceptable. Per-pixel fragment lists, on the other hand, require only one single rendering pass, yet they require storing all fragments and can quickly run out of memory on current GPUs. This also holds when depth peeling is applied to fragments bucketed by depth [18], even though the number of rendering passes can be reduced. As an alternative to the exact simulation of transparency, stochastic transparency [19] uses sub-pixel masks to stochastically sample the fragments' transparency contributions. Stochastic transparency requires only a rather small and fix number of rendering passes, yet it transforms directional structures into noise. This is especially undesirable in our scenarios, where only lines are rendered and even in transparent regions their directional structure should be preserved.

The technique most closely related to ours is the one by Schussman and Ma [20]. They voxelize streamlines into a regular grid, and compute a spherical harmonics representation of the lighting distribution caused by the line segments in every voxel. Voxel-based rendering with sub-pixel illumination can then be performed, yet single line primitives cannot be determined any more, for instance, to render the initial lines or compute exact occlusions.

Voxel-based ray-tracing

Advances in hardware and software technology have shown the potential of ray-tracing as an alternative to rasterization, especially for high-resolution models with many inherent occlusions. Developments in this field include advanced space partitioning and traversal schemes [21], [22], [23], and optimized GPU implementations [24], [25], [26], to name just a few. All these approaches can be classified as "conventional ray-tracing approaches", since they operate on the polygon object representation and perform classical ray-polygon intersection tests. Recently, Wald et al. [27] proposed the use of ray-tracing for particle rendering, by using a tree-based search structure for particle locations to efficiently find those particles a ray has to be intersected with. The integration of global illumination effects like ambient occlusion into particle visualizations has been demonstrated by Wald et al. [27] and Staib et al [28].

Voxel models have been first introduced 1993 by Kaufman in the seminal paper [29]. Compared to polygonal representations, they provide an interesting set of advantages like easier level of detail computation and combined storage of surface and geometry information. A detailed investigation of algorithms for line voxelization has been provided by Cohen-Or and Kaufman [6]. Interestingly, a

voxel model has also been used in the very first published work on iso-surface visualization: *Cuberilles* [30]. The Cuberille method—or opaque cubes—works by computing the set of grid cells that contain a selected iso-surface and rendering those as small cubes.

In the last years, there has been a lot of research into large octrees to render high-resolution voxel models [2], [3], [31]. All these approaches subdivide the model into a sparse volume, storing only small volume "bricks" along the initial model surface, and use a compute-based octree traversal to render the contained surface. The potential of voxel-based rendering approaches to efficiently simulate global illumination effects has been demonstrated, for instance, in [5], [32]. The survey by Joenssen et al [33] provides a general overview of approaches for simulating global illumination effects on volumetric data sets.

3 VOXEL-BASED CURVE DISCRETIZATION

The input of our method consists of a set of curves, e.g. streamlines, where each curve is approximated by a connected line set. Each line set is represented by a sequence of vertices v_i and corresponding attributes a_i. The attributes can be specified optionally, for instance, to enable the use of a transfer function to interactively change the curves' colors or transparencies. The curve discretization is performed in a pre-process on the CPU, and the generated data structure is then uploaded to the GPU where rendering is performed. This pre-process can also be performed on the GPU, enabling instant update operations when curves are removed or new curves are added. This, however, requires some modifications, the discussion of which we delay until the end of the current section. We propose a two-level grid structure for discretizing the initial curves, as illustrated in Fig. 1.

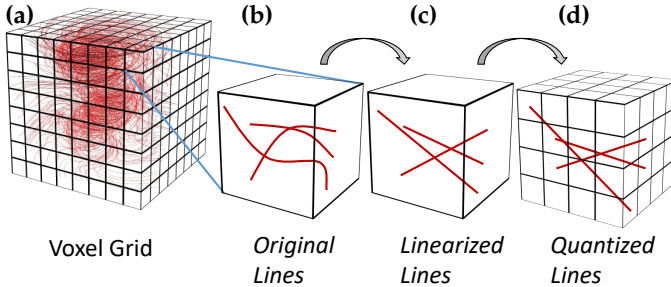**(a)** Voxel Grid   **(b)** *Original Lines*   **(c)** *Linearized Lines*   **(d)** *Quantized Lines*

Fig. 1: Two-level grid structure. First level: Voxel grid, curves are clipped to the voxel faces (b), and per-voxel linear line segments are generated (c). Second level: Line vertices are quantized based on a uniform subdivision of voxel faces (d).

The first level is given by a voxel grid, in which each curve is approximated by a set of per-voxel linear line segments. A regular subdivision of each voxel face yields the second level, where the endpoints of the per-voxel line segments are quantized to the center points of sub-faces.

The use of a two-level structure allows controlling the approximation quality of the voxelization and storing the line segments per voxel in a compact form. The resolution of the voxel grid controls the size of the geometric features that

can get lost when approximating these features by linear segments in every voxel. With increasing grid resolution better approximation quality is achieved, yet at the cost of increasing memory requirement. The end points of each per-voxel line segment can either be stored exactly using floating point values, or they can be quantized to a set of points on the voxel faces. This allows for a compact encoding of the line segments per voxel, and it can be used to control how many lines passing through one voxel are collapsed to one single line. While we let the user select the quantization resolution, more advanced quantization strategies can generate a locally adaptive quantization to assure that local line features are well preserved [34].

3.1 Curve voxelization

Initially, the resolution $r_x \times r_y \times r_z$ of the 3D voxel grid into which the curves are voxelized is set. We use cube-shaped voxels of side length 1 and set the resolution so that the aspect ratio of the bounding box of the initial curves is maintained. The vertex coordinates v_i are then transformed to local object coordinates in the range from $(0, 0, 0)$ to (r_x, r_y, r_z).

For each curve and starting with the first vertex, every pair of vertices v_i and v_{i+1} is processed consecutively. A line through v_i and v_{i+1} is clipped against the voxel boundaries, via line-face intersection tests in the order of their occurrence from v_i to v_{i+1}. If v_i and v_{i+1} are located in the same voxel, no new intersection point is generated. This gives a sequence of voxel-face intersections, and every pair of consecutive intersections represents a line that enters into a voxel and exits that voxel. In general the first and last vertex of a line do not lie on a face, we hence omit the segments from the first vertex to the first face intersection and from the last face intersection to the last vertex.

Fig. 2 illustrates the voxelization process, demonstrating increasing approximation quality with increasing grid resolution, as well as limitations of the piecewise linear curve approximation when the grid resolution is too low.
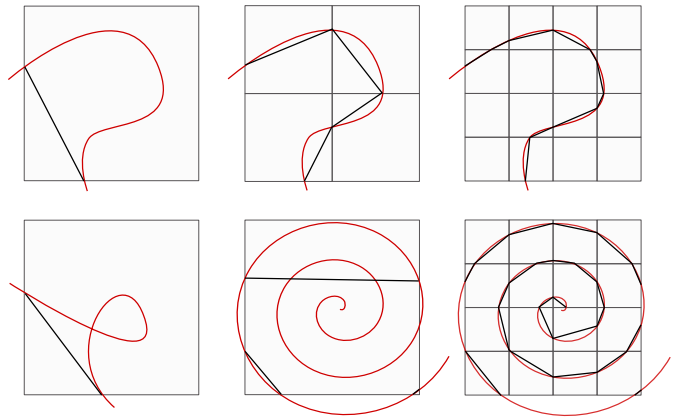

Fig. 2: Curve voxelization: Initial curve in red and piecewise linear approximation in black. Top: Decreasing approximation error with increasing voxel grid resolution. Bottom: Geometric details can be missed if the voxel grid resolution is too low.

For every voxel in the voxel grid, a linked list is used to store the lines intersecting that voxel. Line attributes, interpolated to the line-face intersection points, and IDs that identify to which curve a line belongs can be stored in addition.

3.2 Line quantization

For quantizing the coordinates of the curve-voxel intersection points, we use a small number of bins per face: As illustrated in Fig 1d, every square voxel face is subdivided regularly into $N \times N$ smaller squares. The coordinates of the intersection points are then quantized to the centers of these squares. Thus, every vertex can be encoded via a $2N$ bit pattern (indicating the sub-square in a voxel face) and a 3 bit face ID (indicating on which face the vertex is located). The bits are packed into a single word that is just large enough to store them. Since voxels have unit size and each voxel face is subdivided equally, the vertex location in object coordinates (quantized to the per-face square centers) can be computed from a bit pattern stored per-vertex. Fig. 3 illustrates the quantization process and the effects of different values of N on the curve approximations.
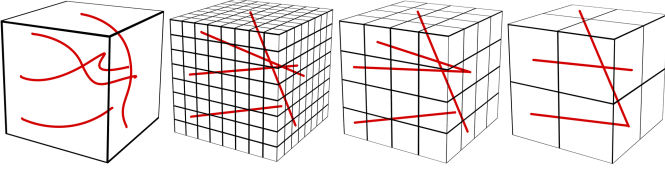

Fig. 3: Line quantization. From left to right: Original curves, per-voxel linear approximation using vertex quantization to $8^2, 4^2$, and 2^2 sub-faces (bins) per voxel face, respectively. If the number of bins is too low, lines can fall onto each other.

The quantization process introduces an additional approximation error that depends on the selected subdivision of voxel faces. This error does not cause any geometric details to be lost, yet it slightly jitters the vertex locations and, thus, affects a line's orientation. In particular, different lines might be mapped onto the same quantized line because their vertices are quantized to the same locations.

Fig. 4 shows a direct comparison between the original curves and the voxelized curves using a voxel grid resolution and per-face subdivision at which the discretization errors can only just be perceived. In both cases the lines are rendered via GPU rasterization; by starting at the first vertex and then either by traversing the original set of vertices and constructing tubes around each line on-the-fly in a geometry shader, or by performing the same construction on the discretized line set. Especially the straight curves in the foreground show some subtle bumpiness that is caused by the quantization of line vertices.

To demonstrate how, in general, the voxel grid resolution and the voxel face subdivision affect the final reconstruction quality, Fig. 5 shows some extreme cases which also reveal the interplay between both resolutions. A detailed analysis of the dependencies between resolution, quality and memory consumption for different resolutions is given in Sec. 5. While a low resolution of the voxel grid affects in particular the per-voxel approximation error, a low degree of face
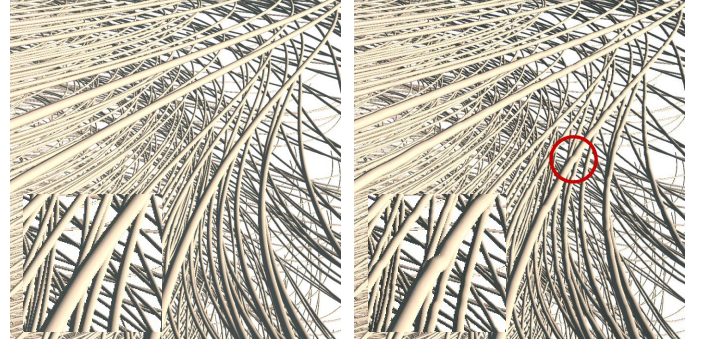

Fig. 4: GPU rasterization of initial lines (left) vs. rasterization of voxelized lines using 256^3 voxel grid and curve-face intersections quantized to 16^2 bins (right).

subdivision can introduce additional C^1-discontinuities at voxel transitions, i.e., by jittering a vertex that is shared by two lines with the same orientation. Especially high frequent variations that occur when a short line is jittered are perceptually noticeable, as indicated by the last example in Fig. 5.

3.3 GPU implementation

The per-voxel linked list representation used to store the lines is neither memory efficient, as each line stores a pointer to the following element, nor can subsequent lines be accessed quickly, as they are not stored consecutively in memory and, thus, cached reads are prohibited. Therefore, before uploading the data to the GPU, the lines are reordered in memory so that all lines passing through the same voxel are stored in consecutive elements of a linear array. The final data structure stores for each voxel a header, which stores the number of lines for that voxel, and a pointer to the array element storing the first line passing through it.

Constructing the voxel model on the GPU consists of three stages: Firstly, in parallel for every initial curve we compute the lines per voxel and write them into a linear append buffer. Every segment is assigned the unique ID of the voxel it is contained in. Next, the buffer is sorted with respect to the voxel IDs, so that all lines falling into the same voxel are located consecutively in the buffer. An exclusive parallel prefix sum is then computed in-place over the buffer to count the number of line per voxel. Now, for every voxel the start index of its line set can be determined from the content of the buffer, and stored in a separate voxel buffer at the corresponding location. This buffer is used at render time to look up at which position in the global buffer the lines for a particular voxel are stored.

3.4 LoD construction

One important property of classical voxel models for surfaces is that a LoD structure can be generated efficiently by simple averaging operations on the voxel values to aggregate information. For a line set that is voxelized as proposed in our work, such an averaging operation cannot be applied immediately because a) an averaging operator for lines first needs to be defined, and b) every voxel might store not only one but many lines. It is worth noting that

Fig. 5: Rasterization of streamlines in the 256^3 Aneurysm II dataset. (a) Initial lines. (b) Line encoding in 64^3 voxel grid with exact curve-voxel intersections. (c) Line encoding in 64^3 voxel grid with curve-voxel intersections quantized to 8^2 bins. (d) Line encoding in 256^3 voxel grid with curve-voxel intersections quantized to 8^2 bins.

regardless of how a) and b) are addressed, it is impossible, in general, to represent the lines in one voxel by one single average line so that continuity with the average lines in adjacent voxels is ensured.

We address the problem of LoD construction for a voxelized line set as follows: Since we intend to use the LoD structure in particular to accelerate the simulation of global illumination effects, we derive a scalar indicator for the amount of light that is blocked by the lines in a single voxel. By using standard averaging operators, an octree LoD structure can then be generated in a straight forward way from the indicator field. For every voxel, we compute an average *density* value ρ from the lines passing through it, by taking into account the lines' lengths and opacities:

$$\rho = \sum_{l_i \in L} length(l_i)\sigma_i, \tag{1}$$

where L is the set of lines in the voxel, and l_i and σ_i are the length and opacity of the i-th line in that voxel, respectively. The lines' opacities are either set to a constant values or assigned individually via a transfer function. Finally, an octree is build bottom-up by averaging the density values in 2^3 voxels into one voxel at the next coarser level (see Fig. 6).

per voxel and, thus, increasing the memory consumption significantly.

In addition to the per-voxel opacity values, one representative average line is computed for every voxel; by averaging separately the start points and end points of all lines in a voxel. This gives two average points which are snapped to the closest bin on the voxel boundaries, and from which the average line is computed. Care has to be taken regarding the orientation of lines, i.e., when two averaged lines have vastly opposite directions. Therefore, before considering a new pair of start and end point, we first test whether the corresponding line has an angle of more than 90 with the previous line, and we flip the line if this is the case. We also attempt to establish connectivity between representative lines in adjacent voxels if their endpoints are in the same voxel face, by snapping the endpoints to the quantization bin into which the average of both points is falling. In this way, we can often achieve continuous representative curves, even though it is clear that in general such a continuous fit is not possible. Fig. 7 illustrates a multi-resolution representation of a set of curves. Note in particular how well in certain regions the average curves even on the coarser resolution levels represent the initial curves.
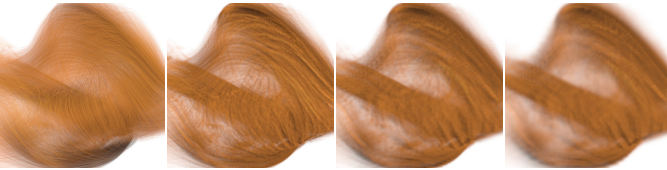

Fig. 6: Volume rendering of the initial line density per voxel as computed via eqn. (1), and the line density values at the first, second, and third LoD.

Fig. 7: Rasterization of the initial curves (left) compared to 1st (middle) and 2nd (right) LoD.

The density values can be interpreted as per-voxel opacities telling how much light is blocked by a voxel. Since the amount of blocking depends also on how the lines are oriented with respect to the incoming light, direction-dependent opacity values are favorable in principle [35]. Even though they can be generated in a straight forward way, by simulating the attenuation from a number of selected directions via ray-casting, we did not consider this option in the current work to avoid storing many values

The representative lines are stored in addition to the per-voxel line set, and a LoD representation can be computed by propagating these lines to ever coarser resolution levels. The average per-voxel opacity values at different resolution levels are stored in a set of 3D texture maps. By using these quantities we can efficiently realize a number of rendering accelerations and effects, which we will introduce in Sec. 4.
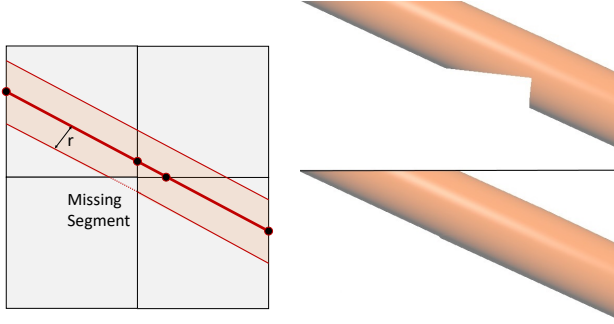
Fig. 8: When a tube extends into a voxel, and a ray intersects that voxel but none of the voxels in which the tube's center line is encoded (left), no intersection is found and a piece of the tube is missing (right top). By including neighbouring voxels in the intersection test, missing intersection points are found (right bottom).

4 VOXEL-BASED LINE RAYCASTING

Once the voxel-based line representation has been constructed and the data is residing in GPU memory, volume ray-casting can be used to render the lines in front-to-back order. For every pixel a ray is cast through the voxel grid, thereby going from voxel face to voxel face using a digital differential analyzer algorithm. The voxel grid serves as a search structure to efficiently determine those lines that need to be tested for an intersection with the ray.

Whenever a voxel is hit, the voxel header is read to determine how many lines are stored in that voxel, and if a voxel is empty, it is skipped. Otherwise, the lines are read consecutively and intersected with the ray. Here it is assumed that the lines are in fact tubes with a user-defined radius, so that the intersection test becomes a ray-tube intersection test. This test yields an entry and exit point, from which the distance the ray travels inside the tube can be computed and used, for instance, to simulate attenuation effects. If more than one intersection with tubes are determined, the intersections are first computed and then sorted in place in the correct visibility order.

4.1 Ray-tube intersections

During ray-casting, a problematic case can occur if a tube stands out of the voxel in which its center line is defined. Since the tube expands into a voxel which may not know the center line, the piece in this voxel cannot be rendered if the ray doesn't intersect any of the voxels in which the center line is encoded. This situation is illustrated in Fig. 8. A straightforward solution to this problem is for a given ray and voxel intersection to also test for intersections with lines from adjacent voxels, a solution that unfortunately decreases rendering performance due to the increased number of intersection tests. However, our experiments have shown that the artifacts that occur when neglecting the missing segments are only rarely visible. In particular when rendering tubes with transparency the artifacts are hardly perceivable. As a compromise, we hence during interactive navigation restrict our method to testing against the lines in the voxel hit by the ray. As soon as the camera stands still, adjacent voxels are also taken into account.
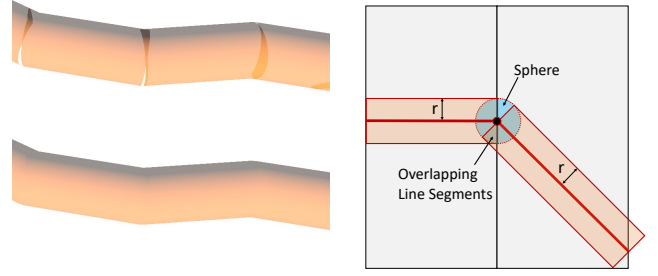

Fig. 9: Left top: Overlapping tubes are blended incorrectly and parts are missing. Rendering a sphere at line joints (right) continuously closes the gaps (left bottom).

Another problematic situation occurs at the joint between adjacent lines, i.e., a gap is produced if the lines do not have the same direction (see Fig. 9). The gap is filled by rendering spheres at the end points of the line segments with a radius identical to the tube's radius. To avoid blending the same line twice, we keep track of the IDs of intersected lines using bit operations and consider an intersection point only once.

At every entering ray-tube intersection point, we calculate the tube normal and evaluate a local illumination model. If the tubes are rendered opaque, this only has to be done for the first intersection point. The resulting color value is combined with the tube color, and this color is used as pixel color. Fig. 10 compares the rendering of opaque tubes via GPU ray-casting to GPU line rasterization.

Fig. 10: Opaque line rendering. Left: GPU ray-casting of voxelized lines (8 ms, 256^3 voxel grid, curve-face intersections quantized to 16^2 bins). Right: GPU line rasterization (12 ms).

Interestingly, in terms of rendering quality rasterization and ray-casting do not seem to show any significant differences at the selected discretization resolution, yet even for opaque lines ray-casting renders already faster than rasterization. The main reason is that ray-casting can effectively employ early-ray termination once the first intersection with a tube is determined. Since rasterization renders the lines in the order they are stored, which is not the visibility order in general, it needs to generate a considerably larger amount of fragments.

4.1.1 Transparency rendering

If the tubes are rendered semi-transparent, at every entering ray-tube intersection an opacity value α is either read from the voxel header or assigned via a transfer function. The opacity value is then used to modulate the tube color, and this color is blended with the pixel color using front-to-back α-compositing, i.e., in the order in which the tube intersections are determined along the ray. The described way of handling opacity is exactly how the final pixel color is computed when semi-transparent lines are rendered via GPU rasterization. In contrast to ray-casting, however, in GPU rasterization all generated fragments first have to be stored and finally sorted per-pixel with respect to increasing depth. Only then can the fragments' colors be blended in correct order.

Fig. 11 shows colored and semi-transparent lines, once rendered via GPU rasterization and once via GPU ray-casting. In this situation, the advantage of ray-casting comes out most significantly: Since the ray-tube intersection points are computed in correct visibility order, there is no need to store and finally sort these points for blending. Due to this, the performance gain of ray-casting compared to rasterization now becomes significant; about a factor of 7 for the used dataset.
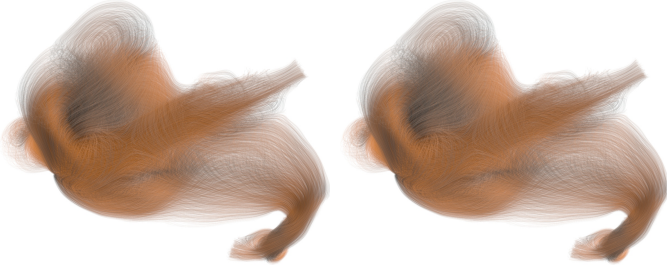
Fig. 11: Left: Transparency rendering using fragment linked lists on the GPU (160 ms, 600 MB fragment list). Right: GPU ray-casting (24 ms, 256^3 voxel grid, curve-face intersections quantized to 16^2 bins, 60 MB voxel representation).

Furthermore, additional acceleration and quality improvement strategies can be integrated into ray-casting in a straight forward way. Firstly, α-termination, i.e., the termination of a ray once the accumulated opacity exceeds a user-defined threshold, can be used to reduce the number of ray-tube intersection points. If a LoD representation is available, even opacity-acceleration [36] can be employed, i.e., increasing the step size along the ray and simultaneously sampling the opacity from ever coarser resolution levels with increasing optical depth. Secondly, instead of considering a constant opacity per tube, the opacity can be made dependent on the distance the ray travels within the tube. Since together with the ray entry point also the ray exit point is computed, this distance is immediately available. Even the handling of penetrating tubes does not impose any conceptual problem, and only requires to sort the ray-tube intersections locally per voxel.

4.1.2 Shadow simulation

The possibility to efficiently trace arbitrary rays through the voxelized 3D curves can be employed to efficiently simulate global illumination effects such as shadows. Shadows provide additional depth and shape cues, and can significantly enhance the visual perception of the curves geometry and their spatial relationships.

The simulation of hard shadows of a point light source can be realized by sending out shadow rays and testing if the ray hits another tube before it hits a light source. However, as demonstrated in Fig. 12b, due to the high frequency shadow patterns that are caused by a dense set of curves, hard shadows rather disturb the visual perception than help to improve it.

We propose the following two approaches to incorporate shadows into the rendering of large line sets without introducing high-frequency shadow patterns. The first approach is to test the shadow rays against the representative lines at a coarser LoD, thus reducing the number of lines that throw a shadow and making the shadows wider and more contiguous (see Fig. 12c). The second approach replaces hard shadows by soft shadows, by sampling the line density values along the shadow rays to measure the amount of blocking. (see Fig. 12d). Both approaches require to traverse only one single shadow ray towards the light source and can be performed efficiently on the proposed LoD representation.
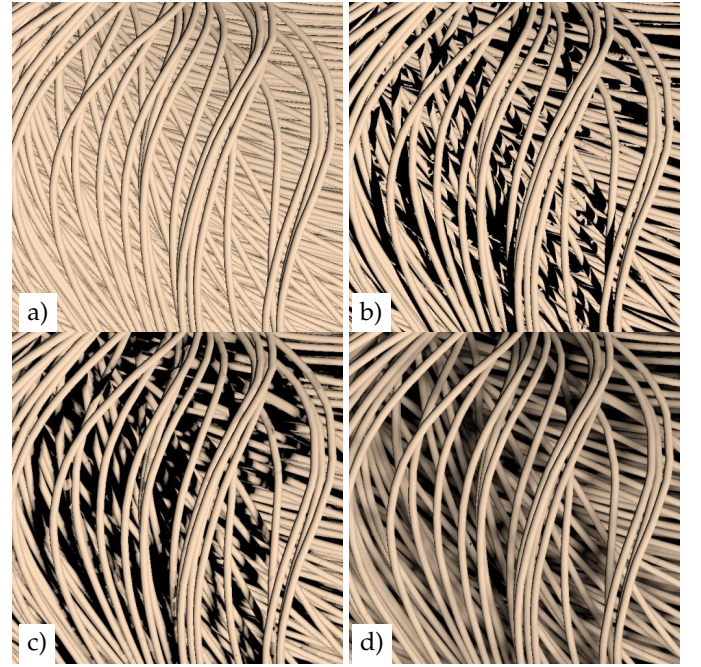

Fig. 12: Voxel-based line ray-casting with (a) local illumination (10 ms), (b) point light shadows (22 ms), (c) point light shadows from representative lines at first LoD (12 ms), (d) soft shadows from line density values via cone-tracing (11 ms).

In the first approach, the shadow rays traverse the voxels of a selected LoD and test for intersections with the representative line in every voxel. Even though we do not avoid hard shadows in this way, by using one single yet thicker line to represent many thinner lines, the frequency of variations from shadow to non-shadow can be reduced significantly. The second approach mimics the effect of an area light source, requiring, in general, to use many rays

to estimate how much of the light leaving the area light is blocked. Instead, we sample the line density values with one ray in a way similar to cone-tracing [4], i.e., by sampling from ever coarser resolution levels with increasing distance from the illuminated point. In particular, we simulate the amount of light that falls onto the point along a cone with an opening angle that subtends one voxel at the finest level.

When comparing the results of both approaches to the rendering of hard shadows in Fig. 12a, one can see that the shadow frequency is considerably reduced and the visual perception is improved. The major shadowing effects, on the other hand, are still present in the final renderings, and the spatial relationships between the lines are effectively revealed. When using the first approach, the render time increases about 20% compared to the rendering without shadows; the second approach yields a decrease of about 10%. These only marginal decreases are due to the use of the LoD representation, which requires testing against only one single line per voxel when using the first approach, and interpolating trilinearly in the line density fields at different resolution levels when using the second approach. Since a texture lookup operation takes less time than an explicit ray-tube intersection test, the second approach performs even faster than the first one at almost similar quality.

4.1.3 Ambient occlusion

Both rendering approaches for low-frequency shadows can also be used to simulate ambient occlusions (AO), i.e., soft shadows that occur in the cavities of a 3D object when indirect lighting is cast out onto the scene. As demonstrated in Fig. 13, the soft shadows from ambient occlusions help in particular to enhance the spatial separation between individual lines or bundles of lines.
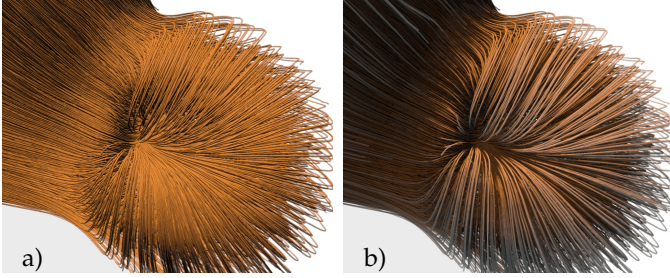

Fig. 13: Local lighting (a) vs. ambient occlusion (b).

AO is simulated by casting out rays to sample the surrounding geometry, and computing how much light from an environment map is blocked by this geometry. For simulating shadows we restrict the direction along which we send out rays to the direction of a directional light or towards a point light source, yet ambient occlusion requires to send out rays into the entire upper hemisphere with respect to the normal direction at a surface point. AO can be integrated in a straight forward way into the ray-based rendering pipeline, by spawning at every visible point a number of secondary rays into the hemisphere and calculating whether the environment light is blocked or not. In Fig. 14a, 2500 rays per visible point were used to uniformly sample the hemisphere, and these rays were intersected against the tube segments in every voxel to determine whether the light is
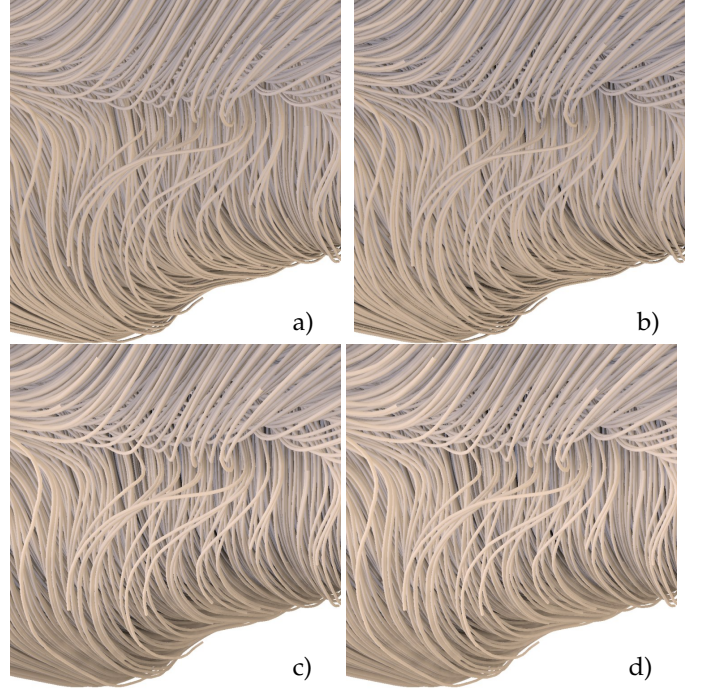

Fig. 14: Top: Hemisphere ambient occlusion using 2500 shadow rays. (a) Rays are tested against line geometry (>10 s). (b) Occlusions are estimated from line density values at first LoD (600 ms). Bottom: Pre-computed per voxel spherical occlusions, sampled trilinearly along 2500 (c) and 25 (d) shadow rays at 12 ms and 1130 ms, respectively.

blocked or not. The higher variance in the ambient occlusion values when less rays are used can clearly be seen.

To enable interactive updates of AO values when the scene or light situation changes, the number of rays as well as the number of objects against which the shadow rays are tested need to be reduced. Screen-space approaches [37] compute a rough AO estimation by using a few rays in 2D screen-space (between 8 and 20 in realtime applications), and by testing these rays against rendered surface points in a short radius of influence.

To overcome visual shortcomings of screen-space calculation, we compute AO values in 3D space, and present two acceleration strategies to efficiently approximate the amount of occlusion per tube point. Similar to screen-space ambient occlusion, we restrict the sampling of occluding structures to a radius of influence (in our case 15 voxels at the finest voxel resolution).

The first approach is to approximate the amount of occlusion along a ray by sampling the line density values at the finest level voxel grid via trilinear interpolation (Fig. 14b), instead of testing against the tube geometries at this level (Fig. 14a). The values along a ray are accumulated until either full blocking is reached (line density ≥ 1) or the ray reaches beyond the radius of influence of leaves the domain. The occlusion values are finally integrated over all rays and normalized by dividing through the number of rays.

The renderings in Figs. 14a and b show that the AO values vary strongly around the tube axis, which is due the hemisphere sampling of occlusions wrt to the varying normal direction. Due to the many tubes that are rendered,
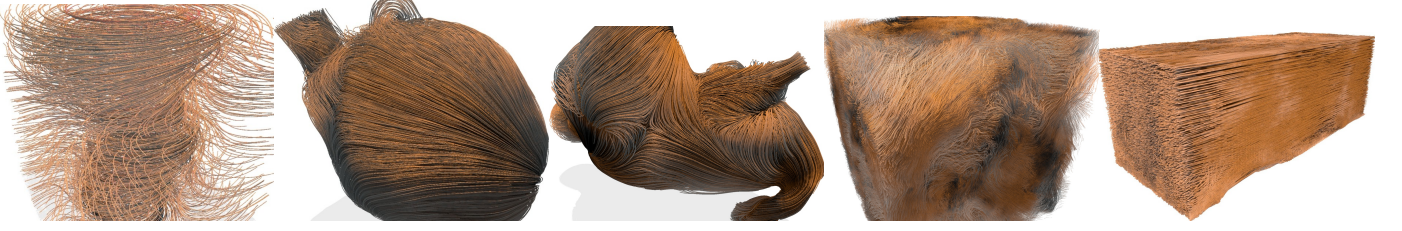
Fig. 15: The datasets we have used in our experiments: Tornado, Aneurysm I, Aneurysm II, Turbulence, Weather Forecast.

this adds high frequent intensity variations which rather disturb the visual impression than help to enhance the spatial relationships between the tubes. To avoid this effect, we propose a second approach which computes pointwise AO values independent of the surface orientation by considering occlusions in the entire sphere around each visible point (see Fig. 14c and d). The AO values are first approximated per voxel in a pre-process, and at runtime these values are trilinearly interpolated at the locations of the visible tube points. In this way the AO values reflect the local spherical surrounding of a line rather than the surrounding in the normal direction at the tube points. This emphasizes in a far better way the embedding of a line in the surrounding set of lines.

5 RESULTS AND DISCUSSION

In this section, we analyze the quality, memory consumption, and performance of our approach. All times were measured on a standard desktop PC, equipped with an Intel Xeon E5-1650 v3 CPU with 6×3.50 GHz, 32 GB RAM, and an NVIDIA GeForce GTX 970 graphics card with 4 GB VRAM. For all renderings, the view port was set to 1920×1080.

We used the following datasets to analyze the quality, memory requirements, and performance of the proposed voxel-based rendering approach for line sets (see Fig. 15):

- **Tornado**: 1000 randomly seeded streamlines in a flow forming a tornado.
- **Aneurysm I/II**: 4700 / 9200 randomly seeded streamlines in the interior of two aneurysms [38]. Streamlines were advected up to the vascular wall, resulting in empty space up to the cuboid domain boundaries.
- **Turbulence**: 50000 domain-filling streamlines advected in a forced turbulence field of resolution 1024^3 as described by Aluie et al. [39].
- **Weather Forecast**: 212000 domain-filling path lines computed over 96 hours each on the wind field of a forecast by the European Centre for Medium-Range Weather Forecasts.

5.1 Quality analysis

To analyze the effect of the resolution of the voxel grid and the quantization structure on reconstruction quality, we performed a number of experiments with the datasets listed above. The subjective visual quality of the rendered images is hard to measure, wherefore we attempt to quantify the quality objectively by measuring the following quantities: For different resolutions we measured the mean Hausdorff

distance between the original curves and their piecewise linear approximations (Fig. 16a), the mean angle between the tangents at the original curve points and the curve's piecewise linear approximations (Fig. 16b), and the number of line segments falling onto each other due to the quantization of vertex coordinates (Fig. 16c).

Fig. 16a indicates that already at a voxel grid resolution of 128^3 and a quantization resolution of 32, suitable reconstruction accuracy is achieved. At a voxel grid resolution of 64^3, geometric features get lost and cannot be faithfully reconstructed even at high quantization resolution. Fig. 16b shows essentially the same dependencies, yet one can observe a stronger effect of the quantization resolution. The local directional changes of the curve tangents due to displacements of the vertex coordinates is scale independent and depends mainly on the quantization resolution. From Fig. 16c it can be seen that at a voxel grid resolution of 128^3 and higher, and starting at a quantization resolution of 32, the number of duplicate lines is below 0.1% of all encoded lines and doesn't change significantly beyond these resolutions. The maximum Hausdorff distance, on the other hand, is always bounded by the voxel diagonal, and the directional error can be up to 180 degrees, if a curve makes a loop in a voxel.

Supported by our analysis, and further verified by comparing the visual quality of the original curves and the voxel-based curve representations, we found that a voxel grid resolution of 256^3 and a quantization level of 32 is for all cases the resolution at which a further increase in voxel resolution or bin size only causes little improvement in error quantities This is also evidenced by the close-up views in Fig. 18, where no apparent differences between the original curves and their voxelized counterparts can be perceived. Due to this, we decided to use these resolutions in the performance analysis and the analysis of illumination effects below.

5.2 Memory statistics

Our approach requires a 5 byte header for every voxel, to indicate how many lines are encoded per voxel (1 byte, restricting to a maximum of $2^8 - 1$ lines per voxel) and to reference the memory address where the lines are stored (4 bytes). In addition, every line is encoded by specifying at which of the 6 voxel faces the two endpoints are located ($2 \cdot 3$ bits), and addressing the sub-face on each voxel face to which the endpoints are quantized ($2 \cdot 10$ bits for a quantization resolution of 32^2). Furthermore, we use 1 byte per line to map to a transparency or color value and 5 bits to encode a local line ID. For a selected quantization resolution, the minimum number of bytes that is required to encode this
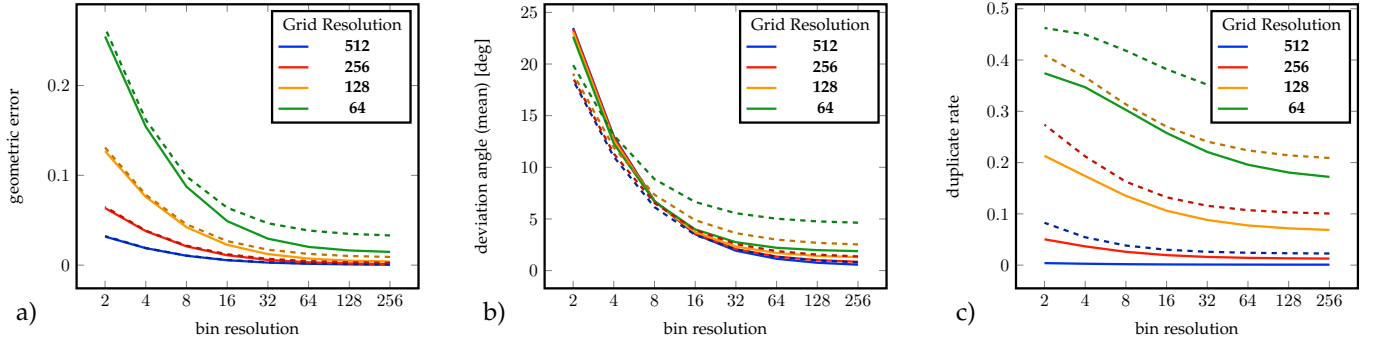
Fig. 16: Comparing the Aneurysm II (solid line) with the Weather Forecast (dashed line) dataset using different metrics: (a) Distance Error (in units of voxel size at a grid of 64^3), (b) tangent deviation, (c) Number of duplicates in proportion to the number of generated segments

information is used. Overall, our approach requires 4, 5, and 6 bytes per line, respectively, when quantization resolutions of (4,8), (16,32) and (64,128) are used.

Table 1 compares the memory consumption for different resolution and quantization levels with the memory that is required to store the original lines on the GPU, using 32 bit float values per vertex component and 1 byte per vertex to map to transparency or color. In the second column we show in brackets the memory that is required in average over multiple views to store the per-pixel fragment lists when transparent lines are rendered. It can be seen that for all but the Tornado dataset the voxel-based representation at our selected resolution level ($256^3/32^3$) has a significantly lower memory footprint. Even when opaque lines are rendered and the extra memory for storing the fragment lists is not required, a rather moderate increase of about a factor of 4 to 5 is observed. One exception is the Weather Forecast dataset, which is comprised of many curves with only few long lines. These lines are split into many smaller lines and stored in the voxel representation, so that the memory requirement is significantly increased compared to the original line representation.

It is in particular interesting that due to memory limitations on the GPU the Weather Forecast dataset cannot be rendered via GPU rasterization if transparency is used (see Fig. 17). Our approach requires only 144 MB to store the view-independent voxel-based representation on the GPU.
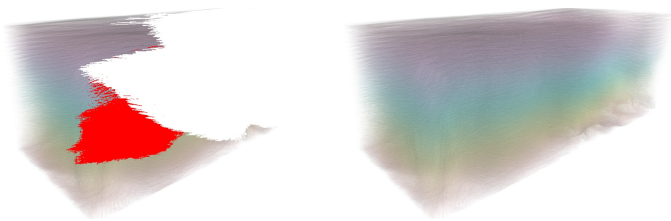

Fig. 17: Left: GPU rasterization using fragment linked lists fails due to memory limitations. Red indicates 512 or more fragments fall onto a pixel, when not all fragments could be stored due to memory limitations the pixel is set to white. Right: Voxel-based ray-casting renders at 23 ms.

5.3 Performance analysis

For all datasets, we compare the performance of rasterization-based line rendering with and without frag-

ment lists to voxel-based line ray-casting (see Tab. 2). For rasterization-based rendering, which unfolds a given line primitive into a number of triangles approximating a cylinder, we can also unfold the line into a quadrilateral and let the generated fragments perform an analytical ray-cylinder intersection test. This approach, even though it frees the geometry shader, requires some additional stitching geometry to achieve the appearance of a continuous tube. Furthermore, it increases the load in the pixel shader, which is already the bottleneck when using transparency; the use case we focus on in this work. Since even for opaque lines we observed only a slight performance increase when using this approach, we decided to refrain from using it. In all of our experiments we let the geometry shader generate 8×2 triangles per line segment.

To also compare the performance of our approach to that of triangle-based GPU ray-tracing, for all datasets we saved the generated triangles to a file and used them as input geometry for the OptiX ray-tracing framework [40]. We selected Spatial Splits in Bounding Volume Hierarchies (SBVH) [41] as acceleration structure. Besides the fact that both the Turbulence and Weather Forecast datasets could not be rendered due to memory limitations, for the smaller datasets (Aneurysm I and II) we observed almost similar frame rates when rendering opaque tubes. When rendering transparent tubes, however, the performance dropped significantly, of up to a factor of 10. We attribute this to the fact that SBVH allows skipping empty space surrounding the line sets efficiently, yet when the lines are dense in the interior (as in Aneurysm I and II), repeated traversal operations slow down the performance. If more space in the interior is empty, this limitation is reduced: OptiX renders the opaque lines in Tornado at 160 fps, while our approach renders at 83 fps. However, it is fair to say that in this case also our approach can be speeded up significantly by using the octree voxel grid to skip empty space. At a per-line opacity of 25 percent, OptiX and our approach (w/o empty space skipping) render at 35 fps and 67 fps, respectively.

In Fig. 21 we show the use of transparency to reveal interior structures that are occluded when opaque lines are rendered. Even when opaque lines are rendered, so that fragment lists and sorting is not required in GPU rasterization, the larger datasets can be rendered at higher rates using voxel-based ray-casting. The main reason is that a ray can be terminated immediately when the first

TABLE 1: Statistics on memory consumption and preprocessing time. The original geometry is encoded in three float values per vertex. In the second column, we give in brackets the memory that is required to store the per-pixel fragment lists on the GPU. The remaining columns show the memory that is used by the voxel-based representation—with voxel grid resolution V^3 and quantization resolution Q^2 given as V^3/Q^2—, and in brackets the preprocessing time to generate the representations. Per vertex and per quantized line a 1 byte index is stored.

Dataset	Lines (vertices per line)	Original Geometry (Linked List)	$512^3/256^2$ (Preprocess)	$256^3/128^2$	$256^3/32^2$	$128^3/128^2$	$128^3/32^2$	$64^3/8^2$
Tornado	1000 (250)	3 MB (100 MB)	811 MB (1.4 s)	103 MB (0.2 s)	103 MB (0.2 s)	14 MB (0.1 s)	14 MB (0.1 s)	2 MB (0.1 s)
Aneurysm I	4700 (410)	25 MB (600 MB)	731 MB (1.6 s)	111 MB (0.4 s)	107 MB (0.4 s)	25 MB (0.2 s)	22 MB (0.2 s)	6 MB (0.3 s)
Aneurysm II	9200 (367)	44 MB (750 MB)	702 MB (1.9 s)	116 MB (0.6 s)	109 MB (0.6 s)	29 MB (0.5 s)	26 MB (0.5 s)	8 MB (0.4 s)
Turbulence	50000 (220)	143 MB (2500 MB)	1087 MB (3.5 s)	221 MB (1.4 s)	201 MB (1.4 s)	72 MB (1.3 s)	62 MB (1.3 s)	21 MB (1.0 s)
Forecast	212000 (13)	36 MB (>4000 MB)	573 MB (5.6 s)	209 MB (4.6 s)	177 MB (4.6 s)	98 MB (5.0 s)	82 MB (5.0 s)	31 MB (4.7 s)

ray-tube intersection is computed, while GPU rasterization always needs to generate all fragments even if the early depth-test can discard many of them before entering the fragment stage. On the other hand, for the Tornado dataset, where the line density is rather low so that many rays need to be traversed through the entire voxel grid, voxel-based ray-casting performs slower than rasterization-based rendering. This still holds when transparency is used. The fact that in rasterization-based rendering using transparency all fragments need to be sorted explains the over-linear increase of the render times, while the render times remain almost constant when ray-casting is used on the voxel-based representation.

The worst case scenario for the ray-caster is a very dense line set with very low line opacity. This prevents rays from terminating early, so that all lines along the rays have to be accessed and tested for intersections. Even though this scenario is quite unusual, we analyzed the rendering performance in this situation for the Weather Forecast and Turbulence dataset. In this case, the Forecast dataset can still be ray-cast in roughly 144 ms, while the dataset cannot be rendered via rasterization because the fragment lists exceed the available GPU memory. When using the Turbulence dataset with very low line opacity, GPU rasterization and voxel-based ray-casting render at 380 ms and 124 ms, respectively, demonstrating the efficiency or our approach even in this extreme situation.

5.4 Illumination effects

The secondary rays used to simulate global illumination effects like shadows and ambient occlusions are solely traversed on the first LoD level of the voxel grid. Therefore, these effects cause only a moderate increase in the render time, yet they can significantly improve the spatial perception of the rendered line structures. This is demonstrated in Fig. 19, where opaque lines are rendered with soft shadows and ambient occlusions (right).

For instance, for 100 rays sampling in a sphere with a radius of roughly 5 times the voxel size (with a step size of 1 voxel size), ambient occlusions for all 256^3 voxels can be computed in roughly 9 ms. Soft shadows can be computed in about 2 ms. The computed illumination values can be stored inside our voxel representation and only have to be updated if the scene changes.

Fig. 18 shows a comparison between reference images produced by a rasterizer using fragment linked lists (left) and images produced using our ray-casting approach using a memory efficient representation (right). In all images an attribute stored for each line segment was mapped by a transfer function to a color value. Opacity values were manually predefined. The memory efficient representation introduces some artifacts like jittered line segments and visible gradations at the voxel boundaries as attributes are not interpolated in between. Comparing the reference image and the high resolution represenation, no significant differences are observable.

In Fig. 19, lines are rendered with local illumination (left) and soft shadows in combination with ambient occlusions (right). When rendering huge amounts of semi-transparent lines, single lines cannot be distinguished any longer. Ambient occlusions help to distinguish individual curves, since less light is reaching occluded curves. Shadow effects enhance the overall structure of the dataset and indicate the relative position of curves to each other. In combination with ambient occlusions, details in shadowed regions are preserved.

6 CONCLUSION AND FUTURE WORK

We have introduced a new approach for visualizing large 3D line sets, by using GPU ray-casting on a novel voxel-based line representation. For large sets of transparent lines we have shown significant performance improvements over rasterization-based approaches. The voxel-model gives rise to an efficient integration of local and global illumination effects.

A limitation of our approach is the lack of hardware accelerated anti-aliasing. While anti-aliasing in triangle rasterization does not have a significant effect on performance, it is expensive in ray-casting since several rays have to be traced per pixel. Another limitation is with respect to the addition of new lines. Since the voxels are densely packed in one linear array on the GPU, the whole memory needs to be restructured when new lines are added. On the other hand, since generating the voxel model is sufficiently fast, even a complete re-voxelization of all lines can be performed at high speed.

In the future we will further optimize the voxelization process so that even time-varying line sets can be visualized at interactive rates, and we will incorporate empty space skipping using the voxel hierarchy. Moreover, it will be interesting to investigate the use of adaptive line quantization strategies, for instance, based on line curvature, to generate an adaptively refined voxel model.
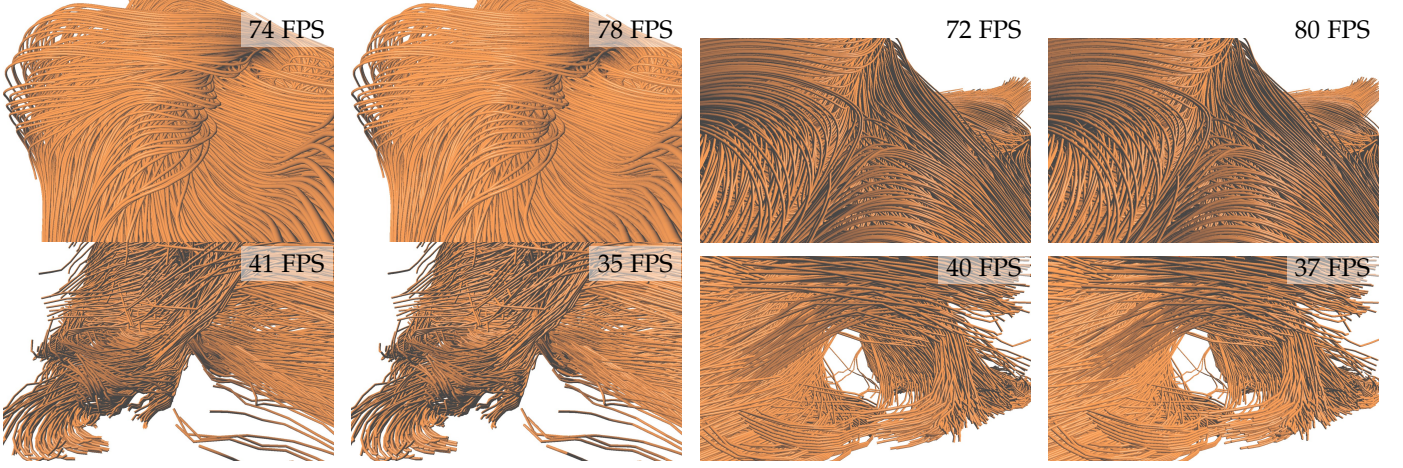
74 FPS

78 FPS

72 FPS

80 FPS

41 FPS

35 FPS

40 FPS

37 FPS

Fig. 18: Comparison between rasterization-based rendering (left) and voxel-based line raycasting using 256^3 voxels partitioned into 32^2 bins per face (right), for the Aneurysm II and the Weather Forecast dataset. The lines are rendered opaque with local illumination.

69 FPS

43 FPS
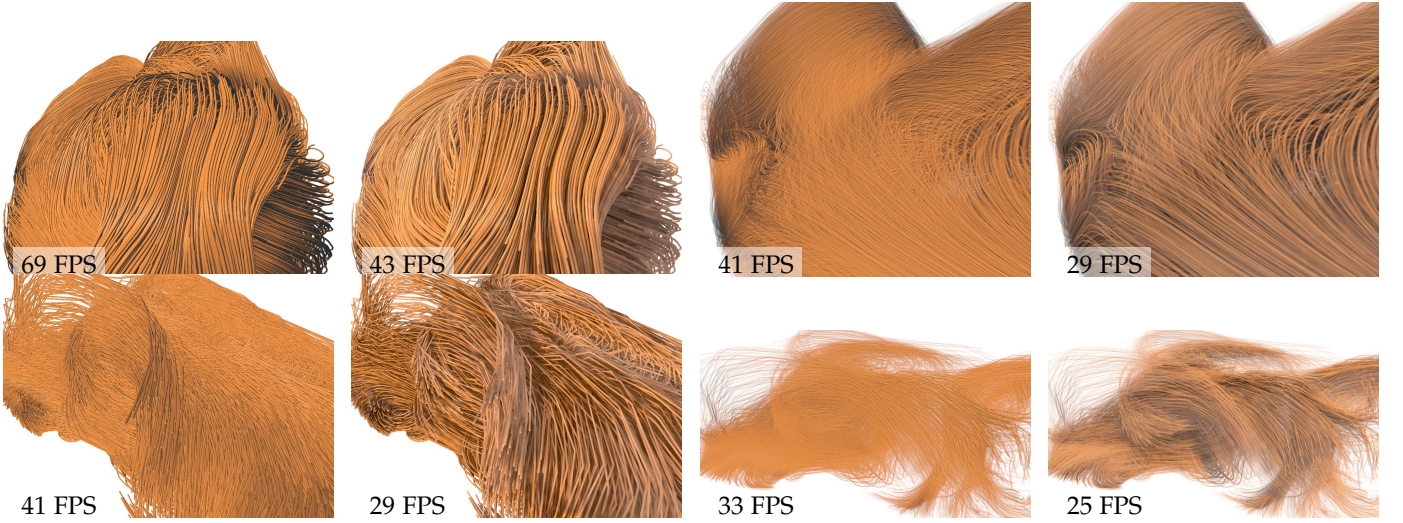
41 FPS

29 FPS

41 FPS

29 FPS

33 FPS

25 FPS

Fig. 19: Lines are rendered with local illumination (left) and soft shadows in combination with ambient occlusions (right). When rendering huge amounts of semi-transparent curves, single curves cannot be distinguished any longer. Ambient occlusions help to distinguish individual curves, since less light is reaching occluded curves. Shadow effects enhance the overall structure of the dataset and indicate the relative position of curves to each other. In combination with ambient occlusions, details in shadowed regions are preserved.
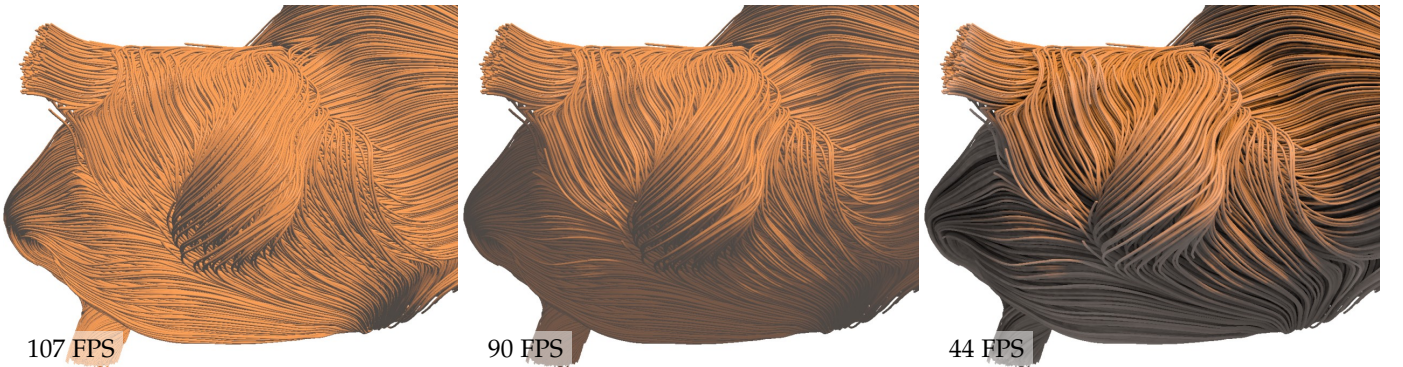
107 FPS

90 FPS

44 FPS

Fig. 20: First: Local illumination. Second: Soft shadows from line density values via cone-tracing. Third: Soft shadows in combination with ambient occlusion using 50 sample rays per voxel. The Aneurysm II dataset is shown, while continuously updating corresponding illumination values.
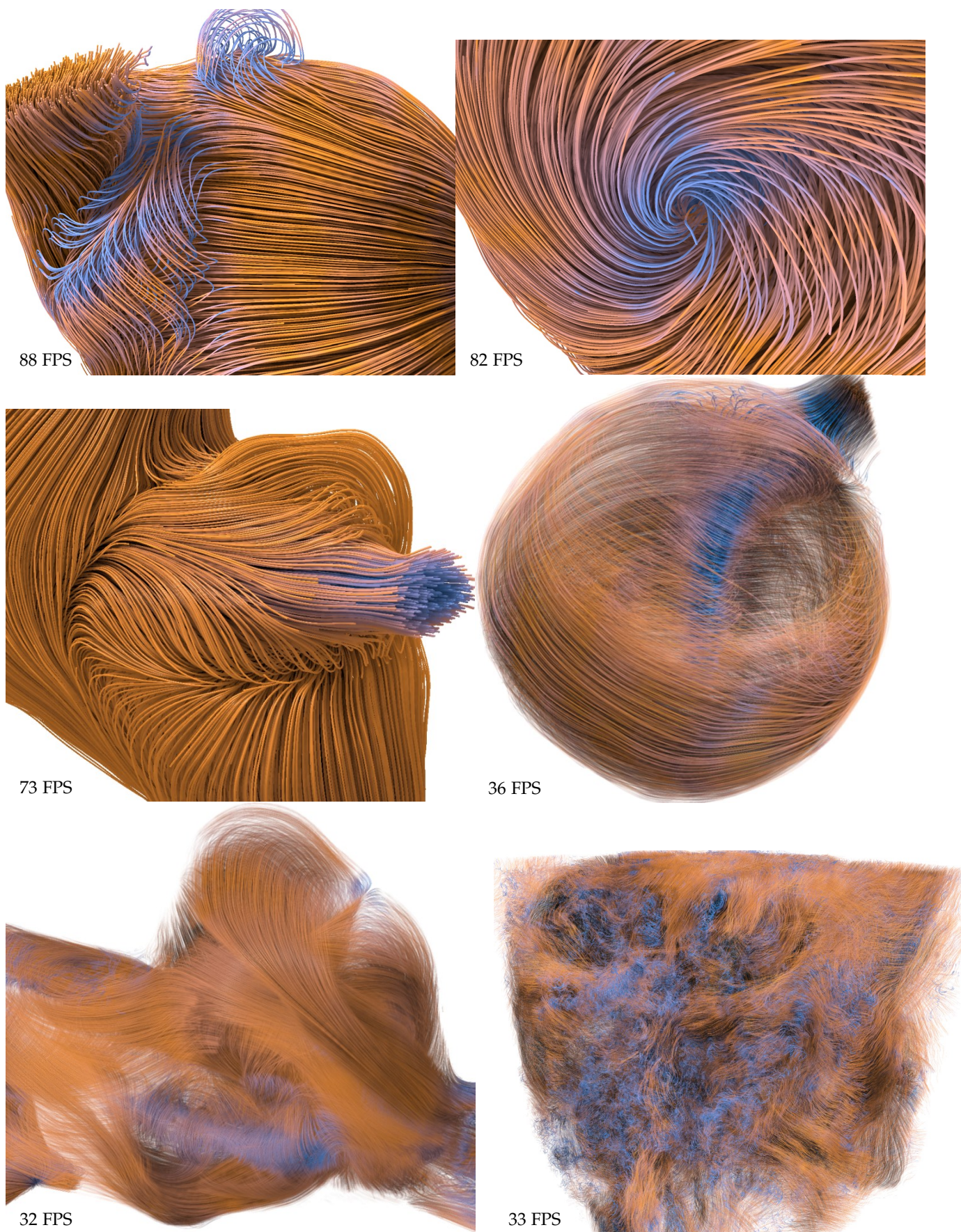
Fig. 21: Renderings of Aneurysm I (top and middle right), Aneurysm II (middle right and bottom left), and Turbulence (bottom right). Soft shadows and ambient occlusions are enhancing the depth perception. Semi opaque renderings are revealing inner structures.

TABLE 2: Performance statistics for different datasets. Rendering performance was evaluated using a constant per-line opacity value of 25 percent and 100 percent, respectively.

Dataset	Timings Semi Opaque		Timings Opaque	
	Linked List	Raycasting	Rasterization	Raycasting
Tornado	4 ms/250 fps	15 ms/67 fps	2 ms/500 fps	12 ms/83 fps
Aneurysm I	46 ms/22 fps	25 ms/40 fps	8 ms/125 fps	9 ms/111 fps
Aneurysm II	160 ms/6 fps	27 ms/37 fps	14 ms/71 fps	12 ms/83 fps
Turbulence	380 ms/3 fps	24 ms/42 fps	65 ms/15 fps	6 ms/167 fps
Forecast	overflow	23 ms/43 fps	29 ms/34 fps	4 ms/250 fps

REFERENCES

[1] N. Thibieroz and H. Gruen, "Oit and indirect illumination using dx11 linked lists," in *Game Developers Conference*, 2010.

[2] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 2009, pp. 15–22.

[3] S. Laine and T. Karras, "Efficient sparse voxel octrees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1048–1059, 2011.

[4] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing: A preview," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 2011, pp. 207–207.

[5] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller, "Voxel-based global illumination," in *Symposium on Interactive 3D Graphics and Games*, 2011, pp. 103–110.

[6] D. Cohen-Or and A. Kaufman, "3d line voxelization and connectivity control," *IEEE Comput. Graph. Appl.*, vol. 17, no. 6, pp. 80–87, 1997.

[7] M. Zöckler, D. Stalling, and H.-C. Hege, "Interactive visualization of 3d-vector fields using illuminated stream lines," in *Proceedings of the 7th Conference on Visualization '96*, ser. VIS '96, 1996, pp. 107–ff.

[8] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo, "Illuminated lines revisited," in *VIS 05. IEEE Visualization, 2005.*, 2005, pp. 19–26.

[9] C. Stoll, S. Gumhold, and H. P. Seidel, "Visualization with stylized line primitives," in *IEEE Visualization*, 2005.

[10] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg, "Depth-dependent halos: Illustrative rendering of dense line data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1299–1306, 2009.

[11] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg, "Interactive illustrative line styles and line style transfer functions for flow visualization," *CoRR*, vol. abs/1503.05787, 2015.

[12] S. W. Park, H. Yu, I. Hotz, O. Kreylos, L. Linsen, and B. Hamann, "Structure-accentuating dense flow visualization," in *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Conference on Visualization*, 2006, pp. 163–170.

[13] A. Kuhn, N. Lindow, T. Günther, A. Wiebel, H. Theisel, and H.-C. Hege, "Trajectory density projection for vector field visualization," in *EuroVis - Short Papers 2013*, 2013, pp. 31–35.

[14] T. Günther, C. Rössl, and H. Theisel, "Opacity optimization for 3d line fields," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 120:1–120:8, 2013.

[15] O. Mishchenko and R. Crawfis, "On perception of semi-transparent streamlines for three-dimensional flow visualization," *Computer Graphics Forum*, vol. 33, no. 1, pp. 210–221, 2014.

[16] C. Everitt and L. Williams, "Interactive order-independent transparency," in *White paper available online at http://developer.nvidia.com*, 2001.

[17] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," in *Proceedings of the 21st Eurographics Conference on Rendering*, ser. EGSR'10, 2010, pp. 1297–1304.

[18] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Efficient depth peeling via bucket sort," in *Proceedings of the Conference on High Performance Graphics 2009*, 2009, pp. 51–57.

[19] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1036–1047, 2011.

[20] G. Schussman and K.-L. Ma, "Anisotropic volume rendering for extremely dense, thin line data," in *Proceedings of the Conference on Visualization '04*, ser. VIS '04, 2004, pp. 107–114.

[21] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," *ACM Transactions on Graphics*, pp. 485–493, 2006.

[22] S. Woop, G. Marmitt, and P. Slusallek, "B-KD Trees for hardware accelerated ray tracing of dynamic scenes," in *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. New York, NY, USA: ACM, 2006, pp. 67–77.

[23] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, and P. Navratil, "Ospray - a cpu ray tracing framework for scientific visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 931–940, 2017.

[24] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proc. High-Performance Graphics 2009*, 2009, pp. 145–149.

[25] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 55–63.

[26] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 66:1–66:13, 2010.

[27] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka, "Cpu ray tracing large particle data with balanced p-k-d trees." in *IEEE SciVis*, 2015.

[28] J. Staib, S. Grottel, and S. Gumhold, "Visualization of particle-based data with transparency and ambient occlusion," *Computer Graphics Forum*, vol. 34, no. 3, pp. 151–160, 2015.

[29] A. Kaufman, D. Cohen, and R. Yagel, "Volume graphics," *Computer*, vol. 26, no. 7, pp. 51–64, Jul. 1993.

[30] G. T. Herman and H. K. Liu, "Three-dimensional display of human organs from computed tomograms," *Computer Graphics and Image Processing*, vol. 9, no. 1, pp. 1–21, 1979.

[31] F. Reichl, M. G. Chajdas, K. Bürger, and R. Westermann, "Hybrid Sample-based Surface Rendering," in *Vision, Modeling and Visualization*, 2013, pp. 47–54.

[32] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, vol. 30, no. 7, 2011.

[33] D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski, "A survey of volumetric illumination techniques for interactive volume rendering," *Computer Graphics Forum*, vol. 33, no. 1, pp. 27–51, 2014.

[34] H. Freeman and J. M. Glass, "On the quantization of line-drawing data," *IEEE Transactions on Systems Science and Cybernetics*, vol. 5, no. 1, pp. 70–79, 1969.

[35] E. Gobbetti and F. Marton, "Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 878–885, 2005.

[36] J. Danskin and P. Hanrahan, "Fast algorithms for volume ray tracing," in *Proceedings of the 1992 Workshop on Volume Visualization*, 1992, pp. 91–98.

[37] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion," in *ACM SIGGRAPH 2008 Talks*, 2008, pp. 22:1–22:1.

[38] G. Byrne, F. Mut, and J. Cebral, "Quantifying the large-scale hemodynamics of intracranial aneurysms," *Amer. J. of Neuroradiology*, vol. 35, pp. 333–338, 2014.

[39] H. Aluie, G. Eyink, V. E., S. C. K. Kanov, R. Burns, C. Meneveau, and A. Szalay, "Forced MHD turbulence data set," http://turbulence.pha.jhu.edu/docs/README-MHD.pdf, 2013 (accessed March 22, 2017).

[40] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 66:1–66:13, 2010.

[41] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics*, 2009, pp. 7–13.

Mathias Kanzler is a PhD candidate in the Computer Graphics and Visualization Group at the Technical University of Munich (TUM). He received the M.Sc. in computer science from TUM in 2015. His research interests include visualization and real-time rendering.

Marc Rautenhaus is a postdoctoral researcher in the Computer Graphics and Visualization Group at the TUM. He received the M.Sc. in atmospheric science from the University of British Columbia, Vancouver, in 2007, and the Ph.D. in computer science from TUM in 2015. Prior to joining TUM, Marc worked as a research associate at the German Aerospace Center's Institute for Atmospheric Physics. His research interests focus on the intersection of visualization and meteorology.

Rüdiger Westermann studied computer science at the Technical University Darmstadt and received the Ph.D. in computer science from the University of Dortmund, both in Germany. In 2002, he was appointed the chair of Computer Graphics and Visualization at TUM. His research interests include scalable data visualization and simulation algorithms, GPU computing, real-time rendering of large data, and uncertainty visualization.