GPU-based Raycasting of Hermite Spline Tubes

Benjamin Russig* TU Dresden
Mirco Salm† TU Dresden
Stefan Gumhold‡ TU Dresden

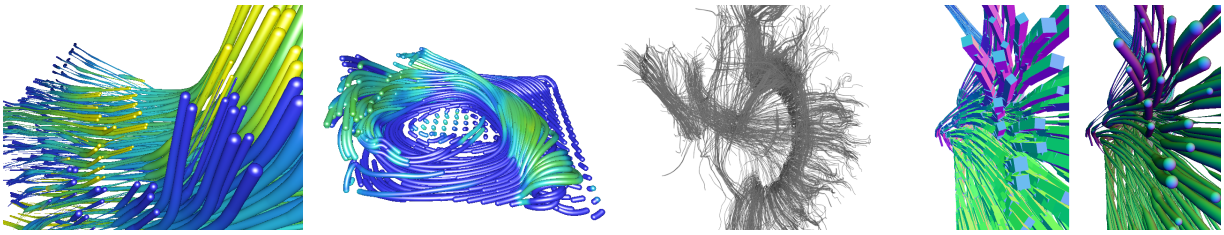

Figure 1: Rendering results from the technique presented in this paper. From left to right: example datasets *Fibers*, *HotRoom* and *Brain*, as well as a side-by-side comparison of the bounding box geometry we use for raycasting and the tube segments they cover.

Abstract

Visualizing curve and trajectory data is a common task in many scientific fields including medicine and physics. Tubes are an effective visualization primitive for this sort of data, but they require highly specialized renderers to achieve high image quality at frame rates sufficient for interactive visualization. We present a rendering algorithm for Hermite spline tubes, i.e. tubes that result from Hermite splines interpolating the data, with support for varying-radii circular tube cross sections. Our approach employs raycasting and works directly on this continuous representation without the need for surface tessellation, made possible by an efficient ray-tube intersection routine suitable for execution on modern GPUs.

Index Terms: Computing methodologies—Computer graphics—Rendering; Human-centered computing—Visualization—Visualization Application domains—Scientific visualization

1 Introduction

Tubes are employed as a primitive for visualizing a wide variety of natural and abstract structures, ranging from white matter tracts and blood vessels (Merhof et al. [8]) over trajectories of physical particles and objects (Fraedrich et al. [4]) to stream- and pathlines emerging from vector fields (Zöckler et al. [14], Meuschke et al. [9]).

Formed by a (potentially varying) cross section extruded from a central axis, they pose several challenges for efficient rendering, usually addressed by striking a compromise between image fidelity and render time. Traditionally, interactive realtime visualizations using tubes rely on adaptive surface tessellation to maximize throughput with minimal loss of quality (e.g. Stoll et al. [12], Nunes et al. [10]). Since this still means discretization of the tube surface, a lot of additional data is being generated for rendering, worsening the geometry bottleneck for large, complex datasets.

Raycasting of analytically or algebraically defined tube surfaces can circumvent this problem, but compact formulations of the ray-surface intersection for the former or even just the tube surface itself for the latter are not straight-forward or – depending on the versatility of the tube primitive – non-existent.

We tackle this problem by defining the tube from the volume traced out by a sphere moving along the central axis, which we describe using a Hermite spline. Other tube attributes, notably the radius, also follow a Hermite spline. During rendering, the *Hermite spline tubes* are processed per *Hermite spline segment*, where each segment is defined by two nodes with information about position, radius, and their respective first derivatives.

The remainder of this paper will give a focused overview of related methods in Sect. 2, explain our raycasting algorithm in detail in Sect. 3, discuss preliminary results in section Sect. 4 and provide a brief outlook on future work in Sect. 5.

2 Related Work

Generalized cylinders were originally introduced by Agin and Binford [1]. In contrast to a regular cylinder, the cross section of a generalized cylinder can be arbitrarily shaped and its axis is described by a curve. Tubes are a form of generalized cylinder with – at least when characterizing common usage of the term in the visualization community – rather simple, mostly convex cross sections. In this section, we give a brief overview of related methods for rendering tubes. For conciseness, we restrict ourselves to methods that employ raycasting or -tracing, as they are most related to our approach. We categorize these methods according to whether they work on some continuous description of the tube directly (we refer to that as *spline tubes*) or rely on some sort of line primitives to connect data points.

Spline Tubes. Bronsvoort and Klok [2] presented a method for raytracing tubes defined by parametric curves. Their method employs a generic subdivision scheme that does not make any assumptions about curve parameterization or cylinder cross sections, resulting in a highly versatile but expensive intersection routine targeted at offline renderers. Reina et al. [11] present a GPU-based raycasting scheme for spline tubes with elliptical cross section. Ellipsoids have a discernible orientation, and this translates to intuitively understandable corkscrew patterns on the tube surface. They cannot provide a compact implicit function for the tube surface, so they employ the distance-bound raymarching strategy first presented by Hart [6] to find ray intersections. Although raymarching tends to be slower than direct raycasting, their profiling results suggest that tessellation-based techniques will be slower still for sufficiently large datasets as the GPU hits a vertex bottleneck. More recently, the open source CPU raytracing framework Embree [13] added support for a variety of spline curves for use as geometry primitives. The tube variants of these primitives sweep a planar circle along the axis, which can create bulge artifacts in locations of very high curvature. Thus, they restrict valid cross section radii at a point to be smaller than the local curvature radius of the axis.

Line Primitives. The other major category of relevant rendering techniques is based on discretization into piecewise linear segments, relying on a dense sampling for smooth-looking curves.

*e-mail: benjamin.russig@tu-dresden.de
†e-mail: mirco.salm@mailbox.tu-dresden.de
‡e-mail: stefan.gumhold@tu-dresden.de

Han et al. [5] raytrace cone stumps and spheres on the CPU. To support artifact-free transparency despite composing their tube primitive from multiple objects, they employ a CSG-derived interpretation of ray surface hits. Kanzler et al. [7] utilize a novel voxel representation for lines, enabling fast image-order raycasting of line datasets as tubes. Their main contribution is the voxel representation itself, as it supports level-of-detail by means of a tailor-made averaging operation, as well as fast voxel-based approximate simulation of global illumination phenomena. For actual intersection of the rays with tube segments, they employ a similar approach to Han et al. [5], albeit without support for varying radii.

3 Hermite Spline Tube Raycasting

3.1 Preliminaries

Terminology. A Hermite spline tube is defined by a sequence of control points (henceforth called *nodes*) that provide position, radius, and color values as well as derivatives. The shape and coloring of a tube in between nodes is determined by a cubic Hermite interpolation of the node attributes or, for the purpose of ray casting, a piecewise quadratic approximation thereof (see Sect. 3.2). We will refer to the value of an attribute of some node i as n_i and its derivative as t_i.

The *position spline* defines a curve in world space that forms the generalized cylinder axis. We will mathematically refer to the position spline as $\boldsymbol{p}(t)$ and its component functions as $p_{x|y|z}(t)$, respectively. A *tangent spline* $\vec{\boldsymbol{t}}(t)$ can be derived from the position spline by differentiation with regard to the curve parameter, and it will interpolate the node tangents just like the position spline interpolates the node positions. The *radius spline* $r(t)$ defines the radius of the sphere extruding the tube at t, and the *color spline* $\ddot{c}(t)$ describes the color value attributed to t.

Finally, pairs of adjacent nodes form tube *segments*, resulting in individual per-segment polynomial curves. Of particular interest for our algorithm are the *position curves* $\boldsymbol{p}_k(t)$ and *radius curves* $r_k(t)$, $t = 0..1$ of a segment k.

Data organization. During rendering, node data is stored as vertex attributes in a vertex buffer, each vertex representing one node. To form segments, pairs of nodes are referenced by means of an index buffer. The whole dataset can then be drawn with a single indexed draw call for line lists.

Note that we impose no notion of individual tubes; they form naturally from segments that fit together. Sharp corners are supported by duplicating a node with a differing tangent, at the cost of introducing redundancy for the other attributes of that node. Bifurcations are supported by referencing the same node more than twice, or optionally – if discontinuity of any of the attributes is desired – referencing another duplicate of that node from the bifurcating segment.

3.2 Rendering

Algorithm. We begin with a high-level view of the rendering algorithm. Our method targets rasterizer systems, meaning we have to first identify the fragments for which to cast rays at the scene. This necessitates the use of silhouette geometry. We opted to employ oriented bounding boxes around the tube segments in world space.

The algorithm starts with a single indexed line list draw call for the whole dataset. The vertex shader is purely pass-through, providing the geometry shader with the start and end node of a segment. The geometry shader takes the line formed by the two nodes as an input and outputs triangle strips for two oriented bounding boxes tightly covering the whole tube segment. We output two boxes per segment because we actually subdivide the cubic curves of a segment into two quadratics at the geometry shader stage – we will explain our reasoning for doing this when discussing the intersection routine later on.

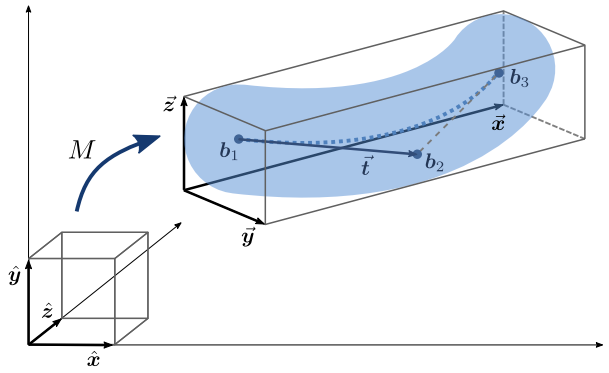

Figure 2: To obtain the oriented bounding box for a quadratic sub-segment, we compute a matrix M that transforms the unit cube in the positive octant such that it tightly fits the tube segment geometry. The direction from start to end control point ($\boldsymbol{b}_1$ and $\boldsymbol{b}_3$, respectively) and the tangent $\vec{\boldsymbol{t}} = \boldsymbol{b}_2 - \boldsymbol{b}_1$ determine the orientation.

Rays are then cast in the fragment shader for every fragment generated by the rasterizer, yielding color and depth information for the closest intersection of the ray with the tube segment inside the silhouette.

Segment subdivision. To obtain the two sub-segments, we create two quadratic Bezier curves per attribute, the control points of which we denote as a_1, a_2, a_3 and b_1, b_2, b_3, respectively. For a segment referencing nodes i, j, we set the control points at the segment nodes to $a_1 = n_i$ and $b_3 = n_j$. The respective middle control points are then calculated as $a_2 = n_i + c_1 t_i$ and $b_2 = n_j - c_2 t_j$. We choose $c_1 = c_2 = 1/3$, motivated by the Hermite/Bezier basis transformation. Setting $a_3 = b_1 = (a_2+b_2)/2$ connects the sub-segments with C^1 continuity at the junction. The Bezier control points are provided to the fragment shader as vertex attributes of the respective bounding box.

Silhouettes. In order to minimize the number of ray misses, the silhouettes should be as tight-fitting around the tube sub-segments as possible. At the same time, geometry load on the GPU should be minimized in order to not lose the performance advantage of raycasting for large datasets. We believe that oriented bounding boxes are a reasonable choice. Despite their simplicity, they are likely to be a good fit for the majority of tube segments in a dataset since individual segments curve relatively little in practice.

To determine the bounding box for a sub-segment, we first select a suitable orientation. We found the following strategy to yield reasonable results (the sub-segment position curve $\boldsymbol{b}(t)$ is again determined by Bezier control points $\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3$):

- *x-axis*: use the normalized vector from start node to end node, i.e. $\hat{\boldsymbol{x}} = \frac{\boldsymbol{b}_3 - \boldsymbol{b}_1}{\|\boldsymbol{b}_3 - \boldsymbol{b}_1\|}$.
- *y-axis*: project the tangent $\vec{\boldsymbol{t}} = \boldsymbol{b}_2 - \boldsymbol{b}_1$ onto a plane with normal $\hat{\boldsymbol{x}}$ and normalize, i.e. $\hat{\boldsymbol{y}} = \vec{\boldsymbol{v}}/\|\vec{\boldsymbol{v}}\|$ where $\vec{\boldsymbol{v}} = \vec{\boldsymbol{t}} - \hat{\boldsymbol{x}}\langle\vec{\boldsymbol{t}}, \hat{\boldsymbol{x}}\rangle$. In case the tangent and $\hat{\boldsymbol{x}}$ are colinear, we choose any unit vector orthogonal to $\hat{\boldsymbol{x}}$.
- *z-axis*: results from the cross product $\hat{\boldsymbol{z}} = \hat{\boldsymbol{x}} \times \hat{\boldsymbol{y}}$.

We can now construct a matrix $R = (\hat{\boldsymbol{x}}\ \hat{\boldsymbol{y}}\ \hat{\boldsymbol{z}})^T$ representing a rotated version of world space that has its basis vectors aligned with the final bounding box edges. Transforming control points into this frame yields the transformed position curve $\boldsymbol{q}(t) = R \cdot \boldsymbol{b}(t)$, which we use for determining the bounding box extents by calculating the extrema of the component functions $q_{x|y|z}(t) \pm r(t)$.

Let $q_{x|y|z,-}(t) = q_{x|y|z}(t) - r(t)$ and $q_{x|y|z,+}(t) = q_{x|y|z}(t) + r(t)$. The segment extrema can be found by evaluating each component function at $t = 0, 1$ as well as an additional candidate $t_{x|y|z}$ acquired by analytically computing the roots of the respective (linear) first derivatives $q'_{x|y|z,-}(t)$ and $q'_{x|y|z,+}(t)$. This candidate will be

considered only if it falls inside the parameter range $[0,1]$.

We denote the minimum and maximum value of a component as $\{x|y|z\}_{min}$ and $\{x|y|z\}_{max}$. With the extrema of each component function known, they result from the respective smallest and largest values. For the x-extent, this yields:

$$\begin{aligned} x_{min} &= \min\left(q_{x,-}(0),\ [q_{x,-}(t_x),]\ q_{x,-}(1)\right) \\ x_{max} &= \max\left(q_{x,+}(0),\ [q_{x,+}(t_x),]\ q_{x,+}(1)\right) \end{aligned}$$

With this, scaling and translation for the oriented bounding box can be fixed. The corresponding homogeneous transformation matrix for use on the positive unit cube (see Fig. 2) is calculated as follows:

$$M = \begin{bmatrix} R^T & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{max}-x_{min} & 0 & 0 & x_{min} \\ 0 & y_{max}-y_{min} & 0 & y_{min} \\ 0 & 0 & z_{max}-z_{min} & z_{min} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Intersection. Since no compact implicit function describing a spline tube segment is available, direct raycasting (i.e. without employing a ray marching strategy) requires a parametrized ray-segment intersection to be formulated first. To that end, we parametrize the ray-sphere intersection problem such that position and radius of the sphere are determined by the respective attribute curves. We then construct a function $l = f(t)$ that directly relates the curve parameter t to the ray parameter l corresponding to the front-facing intersection with the sphere at t. Thus, the smallest local minimum of that function within the segment domain $t = 0..1$ yields the closest intersection of the ray with the tube segment.

We further simplify the problem by formulating the intersection in *ray space* (see Fig. 3). We define the ray as $\boldsymbol{r}(l) = \boldsymbol{o} + l \cdot \hat{\boldsymbol{d}}$, where the ray origin $\boldsymbol{o}$ is the eye position and $\hat{\boldsymbol{d}}$ gives the direction to the point on the image plane corresponding to the fragment. In ray space, the coordinate system origin is at $\boldsymbol{o}$, while the ray direction $\hat{\boldsymbol{d}}$ coincides with the x-axis (any two vectors forming an orthonormal basis with $\hat{\boldsymbol{d}}$ may be chosen as y and z axes).

We start with the intersection of a sphere (defined by its position $\boldsymbol{p}$ and radius r) and the x-axis, which is computed as:

$$s(\boldsymbol{p}, r) = p_x \pm \sqrt{r^2 - p_y^2 - p_z^2} \tag{1}$$

Plugging in the segment attribute curves for ray-space position and radius (we omit the segment ID k for legibility) and chosing the front-facing intersection yields:

$$f(t) = p_x(t) - \sqrt{r(t)^2 - p_y(t)^2 - p_z(t)^2} \tag{2}$$

We minimize f via differentiation with respect to t. For convenience, we denote the individual polynomials outside and inside the square root as $h(t) = p_x(t)$ and $g(t) = r(t)^2 - p_y(t)^2 - p_z(t)^2$, i.e. $f(t) = h(t) - \sqrt{g(t)}$, yielding the derivative

$$f'(t) = h'(t) - \frac{g'(t)}{2\sqrt{g(t)}} \tag{3}$$

f and f' are real only within intervals where $g(t) \geq 0$, so finding the minima of f via root search on f' requires knowing the real roots of g first. Then, root finding on f' has to be performed on the following intervals of $t = 0..1$:

(a) $t_{min} = 0$ and the first real root of g at t_0, iff. $g(0) > 0$
(b) two subsequent real roots t_a and t_b of g, iff. $g((t_a+t_b)/2) > 0$
(c) the last real root of g at t_n and $t_{max} = 1$, iff. $g(1) > 0$

Intuitively, these intervals can be thought of as ranges of the curve parameter t where the tube segment and the ray spatially overlap, and the number of these intervals corresponds the the number of times the ray hits the tube segment (see Fig. 4).

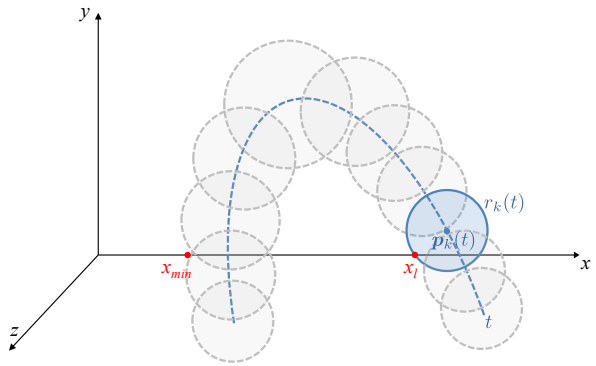

Figure 3: Spheres along a spline segment in ray space are intersected with the x-axis (the ray), resulting in an intersection point x_1. The point $\boldsymbol{p}_k(t)$ on the position curve and corresponding sphere radius $r_k(t)$ are parametrized by the local curve parameter $t = 0..1$ of tube segment k. The first intersection x_{min} with the tube segment is found by minimizing x with respect to t.

When using the cubic position and radius curves of the segment as is, g will become a polynomial of degree 6. If we use the quadratic sub-segment curves instead, we can get g down to degree 4. We could now analytically calculate all real roots of g by applying the quartic equation. However, due to the large number of arithmetic operations involved, we opted to use the same fast iterative approach we apply to f'.

Root finding. In the following, we describe the strategy we use for solving all root finding problems posed by our method. Let $u(x)$ be a polynomial. We bracket the real roots we are interested in by some isolating interval $[x_a, x_b]$ such that $u(x_a)$ and $u(x_b)$ evaluate to opposite signs. We then use the bisection method [3] to approximate them. We believe bisection to be a reasonable choice for our use case because of its low per-iteration cost and fixed convergence rate, but investigating methods with faster theoretical convergence is a promising avenue for future work.

Since every real root of u within some interval $[a,b]$ of interest lies in between two subsequent extrema that evaluate to opposite signs, its extrema provide all the information needed to form isolating intervals (in this context, the interval borders can be extrema as well). Determining the extrema of u is equivalent to finding the roots of its derivative. Therefore, starting with some derivative for which it is known that only a single root inside $[a,b]$ exists, it is possible to recursively apply bisection to determine the isolating intervals for all lower derivatives of u up to the original polynomial.

If u is of degree n, the roots of n derivatives need to be found. Additionally, the number of potential real roots of each derivative equals its degree. Consequently, the worst-case complexity of this algorithm is $O(^{(n^2+n)}/_2)$. However, some root searches can be saved by analytic computation as soon as the derivative is of sufficiently low degree. For our g of degree 4, we chose to analytically compute the real roots of g'' (which is quadratic), necessitating 7 numerical root searches in the worst case.

f is not a polynomial, so we cannot stop at some derivative for which a closed-form solution exists. However, we observed from empirical study of f that the number of roots consistently decreases with every differentiation, and that f' has a maximum of 3 roots per real interval (we would like to follow up with a formal proof for these claims in future work). This means that starting from $f^{(3)}$, 6 numerical root searches per interval are required in the worst case to find all real roots of f'. Also, we observed that in every real interval where f' has 3 real roots, the middle one always corresponds to a local maximum of f and can thus be ignored, bringing down the worst-case number of root searches per interval to 5. For some illustrative plots of f and its companions, see Fig. 4.

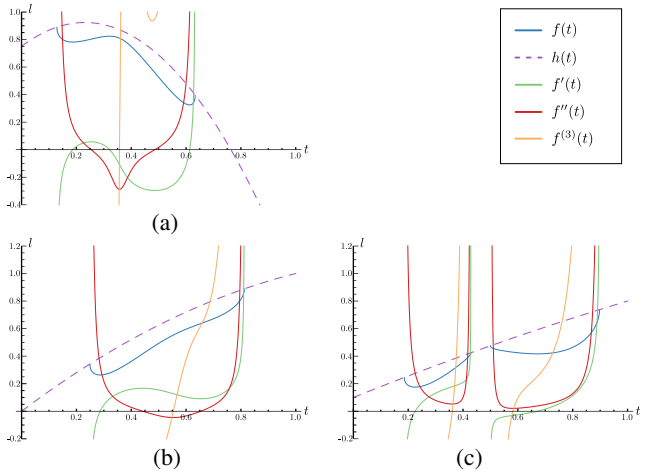

Figure 4: Example plots of f. Every interval where f is real corresponds to an intersection with the tube. Most configurations will result in just one such interval (examples (a) and (b)). Several disjunct intervals are possible if the ray leaves and enters the tube several times (example (c)).

Finally, we evaluate f at the roots of f' and chose the smallest value as the ray parameter corresponding to the closest intersection.

Shading. For demonstration purposes, we apply simple local lighting with diffuse Lambertian reflectance and a Blinn-Phong specular term to the tube surface. The surface normal required for lighting calculations equals the normal of the sphere at the curve parameter t_0 corresponding to the closest intersection, i.e. $\hat{\boldsymbol{n}} = \vec{\boldsymbol{n}}/\|\vec{\boldsymbol{n}}\|$ with $\vec{\boldsymbol{n}} = \boldsymbol{r}(f(t_0)) - \boldsymbol{p}(t_0)$.

Currently, we use just the color obtained from $\ddot{\vec{c}}(t_0)$ to determine surface albedo. Adding texture would be possible by adopting a suitable surface parameterization.

4 Results

For example renderings of the five test datasets using our method, see Fig. 1 and Fig. 5. These datasets are characterized by the following statistics:

	Fibers	*HotRoom*	*Bundle*	*Brain*	*Furball*
tubes	241	722	1250	1701	10000
segments	3530	30948	105640	114489	787327

We compare the performance of our method against a tessellation-based strategy with comparable capabilities that employs dynamic hardware tessellation for view-dependent level-of-detail, similar in spirit to the method proposed by Nunes et al. [10]. In contrast to our raycasting method, the tube surface model of the tessellation approach results from sweeping a planar circle along the position spline instead of a sphere. The cross section at every sample is made up of 6 vertices at the base tessellation level and may get refined to up to 30 vertices to enable a visual fidelity comparable to our method. The tessellator is also allowed to insert up to 5 additional samples along the spline.

For measuring rendering performance, we rendered each dataset in a 1920x1080, 45° vertical FoV viewport from two different viewing configurations, which we call *close* and *far*. The view was then rotated in an orbit around the center of the dataset for 1000 frames. For the *far* configuration, we chose a distance from the center such that the whole dataset fits the viewport, for the *close* configuration we chose the distance to be $1/3$ of that. To estimate the impact of generating the silhouette geometry, we also performed these tests with an empty fragment shader that discards all fragments. For render time measurements, performed on a *Geforce RTX 2080 Ti* GPU, see Fig. 5. Our experiments confirmed that our method becomes

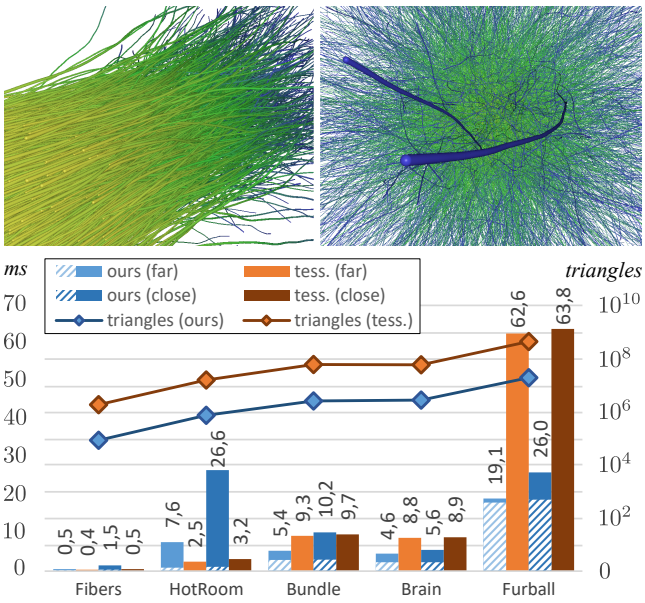

Figure 5: **Top**: Procedurally generated datasets *Bundle* (left) and *Furball* (right). **Bottom**: Comparison of average render times (in milliseconds) and triangle count for each test dataset and view configuration. For our method, the hatched area represents the average render time without any fragment shading performed. The triangle count measured for tessellation is the average over all camera orbits performed on a given dataset.

competitive with and finally surpasses tessellation as datasets increase in size, in line with the findings reported for other raycasting techniques [7, 11]. Zooming in on a dataset incurs a noticeable performance penalty for our method as it is highly fillrate-bound.

In general, performance of tessellation is more strongly coupled to dataset size. Our method scales favorably in this regard, but overdraw is a major performance factor even for smaller-sized data, as evidenced by the anomaly exposed by dataset *HotRoom*, which is made up of segments much smaller in length than their average tube radius, causing significant silhouette overlap within a tube (even the spheres at the bottom consist of dozens of segments each). In addition, we noticed a considerable impact of the geometry stages for very large datasets, suggesting alternative means of generating silhouettes are worth investigating. Irrespective of these considerations, our method achieved fully interactive frame rates for all five datasets.

5 Conclusion and Outlook

We presented an efficient algorithm for raycasting Hermite spline tubes. While initial results are promising, we identified many possible improvements and directions for further research.

For one, cubics are a third-order approximant, opening up the opportunity for drastic data reduction. We envision a pre-processor that automatically merges segments, observing a user-defined error bound. GPU-side tessellation could then be used to generate more complex silhouette geometry if overall curvature of a segment makes tight fitting with a single box impossible.

Complex datasets incur a lot of overdraw due to occlusion. While existing strategies can be used to tackle this problem in a rasterizer system, we think that with an efficient intersection routine available, realtime raytracing (e.g. using NVIDIA RTX) is a promising alternative for rendering large numbers of Hermite spline tubes.

Acknowledgments

This work has received funding from DFG through TRR 248 (grant 389792660) and the two Clusters of Excellence CeTI (EXC2050/1 grant 390696704) and PoL (EXC2068 grant 390729961).

REFERENCES

[1] G. Agin and T. O. Binford. Computer description of curved objects. *IEEE Transactions on Computers*, C-25(4):439–449, 1976. doi: 10.1109/TC.1976.1674626

[2] W. F. Bronsvoort and F. Klok. Ray tracing generalized cylinders. *ACM Trans. Graph.*, 4(4):291–303, Oct. 1985. doi: 10.1145/6116.6118

[3] J. D. Burden, R. L. Faires, and A. M. Burden. *Numerical Analysis*. Cengage Learning, Clifton Park, NY, USA, 10 ed., 2015.

[4] R. Fraedrich and R. Westermann. Motion visualization in large particle simulations. In P. C. Wong, D. L. Kao, M. C. Hao, C. Chen, R. Kosara, M. A. Livingston, J. Park, and I. Roberts, eds., *Visualization and Data Analysis 2012*, vol. 8294, pp. 274 – 285. International Society for Optics and Photonics, SPIE, 2012. doi: 10.1117/12.904668

[5] M. Han, I. Wald, W. Usher, Q. Wu, F. Wang, V. Pascucci, C. D. Hansen, and C. R. Johnson. Ray tracing generalized tube primitives: Method and applications. *Computer Graphics Forum*, 38(3):467–478, jul 2019. doi: 10.1111/cgf.13703

[6] J. Hart. Sphere Tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1996. doi: 10.1007/s003710050084

[7] M. Kanzler, M. Rautenhaus, and R. Westermann. A voxel-based rendering pipeline for large 3D line sets. *IEEE Transactions on Visualization and Computer Graphics*, 25(7):2378–2391, July 2019. doi: 10.1109/TVCG.2018.2834372

[8] D. Merhof, M. Sonntag, F. Enders, C. Nimsky, P. Hastreiter, and G. Greiner. Hybrid visualization for white matter tracts using triangle strips and point sprites. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1181–1188, 2006. doi: 10.1109/TVCG.2006.151

[9] M. Meuschke, S. Oeltze-Jafra, O. Beuing, B. Preim, and K. Lawonn. Classification of blood flow patterns in cerebral aneurysms. *IEEE Transactions on Visualization and Computer Graphics*, 25(7):2404–2418, 2019.

[10] 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. doi: 10.1080/2165347X.2012.659610

[11] G. Reina, K. Bidmon, F. Enders, P. Hastreiter, and T. Ertl. GPU-based hyperstreamlines for diffusion tensor imaging. In B. S. Santos, T. Ertl, and K. Joy, eds., *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*. The Eurographics Association, 2006. doi: 10.2312/VisSym/EuroVis06/035-042

[12] C. Stoll, S. Gumhold, and H. P. Seidel. Visualization with stylized line primitives. In *VIS' 05. IEEE Visualization*, pp. 695–702, 2005.

[13] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4), July 2014. doi: 10.1145/2601097.2601199

[14] 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*, VIS '96, p. 107ff. IEEE Computer Society Press, Washington, DC, USA, 1996. doi: 10.5555/244979.245023