High-quality real-time raycasting and raytracing of streamtubes with sparse voxel octrees

Tim McGraw*
Purdue University

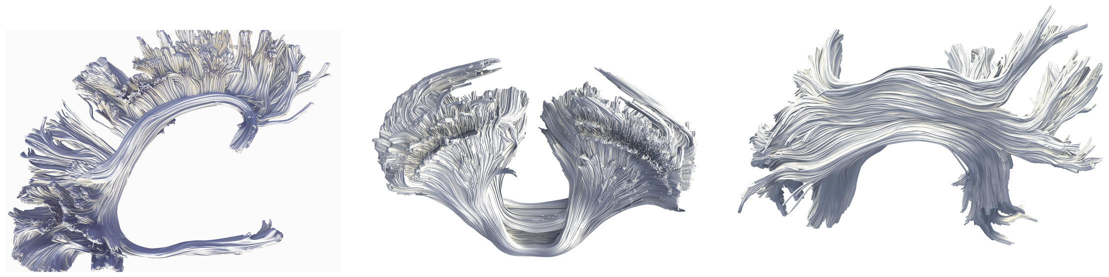

Figure 1: Neuronal fibers of the cingulum bundle (left), middle cerebellar peduncle (middle) and superior longitudinal fasciculus (right) rendered by hybrid raytracing / raycasting a sparse voxel octree.

Abstract

Voxel-based rendering and signed distance function (SDF) raycasting have been active areas of graphics research recently because they simplify high-quality graphical effects like ambient occlusion and shadowing as compared to rasterization. Much work has centered around converting triangle meshes to sparse voxel octrees (SVOs) and developing memory efficient storage schemes but little exploration of the implications to scientific visualization has taken place. In this work we explore techniques for high-performance rendering of tubes, such as streamtubes used for visualizing vector fields and fiber tracts from diffusion tensor MRI. We first present our method for generating a voxelization of the tubes, and then describe several methods for rendering: raytracing a SVO storing straight tube segments in the leaves, and hybrid raytracing/raycasting a SVO storing curved tube segments. We discuss the tradeoffs inherent in these different representations and compare the rendering techniques and results. Compared to standard graphics pipeline approaches, like using the geometry shader, we achieve joining, capping, and smooth circular cross-sections with minimal additional effort.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Scientific visualization;

1 Introduction

Rendering tube-like structures is a common task in scientific visualization. Streamtubes of vector fields, and fiber tracts computed from diffusion tensor MRI are two examples. In interactive graphics, polygonal representations of the tubes are most often used. The GPU and real-time graphics APIs are built around the idea of rasterizing triangle meshes. But polygonal representations have several drawbacks. Polygons are well-suited to representing nearly planar surfaces, but many polygons are required to give the impression of smooth curved surfaces. Volumetric phenomena are not well-represented by triangle meshes. Polygon meshes lack the regularity of raster data, which can lead to visual artifacts from cracks and T-junctions.

Two alternative approaches to geometry representation which we will explore are voxels and signed distance functions (SDFs). Voxels are commonly encountered in scientific visualization as a representation of scalar fields discretized on a regular 3D grid. Voxels can also be used to represent volumetric phenomena, like fluids, as well as fur, foliage, and semi-transparent objects which may not be accurately rendered by rasterizing polygon meshes [1]. Since most of the voxels in a typical scene would be empty, a full 3D volume is not necessary, and would be an inefficient way to store this data. Hierarchical decompositions of space that exploit scene sparsity are commonly used, e.g. the sparse voxel octree (SVO). In the SVO, empty space is not explicitly stored, leading to a huge reduction in memory requirements for most scenes.

Signed distance functions are a special case of implicit functions for representing surfaces. The signed distance function has the property that the distance to the embedded surface from the point x,y,z is given by $|f(x,y,z)|$, and $f(x,y,z)=0$ for points on the surface. The sign of $f(x,y,z)$ differentiates points that are inside the surface from points outside the surface. Equations can be written for SDFs representing simple primitives, and more complicated surfaces can be built by combining primitives using Boolean operations, or by blending [3]. However, combining a large number of primitives, such as all the union of all the line segments in a collection of streamlines, can be prohibitively expensive. A key strength of our approach is that each node in our SVO stores only the segments which contribute to each voxel, greatly accelerating the union operation. Since SDF primitives appear perfectly smooth at any desired image resolution we can generate high-quality images of streamlines from any viewing distance, unlike polygonal and voxel representations. A benefit of both SDFs and SVOs is that they allow empty space to be efficiently skipped during rendering. For SDFs, $|f(x,y,z)|$ tells you how far you can safely move along a viewing ray without intersecting the surface. For the SVO, large empty nodes can be skipped with ray-plane intersection tests [18].

In this paper we explore two representations for scenes consisting of a large number of tubes, and methods for generating and rendering these representations. Our contributions are

- A GPU-based voxelization technique which can convert geometric streamlines into a streamtube signed distance function.
- A SVO representation for real-time raytracing of streamtubes with straight segments

*e-mail: tmcgraw@purdue.edu

- A hybrid SVO / SDF representation for raytracing/raycasting streamtubes with curved segments

We demonstrate that our methods can combine benefits of both voxel and SDF representations. We can efficiently compute the union of a large number of SDF primitives representing a collection of streamtubes to achieve smooth surfaces at any viewing distance, and render them at interactive rates.

2 Related work

Hierarchical data structures, like octrees, are popular for storing volume data and accelerating spatial queries. Many variations of the structures have been developed to solve specific problems. Knoll et al. [14] developed efficient methods for raytracing isosurfaces stored in an octree which stores minimum and maximum values of a scalar field at each voxel. Octrees containing contour information, in the form of a pair of parallel planes which bound a surface, were used by Laine et al. [16] to reduce blockiness in SVO renderings. Many other sparse hierarchical structures for rendering volumes (e.g. Gigavoxels [6], SparseLeap [11]) and embedded surfaces (e.g. VDB [19]) have been proposed. An SVO can used as an auxilliary scene representation in conjunction with polygons. The SVO can then be used to approximate global illumination [7] and to perform other spatial queries when shading polygons.

Much voxelization literature focuses on the challenge of converting triangle meshes into voxels. When voxelizing meshes it is important to generate a tree which contains every voxel intersected by a triangle. However, the rasterization rules of many graphics APIs will only generate a pixel when the triangle overlaps a specific sample point (or points). A common approach to voxelizing a scene proceeds slice-by-slice [9, 26], rendering the scene with a sequence of viewing volumes. Some approaches rasterize into a full intermediate 3D texture, but it is possible to rasterize directly into an SVO or other sparse hierarchy [5, 23].

Implicit surfaces are another alternative to polygonal scene representation. John Hart [12] presented methods for modeling with implicit surfaces by blending multiple primitives and rendering them with *sphere tracing* (often referred to now as *raycasting*), which exploits the empty-space skipping property of SDFs. Evans [8] used SDFs discretized into textures to approximate global illumination effects, including the ambient occlusion technique we use in our results. Reiner et al. [22] described further methods for modeling more complex scenes with SDFs. The scenes are represented as trees with internal nodes representing blending and Boolean operations. One failure case of the space-skipping property of raycasting occurs when rays graze the surface of an object. The process may proceed in many small steps until the ray gets far enough from the surface. Galin et al. [10] developed a method for raycasting complex shapes by establishing local Lipschitz bounds that enable larger steps to be taken along these grazing rays.

Streamtubes are often rendered as rasterized triangles, but there are alternatives to rendering a full polygonal approximation to the cylindrical shape. Many approaches based on simplified tube geometry have been developed. Imposter-based approaches create view aligned primitives and shade them to give the illusion of true 3D tubes. Petrovic et al. [21] perform a ray-tube intersection test in the fragment shader so that intersections between tubes can be accurately rendered. Imposter-based methods suffer from artifacts which break the illusion of 3D appearance when viewing along the streamline direction. Many methods address this special case in an ad hoc way by drawing a view aligned quad to represent the streamtube cross-section or end cap.

A common approach to generating tubes from curves relies on the geometry shading (GS) stage of the programmable graphics pipeline [15]. The process involves collecting multiple streamline vertices in the GS and generating a triangle strip that approximates a cylindrical tube segment. Many vertices must be generated to approximate a smooth cross-section. This process of geometry amplification in the GS is known to cause performance issues. The tessellation shader can also be used to generate tubes, as demonstrated by Nunes et al. [20]

3 Methods

In this section we describe how we perform tube voxelization and construct the SVOs used for rendering straight and curved tubes.

3.1 Tube SDF voxelization

When voxelizing, we take a slice-by-slice GPU rasterization approach, as many existing mesh voxelization approaches do, so we do not detail the full method here. Instead we focus on the differences with typical slicing approaches. The tubes we wish to display are defined by a collection of line segments which we render as quadrilateral imposters. These imposters represent a slice of the distance function around a tube. Note that when the view direction and streamline direction are parallel this slice of the distance function is the distance to the circular cross-section of the tube and the impostor is still a quadrilateral. The distance of each fragment to the tube is computed in the fragment shader.

If $f(\mathbf{p})$ is the distance to a line segment, then $f(\mathbf{p})-r$ is the signed distance to a tube of radius, r. The distance from the point $\mathbf{p}=(x,y,z)$ to the line segment between points $\mathbf{a}$ and $\mathbf{b}$ is given by

$$\begin{aligned} \mathbf{v} &= \mathbf{b}-\mathbf{a} \\ \mathbf{w} &= \mathbf{p}-\mathbf{a} \\ t &= \min(\max(\frac{\mathbf{v}\cdot\mathbf{w}}{\mathbf{v}\cdot\mathbf{v}},0),1) \\ f &= \|\mathbf{w}-\mathbf{v}t\|. \end{aligned} \tag{1}$$

Curved tube segments are generated by domain transformation, $f(\mathbf{Mp})$. For curved streamtubes we choose $\mathbf{M}$ to be the quadratic interpolation of 3 matrices along the tube: $\mathbf{M}=\mathbf{M_a}$ at $\mathbf{p}=\mathbf{a}$, $\mathbf{M}=\mathbf{M_b}$ at $\mathbf{p}=\mathbf{b}$, and $\mathbf{M}=\mathbf{I}$ (the identity matrix) at the midpoint of the segment. $\mathbf{M_a}$ and $\mathbf{M_b}$ are the matrices that perform a rotation about the corresponding end which aligns $\mathbf{v}$ with the tangent at that endpoint.

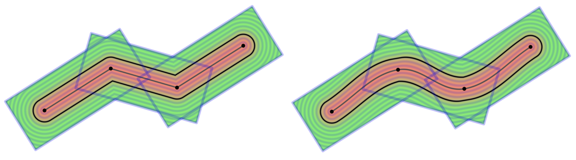
Figure 2: Streamlines are rasterized by rendering a bounding quadrilateral for each line segment. Signed distance to the tube - either a straight tube (left) or a curved tube (right) - is computed in the fragment shader and rendered to a texture. Green/red represent regions of positive/negative distance.

To join multiple tube segments into a streamtube we use the Boolean union operation which is computed as the minimum of SDF values at a fragment. We achieve this by rendering billboarded segments into a floating-point framebuffer object using the minimum blending mode (GL_MIN in OpenGL [24]). The voxelization process proceeds slice-by-slice, setting the projection matrix to cover one slab of voxels plus 2 streamtube radii for each rendering pass. Note that we don't need to join the billboarded segments, in fact we rely on segments overlapping so that the minimum blending mode can compute the join between tube segments. Likewise, we do not need to cap the segments. The signed distance to the tube $f(\mathbf{p})-r$ will have a smooth radius automatically generated at the ends. The process is illustrated in Figure 2. Unlike the process for rasterizing polygonal meshes, we do not need to worry about triangle dominant axis selection or missing thin structures since our scene consists of tubes with nonzero radius.

Due to the finite size of the imposters that we draw, the distance function is only computed in a band around the streamline. The size of that band depends on parameters δ (the width of the imposter), and r, the streamtube radius. Choice of δ depends on the subsequent rendering technique, e.g. isosurface extraction and SVO creation require only a narrow band, while raycasting requires a wide band. Example marching cube and volume raycasting results are shown in Figure 3. The raycasting method we used is similar to the sphere tracing method described by Hart [12], except that we are fetching trilinearly interpolated SDF values from a 3D texture. These methods can produce a reasonable image of streamtubes, however, there are critical drawbacks. The SDF texture is large. A 1024^3 dataset occupies 2GB when represented as half-precision floating point (16 bits per voxel). Even though typical video memory on modern videocards is greater than this, creation of larger 3D textures can fail due to memory fragmentation. Per-streamline and per-segment attributes can't be used, for example by rendering into an auxillary volume, since multiple fibers may intersect a single voxel.

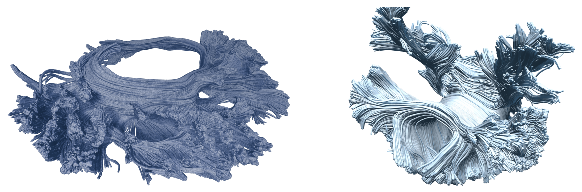
Figure 3: A high resolution mesh extracted from the voxelized SDF of right cingulum bundle using marching cubes (left), and raycasting results for SDF of the corpus callosum (right).

3.2 Sparse voxel octree building and raytracing

In our SVO, nodes are stored in the voxels of a 3D texture, which allows for fast indexing with 3D indices. Minimal data is stored in each node. We don't store node size, depth, or any other property that can be computed during tree traversal. All children of a given node are stored in a $2 \times 2 \times 2$ block of adjacent voxels. This allows the parent node to point to its children by pointing to a single child with a single 3D texture coordinate. Internal nodes store a flag which lets us differentiate internal nodes from leaf nodes. In leaf nodes we store minimum distance to the isosurface, and for maximum flexibility, texture coordinates that refer to auxiliary 2D textures that allow us to decorate the tree with any application-specific data needed. In total, each node occupies 8 bytes.

As each slice of the distance volume is generated during voxelization, we scan through the values and classify voxels as leaf nodes when $\varepsilon > f(x,y,z) > -\varepsilon$. The Morton code for the voxel and the signed distance value are saved for each leaf. When all slices have been processed the leaves are sorted in z-curve order by Morton code and the tree is built bottom-up as described by Baert et al. [2]. The SVO can be rendered with a stackless traversal. We use the push-down variant of the kd-restart traversal presented by Horn et al. [13], but adapted for octrees, as described in Algorithm 1, where node.splitPlaneIntersection(ray) returns the first node split plane intersection between tMin and tMax, and node.child(ray, tMin, tMax) returns the first child node along the ray between tMin and tMax. In terms of memory requirements, the SVO is a vast improvement over the full texture distance representation. Memory requirements are on the order of 10s of MB, rather than several GB. The SVO also enables simple LOD computation by limiting the maximum traversal depth of the SVO when performing queries. A major drawback to the SVO representation is that renderings have a blocky appearance unless the resolution is extremely high. This effect can be seen in Figure 4, which was generated using Algorithm 1 where the leaf node contents are the bounding box of the node. we address this problem in the following sections.

Algorithm 1 Raytracing SVO

```
tMin ← sceneMin; tMax ← sceneMin;
while tMax < sceneMax do
    tMin ← tMax; tMax ← sceneMax;
    node ← root; pushdown ← true;
    while node.type != leaf do
        tSplit ← node.splitPlaneIntersection(ray)
        if tSplit > tMin AND tSplit<tMax then
            tMax ← tSplit; pushdown ← false;
        end if
        node ← node.child(ray, tMin, tMax)
        if pushdown == true then
            root ← node
        end if
    end while
    if node.isEmpty == false then
        return ray intersection with leaf node contents
    end if
end while
No intersection found
```

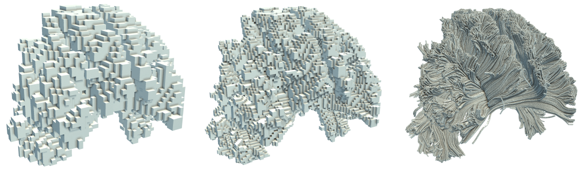
Figure 4: SVO raytracing of corpus callosum at 3 different LODs.

3.3 SVO for raytracing straight tube segments

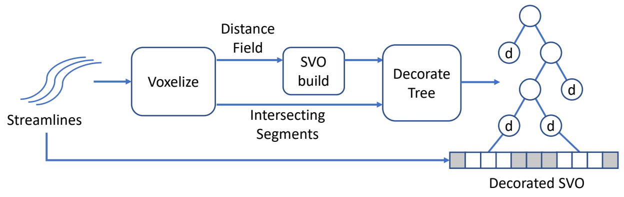

Figure 5: Our processing pipeline for streamtube SVO creation. The streamtube SDF is voxelized and a SVO is built with signed distances in the leaf nodes which are decorated with streamline vertex indices.

In this scheme, illustrated in Figure 5, we store within each SVO leaf node references to the tube segments that intersect the corresponding voxel. When raytracing, if the ray intersects a leaf node the ray intersection points with the tubes are computed. This requires us to store the streamline vertices on the GPU (we use a shader storage buffer) in addition to the SVO. In our case we store 4 vertex indices per leaf node, so that we can resolve 4 streamtube segments intersecting in a single voxel, and we also store the streamline ID in the w-component of each streamline vertex. Since the streamline vertices are stored consecutively in memory, we can index forward and backward to determine adjacent vertices on the streamline. So, for each vertex index, m in a leaf node we render the tube segment between vertex m and $m+1$ if they belong to the same streamline. We can use the same technique to color tubes by orientation, as shown in Figure 7.

During voxelization, We keep track of all of the tube segments that intersect each voxel using an A-buffer [4]. After rendering each slice, we sort the A-buffer layers by distance in a compute shader before augmenting the SVO leaf nodes with the 4 nearest segments. For the results shown in this paper we used an A-buffer with 24 layers.

3.4 SVO for raycasting curved tube segments

This approach, in essence, replaces the cubical proxy geometry used in volume raycasting with tight-fitting voxelized proxy geometry from the SVO. The hybrid raytracing/raycasting rendering technique proceeds by using Algorithm 1 to determine the leaf node intersected by each ray, then performing SDF raycasting on the union of curved tubes within that node. We cannot compute an exact intersection as with straight segments since there is no analytic expression for curved tube-ray intersection. During raycasting we use the same domain transformation $f(M\mathbf{p})$ that was used during curved tube voxelization.

3.5 Results

In this section we describe the results of our technique on neuronal fiber data from the 2015 ISMRM tractography challenge [17]. The tracts were computed from diffusion-weighted MRI data from the Human Connectome Project [25] which were then manually refined and segmented into 25 anatomical regions and saved in the tck file format. These tracts were originally used as the ground-truth for assessing the quality of various tractography algorithms.

Tck file	Streamline buffer (MB)	SVO size (MB)	Decoration size (MB)
CP	0.417	2.44	4.26
SCP R	1.67	5.30	9.22
SLF R	16.2	35.3	61.7
Cingulum L	24.9	60.5	105
MCP	38.7	40.2	70.4

Table 1: SVO memory requirements

We implemented our methods in OpenGL on a PC with Nvidia GTX 1080 Ti (12 GB VRAM) and Intel Core i7-8700K 3.7 GHz and 32 GB RAM. In our images we have used a low-resolution dense distance field created by our voxelizer to compute shadowing and ambient occlusion. Voxelizing only distance for applications such as isosurface extraction is fast (2s to 5s per dataset), but voxelizing the streamline vertex indices using the A-buffer is much more time consuming (114s to 136s). However, this is a preprocessing step which can be performed offline once for each dataset.

Tck file	Straight tube raytracing (ms)	Curved tube raycasting (ms)
CP	3.61	15.3
SCP R	5.87	22.8
SLF R	13.3	73.1
Cingulum L	11.3	59.7
MCP	12.4	77.3

Table 2: Raytracing and raycasting performance (1920×1080)

At runtime our system needs access to a buffer of streamline vertices, the SVO 3D texture, and the node decoration texture. In Table 1 we show the GPU memory requirements of our system for several datasets, ranging from the smallest to largest. These requirements are the same for straight or curved streamline segments. For comparison, a full 3D texture of size 1024^3 containing 16-bit distance values requires 2 GB, and full volumes holding 4 32-bit integer vertex indices would be 16 GB. Even though our data is not sparse everywhere (it contains regions of tightly-packed fiber bundles) good compression is achieved nevertheless. We merged all 25 datasets into a single decorated SVO to obtain a full-brain dataset consisting of 19.5 million fiber vertices occupying a total of 1.92 GB. This is more memory than polygonal streamtubes would require, but we estimate we could handle up to 100 million vertices on our test PC before encountering memory issues.

In Figure 1 raytraced images of the Cingulum L, MCP and SLF R datasets are shown, and Figure 6 shows raytraced images of the full-brain dataset. At high magnification it is apparent that the raytraced tubes are made of straight segments, but the hybrid rendered curved tubes are smooth, as shown in Figure 7. Average render times during interactive sessions rendered with lighting in a 1920×1080 window are shown in Table 2.

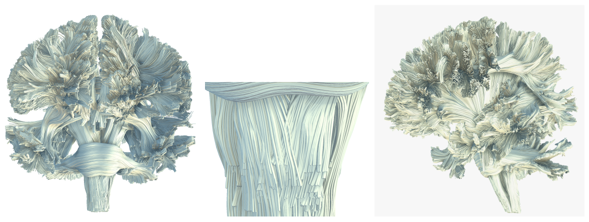
Figure 6: Raytraced images of merged datasets, coronal view (left), intersecting fiber detail (middle) and sagittal view (right).

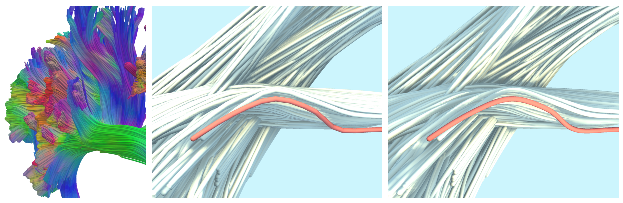
Figure 7: Tubes colored by fiber direction (left). Straight raytraced tubes (middle), curved raycast tubes (right).

4 Conclusions and Future Work

In this paper we have presented a novel method for voxelizing collections of curved tubes, and an efficient SVO representation that allows straight and curved tubes to be rendered at interactive rates. Compared to other approaches, such as imposters generated in the geometry shader, our method allows smooth joins between tube segments, smooth circular cross-section at any pixel resolution, and round end-capping.

A drawback of our approach is that we require that no more than four tube segments intersect a voxel. In our results there are no visible artifacts because we have chosen a sufficiently small voxel size for our datasets. However, our voxelizer has detected regions in the interior of bundles where the 4 tube limit is exceeded. This limits our ability to do cutaways or render with transparency. In future work we plan to implement a preprocessing step that can select an appropriate voxel size based on number of vertex indices per voxel, tube radius, streamline density, and segment lengths.

Our method does not exploit the ability of SVOs to represent multiple levels-of-detail (LOD). In the future we plan to investigate methods for view-dependent LOD. We also wish to explore other application areas, such as molecular model visualization.

Acknowledgments

The authors wish to thank the organizers of the 2015 ISMRM tractography challenge [17] for the datasets used in this paper.

References

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th ed., 2018.

[2] J. Baert, A. Lagae, and P. Dutré. Out-of-core construction of sparse voxel octrees. *Computer Graphics Forum*, 33(6):220–227, 2014.

[3] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[4] C. Crassin. Fast and accurate single-pass A-buffer using OpenGL 4.0+. *Icare 3D Blog, https://blog.icare3d.org//2010//06//fast-and-accurate-single-pass-buffer.html*, 2010.

[5] C. Crassin and S. Green. Octree-based sparse voxelization using the GPU hardware rasterizer. *OpenGL Insights*, pp. 303–318, 2012.

[6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15–22, 2009.

[7] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, 2011.

[8] A. Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, pp. 153–171. ACM New York, NY, USA, 2006.

[9] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

[10] E. Galin, E. Guérin, A. Paris, and A. Peytavie. Segment tracing using local lipschitz bounds. In *Computer Graphics Forum*, 2020.

[11] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. Sparseleap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization & Computer Graphics*, 24(1):974–983, 2018.

[12] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *Visual Computer*, 12(10):527–545, 1996.

[13] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive kD tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pp. 167–174, 2007.

[14] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 115–124. IEEE, 2006.

[15] M. Krone, K. Bidmon, and T. Ertl. GPU-based visualisation of protein secondary structure. In *Theory and Practice of Computer Graphics*. The Eurographics Association, 2008.

[16] S. Laine and T. Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2010.

[17] K. H. Maier-Hein, P. F. Neher, J.-C. Houde, M.-A. Côté, E. Garyfallidis, J. Zhong, M. Chamberland, F.-C. Yeh, Y.-C. Lin, Q. Ji, et al. The challenge of mapping the human connectome based on diffusion tractography. *Nature communications*, 8(1):1–13, 2017.

[18] A. Majercik, C. Crassin, P. Shirley, and M. McGuire. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques*, 7(3), 2018.

[19] K. Museth. VDB: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics*, 32(3):1–22, 2013.

[20] G. Nunes, A. Valdetaro, A. Raposo, B. Feijó, and R. de Toledo. Rendering tubes from discrete curves using hardware tessellation. *Journal of Graphics Tools*, 16(3):123–143, 2012.

[21] V. Petrovic, J. Fallon, and F. Kuester. Visualizing whole-brain dti tractography with GPU-based tuboids and lod management. *IEEE Transactions on Visualization & Computer Graphics*, 13(6):1488–1495, 2007.

[22] T. Reiner, G. Mückl, and C. Dachsbacher. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Computers & Graphics*, 35(3):596–603, 2011.

[23] M. Schwarz and H.-P. Seidel. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics*, 29(6):1–179, 2010.

[24] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane. *OpenGL programming guide: The official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[25] D. C. Van Essen, S. M. Smith, D. M. Barch, T. E. Behrens, E. Yacoub, K. Ugurbil, W.-M. H. Consortium, et al. The WU-Minn human connectome project: an overview. *Neuroimage*, 80:62–79, 2013.

[26] L. Zhang, W. Chen, D. S. Ebert, and Q. Peng. Conservative voxelization. *The Visual Computer*, 23(9-11):783–792, 2007.