

Real-Time Rendering of Dynamic Line Sets using Voxel Ray Tracing

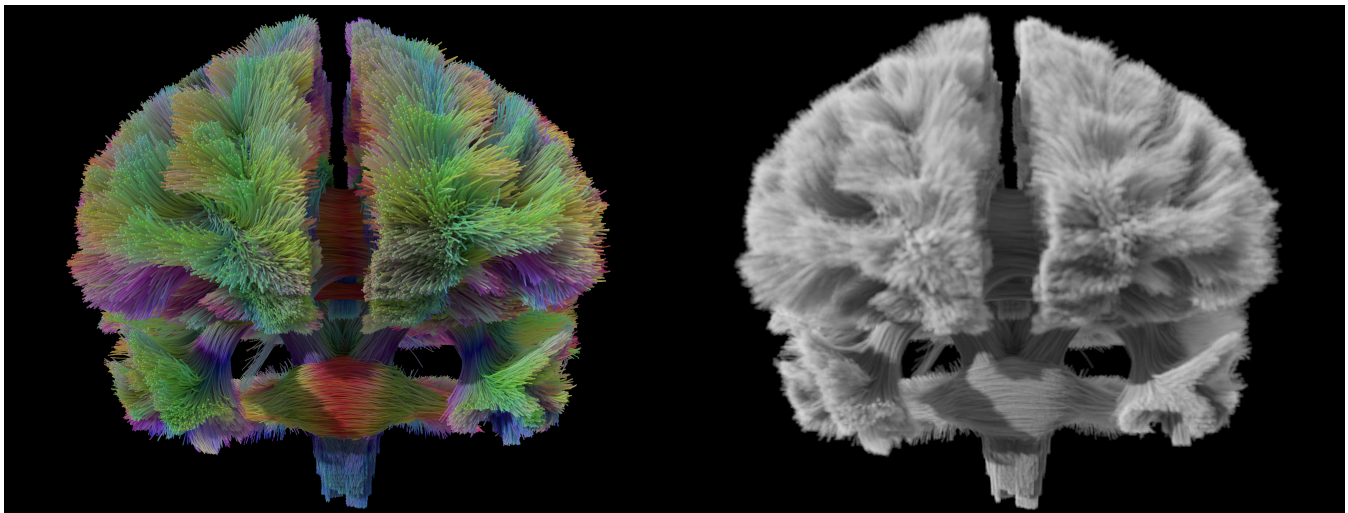
B. Kraaijeveld¹ , A.C. Jalba¹ , A. Vilanova¹  & M. Chamberland¹ ¹Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

Figure 1: Renderings of the Bundles Large tractography line set. Left: a voxel-based ray-traced rendering with lines colored by their tangent direction as is customary in tractography. Right: a volume render of the per-voxel occupancy as produced by our voxelization method. Both renderings are shaded using voxel-cone-traced ambient occlusion and directional shadows.

Abstract

Real-time rendering of dynamic line sets is relevant in many visualization tasks, including unsteady flow visualization and interactive white matter reconstruction from Magnetic Resonance Imaging. High-quality global illumination and transparency are important for conveying the spatial structure of dense line sets, yet remain difficult to achieve at interactive rates. We propose an efficient voxel-based ray-tracing framework for rendering large dynamic line sets with ambient occlusion and ground-truth transparency. We introduce a voxelization algorithm that supports efficient on-the-fly construction of acceleration structures for both voxel cone tracing and ray tracing. To further reduce per-frame preprocessing cost, we propose a voxel-based culling method that restricts acceleration structure construction to camera-visible voxels. Together, these contributions enable real-time rendering of large-scale dynamic line sets with high quality and physically accurate transparency. We demonstrate that our method outperforms the state of the art in quality and performance when rendering (semi-)opaque dynamic line sets.

CCS Concepts

- **Computing methodologies** → *Visibility; Ray tracing; Rasterization;*
- **Human-centered computing** → *Scientific visualization;*

1. Introduction

Real-time rendering of 3D line sets has many applications in scientific visualization and in the game industry. While some line sets are inherently static, others are dynamic: they vary over time as a result of a simulation, animation, or user interaction. Visualizing dynamic line sets is relevant when dealing with time-varying flow data [MLP*10], particle trajectories, hair and fur simulation [WBK*07],

and when supporting user interactions such as segmenting diffusion MRI whole-brain tractograms into white matter bundles [ASH*09; SSA*08] or real-time tractography [CWM*14]. Global illumination effects, such as ambient occlusion and soft shadows, facilitate depth and shape perception [DRN*17], while transparency helps reveal otherwise occluded structures [KNM*21].

While global illumination effects can be approximated effi-

ciently and accurately using *voxel cone tracing* [CNS*11; SGG15; HGVV16; KRW19; GG21], transparency remains a key challenge when rendering large, dense line sets in real-time. Accurate transparency requires compositing all geometry in visibility order, which is computationally expensive for dense line sets. Two key strategies are commonly used to resolve transparency: *Object-order* rendering techniques process geometry element by element and rely on various techniques to capture and sort all geometry per pixel. Groß and Gumhold [GG21] present an object-order method that renders transparent line sets by sorting all geometry in visibility order every frame. However, due to the sorting overhead, their method is less efficient for (semi-)opaque datasets. *Image-order* rendering methods precompute various acceleration structures and resolve transparency by tracing rays from the camera's perspective. Kanzler et al. [KRW19] present a voxel-based quantized line representation that allows for efficient ray-traced transparency, but relies on a pre-computed acceleration structure. While ray tracing more naturally handles transparency, there is little to no opportunity to precompute these acceleration structures in the context of dynamic line sets.

In this work, we present an efficient rendering pipeline for high-quality ray tracing of dynamic line sets that computes all acceleration structures on the fly. We achieve real-time performance by leveraging a fast voxelization scheme, which we use to accelerate the remainder of the rendering pipeline. Based on this voxelization, we introduce a voxel-based culling method to construct a ray-tracing acceleration structure on the fly only for voxels visible to the camera. Finally, we use these acceleration structures to render large line sets with voxel-based shading and ground-truth transparency. While we present a pipeline to accelerate rendering of dynamic line sets, our pipeline could be extended to render almost any geometric primitive. To summarize, our main contribution is a high-performance voxel-based ray tracing pipeline, which is made possible by:

- A GPU-based method for efficient and high-quality conservative voxelization of capsule primitives.
- A voxel-based culling approach that only constructs a ray-tracing acceleration structure for voxels visible to the camera, saving memory and increasing performance.
- A voxel-based ray-tracing technique that renders ground-truth transparency with high performance for (semi-)opaque line sets.

2. Related work

Our work is closely related to prior research line set rendering, voxelization, and transparency techniques.

2.1. Rendering of Line Sets

In object-order rendering, accurate transparency is achieved by compositing geometry in visibility order using order-dependent and order-independent techniques. Depth peeling [Eve01] is an order-independent technique that resolves transparency by rasterizing all geometry as many times as the maximum depth complexity. However, since a single rasterization pass is already computationally expensive for dense line sets, depth peeling has poor performance. Staib et al. [SGG15] achieve transparency in a single rasterization pass by sorting all geometry in visibility order, which is sufficient when rendering small spherical particles. However, when rendering

line segments, transparency has to be resolved for each individual pixel. Groß and Gumhold [GG21] combine order-dependent and order-independent transparency techniques to render accurate transparency for line segments in a single rasterization pass. While object-order methods can achieve accurate transparency and support dynamic data, sorting and other transparency techniques incur a substantial performance overhead compared to image-order methods.

Image-order methods are a more natural fit for rendering transparency, but often rely on precomputed acceleration structures, which limits their use to static datasets. Schussman and Ma [SM04] precompute a per-voxel spherical harmonics representation that allows for efficient rendering of large line sets. Kanzler et al. [KRW19] precompute a voxel-based line segment quantization and a level-of-detail hierarchy, enabling efficient rendering of large transparent line sets at the cost of accuracy. McGraw [McG20] precomputes a sparse voxel octree to accelerate ray marching of high-quality interpolated line sets with ray-marched shadows. Wald et al. [WMZ*20] repurposed GPU ray-tracing hardware to accelerate the ray tracing of line sets based on a bounding volume hierarchy of oriented bounding boxes. Our method is most similar to that of Kanzler et al. [KRW19] but supports dynamic datasets by constructing all acceleration structures on the fly. Furthermore, we improve the rendering quality and support ground truth transparency, while being as efficient in rendering dynamic (semi-)opaque line sets.

Global illumination for line sets is tackled using, e.g., ray tracing [WMZ*20], sparse voxel octrees [McG20], or screen-space techniques [EHS13]. Voxel cone tracing [CNS*11] is particularly well suited for rendering soft shadows and ambient occlusion for dense line sets [KRW19; GG21] and molecular datasets [SGG15; HGVV16]. Voxel cone tracing provides great performance, while delivering high-quality low-frequency shading, which improves visual clarity over high-frequency shading techniques [KRW19]. In our work, we repurpose the voxel cone tracing acceleration structure to accelerate rendering by employing voxel-based primitive culling, texture-space shading, and empty-space skipping.

2.2. Voxelization of Line Sets

Voxel cone tracing relies on an acceleration structure called an *occupancy pyramid* [HGVV16]: a 3D texture hierarchy where each voxel encodes the fraction of its volume occupied by geometry. In prior work, the occupancy pyramid is often constructed using voxelization techniques [SGG15; HGVV16; KRW19; GG21].

Voxelization of line segments has been a longstanding topic in computer graphics. An early influential algorithm is Bresenham's line drawing algorithm [Bre65], which may be extended to 3D for voxelization. While Bresenham's algorithm produces a single-pixel-width line, it is not conservative, i.e., not all pixels/voxels intersected by the line are recognized. Xiaolin Wu proposed a method to rasterize anti-aliased lines [Wu91], removing the staircasing effect at the cost of being more expensive than Bresenham. Amanatides & Woo's Digital Differential Analyzer (DDA) algorithm [AW87] provides a conservative voxelization for infinitely thin line segments and is widely used as a ray-voxel traversal algorithm. However, none of these algorithms guarantee a conservative voxelization of a line segment with a radius, limiting their applicability for constructing ray-tracing acceleration structures.

Voxelization of line segments can be used to generate an occupancy pyramid to enable real-time voxel cone tracing, as is shown by Groß and Gumhold [GG21] and Kanzler et al. [KRW19], who respectively use rounded cone and cylinder primitives during rendering, but infinitely thin lines during voxelization. Both demonstrate how non-conservative voxelization of line sets can generate high-quality occupancy pyramids. Kanzler et al. [KRW19], however, show non-conservative voxelization leads to missing-segment artifacts when used as the basis for ray tracing. While they address these artifacts by including all voxel neighbors in ray-tracing intersection tests, this approach adds a significant performance overhead. We demonstrate how our conservative capsule voxelization addresses these artifacts, while maintaining high ray-tracing performance.

Molecular rendering methods generate an occupancy pyramid every frame to enable real-time voxel cone tracing for dynamic data [SGG15; HGVV16]. Staib et al. [SGG15] voxelize spherical particles by rendering planar slices to a 3D texture, using multi-sampling to create a conservative, anti-aliased voxelization. Herosilla et al. [HGVV16] voxelize molecules that consist of atoms and bonds, modeled by sphere and cylinder primitives, respectively. They estimate occupancy by evaluating the primitive's signed distance function at the center of each voxel within the primitive's *Axis-Aligned Bounding Box* (AABB). Like Herosilla et al. [HGVV16] and Hsieh et al. [HCTS10], we use signed distance functions to estimate occupancy. However, since our line sets contain line segments that may be long, thin, and not axis-aligned, it can be expensive to evaluate all voxels within the primitive's AABB. To improve performance for conservative voxelization of line primitives, we present a voxel traversal algorithm that scales better with line segment length.

2.3. Order-Independent Transparency

A-Buffer techniques [Car84] improve performance over depth peeling by capturing multiple fragments per pixel in a single rasterization pass. A key challenge in A-Buffer techniques is the unknown number of fragments per pixel, which results in unbounded memory cost and indeterminate fragment write locations. Per-Pixel Linked Lists [YHGT10] capture all fragments in a single rendering pass but introduce significant memory overhead due to storing a node pointer with every fragment. Fragment Pages reduce this overhead by providing GPU support for unrolled linked lists [Cra10]. The Linearized Layered Fragment Buffer [KLZ12] instead captures all fragments in two rendering passes. The first pass computes per-pixel fragment counts, which are converted into per-pixel memory offsets using a parallel prefix sum [HG12]. A secondary rendering pass then captures all fragments into contiguous memory. Because these A-Buffer methods capture all fragments per pixel, they introduce significant memory and bandwidth costs for scenes with high depth complexity. Furthermore, while these methods capture all fragments, they still need to be sorted to produce correct transparency. Because generally more fragments are generated than there are primitives, sorting fragments is more expensive than sorting primitives. In our work, we extend the Linearized Layered Fragment Buffer [KLZ12] for the voxel domain with a two-pass voxelization scheme to capture per-voxel fragment lists. By only capturing per-voxel fragment lists for the voxels visible to the camera, our method saves memory and improves performance when rendering dynamic line sets.

3. Voxel-Based Ray Tracing of Dynamic Line Sets

In this section, we describe our rendering pipeline for real-time visualization of dynamic line sets with ambient occlusion and transparency. An overview of the pipeline is provided in Figure 2.

We voxelize all line segments every frame to compute the *occupancy pyramid* and *per-voxel primitive counts*, see Section 3.1. Based on this voxelization, we use voxel-based culling to determine the camera-visible voxels, see Section 3.2. Then, we compute *per-voxel fragment lists* for just the camera-visible voxels, described in Section 3.3. We compute shading in voxel space using voxel cone tracing, see Section 3.4. Finally, we ray trace line sets with ground truth transparency and ambient occlusion, see Section 3.5.

3.1. Voxelization

To support real-time voxel cone tracing for dynamic line sets, we voxelize the entire line set every frame to generate the *occupancy pyramid*. To enable ray tracing, we use conservative voxelization to capture all line-voxel intersections. Each intersection contributes to the *per-voxel primitive count* and the estimated voxel *occupancy*.

During voxelization, we represent each line segment as a Quilez-style capsule primitive [Qui08], defined by vertices \mathbf{v}_0 and \mathbf{v}_1 and radius r . Capsule primitives connect adjacent line segments without leaving gaps but constrain each polyline to have a uniform radius. We traverse the axis corresponding to the largest-magnitude component— x , y or z —of the line delta $\mathbf{d} = \mathbf{v}_1 - \mathbf{v}_0$. At each step along the major axis, we visit the voxels along the remaining two (minor) axes. Unlike a digital differential analyzer (DDA), our method ensures conservative voxelization for capsules, while visiting fewer voxels than axis-aligned bounding box (AABB) voxelization. Figure 3 shows an illustration of our voxel traversal algorithm.

We sort the components of the line delta \mathbf{d} by absolute magnitude to obtain the axis index vector $\mathbf{a} = (a_0, a_1, a_2)$, where a_0 denotes the major axis and a_1, a_2 the minor axes. To simplify traversal, we swap vertices \mathbf{v}_0 and \mathbf{v}_1 if the major axis has a negative direction. By normalizing \mathbf{d} relative to its largest vector component d_{a_0} , we obtain a step vector $\mathbf{s} = \mathbf{d}/d_{a_0}$ as the vector from one major axis voxel boundary to the next. To account for the capsule end caps, we extend the line segment in both directions with $\mathbf{s} \cdot r$ to obtain the extended vertices $\hat{\mathbf{v}}_0$ and $\hat{\mathbf{v}}_1$. Finally, we compute the projected capsule radii \hat{r}_1 and \hat{r}_2 along the minor axes a_1 and a_2 using:

$$\hat{r}_i = \frac{r}{\sqrt{1 - (d_{a_i}/\|\mathbf{d}\|)^2}} \quad \text{where } i \in 1, 2. \quad (1)$$

We traverse along the major axis a_0 , from extended vertex $\hat{\mathbf{v}}_0$ to $\hat{\mathbf{v}}_1$, and visit every voxel boundary in between. For each consecutive pair of voxel intersection points along the major axis a_0 , defined as \mathbf{p}_n and \mathbf{p}_{n+1} , we compute a 2D bounding box in the minor axes a_1, a_2 using radii \hat{r}_1, \hat{r}_2 . We visit all voxels that overlap this bounding box. The voxel traversal pseudocode is provided in Algorithm 1.

While AABB voxelization exhibits linear, quadratic, or cubic time complexity depending on line orientation, like DDA, our capsule voxelization scales linearly with line segment length, regardless of line orientation. Because most line sets contain a wide range of line

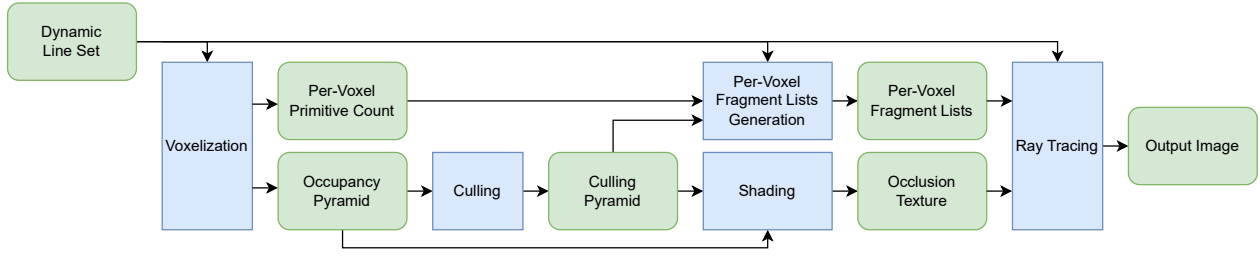


Figure 2: Schematic overview of our rendering pipeline. The stages of our rendering pipeline are represented by the blue blocks and are described in Sections 3.1 to 3.5. Green blocks represent data structures, while arrows show the inputs and outputs of each pipeline stage.

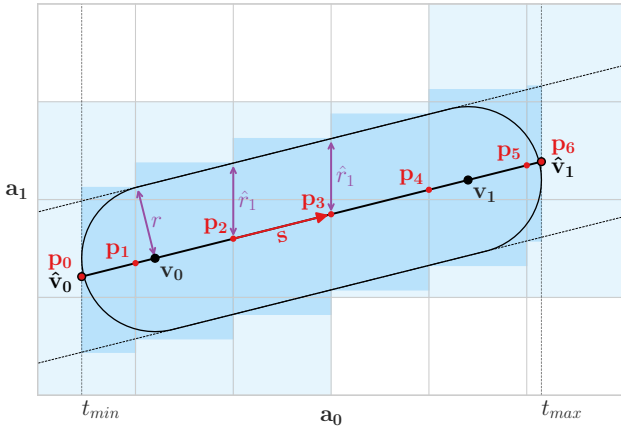


Figure 3: 2D Illustration of our voxel traversal scheme. Our algorithm traverses axis a_0 , visiting points \mathbf{p}_0 – \mathbf{p}_6 . For each consecutive pair of points, a bounding box (dark blue) is computed along the minor axes a_1, a_2 using the projected radii \hat{r}_1, \hat{r}_2 . All intersected voxels (light blue) are voxelized. See the text for details.

orientations, our method is expected to outperform AABB when voxelizing longer line segments.

To compute the base level of the occupancy pyramid, we iterate over all capsule-voxel intersections using our voxel traversal algorithm. As no closed-form expression for the capsule-voxel intersection volume exists, we approximate occupancy using the primitive's signed distance function, following Hermosilla et al. [HG16] and Hsieh et al. [HCTS10]. However, when representing line segments as capsules, the endpoints of adjacent capsules overlap. This results in spherical artifacts at each interior vertex, which lead to an over-estimation of occupancy, particularly for densely packed vertices. To prevent overlapping geometry, we follow Groß and Gumhold [GG21] in defining a clipping plane normal \mathbf{n}_i as the tangent at \mathbf{v}_i computed via central differences. In Algorithm 2, we extend the Quilez capsule SDF [Qui08] to define a clipped capsule SDF. This SDF computes the signed distance from point \mathbf{p} to a clipped capsule with vertices $\mathbf{v}_0, \mathbf{v}_1$, clipping planes $\mathbf{n}_0, \mathbf{n}_1$ and radius r . The signed-distance-based occupancy estimation is illustrated in Figure 4.

The clipped capsule signed distance function provides a good occupancy approximation for capsules with large radii relative to

Algorithm 1: Conservative Capsule Voxelization

```

 $t_{min} \leftarrow \hat{v}_{0,a_0}$  // major axis component of  $\hat{v}_0$ 
 $t_{max} \leftarrow \hat{v}_{1,a_0}$  // major axis component of  $\hat{v}_1$ 
 $t_0 \leftarrow t_{min}$ 
 $\mathbf{p}_0 \leftarrow \hat{v}_0$ 

while  $t_0 < t_{max}$  do
    // Compute next intersection point
     $t_1 \leftarrow \min(t_{max}, \text{floor}(t_0 + 1))$ 
     $\mathbf{p}_1 \leftarrow \hat{v}_0 + \mathbf{s}(t_1 - t_{min})$ 

    // Define 2D box to voxelize
     $j_{min} \leftarrow \text{int}(\min(p_{0,a_1}, p_{1,a_1}) - \hat{r}_1)$ 
     $j_{max} \leftarrow \text{int}(\max(p_{0,a_1}, p_{1,a_1}) + \hat{r}_1)$ 
     $k_{min} \leftarrow \text{int}(\min(p_{0,a_2}, p_{1,a_2}) - \hat{r}_2)$ 
     $k_{max} \leftarrow \text{int}(\max(p_{0,a_2}, p_{1,a_2}) + \hat{r}_2)$ 

    // Visit all voxels within 2D box
    for  $j = j_{min}$  to  $j_{max}$ ,  $k = k_{min}$  to  $k_{max}$  do
         $\mathbf{C}_{a_0} = \text{int}(t_0)$ ,  $\mathbf{C}_{a_1} = j$ ,  $\mathbf{C}_{a_2} = k$ 
        Visit( $\mathbf{C}$ )
    end

    // Move to next intersection point
     $t_0 \leftarrow t_1$ 
     $\mathbf{p}_0 \leftarrow \mathbf{p}_1$ 
end

```

Algorithm 2: Clipped Capsule Signed Distance Function

```

Function ClippedCapsuleSDF( $\mathbf{p}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{n}_0, \mathbf{n}_1, r$ ):
     $\mathbf{d} \leftarrow \mathbf{v}_1 - \mathbf{v}_0$ 
     $\mathbf{pv}_0 \leftarrow \mathbf{p} - \mathbf{v}_0$ 
     $\mathbf{pv}_1 \leftarrow \mathbf{p} - \mathbf{v}_1$ 
     $h \leftarrow \text{clamp}((\mathbf{pv}_0 \cdot \mathbf{d}) / (\mathbf{d} \cdot \mathbf{d}), 0, 1)$ 

     $sd_{f_{capsule}} \leftarrow \|\mathbf{pv}_0 - \mathbf{d} \cdot h\| - r$ 
     $sd_{f_{n_0}} \leftarrow -\mathbf{pv}_0 \cdot \mathbf{n}_0$ 
     $sd_{f_{n_1}} \leftarrow -\mathbf{pv}_1 \cdot \mathbf{n}_1$ 

    return  $\max(sd_{f_{capsule}}, sd_{f_{n_0}}, sd_{f_{n_1}})$ 

```

voxel size. However, aliasing is introduced when the capsule radius is smaller than half the voxel size. To reduce aliasing, we adapt Phone Wire Anti-Aliasing (PWAA) [Per12], which clamps the radius of thin line segments to the pixel size and fades alpha instead. We adapt PWAA to the voxel domain by clamping the clipped capsule radius to a minimum value r_{min} and correcting for the radius difference in the occupancy estimation. The occupancy estimation algorithm is shown in Algorithm 3.

Algorithm 3: Clipped Capsule Occupancy Estimation

```

Function CapsuleOccupancy( $\mathbf{p}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{n}_0, \mathbf{n}_1, r, r_{min}$ ):
     $r_{clamp} \leftarrow \max(r, r_{min})$ 
     $r_{correction} = (r/r_{clamp})^2$ 

    // Compute Occupancy from SDF
     $sdf \leftarrow \text{ClippedCapsuleSDF}(\mathbf{p}, \mathbf{v}_0, \mathbf{v}_1, \mathbf{n}_0, \mathbf{n}_1, r_{clamp})$ 
     $occupancy = \text{clamp}(0.5 - sdf, 0, 1)$ 

    return  $occupancy \cdot r_{correction}$ 

```

To construct the base level of the occupancy pyramid on the GPU, each thread voxelizes a single line segment at a time, summing per-voxel occupancy using atomic operations. To complete the occupancy pyramid, an averaging mipmap is generated. The base level of the occupancy pyramid is shown in Figure 1. To compute the first pass of the Linearized Layered Fragment Buffer [KLZ12], we compute the *per-voxel primitive count* alongside the occupancy pyramid. Since our voxelization is bandwidth bound, we pack both occupancy and primitive count into a single 32-bit integer, reserving 16 bits for each value. Only a single 32-bit atomic addition is needed to update both values for each voxel, significantly improving performance.

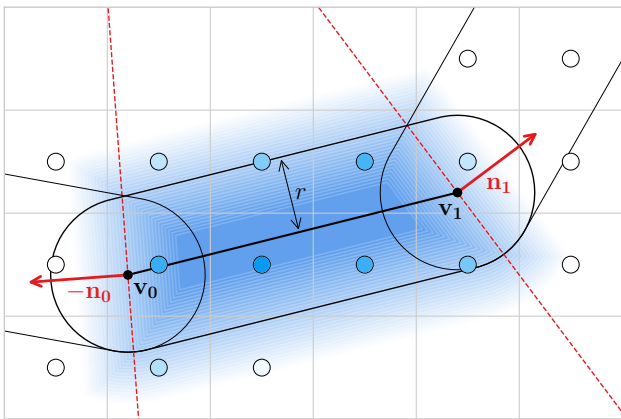


Figure 4: 2D Illustration of our SDF-based occupancy estimation. To compute the occupancy for a given capsule-voxel intersection, the clipped capsule occupancy function (Algorithm 3, visualized as a blue gradient) is sampled at the center of each intersecting voxel (filled white-blue circles). See text for more details.

3.2. Culling

We use the occupancy pyramid to cull voxels that are not visible to the camera. To determine visibility, we trace a ray from the center of each occupied voxel towards the camera and store the result in the base level of a new 3D texture hierarchy called the *culling pyramid*. We then generate a mipmap hierarchy in which each parent voxel encodes whether any of its child voxels are visible to the camera.

Determining visibility based on the occupancy pyramid can result in erroneously culled voxels, particularly for dense line sets viewed at grazing angles. These errors arise from the relatively low resolution of the occupancy pyramid, which allows rays to pass through voxels that appear occupied but are partially empty. To create a more conservative occupancy estimation, we apply morphological erosion to the base level of the occupancy pyramid by taking the minimum occupancy of each voxel's 6-neighborhood. Tracing rays through the eroded occupancy volume yields a more conservative culling pyramid and eliminates culling artifacts.

3.3. Per-Voxel Fragment Lists

Next, we create a ray-tracing acceleration structure by storing all fragments, i.e. capsule-voxel intersections, in *per-voxel fragment lists*, where each entry records the intersecting line segment index.

A common strategy for generating per-voxel fragment lists is to sort all fragments based on the Morton-encoded voxel coordinates. This *voxelize-sort-scan* strategy is used in the context of line rendering by, e.g., Kanzler et al. [KRW19] and McGraw [McG20]. However, even the fastest sorting algorithm comes with a notable performance overhead. OneSweep, a state-of-the-art parallel radix-sort algorithm, sorts n fragments using $5n$ memory operations, given 32-bit keys [AM22]. While many methods rely on sorting, fragment order within a voxel is inherently undefined, since it depends on the ray that intersects the voxel. We can therefore generate the acceleration structure using a more efficient bucket sort algorithm, where we treat each voxel as a bucket. Instead of sorting, our method extends the Linearized Layered Fragment Buffer [KLZ12] to the voxel domain. This *voxelize-scan-voxelize* strategy uses two voxelization passes, i.e. $2n$ memory operations, and one scan over all voxels to construct the per-voxel fragment lists. Because we use a two-pass voxelization scheme, we can improve performance by accelerating the secondary pass based on the culling pyramid. This results in the *voxelize-cull-scan-voxelize* strategy.

To construct the per-voxel fragment lists, we first compute a parallel prefix-sum over the per-voxel primitive count buffer for all voxels marked visible in the culling pyramid. This prefix-sum yields a *per-voxel memory offset* for all camera-visible voxels. After culling entire line segments by testing their axis-aligned bounding box against the culling pyramid, we voxelize the remaining line segments. For all capsule-voxel intersections, we sample the base level of the culling pyramid to check whether the fragment entry should be written, preventing erroneous writes to culled voxels. For all remaining capsule-voxel intersections, we atomically increment the per-voxel memory offset and write the line segment index. Since we only consider polylines that neither branch nor loop, we use primitive index i to refer to the line segment from vertex \mathbf{v}_i to vertex \mathbf{v}_{i+1} , which reduces the size of the per-voxel fragment lists.

line set	# polylines	# segments	avg. segment length	description
Bundles Small	24.000	735.080	6.17	Segmented Tractogram (Four Bundles)
Aneurysm	9.213	2.267.219	1.75	Streamlines in anterior of Aneurysm
Bundles Large	216.000	4.963.145	5.27	Segmented Tractogram (All Bundles)
Brain 200k	200.000	10.846.113	0.97	Whole Brain Tractogram
Turbulence	80.000	17.468.339	1.10	Streamlines advected in forced turbulence field
Brain 1M	1.000.000	54.240.953	0.31	Whole Brain Tractogram

Table 1: Line sets used in our experiments. For each line set, we provide their polyline count, segment count, average segment length relative to the voxel size at a resolution of 128^3 , and a short description. Renderings of these line sets can be found in [Appendix A](#).

3.4. Shading

Even though voxel cone tracing is an efficient global illumination method, evaluating it for every ray-capsule intersection becomes impractical when rendering dense line sets with transparency. Therefore, we compute and store global illumination effects in voxel space before the ray-tracing step. This voxel-based shading scheme makes evaluating global illumination inexpensive during ray tracing, requiring only a single texture sample per ray-capsule intersection.

We compute ambient occlusion by sampling the occlusion pyramid using voxel-cone-traced cones whose directions are equally distributed around the sphere. Similarly, we compute directional shadows by tracing a ray from the voxel center toward the directional light source. As an optimization, we only compute shading for voxels marked visible in the culling pyramid.

3.5. Ray-Tracing

Using the per-voxel fragment lists, we render all lines front to back using voxel-based ray tracing. We trace a single ray for every pixel and use octree traversal to efficiently identify the first occluded voxel of the culling pyramid. We then traverse the voxel grid using DDA and query each voxel’s fragment list for intersecting line segments. We use a ray-capsule intersection test [Qui13], combined with clipping planes, to compute the intersection point with the clipped capsule. To guarantee correct hit ordering, we only consider intersection points within the current voxel. Thanks to conservative voxelization, every ray passes through a voxel containing both the primitive index and intersection point of each intersecting primitive.

When rendering opaque geometry, only the closest intersection point within a voxel needs to be determined, after which we employ early ray termination and proceed to shading, making ray tracing especially effective for opaque geometry. When rendering transparent geometry, we need to capture all ray-capsule hits within a voxel in visibility order. Storing all hits in private GPU memory, however, increases register pressure and reduces performance. We therefore encode each hit in a single 32-bit integer by storing the per-voxel hit depth in the most significant 16 bits and the per-voxel fragment index in the least significant 16 bits. We obtain the closest k hits by insertion-sorting the depth-index keys in private memory as fragments arrive. Only then do we evaluate shading and blend the fragments to the screen. When a voxel contains more than k hits, we repeat the procedure, considering only fragments deeper than the k th entry from the last iteration. Using this sorting strategy, we achieve ground-truth transparency. [Figure 1](#) shows an example of a semi-transparent rendering using this technique.

4. Results & Discussion

We implemented and evaluated our methods using Rust and wgpu on an Apple M3 MacBook with an 18-core GPU and 36 GB of RAM. We have made the source code and a web demo available on <https://github.com/as-the-crow-flies/vibrant-tractography>.

We tested our method on several line sets, which are summarized in [Table 1](#). Our tractography line sets are based on diffusion MRI data from the Human Connectome Project [ESB*13]. The *Brain 200k* and *Brain 1M* line sets were obtained using MRtrix [TSR*19]. The *Bundles Small* and *Bundles Large* line sets were obtained using TractSeg [WNM18]. The *Aneurysm* and *Turbulence* flow line sets are part of the public dataset from Kern et al. [Ker20].

In [Section 4.1](#), we evaluate the quality and performance of our conservative capsule voxelization. In [Section 4.2](#), we evaluate the performance of and the effect of culling on constructing the per-voxel fragment lists. In [Section 4.3](#), we evaluate and compare our overall dynamic rendering approach with the methods of Kanzler et al. [KRW19] and Groß and Gumhold [GG21].

4.1. Voxelization

We first evaluate the performance of our *capsule* voxelization method against two commonly used methods:

- **DDA:** the Digital Differential Analyzer algorithm by Amanatides and Woo [AW87], which generates a conservative voxelization for infinitely thin lines, but is not conservative for capsule primitives. Occupancy is estimated as in Groß and Gumhold [GG21].
- **AABB:** Axis-Aligned Bounding Box voxelization following Hermosilla et al. [HGVV16], which visits all voxels within the primitive’s AABB, sampling the primitive’s signed distance function (SDF) to estimate occupancy.

To compare the performance of these three voxelization strategies, we measured the voxelization time for the six line sets at various voxel-grid resolutions. The results are shown in [Table 2](#). For all tests, we used a line segment radius of 0.2 times the voxel size. Since *capsule* voxelization is conservative and therefore visits more voxels, it is always slower than *DDA* voxelization. For very short line segments, *capsule* voxelization shows no performance advantage over *AABB* voxelization and even shows reduced performance at lower voxel resolutions. However, for the *Bundles Small & Large* line sets, which have longer line segments, a significant performance advantage is observed, especially at higher grid resolutions.

Resolution Dataset	128 ³			256 ³			512 ³		
	DDA	capsule	AABB	DDA	capsule	AABB	DDA	capsule	AABB
Bundles Small	3.38	3.61	4.72	4.70	6.20	13.6	28.8	31.6	72.4
Aneurysm	1.29	2.70	2.81	5.22	5.24	5.22	29.2	30.8	31.9
Bundles Large	3.32	6.07	10.7	6.14	12.3	36.5	44.0	51.5	179
Brain 200k	2.70	5.19	3.92	8.97	10.2	10.2	40.8	47.3	48.8
Turbulence	6.53	8.15	7.62	22.8	28.0	29.2	64.9	78.8	85.2
Brain 1M	9.28	20.0	16.7	29.1	34.9	35.6	114	146	153

Table 2: Voxelization performance, measured in milliseconds, between our capsule traversal and AABB traversal, with DDA as a reference.

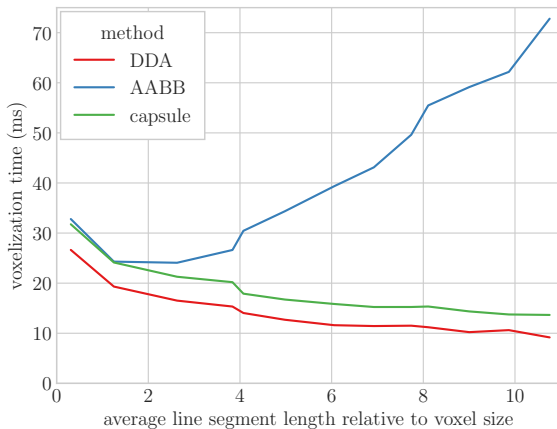


Figure 5: Voxelization performance as a function of segment length.

To further investigate the performance of the three voxelization strategies, we evaluated the effect of line segment length on performance, as shown in Figure 5. We computed various levels of detail of the high-resolution *Whole Brain 1M* dataset by connecting every n th vertex to produce level of detail n . The resulting line sets span roughly the same number of voxels but have increasing average line segment lengths. For all tests, we measured voxelization performance using a resolution of 256^3 and a line radius of 0.2 times the voxel size. Figure 5 shows the advantage of *capsule* voxelization algorithm over the *AABB* approach for longer line segments. *Capsule* voxelization adds a fixed overhead compared to *DDA* voxelization. These findings corroborate the theoretical efficiencies in Section 3.1.

Now that we have investigated the performance of our voxelization strategy, we examine the qualitative aspects of voxelization. Since *DDA* only visits voxels that are directly intersected by an infinitely thin line segment, occupancy estimation is restricted to those voxels, leading to aliasing for larger line radii. Conservative voxelization algorithms visit all voxels intersected by the capsule primitive, allowing for more accurate occupancy estimation. We evaluate how clipping planes and Phone-Wire Anti-Aliasing (PWAA) improve SDF-based occupancy estimation compared to *DDA*.

Figure 6 compares *DDA*-based occupancy and SDF-based occupancy both with and without clipping planes. The *DDA*-based occupancy, shown in Figure 6a, clearly exhibits aliasing for larger line radii. SDF-based occupancy without clipping planes, shown in Figure 6b, reduces aliasing but introduces spherical artifacts at every interior vertex. These artifacts cause an overestimation of oc-

cupancy, especially when voxelizing short segments with large radii. Our clipped capsule signed distance function, shown in Figure 6c, removes the artifacts and reduces aliasing compared to *DDA*.

Figure 7 compares *DDA*-based occupancy with SDF-based occupancy with and without Phone-Wire Anti-Aliasing (PWAA). Shown in Figure 7a, *DDA*-based occupancy estimation produces minimal aliasing for thin line segments. However, SDF-based occupancy without PWAA, shown in Figure 7b, introduces severe aliasing. PWAA, shown in Figure 7c, reduces aliasing and matches or slightly improves upon the results of *DDA*-based occupancy.

Figure 8 illustrates missing-segment artifacts during ray tracing, which occur when a ray passes through a voxel that does not contain the intersecting primitive. These artifacts appear as cubical cutouts and become more apparent with larger line radii relative to voxel size. Conservative voxelization completely eliminates these artifacts.

To conclude, these results show that the choice of voxelization algorithm depends on the scenario. *DDA* voxelization offers the best performance and shows minimal aliasing for thin line segments but is not conservative. Conservative voxelization should be used for ray tracing and for thicker line segments. As expected, *capsule* voxelization scales linearly with segment length, compared to the worst-case cubic time complexity of *AABB* voxelization.

4.2. Per-Voxel Fragment Lists

We capture per-voxel fragment lists using a two-pass voxelization scheme: *voxelize-scan-voxelize* (VSV). To improve performance for (semi-)opaque line sets, we introduced a culling step to accelerate the secondary voxelization pass: *voxelize-cull-scan-voxelize* (VCSV). We compare these two approaches to another common method: *voxelize-sort-scan* (VSS) [KRW19; McG20], using OneSweep [AM22] as a fast sorting implementation. In Table 3 we compare these strategies to compute both the occupancy pyramid and the per-voxel fragment lists, i.e., all per-frame preprocessing required for dynamic ray tracing and ambient occlusion.

As expected, *voxelize-scan-voxelize* (VSV) outperforms *voxelize-sort-scan* (VSS), while producing the same per-voxel fragment lists and occupancy pyramid. The threefold speedup aligns with the theoretical advantage of VSV. VSV requires only two passes over all fragments and one scan over all voxels, whereas VSS needs seven passes over all fragments (one write, five for sorting, one scan). While Morton-order sorting generates a cache-efficient acceleration structure, similar efficiency can be achieved by the choice of scan pass in the *voxelize-scan-voxelize* strategy.

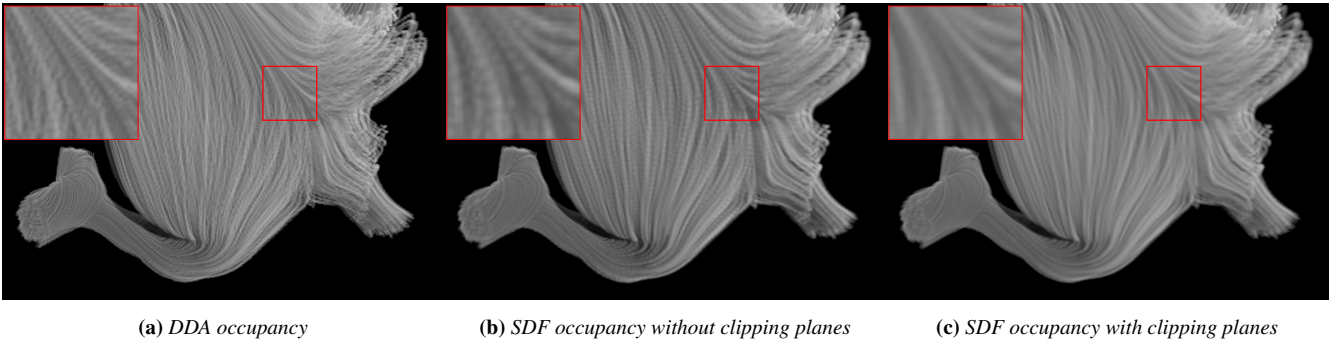


Figure 6: The effect of clipping planes on voxelization quality. Without clipping planes, spherical artifacts appear at interior vertices. Using the clipped capsule SDF removes these artifacts and significantly reduces aliasing compared to DDA voxelization.

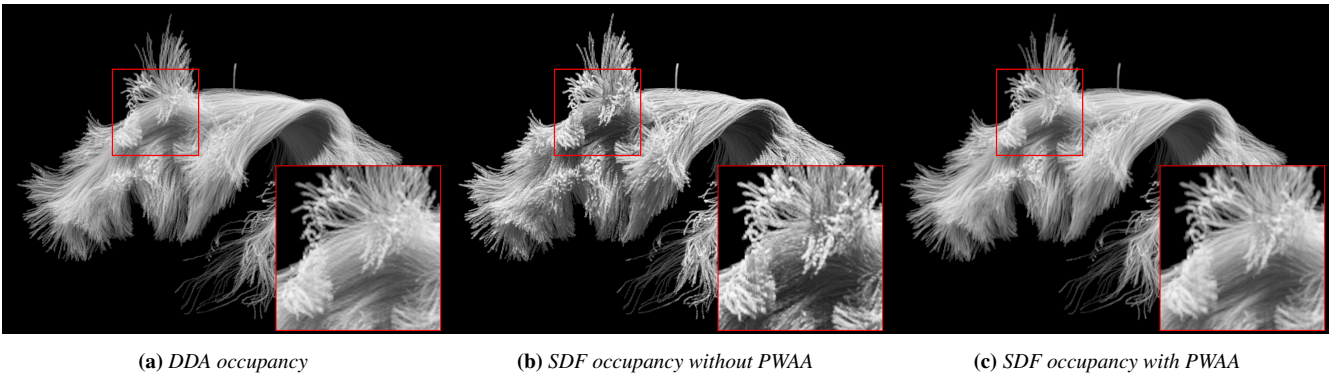


Figure 7: The effect of Phone-Wire Anti-Aliasing on voxelization quality. Without Phone-Wire Anti-Aliasing, SDF-based voxelization of thin lines introduces aliasing and an overestimation of total occupancy. Applying Phone-Wire Anti-Aliasing reduces these artifacts.

Dataset	VSS	VSV (ours)	VCSV (ours)
Bundles Small	14	5	4
Aneurysm	21	6	4
Bundles Large	75	18	12
Brain 200k	67	20	12
Turbulence	120	46	21
Brain 1M	1015	104	50

Table 3: Performance in milliseconds for generating both the occupancy pyramid and the per-voxel fragment lists at a resolution of 128^3 for opaque geometry, using: voxelize-sort-scan (VSS), voxelize-scan-voxelize (VSV) and voxelize-cull-scan-voxelize (VCSV).

Another advantage of the *voxelize-scan-voxelize* (VSV) strategy is the potential of accelerating the secondary voxelization pass. The occupancy pyramid generated in the primary voxelization pass enables view-dependent culling of the secondary voxelization pass, resulting in the *voxelize-cull-scan-voxelize* (VCSV) strategy. When rendering opaque geometry, VCSV improves performance up to two times compared with VSV. The exact gain depends on line set density and camera angle. Altogether, our method cuts down per-frame preprocessing time by a factor of three when rendering transparent data (VSV) or a factor of six for opaque data.

4.3. Rendering Pipeline

Next, we compare our dynamic rendering pipeline to the methods of Kanzler et al. [KRW19] and Groß and Gumhold [GG21], who both render line sets with ambient occlusion and transparency. We test the dynamic aspect of our method and these state-of-the-art methods by including all per-frame preprocessing costs in all performance measurements. By building all acceleration structures from scratch every frame, all rendering pipelines can handle arbitrary dynamic behavior. Since these state-of-the-art methods rely on fast sorting, we employ the OneSweep radix-sort algorithm [AM22].

Groß and Gumhold’s rasterization method [GG21] handles dynamic data, except for the construction of the occupancy pyramid. We therefore generate the occupancy pyramid every frame using fast DDA voxelization, making their method fully dynamic. Otherwise, we keep all aspects the same as in the original paper.

We use the GPU quantization method of Kanzler et al. [KRW19] to construct the occupancy pyramid and per-voxel fragment lists every frame, making their method fully dynamic. We use a quantization resolution of 64^2 , which makes each segment fit in 32 bits for optimal performance. While the authors included a level-of-detail system, we omit it in our experiments, because such a system could be implemented for both Groß and Gumhold and our method. Because it reduces performance, we exclude the authors’

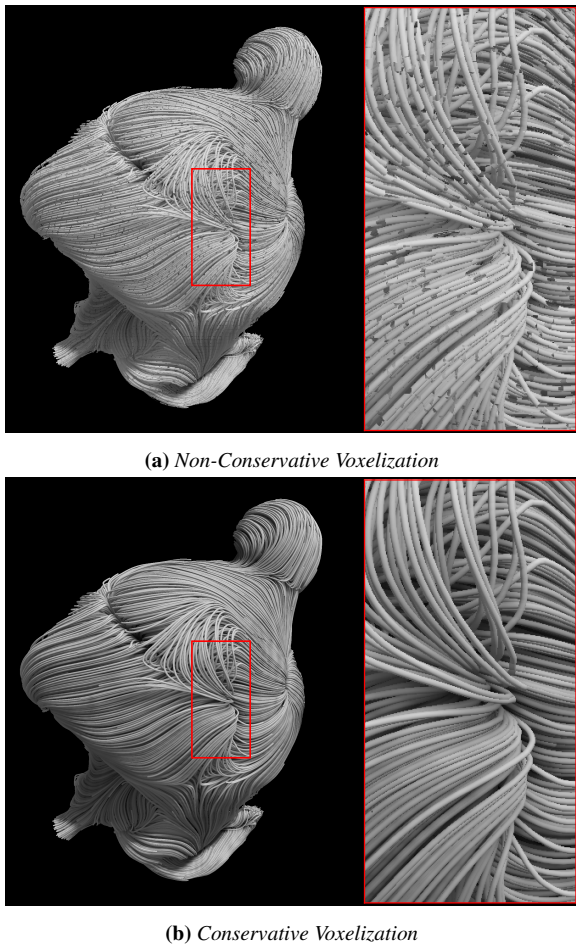


Figure 8: Comparison of ray tracing using conservative versus non-conservative voxelization for the Aneurysm dataset. Cube-like artifacts are visible when employing non-conservative voxelization. Conservative voxelization correctly renders the capsule primitives.

voxel neighborhood sampling in our performance and quality evaluation. As a performance reference, we compare these methods to an opaque rasterization *Baseline*. This baseline implements as Groß and Gumhold’s method [GG21] without sorting and culling, using DDA to compute the occupancy pyramid.

All experiments target a screen resolution of 1920×1080 . We measure the time it takes to complete the entire dynamic rendering pipeline, assuming that all acceleration structures are recomputed every frame for each method. We chose a voxel resolution of 128^3 , which provides optimal ray-tracing performance according to our experiments and does not affect line rendering quality for our method. We use a line radius of 0.2 times the voxel size.

First, we evaluate rendering quality using semi-opaque renderings of the *Bundles Small* line set, shown in Figure 9, which presents a challenge for the state of the art. All methods achieve similar overall renderings, but closer inspection reveals qualitative differences. While their method handles shorter line segments well, Groß and Gumhold’s K-Buffer-based order-correcting algorithm fails to

Dataset	Baseline	Groß	Kanzler	Ours
Bundles Small	13	12	13	10
Aneurysm	25	20	13	9
Bundles Large	83	44	39	22
Brain 200k	138	81	30	28
Turbulence	209	127	47	41
Brain 1M	822	417	119	117

Table 4: Opaque rendering performance, measured in milliseconds, including all per-frame preprocessing costs.

Dataset	Groß	Kanzler	Ours
Bundles Small	32	35	54
Aneurysm	39	30	55
Bundles Large	81	67	89
Brain 200k	127	55	117
Turbulence	175	97	199
Brain 1M	596	158	276

Table 5: Transparent rendering performance (opacity 0.1), measured in milliseconds, including all per-frame preprocessing costs.

properly sort the longer line segments in this line set. Being non-conservative, Kanzler et al.’s method shows missing-segment artifacts, which reduces the rendering quality for (semi-)opaque lines. These artifacts are less noticeable when rendering fully transparent geometry. Our voxel-based ray tracing guarantees correct depth ordering and eliminates missing-segment artifacts.

As shown in Table 4, our method achieves the highest render performance across all dynamic opaque line sets. Our method is closely followed by that of Kanzler et al.; however, their method exhibits quantization and missing-segment artifacts. Our performance gains are attributable to our proposed acceleration structure construction method, which is especially effective when rendering opaque line sets. Compared to the rasterization techniques (*Baseline* and *Groß*), the ray-tracing techniques (*Kanzler* and *Ours*) have the advantage of early ray termination and show superior performance on larger line sets. As expected, all methods outperform *Baseline* rasterization.

Table 5 shows rendering performance for transparent geometry using an opacity of 0.1. When rendering transparent geometry, Kanzler et al. show the best performance for most datasets. While our method performs well on opaque line sets, performance is significantly reduced when rendering full transparency. This performance reduction is expected, since conservative voxelization increases the number of fragments and ray-capsule intersection tests, which dominate the rendering time for fully transparent line sets. Kanzler et al.’s quantized method increases performance by limiting the number of ray-capsule intersections compared to more accurate methods.

To investigate the impact of line opacity on dynamic rendering performance, we evaluated the rendering performance of the *Aneurysm* line set at various opacities, as shown in Figure 10. This figure clearly shows the advantage of early ray-termination over the hierarchical culling strategy of Groß and Gumhold. Given that our method renders correct transparency without quantization and artifacts, it performs favorably compared to the method of Kanzler et al., even surpassing their method for very high line opacities.

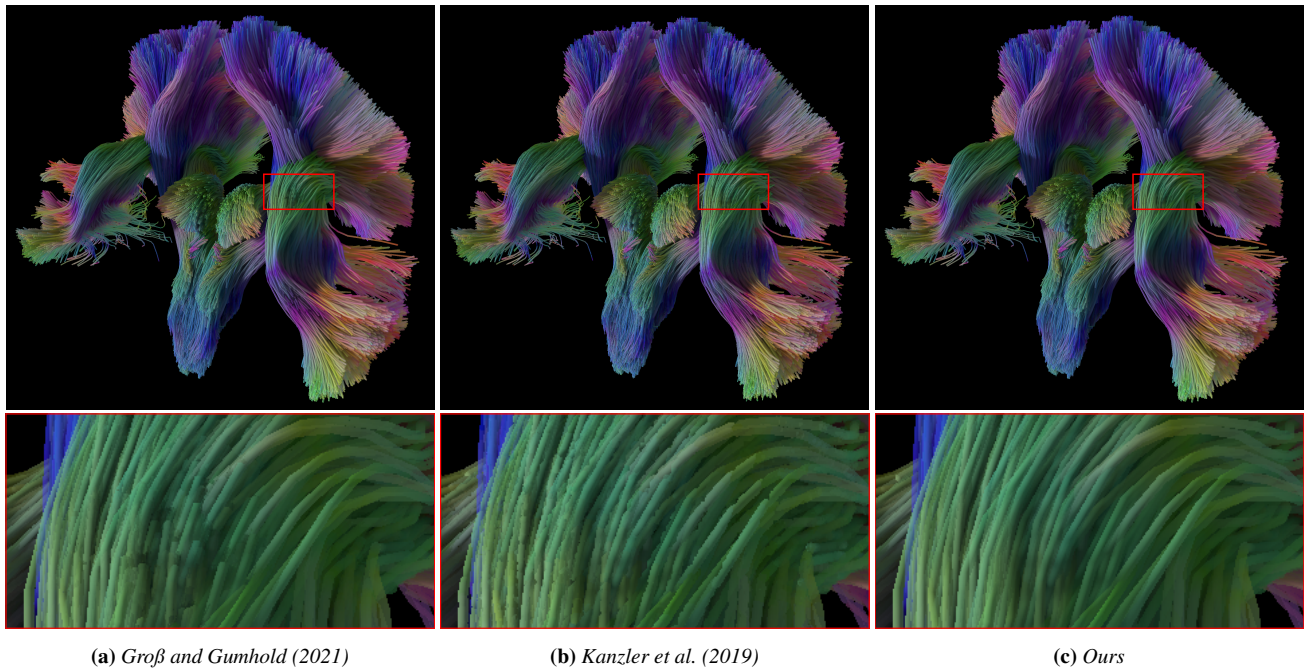


Figure 9: Qualitative comparison between rendering methods. For the Bundles Small dataset, all methods produce similar renderings. However, Groß and Gumhold’s method suffers from sorting artifacts and Kanzler et al.’s method exhibits missing-segment artifacts. Our voxel-based ray-tracing method resolves both types of artifacts.

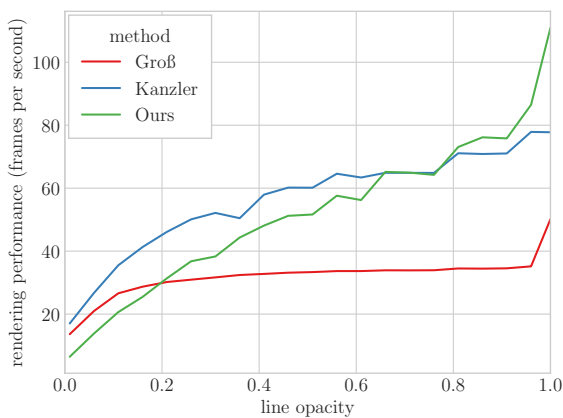


Figure 10: Rendering performance at varying line opacities for the Aneurysm line set, in frames per second, including all preprocessing.

Finally, a performance breakdown of our rendering pipeline is shown in Appendix B, reporting the performance of each step for opaque geometry. As in our other experiments, we chose a grid resolution of 128^3 , a screen resolution of 1920×1080 , and a line radius of 0.2 times the voxel size. Culling only adds a small, fixed overhead for our chosen grid resolution, since it is performed in voxel space. The per-voxel fragment list generation step remains a major contributor in the per-frame preprocessing time, even after our optimizations. We also perform shading in voxel space, which

adds a small overhead depending on the number of visible voxels. Precomputation still accounts for most of the total rendering time for opaque geometry. When rendering static line sets, only the ray-tracing step needs to be computed every frame, which results in about a twofold speedup compared to dynamic line sets.

5. Conclusion and Future Work

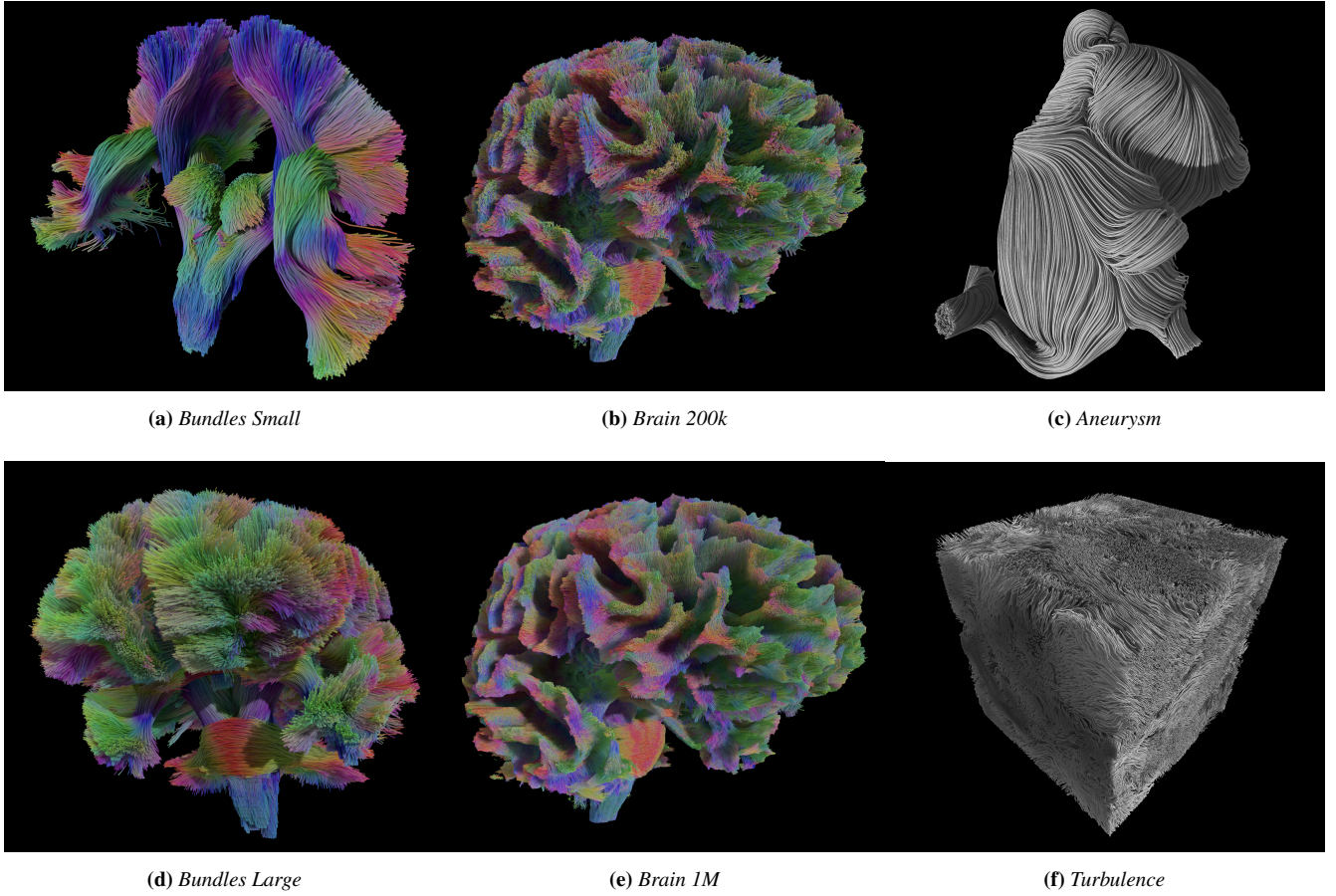
We have presented a real-time rendering pipeline for voxel-based ray tracing of dynamic line sets. Our method improves rendering quality by introducing a conservative capsule voxelization method that supports ray tracing. We improve rendering performance by using a view-dependent culling approach to efficiently compute per-voxel fragment lists. By using our voxel ray tracing technique, we render (semi-)opaque line sets with ambient occlusion and ground truth transparency in real time.

A limitation of our approach is the low rendering performance when rendering lines at low opacity values. This may be improved by adding level-of-detail rendering or by exploiting frame coherence. In the future, we will investigate how interactive diffusion MRI tractography segmentation can benefit from our rendering approach.

Acknowledgments

This publication is part of the project *VIBRANT* with file number *OCENW.M.22.352* of the research programme *Open Competition Domain Science – M*, which is (partly) financed by the Dutch Research Council (NWO).

Appendix A: Line Set Renderings



Appendix B: Performance Breakdown

Dataset	Voxelization	Culling	Per-Voxel Fragment List	Shading	Total Precompute	Ray Tracing
Bundles Small	3.6	1.3	2.7	1.3	8.9	5.1
Aneurysm	2.7	1.3	2.7	2.4	9.1	4.9
Bundles Large	6.1	1.3	6.1	3.4	16.9	7.7
Brain 200k	5.2	1.3	7.7	1.3	15.5	10.3
Turbulence	8.2	1.3	12.7	2.9	25.1	11.6
Brain 1M	20.0	1.3	30.2	2.9	54.4	32.6

Table 6: Performance, in milliseconds, for each stage of our rendering pipeline, corresponding with Sections 3.1 to 3.5, for dynamic opaque line sets. When rendering static line sets, only the ray-tracing step needs to be computed each frame.

References

- [AM22] ADINETS, A. and MERRILL, D. *Onesweep: A Faster Least Significant Digit Radix Sort for GPUs*. 2022. arXiv: [2206.01784](https://arxiv.org/abs/2206.01784) [cs.DC].
- [ASH*09] ANWANDER, A., SCHURADE, R., HLAWITSCHKA, M., SCHEUERMANN, G., and KNÖSCHE, T. “White Matter Imaging with Virtual Klingler Dissection”. *NeuroImage* 47 (2009), S105. ISSN: 1053-8119. DOI: [10.1016/S1053-8119\(09\)70916-4](https://doi.org/10.1016/S1053-8119(09)70916-4).
- [AW87] AMANATIDES, J. and WOO, A. “A Fast Voxel Traversal Algorithm for Ray Tracing”. *8th European Computer Graphics Conference and Exhibition, Eurographics 1987, Amsterdam, The Netherlands, August 24-28, 1987, Proceedings*. Ed. by MARÉCHAL, G. Geneva, Switzerland: Eurographics Association, 1987. DOI: [10.2312/EGTP.19871000](https://doi.org/10.2312/EGTP.19871000).
- [Bre65] BRESENHAM, J. “Algorithm for Computer Control of a Digital Plotter”. *IBM Syst. J.* 4.1 (1965), 25–30. DOI: [10.1147/SJ.41.0025](https://doi.org/10.1147/SJ.41.0025).
- [Car84] CARPENTER, L. C. “The A-buffer, an antialiased hidden surface method”. *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*. Ed. by CHRISTIANSEN, H. New York, NY: ACM, 1984, 103–108. DOI: [10.1145/800031.808585](https://doi.org/10.1145/800031.808585).

- [CNS*11] CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., and EISEMANN, E. "Interactive Indirect Illumination Using Voxel Cone Tracing". *Computer Graphics Forum* 30.7 (2011), 1921–1930. DOI: [10.1111/J.1467-8659.2011.02063.X](https://doi.org/10.1111/J.1467-8659.2011.02063.X).
- [Cra10] CRASSIN, C. *OpenGL 4.0+ abuffer v2.0: Linked lists of fragment pages*. 2010. URL: <https://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html> (visited on 12/18/2025).
- [CWM*14] CHAMBERLAND, M., WHITTINGSTALL, K., MATHIEU, D., FORTIN, D., and DESCOTEAUX, M. "Real-time multi-peak tractography for instantaneous connectivity display". *Frontiers in Neuroinformatics* 8 (2014), 59:1–59:15. DOI: [10.3389/FNINF.2014.00059](https://doi.org/10.3389/FNINF.2014.00059).
- [DRN*17] DÍAZ, J., ROPINSKI, T., NAVAZO, I., GOBBETTI, E., and VÁZQUEZ, P. "An experimental study on the effects of shading in 3D perception of volumetric models". *Vis. Comput.* 33.1 (2017), 47–61. DOI: [10.1007/S00371-015-1151-6](https://doi.org/10.1007/S00371-015-1151-6).
- [EHS13] EICHELBAUM, S., HLAWITSCHKA, M., and SCHEUERMANN, G. "LineAO - Improved Three-Dimensional Line Rendering". *IEEE Transactions on Visualization and Computer Graphics* 19.3 (2013), 433–445. DOI: [10.1109/TVCG.2012.142](https://doi.org/10.1109/TVCG.2012.142).
- [ESB*13] ESSEN, D. C. V., SMITH, S. M., BARCH, D. M., BEHRENS, T. E. J., YACOB, E., and UGURBIL, K. "The WU-Minn Human Connectome Project: An overview". *NeuroImage* 80 (2013), 62–79. DOI: [10.1016/J.NEUROIMAGE.2013.05.041](https://doi.org/10.1016/J.NEUROIMAGE.2013.05.041).
- [Eve01] EVERITT, C. "Interactive Order-Independent Transparency". *White paper; nVIDIA 2.6* (2001), 7. URL: <https://www.nvidia.com/en-us/drivers/Interactive-Order-Transparency/> (visited on 12/18/2025).
- [GG21] GROSS, D. and GUMHOLD, S. "Advanced Rendering of Line Data with Ambient Occlusion and Transparency". *IEEE Transactions on Visualization and Computer Graphics* 27.2 (2021), 614–624. DOI: [10.1109/TVCG.2020.3028954](https://doi.org/10.1109/TVCG.2020.3028954).
- [HCTS10] HSIEH, H., CHANG, C., TAI, W., and SHEN, H. "Novel Geometrical Voxelization Approach with Application to Streamlines". *J. Comput. Sci. Technol.* 25.5 (2010), 895–904. DOI: [10.1007/S11390-010-9374-5](https://doi.org/10.1007/S11390-010-9374-5).
- [HG12] HARRIS, M. and GARLAND, M. "Optimizing Parallel Prefix Operations for the Fermi Architecture". *GPU Computing Gems Jade Edition*. Ed. by HWU, W.-M. W. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2012, 29–38. ISBN: 978-0-12-385963-1. DOI: [10.1016/B978-0-12-385963-1.00003-4](https://doi.org/10.1016/B978-0-12-385963-1.00003-4).
- [HGVV16] HERMOSILLA, P., GUALLAR, V., VINACUA, À., and VÁZQUEZ, P. "High quality illustrative effects for molecular rendering". *Computers & Graphics* 54 (2016), 113–120. DOI: [10.1016/J.CAG.2015.07.017](https://doi.org/10.1016/J.CAG.2015.07.017).
- [Ker20] KERN, M. *Large 3D Line Sets for Opacity-Based Rendering*. Feb. 2020. DOI: [10.5281/zenodo.3637625](https://doi.org/10.5281/zenodo.3637625).
- [KLZ12] KNOWLES, P., LEACH, G., and ZAMBETTA, F. "Efficient Layered Fragment Buffer Techniques". CRC Press, July 2012, 279–292. ISBN: 978-1-4398-9376-0. DOI: [10.1201/b12288-24](https://doi.org/10.1201/b12288-24).
- [KNM*21] KERN, M., NEUHAUSER, C., MAACK, T., HAN, M., USHER, W., and WESTERMANN, R. "A Comparison of Rendering Techniques for 3D Line Sets With Transparency". *IEEE Transactions on Visualization and Computer Graphics* 27.8 (2021), 3361–3376. DOI: [10.1109/TVCG.2020.2975795](https://doi.org/10.1109/TVCG.2020.2975795).
- [KRW19] KANZLER, M., RAUTENHAUS, M., and WESTERMANN, R. "A Voxel-Based Rendering Pipeline for Large 3D Line Sets". *IEEE Transactions on Visualization and Computer Graphics* 25.7 (2019), 2378–2391. DOI: [10.1109/TVCG.2018.2834372](https://doi.org/10.1109/TVCG.2018.2834372).
- [McG20] MCGRAW, T. "High-quality real-time raycasting and raytracing of streamtubes with sparse voxel octrees". *31st IEEE Visualization Conference, IEEE VIS 2020 - Short Papers, Virtual Event, USA, October 25-30, 2020*. New York, NY: IEEE, 2020, 21–25. DOI: [10.1109/VIS47514.2020.00011](https://doi.org/10.1109/VIS47514.2020.00011).
- [MLP*10] MCLOUGHLIN, T., LARAMEE, R. S., PEIKERT, R., POST, F. H., and CHEN, M. "Over Two Decades of Integration-Based, Geometric Flow Visualization". *Computer Graphics Forum* 29.6 (2010), 1807–1829. DOI: [10.1111/J.1467-8659.2010.01650.X](https://doi.org/10.1111/J.1467-8659.2010.01650.X).
- [Per12] PERSON, E. *Phone-wire AA*. 2012. URL: <https://www.humus.name/index.php?page=3D&ID=89> (visited on 04/16/2025).
- [Qui08] QUILEZ, I. *Distance Functions*. 2008. URL: <https://iquilezles.org/articles/distfunctions/> (visited on 07/30/2025).
- [Qui13] QUILEZ, I. *Intersectors*. 2013. URL: <https://iquilezles.org/articles/intersectors/> (visited on 07/30/2025).
- [SGG15] STAIB, J., GROTTTEL, S., and GUMHOLD, S. "Visualization of Particle-based Data with Transparency and Ambient Occlusion". *Computer Graphics Forum* 34.3 (2015), 151–160. DOI: [10.1111/CGF.12627](https://doi.org/10.1111/CGF.12627).
- [SM04] SCHUSSMAN, G. L. and MA, K. "Anisotropic Volume Rendering for Extremely Dense, Thin Line Data". *15th IEEE Visualization Conference, IEEE Vis 2004, Austin, TX, USA, October 10-15, 2004, Proceedings*. Washington, DC: IEEE Computer Society, 2004, 107–114. DOI: [10.1109/VISUAL.2004.5](https://doi.org/10.1109/VISUAL.2004.5).
- [SSA*08] SCHULTZ, T., SAUBER, N., ANWANDER, A., THEISEL, H., and SEIDEL, H. "Virtual Klingler Dissection: Putting Fibers into Context". *Computer Graphics Forum* 27.3 (2008), 1063–1070. DOI: [10.1111/J.1467-8659.2008.01243.X](https://doi.org/10.1111/J.1467-8659.2008.01243.X).
- [TSR*19] TOURNIER, J., SMITH, R. E., RAFFELT, D., TABBARA, R., DHOLLANDER, T., PIETSCH, M., CHRISTIAENS, D., JEURISSEN, B., YEH, C., and CONNELLY, A. "MRtrix3: A fast, flexible and open software framework for medical image processing and visualisation". *NeuroImage* 202 (2019), 116137:1–116137:17. DOI: [10.1016/J.NEUROIMAGE.2019.116137](https://doi.org/10.1016/J.NEUROIMAGE.2019.116137).
- [WBK*07] WARD, K., BERTAILS, F., KIM, T., MARSCHNER, S. R., CANI, M., and LIN, M. C. "A Survey on Hair Modeling: Styling, Simulation, and Rendering". *IEEE Transactions on Visualization and Computer Graphics* 13.2 (2007), 213–234. DOI: [10.1109/TVCG.2007.30](https://doi.org/10.1109/TVCG.2007.30).
- [WMZ*20] WALD, I., MORRICAL, N., ZELLMANN, S., MA, L., USHER, W., HUANG, T., and PASCUCCI, V. "Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 17:1–17:16. DOI: [10.1145/3406179](https://doi.org/10.1145/3406179).
- [WNM18] WASSERTHAL, J., NEHER, P., and MAIER-HEIN, K. H. "TractSeg - Fast and accurate white matter tract segmentation". *NeuroImage* 183 (2018), 239–253. DOI: [10.1016/J.NEUROIMAGE.2018.07.070](https://doi.org/10.1016/J.NEUROIMAGE.2018.07.070).
- [Wu91] WU, X. "An efficient antialiasing technique". *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1991, Providence, RI, USA, April 27-30, 1991*. Ed. by THOMAS, J. J. New York, NY: ACM, 1991, 143–152. DOI: [10.1145/122718.122734](https://doi.org/10.1145/122718.122734).
- [YHGT10] YANG, J. C., HENSLEY, J., GRÜN, H., and THIBIEROZ, N. "Real-Time Concurrent Linked List Construction on the GPU". *Computer Graphics Forum* 29.4 (2010), 1297–1304. DOI: [10.1111/J.1467-8659.2010.01725.X](https://doi.org/10.1111/J.1467-8659.2010.01725.X).