EUROGRAPHICS 2015 / O. Sorkine-Hornung and M. Wimmer
(Guest Editors)

Volume 34 (2015), Number 2

Interactive Rendering of Spline Tubes

Mirko Salm

Dresden University of Technology, Germany
Chair of Computer Graphics and Visualization

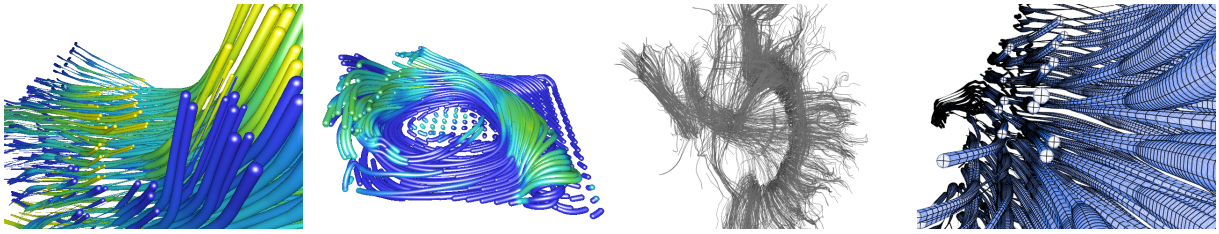

Figure 1: Rendering results of the techniques presented in this paper. The two left most images were generated by the ray casting method while the images to the right were rendered using the tessellation-based approach.

Abstract

Interactive visualizations of tube-based structures find application in various scientific fields. In physics and medicine, for example, they can provide effective means for the exploration of vector field data and white matter tract models. However, rendering tube-based scenes of high complexity requires specialized algorithms to achieve interactive frame rates. This paper presents two such tube rendering approaches (see figure 1). The first technique is tessellation-based and uses a dynamic geometry refinement scheme to maintain qualitatively consistent tube approximations. The second method, in contrast, directly ray casts individual tube segments using a numerical root finding strategy to solve the underlying intersection problem. Advantages and shortcomings of both strategies are discussed and potential improvements are proposed.

1. Introduction

Tubes provide a handy primitive to model a wide variety of natural and abstract structures ranging from white matter tracts [MSE*06] and blood vessels over trajectories of physical particles to stream lines [ZSH96] in vector fields. This makes them attractive for scientific visualizations across various disciplines like math, physics, chemistry, biology, and medicine. Visualizing complex structures at interactive frame rates using tubes requires algorithms that are capable of efficiently rendering those in large numbers. This work presents two such rendering approaches. In particular, they are specialized in rendering a particular kind of tube hereafter referred to as *spline tube*. A spline tube is in the following defined to be a generalized cylinder [AB76] with circular cross-section of varying radius and an axis that follows a spline curve. The first of the presented techniques starts with a coarse world space tessellation of each spline tube. This base tessellation is then adaptively refined at render time using the tessellation unit of the graphics pipeline. The second approach, in contrast, uses ray casting. It computes the intersection points of camera rays and tubes directly and rasterizes only bounding geometry to generate relevant fragments. Since no closed form solution for the intersections exists, a numerical root finding strategy is applied. Both methods are profiled with regard to memory consumption and frame time for various data sets, viewport resolutions, and camera positions. The results are subsequently discussed and the preferred use cases for both approaches are outlined. Potential improvements and extensions are also considered. The main contribution of this work is the proposed ray casting method for quadratic spline tubes and the evaluation of its potential advantages over the more traditional tessellation-based approach.

2. Related Work

Generalized cylinders were originally introduced in [AB76]. In contrast to a regular cylinder, the cross-section of a generalized cylinder can be arbitrarily formed and its axis is described by a space curve. *Spline tubes* form a handy subset of generalized cylinders. Their axes follow spline curves while their cross-sections are restricted to be circular. A

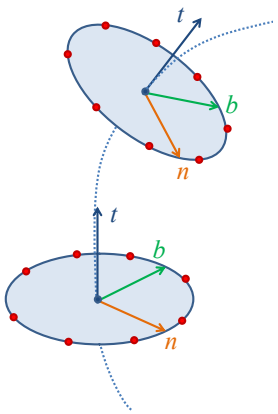

Figure 2: *Cross-section frames along a curved tube axis (dashed line). Each cross-section frame is defined by three vectors forming an orthogonal basis* (t,n,b) *where t corresponds to the tangent of the axis at the considered point. The vectors n and b are used to compute the positions of the tube mesh vertices (red dots)*

common problem often associated with the tessellation of generalized cylinders is that of choosing appropriate *cross-section frames* along the tube axis (see figure 2). Cross-section frames are needed to describe the translation of the vertices from the tube axis to their world space positions. A compilation of several approaches to compute cross-section frames along space curves can be found in [**?**]. Most of the outlined methods use iterative processes to propagate a given frame from the beginning to the end of the curve. Unlike those, the *Frenet frame* [Wikb] can be computed analytically, which allows it to be evaluated efficiently in parallel. However, due to its reliance on the curvature of the tube axis to always be non-zero, it can not be guaranteed to behave well in all cases. The general approach of tessellating a given scene on the CPU and sending the geometry to the GPU for rendering can quickly become bottlenecked by the induced bandwidth demands. To circumvent this problem, all techniques reviewed in the following attempt to transfer only the minimal amount of data necessary to construct the tube geometry at render time. This strategy effectively trades reduced bandwidth demand for processing time.

The approach described in [NvdP98] uses a dynamic screen space tessellation scheme to render spline tubes. Due to the screen space nature of the algorithm, the tube primitives are called *paintstrokes*. The lengthwise subdivision of each tube segment is controlled by several constraints to provide a consistent tessellation quality. The screen space curvature of the axis as well as the varying radius are considered. Additionally, segments are split at inflection points to ensure that the assumptions used to derive the underlying constraints hold. However, problems due to high screen space curvature can still arise and require special handling. For this purpose, ill-formed polygons are split in equivalent pairs of triangles. The quality level of the cross-section subdivision is manually set by the user according to requirements of the use case and is not dynamically adjusted by the algorithm. The surface normals are approximated following a derivation based on infinitesimal right circular conical segments. This model assumes the axis tangents to remain constant along a small curve segment, which is not true in general. The paintstrokes renderer uses an A-Buffer to implement antialiasing. It follows the original implementation introduced in [Car84] albeit with modifications regarding the generation and blending of fragments.

Another tessellation-based spline tube rendering approach is described in [NVR*12]. Unlike the paintstrokes algorithm, it does not perform tessellation in screen space but in world space. Similar to the tessellation technique presented later in this work, it uses the tessellation unit of the GPU to implement a dynamic LOD scheme. However, their refinement strategy differs in that it only affects the cross-section but not the tube resolution along the axis. The refinement criterion is based on a squared distance to the camera and is not viewport size dependent. A pre-processing step adaptively subdivides the tube axis in regions of high curvature to improve the consistency of the tessellation quality. For rendering purposes, the shaders are provided with a list of tangent frames, radii and points defining the Catmull-Rom spline based tube axis. This information is then used to transform a quad-based grid into tube segments at render time. In addition to using MSAA, the screen space radii of the tubes are clamped to a lower bound to reduce aliasing in the form of popping artifacts.

Grisoni and Marchal, too, present a tessellation-based approach with curvature dependent adaptive sampling [GM03]. Their algorithm attempts to guarantee a consistently high frame rate by dynamically adjusting the tessellation quality using a simple energy minimization scheme coupled to a target frame time. The screen space quality of the geometry is not taken into consideration by this process. Their approach also differs in that it supports arbitrarily formed cross-sections and is not limited to circular ones. Furthermore, instead of pre-computing local cross-section frames along the axis, linear blend skinning [Web00] is used to perform the transformation from straight to bent cylinders. While this method drastically reduces the number of required cross-section frames, unfortunately no in-depth analysis of possible artifacts and limitations resulting from the use of this technique is provided by the authors.

In contrast to the approaches outlined above, [RBE*06] apply a ray casting technique to render generalized cylinders with twisted, elliptical cross-sections. Since no analytical ray-segment intersection exists for this case, they use distance bound ray casting [Har94] to find approximate intersection points iteratively. Their profiling results show that compared to a simple tessellation-based approach the ray casting is significantly slower for small and medium-sized data sets. Complex scenes, however, benefit from the reduced load on the graphics bus and vertex load. In this case, choosing a small viewport can result in a favorable fragments-to-vertices ratio that allows the ray casting to perform significantly faster then the tessellation-based approach.

A hybrid spline tube rendering approach combining a coarse CPU-side tessellation with a simple ray casting performed by a fragment shader is presented in [SGpSP05]. While the applied tessellation scheme is similar to the paintstrokes algorithm, significant improvements with regard to the shading quality were achieved due to the higher quality surface representation produced by the ray casting technique.

In [KHK*09] the application of interval arithmetic and affine arithmetic to interactive root finding in the context of ray casting arbitrary implicit functions is explored. An optimized, stack-driven algorithm using SIMD instructions, as well as a stack-less approach that can be efficiently implemented on the GPU, is presented. The performance characteristics of the presented approaches is evaluated for a wide variety of functions.

3. Spline tubes rendering

Commonalities and terminology A spline tube is defined by a sequence of *nodes* that provide position, radius, and color values as well as derivatives. The nodes are assumed to be equidistant positioned in curve parameter space. This implies, for example, particle data to be sampled at constant time intervals. The shape and coloring of a tube in between the nodes is determined by cubic or, as in the case of the ray casting approach, piecewise quadratic interpolation. For this purpose, spline segments for position, radius, and color are constructed from the respective values and derivatives stored at the nodes at render time. Consequently, a pair of subsequent nodes provides the information required to describe the appearance of the tube segment it delimits. The position spline defines a curve in world space that forms the generalized tube axis. A tangent spline can be derived from the position spline by differentiation with regard to the curve parameter. Evaluating the position and tangent splines returns *curve positions* and *curve tangents*, respectively. The *cross-section plane* is defined to be a plane centered at a point of interest on a given tube axis and of which the normal coincides with the axis tangent at this point.

During rendering, per-segment and per-node data is stored in a GPU buffer called the *tubes buffer* (see figure 3). Its particular type is API and implementation dependent (the OpenGL-based reference implementation used to profile both approaches uses a uniform buffer). Each segment accesses its *segment data*, which is kept at the top of the tubes buffer, through its *segment ID*. The segment ID is derived from an instance ID, which is provided by the graphics API (as SV_InstanceID (D3D) or gl_InstanceID (OpenGL)) and sequentially enumerates all instances of an instanced draw call. For both approaches, the segment data at least consists of a memory offset to the *node data* of the two nodes defining the segment. This scheme, although requiring an indirection to access the node data, allows to render multiple tubes using a single instanced draw call without having to provide redundant node data (essentially storing the data of both defining nodes for each segment) or to draw degenerated dummy segments in between subsequent tubes, which would be necessary to ensure that the instance IDs can be directly used to access the node data. Especially in the case of the tessellation-based approach, the relative overhead of processing dummy geometry would be large for data sets containing many short tubes.

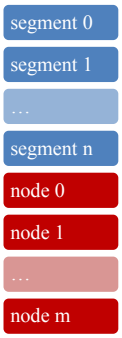

Figure 3: *The basic data layout of a tubes buffer containing n segments and m nodes. The segment data is stored at the beginning of the buffer while the node data is kept at the end.*

3.1. Tessellation-based approach

Base tessellation The base geometry provides a cylinder mesh that is modeled as a stack of vertex rings forming a quad-based topology (see figure 4). The axis of the cylinder is aligned with the z-axis in cylinder space. The z components of the cylinder space vertex positions lie in the interval $[0..1]$. They provide the curve parameters, which are used to sample the set of splines associated with the tube segment. The radius of the cylinder is set to 1 restricting the x- and y-components to the interval $[-1..1]$. First, a vertex shader is used to transform the cylinder mesh to a curved tube segment with varying radius. For a given vertex, the position, tangent, and radius splines are constructed and subsequently evaluated using the z-coordinate of the local position

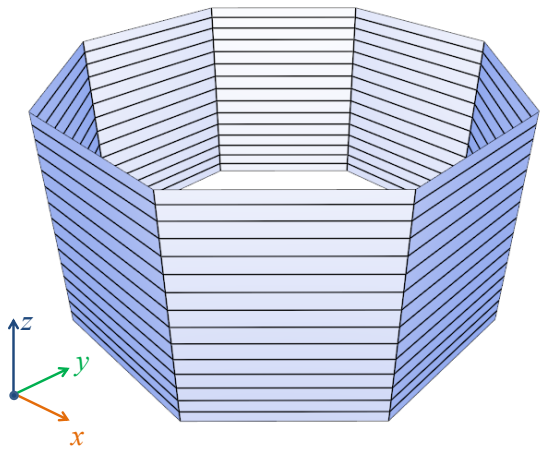

Figure 4: *The cylindrical base geometry from which each tube segment is modeled by the vertex shader.*

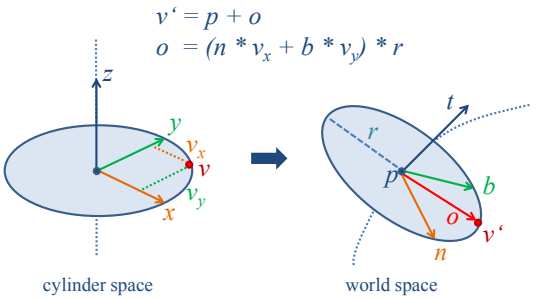

Figure 5: *The world space vertex position v' is computed by translating the curve position p along the offset vector o.*

- *v..local vertex position in cylinder space*
- *v'..global vertex position in world space*
- *o..offset vector*
- *p..curve position*
- *r..cross-section radius*
- *(t,n,b)..cross-section frame*

as curve parameter. The world space position of the vertex is then computed by a translation from the curve position along an offset vector that is orthogonal to the tangent and of which the length equals the radius. This offset vector is computed by transforming the local cylinder space vertex position to world space using an orthogonal cross-section frame (see figure 5).

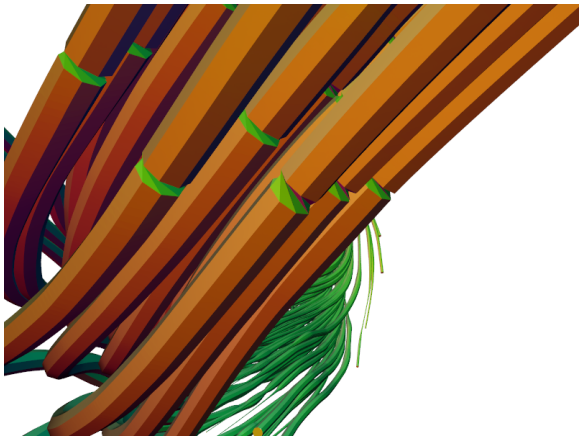

Figure 6: *Artifacts in the base tessellation as a result of successive vertex rings being twisted too strongly (flat-shading is applied to highlight the problem).*

The cross-section frame is constructed by first picking an arbitrary vector orthogonal to the tangent using one of numerous existing strategies [Sam] [Sta] [Wikb]. The remaining vector is then constructed by computing the cross product of the first vector and the normalized tangent. All three vectors together form the cross-section frame. However, none of these strategies is free of discontinuities and consequently can lead to bad topology in the form of strongly twisted consecutive vertex rings (see figure 6). Therefore, instead of calculating the orthogonal vector for a point of interest on the curve independently from all other orthogonal vectors, an iterative approach that propagates an orthogonal vector from start to end of a given curve is applied (similar to those presented in [**?**]). This process is performed as a preprocessing step on the CPU since it can not be directly parallelized. The resulting vectors are provided to the shaders as segment data stored in the tubes buffer. To ensure all tube segments fit seamlessly together, the process is performed for each tube as a whole from start to end. The algorithm starts by picking an arbitrary orthogonal vector at the beginning of the considered tube. This vector is then repeatedly projected onto the cross-section plane associated with the next vertex ring and re-normalized. In the case that the angle between two successive tangents is close to 90 degree, additional sub-steps are inserted to prevent the projected vector from degenerating. For each vertex, the vector describing the offset from curve position to world space position is then calculated by a sum of the two orthogonal vectors

spaning the cross-section plane weighted by the x- and y-components of the local vertex position and scaled by the tube radius (see figure 5). At last, a normal is computed for

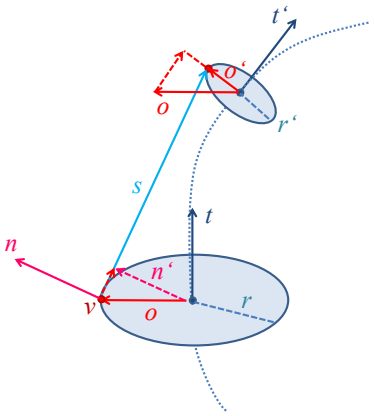

Figure 7: *Depiction of the normal computation scheme used by the tessellation-based approach. For a given vertex v, the normal n is computed by projecting the corresponding offset vector o onto the plane of a numerically determined surface tangent s followed by a normalization.*

- *v..vertex position*
- *n..vertex normal*
- *n'..unnormalized vertex normal*
- *o..offset vector*
- *r..cross-section radius*
- *t..curve tangent*
- *s..surface tangent*

each vertex. Since the tube radius varies along the curve, it is in general not possible to directly use the normalized offset vector as such. Instead, a surface tangent pointing in curve direction is computed using a central difference. The offset vector is then projected onto the plane of this tangent. Normalizing the resulting vector gives the correct normal (see figure 7).

Tessellation refinement and shading To improve the quality of the base tessellation generated by the vertex shader, the geometry is refined by the tessellation unit. First, the subdivision quality is determined in the tessellation control shader. Afterwards, the tessellation evaluation shader, similar to the vertex shader, transforms the then refined cylinder geometry again to a curved tube.

Before the tessellation control shader computes the tessellation levels for a given quad patch, it first performs back-patch culling. A patch is culled iff all of its vertex normals are oriented away from the camera. This provides a more conservative culling criterion than using the patch normal and works well in practice even though in general it is not guaranteed that all of the later generated triangles would be back-facing as well. To determine the inner and outer tessellation levels for the quad patches, the tessellation control shader applies two criteria. The first criterion improves the overall quality of the tessellation by subdividing edges to ensure their screen space lengths (measured in pixels) do not exceed a given maximum. The second criterion aims to specifically increase the resolution of the silhouettes. For this purpose a decreasingly smaller maximal edge length for edges closer to the silhouette is chosen. How close an edge lies to the silhouette is measured by the flatness of the angles between the vertex normals of the given edge and the vertex-to-camera rays. The outer tessellation level corresponding to each edge of a patch is then determined by choosing the maximal value resulting from both criteria. Both values are individually clamped to upper bounds beforehand. To provide an increasingly finer tessellation towards silhouettes, the upper bound of the value resulting from the second criterion is set to be larger then the one of the first. The inner tessellation level controlling the horizontal subdivision of the patch is set to the maximum of the pair of outer levels that control the refinement of the vertical edges. In the same manner, the vertical inner level equals the maximum of the two outer tessellation levels of the horizontal edges.

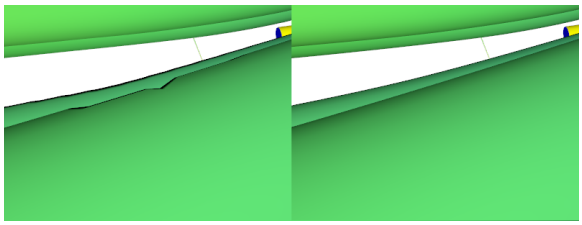

Figure 8: *Left: notchy silhouettes as a result of edges pointing towards the camera getting insufficiently subdivided due to their short lengths in screen space. Right: using a refinement criterion not coupled to the screen space edge length for edges close to the silhouette resolves the artifacts.*

One problem that can arise due to varying tessellation levels are notchy silhouettes (see figure 8). In these cases, the refined geometry actually provides a visually less convincing approximation of the tube than the base tessellation. To prevent these artifacts the silhouette criterion does not directly use screen space edge lengths like the first criterion does. Instead, world space edge lengths are calculated, divided by the w-components of the screen space positions of the edge midpoints and multiplied by the maximum of the viewport size dimensions. This leads to a more isotropic per-quad subdivision close to the silhouette by preventing large differences in tessellation factors of adjacent edges. Furthermore, using a quad patch topology for the base tessellation, too, turned out to be essential in order to prevent this kind of artifact. In comparison to triangle patches, using quad patches additionally reduces the number of redundant vertex shader invocations being issued, thus improving performance. Another more difficult to resolve kind of artifact are bulbs re-

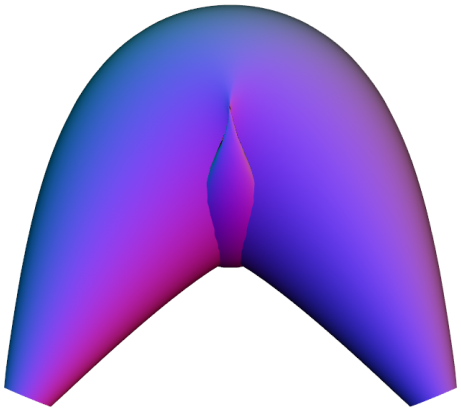

Figure 9: *Bulb artifact resulting from an unfavorable combination of large curvature, radius, and radius derivative (back-face culling is disabled for demonstration purposes).*

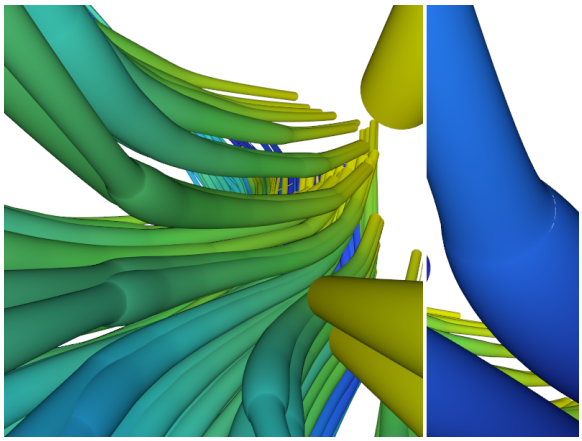

Figure 10: *Sharp radius changes modeled by combining multiple tubes. Right: differences in the normals along the seams of two successive tubes lead to inconsistent tessellation factors, producing non-watertight results.*

sulting from self-intersections (see figure 9). These can occur when the maxima of the radius and position splines are in close proximity. For the majority of cases, the aesthetical benefit of detecting and discarding the affected fragments would presumably be outweighed by the cost of doing so. The test data sets, for example, do not contain any tubes that exhibit bulb artifacts.

Following the tessellation refinement, the new cylinder geometry is again transformed into a tube segment. This is done inside the tessellation evaluation shader. In principle, the same approach already used by the vertex shader can be applied. However, since the curve parameter of each new vertex is not restricted to coincide with those of any of the original vertex rings, none of the pre-computed orthogonal vectors can be directly used to construct the cross-section frames required to compute the world space positions of the vertices. Each orthogonal vector is therefore calculated by linear interpolation of two successive pre-computed vectors read from the tubes buffer, followed by a projection onto the cross-section plane of the considered vertex and a normalization to unit length.

At last, the fragment shader evaluates the color spline, calculates a per-pixel normal as well as a world space position, and from these computes a basic shading.

End caps Until now, the tubes have been assumed to be open at their endings. Adding caps to close them up requires some additional considerations. First of all, the base geometry of the tube segments is extended by additional quads closing up their beginnings and endings. In all cases where the cap geometry of a segment lies inside a tube it is culled by the tessellation control shader. This is also the case for end caps in between too consecutive tubes that are connected to form a single tube. Connecting multiple tubes to a single one is useful to support C^1 discontinuties without having to extent the framework (see figure 10 left). Due to possible differences in the end seam normals of connected tubes, their tessellation levels might differ, which can result in non-watertight geometry (see figure 10 right). Fixing these would require to compute continuous per-vertex normals to ensure that the same tessellation levels are used for the edges shared by the affected tube segments.

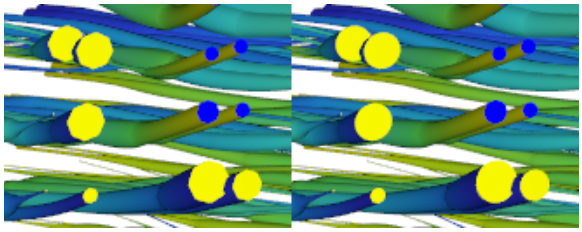

Figure 11: *Left: the seam edges are not sufficiently refined to achieve a smooth appearance (the colors of the end caps are inverted to emphasize the problem). Right: decreasing the maximal screen space edge length for seam edges resolves the problem.*

In general, cap seam edges need to be finer tessellated then the body of the segment to provide a smooth appearance (see figure 11). For this purpose, the maximal screen space edge length is set to a smaller value for the cap seams. The geometry of the interior of the caps on the other hand, being flat in nature, is only minimally refined.

3.2. Raycasting-based appproach

Deriving an intersection routine that allows to render spline tubes directly requires a parametrized ray-segment intersec-

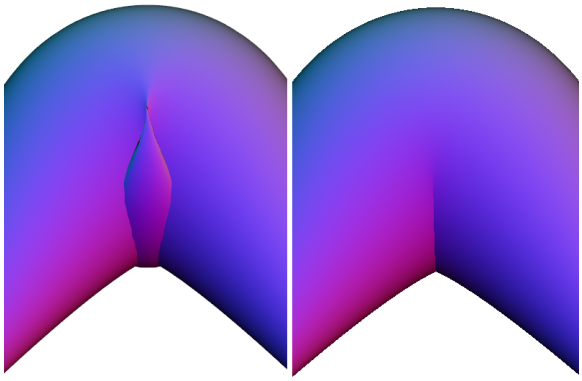

Figure 12: *Left: bulb artifact resulting from the use of a disk-based tube model. Right: a sphere-based model does not produce bulb artifacts.*

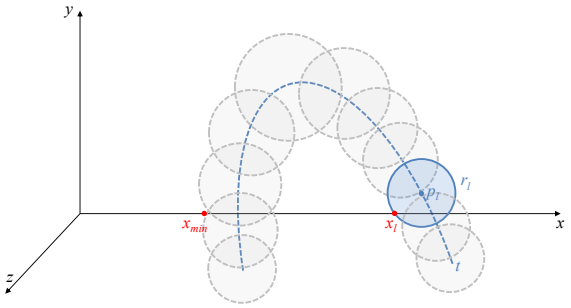

Figure 13: *A sphere in ray space is intersected with the x-axis (the ray), resulting in an intersection point x_l. The position p_l and the radius r_l of the sphere are parametrized by the curve parameter l of the considered tube segment t. The first ray-tube segment intersection x_{min} is found by minimizing the parametrized ray-sphere intersection with regard to the curve parameter.*

tion to be formulated first. However, without a compact implicit form for the tube segment at hand, the common approach of plugging in the ray equation and solving for the ray parameter can not be applied directly. Instead, the intersection of a ray a with geometric primitive of which the position and radius is parametrized by polynomials serves as mathematical foundation. Depending on the underlying primitive, not only do the complexities of the resulting equations differ, but also the geometrical properties of the rendered tube segment. Therefore, both criteria need to be considered. The tessellation approach described previously followed a disk-based model. Unsurprisingly, a ray casting approach based on disks, too, suffers from bulb artifacts even though no problems due to inverted geometry occur. Furthermore, because of the bulb artifacts, up to 6 and 10 intersections (and consequently roots of an equation) need to be found for quadratic and cubic splines, respectively, making the underlying root search too costly in both cases. In contrast, a sphere-based approach does not suffer from bulb artifacts (see figure 12), while, at least for quadratic splines, allows to find the roots with reasonable effort. Therefore, every tube segment attribute is not described by a cubic spline segment, as is the case for the tessellation-based technique, but by two or more quadratic spline *sub-segments*. It is important to note that this approximation introduces C^2 discontinuities on the tube surfaces. While these are not problematic with respect to the quality of the silhouettes, they can become apparent as sudden changes in the brightness of the shading.

Intersection calculation and shading The derivation and evaluation of the intersection routine is simplified by first transforming the considered tube segment and ray to *ray space*. There, the ray itself coincides with the x-axis. An equation returning the ray parameter of a ray-sphere intersection for a given curve parameter is derived as follows (see figure 13). First, the radius R of the circle corresponding to the intersection between a sphere (defined by a position (x,y,z) and a radius r) and the xy-plane is computed:

$$R(x,y,z,r) = \sqrt{r^2 - z^2}. \tag{1}$$

The intersection of this circle with the x-axis is

$$s(x,y,z,r) = x \pm \sqrt{R(x,y,z,r)^2 - y^2}. \tag{2}$$

Inserting (1) into (2) gives

$$s(x,y,z,r) = x \pm \sqrt{r^2 - y^2 - z^2}. \tag{3}$$

Choosing the front side intersection and plugging in the position and radius splines of a tube segment t, parametrized by the curve parameter l, results in

$$f_t(l) = t_x(l) - \sqrt{t_r(l)^2 - t_y(l)^2 - t_z(l)^2}. \tag{4}$$

The minima of every real interval of f_t corresponds to a ray intersection with the tube segment surface. For convenience, $polyA_t$ and $polyB_t$ are declared so as to denote the polynomial terms in front and under the square root, respectively:

$$polyA_t(l) = t_x(l), \tag{5}$$

$$polyB_t(l) = t_r(l)^2 - t_y(l)^2 - t_z(l)^2. \tag{6}$$

Replacing the polynomials in (4) with (5) and (6) gives

$$f_t(l) = polyA_t(l) - \sqrt{polyB_t(l)}. \tag{7}$$

At last, f_t is differentiated with regard to the curve parameter l, resulting in

$$f_t'(l) = polyA_t'(l) - \frac{1}{2}\frac{polyB_t'(l)}{\sqrt{polyB_t(l)}}. \tag{8}$$

The roots of f_t' correspond to potential ray-segment intersections. The challenge is therefore to derive an efficient root finding strategy that can be applied to f_t'. Once the roots are found, f_t is evaluated for all candidates and the intersection closest to the ray starting point in positive direction is identified as the result.

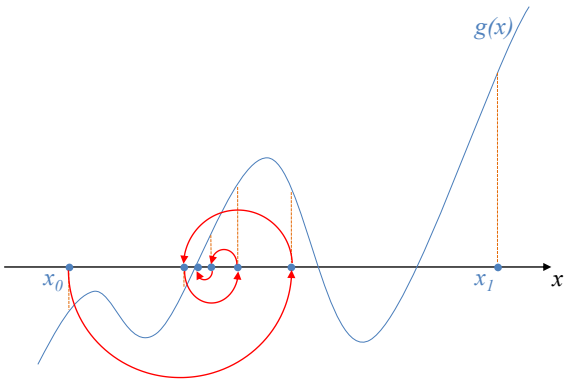

Figure 14: *A root of the function g on the interval* $[x_0..x_1]$ *is approached by performing 5 iterations (red arrows) of a binary search.*

The first step of the root finding strategy is to determine the intervals on which f_t' is real. These in turn correspond to the intervals on which $polyB_t$ is positive. Consequently, the roots of $polyB_t$ need to be found first. Since $polyB_t$ is of degree 4, closed from solutions returning the roots exist [HE95]. However, due to the large amount of arithmetic operations involved in computing an analytical solution, the result can suffer from numerical inaccuracies. Instead, the rather simplistic, numerical root finding strategy depicted in the following is applied to $polyB_t$. Given a function g that is continuous on an interval $[a..b]$, where g evaluates to values of opposite signs for a x_0 and a x_1, a root lying in between x_0 and x_1 can be iteratively approached using a binary search, also know as bisection method [Wika] (see figure 14). Strategies with better theoretical convergence like Newton's method [**?**] and Halley's method [**?**] exist. However, they are less reliable in practice since their actual convergence can be worse then the one of the binary search while the per-iteration cost of the latter is comparably small. Since any root of g on the interval $[a..b]$ lies in between two extrema (or between an interval bound and an extremum, or between both interval bounds) with values of opposite signs, knowing the extrema of g, it is possible to apply the binary search to each of the corresponding intervals to find all roots. Determining the extrema of g, itself, can be formulated as a root finding problem on the derivative of g. Therefore, starting with the n-th derivative of g for which it is known that only a single root on the interval of interest exists, it is possible to iteratively apply the binary root search to determine the search intervals for the next less derived version g. If g is a polynomial 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 complexity of the algorithm grows quadratically with the degree of g when ignoring varying evaluation complexities for derivatives of different degrees. More precisely, for a polynomial of degree n, the number of necessary binary root searches equals $(n^2+n)/2$. However, three binary searches can be saved by computing the roots analytically as soon as the derivative becomes quadratic. Therefore, the roots of $polyB_t$, which is of degree 4, can be found using 7 binary searches. For a cubic spline, $polyB_t$ would be of degree 6. Finding the roots in this case would require 18 binary root searches, turning out rather expensive (although potentially still affordable for offline rendering applications). Knowing the roots of $polyB_t$, the same strategy can be applied to the intervals on which f_t' is real. Although the assumption that the number of maxima and roots decreases with each differentiation step does not hold for general functions, this did not turn out to pose a problem in the case of f_t'. Empirical analyses of various border cases suggest that for any configuration every interval contains either 1 or 3 roots (see figures 15 and 16). In the case of 3 roots, only the outer two are of interest since the middle one corresponds to a local maxima of f_t. Consequently, up to 5 binary searches per real interval are necessary to find all relevant roots of f_t'. However, in most situations every interval contains only a single root. These cases are implicitly detected by checking for opposite signs before performing each binary search and therefore do not induce a major performance hit. In the last step, f_t is evaluated at the found roots. The smaller of the resulting ray parameters is then used to calculate the world space position of the first intersection as well as the corresponding normal, which equals the normalized vector pointing from the curve position to the intersection point (this has only been empirically verified by comparison with numerically computed normals). The final shading is computed just like for the tessellation approach. Additionally, a per-fragment depth is returned for depth testing purposes.

Rendering The vertex shader computes a bounding box for each quadratic tube segment and transforms the cube geometry provided by an instanced draw call accordingly (see figure 17). The x-axis of the bounding box equals the direction

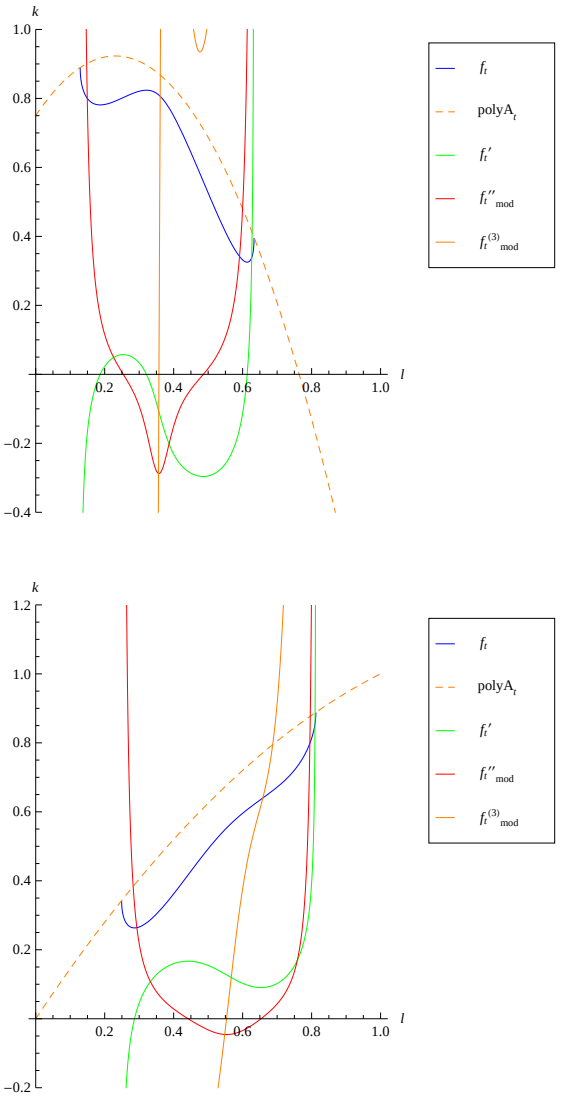

Figure 15: *Resulting graphs of functions relevant to the root search on the real intervals of f_t. The 'mod'-subscript indicates that the function has been modified in a root-preserving fashion. The presented cases occur when a ray intersects a segment in a region where it bends.*

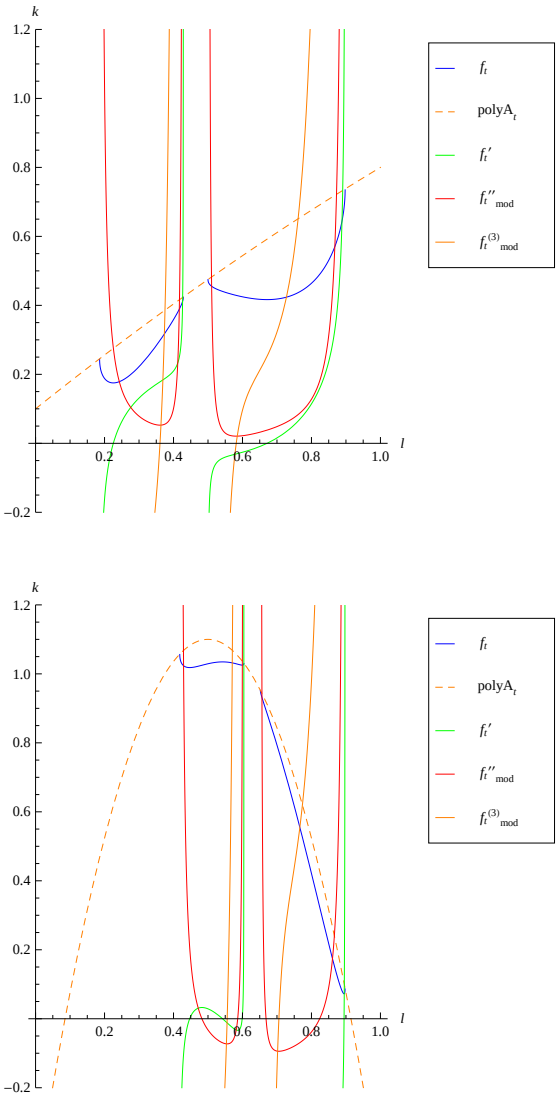

Figure 16: *Resulting graphs of functions relevant to the root search on the real intervals of f_t. The 'mod'-subscript indicates that the function has been modified in a root-preserving fashion. Top: a common case; both intersections could theoretically be found by performing 2 binary searches. Bottom: a border case with 3 relevant roots. (Moving the real intervals close enough together so that they merge does not produce an additional root.)*

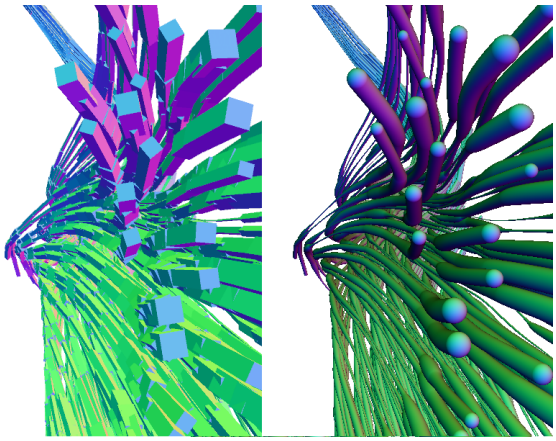

Figure 17: *Left: the bounding boxes used to generate the fragments for which the ray-tube segment intersection computations are performed. Right: the ray casted tube geometry.*

from start to end point of the sub-segment while the vector pointing from start point to control point serves, after being projected into the plane of the x-axis, as y-axis. The z-axis is subsequently determined by a cross product of the two known axes. The extent in the direction of each axis is computed from the extrema of quadratic functions derived from the respective quadratic position and radius splines. In practice, this boils down to solving linear equations. Degenerated cases due to collinear or overlapping points are considered accordingly. The fragment shader, eventually, performs the intersection and shading computations described above.

3.3. Results and Discussion

Frame times for both approaches where measured for three different data sets: *Fibers*, *Fibers2*, and *VecTubes* (see figure 18). They can be coarsely classified as small, large, and medium sized, respectively. Figure 19 provides some basic information regarding the geometry that the data sets contain. Comparing the tubes buffer sizes of both approaches shows that for the ray casting method only about 0.6 times as much data needs to be transferred to the GPU as for the tessellation-based technique. As is later shown, this difference in bandwidth demand has a significant impact on the frame times. The testing system uses an Intel® Core™ 2 Quad Q9550 2,83GHz and a NVIDIA® GeForce™ GTX 560 Ti. The implementation uses OpenGL. For the tessellation-based approach, a cylinder mesh with 16 vertex rings and 8 vertices per vertex ring is used as base geometry. The maximal tessellation factors are set to 5 and 7 for the general refinement criterion and the silhouette criterion, respectively. The ray casting method approximates each cubic segment using 2 quadratic sub-segments and performs 10 iteration per binary search. While using more sub-segments

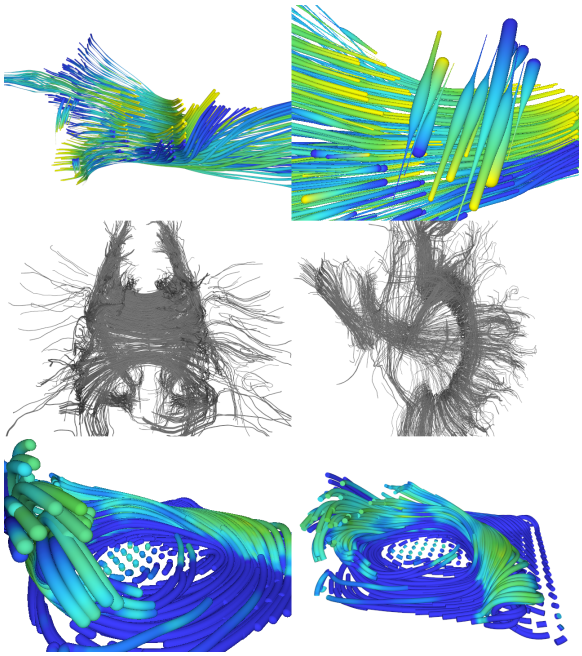

Figure 18: *The data sets used to profile the tessellation-based approach and the ray casting technique. From top to bottom: Fibers, Fibers2, VecTubes.*

	Fibers	Fibers2	VecTubes
tubes count	241	1701	722
segment count	3530	114489	30948
node count	3771	116190	31670
tubes buffer size (Tess)	0.338 MB	10.628 MB	2.888 MB
tubes buffer size (RC)	0.215 MB	6.621 MB	1.805 MB
tubes buffer size ratio (RC/Tess)	0.635	0.623	0.625

Figure 19: *Statistics for the Fibers, Fibers2, and VecTubes data sets. The number of segments directly corresponds to the number of base geometry instances and bounding box instances that need to be drawn for the tessellation-based technique and the ray casting approach, respectively.*

is possible, it turned out to not be worthwhile quality- and performance-wise. The tubes buffer is implemented as a uniform buffer. Compared to a shader storage buffer, the uniform buffer turned out to be faster, even though, due to its size limitation, often more draw calls need to be issued. However, this performance characteristic might be hardware architecture dependent. All results are obtained by averaging 500 frame times measured while the camera orbits at a fixed distance around the scene. Outliers are detected using a median filter operating on a list of the last 10 frames. The final frame time is rounded to integer values of milliseconds. This still provides enough precision to identify general trends. Two viewport sizes in combination with two different camera-to-scene distances are used during the profiling process. The small viewport size equals 800x600 pixels, whereas the medium sized viewport measures 1200x768 pixels, which is about twice as large as the small one. The camera-to-scene distances are chosen per data set so as to achieve a medium amount of screen coverage by the geometry for a camera positioned far from the scene as well as a high coverage for a closely positioned camera.

In figure 20 the frame times achieved by the two approaches are compared for all three data sets. Three trends become apparent:

- the frame times of the tessellation-based approach are much more consistent under varying screen coverage and less dependent on the viewport size
- the ray casting technique is overall slower for small and medium sized data sets (Fibers, VecTubes) but significantly faster then the tessellation-based method for large ones (Fibers2)
- the performance of the ray casting approach also strongly depends on the distribution of the geometry in the scene; compare Fibers2 (even distribution) to VecTubes (clustered geometry)

Figure 21 provides insight in how much render time is consumed by individual processing steps. The following can be derived from these measurements:

- about 10 ms are saved for the ray casting method due to the smaller bandwidth demands
- the largest frame time impact for the tessellation-based approach results from the geometry processing while for the ray casting it stems from the intersection computations performed by the fragment shader

The measurements presented in figure 22 further imply that the geometry processing time of the tessellation-based technique is actually not dominated by the dynamic refinement but by the processing of the base geometry itself.

The measured frame time impact of using multisample antialiasing in conjunction with the tessellation-based approach is shown in figure 23. Considering the significantly improved image quality (see figure 24), the performance costs resulting from the use of MSAA appear to be bear-

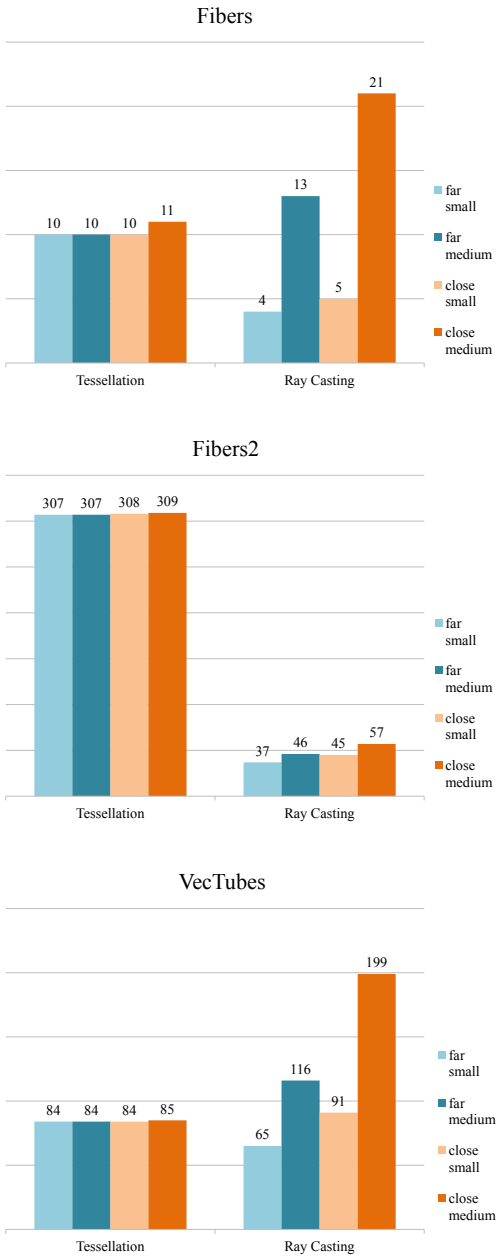

Figure 20: *Comparison of the frame times achieved by the two rendering approaches for the three data sets. The measurements are given in milliseconds.*

- far/close*: camera positioning relative to the scene*
- small/medium*: viewport size (800x600 px/1200x768 px)*

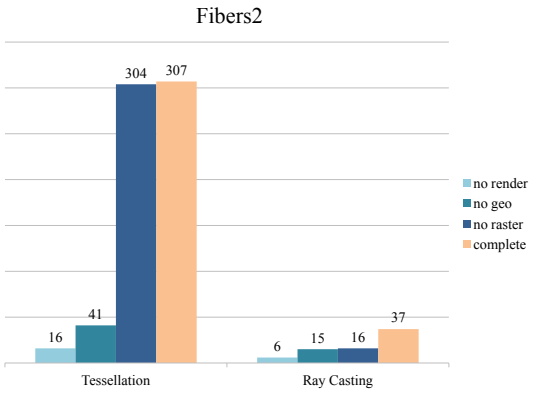

Figure 21: *Comparison of the frame times achieved by the two rendering approaches for the Fibers2 data set. The large camera-to-scene distance and the small viewport size (800x600 px) are used. The measurements are given in milliseconds.*

- no render*: only dummy draw calls are issued (instance count is set to 0); intended to provide insight in the frame time impact of the data transfer from CPU to GPU*
- no geo*: dummy shaders performing minimal work are used; no geometry is rasterized*
- no raster*: the geometry is properly processed but not rasterized*
- complete*: the geometry is properly rendered*

Rasterization of triangles is prevented by negating the z-components of the vertex clip space coordinates returned by the tessellation evaluation shader or vertex shader.

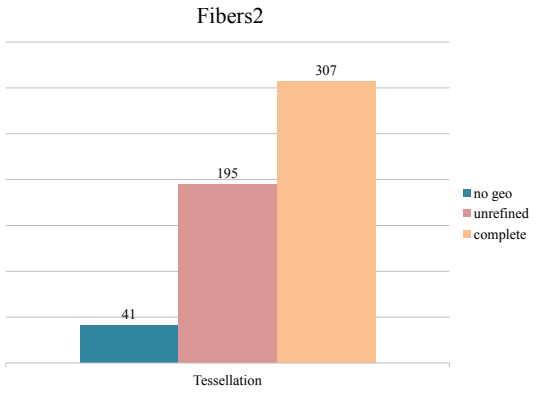

Figure 22: *Comparison of the frame times achieved by the tessellation-based approach for the Fibers2 data set. The large camera-to-scene distance and the small viewport size (800x600 px) are used. The measurements are given in milliseconds.*

- no geo*: dummy shaders performing minimal work are used; no geometry is rasterized*
- unrefined*: the tessellation levels are all set to 1; the geometry is properly rendered*
- complete*: the adaptive, dynamic refinement is used; the geometry is properly rendered*

able. No antialiasing scheme has been implemented for the ray casting technique.

One thing worth noting is that the ray casting technique makes no use of a conservative depth output due to artifacts occuring under OpenGL that could not be resolved. However, an earlier Direct3D-based implementation successfully employed conservative depth with notable performance improvements. Therefore, the frame times measured for the ray casting approach can be considered to be an upper bound on the performance that could be achieved using a functional conservative depth output.

As has been shown, the ray casting method is primarily favorable for rendering large data sets with evenly distributed geometry. For these cases, the tessellation-based technique fails to provide a fluent visualization. However, implementing an additional LOD scheme for the base geometry might improve the performance of the tessellation-based approach considerably. Likewise, the performance of the ray casting method could be improved by dynamically adapting the rendering resolution or by rendering geometry at different resolutions depending on the distance of the geom-

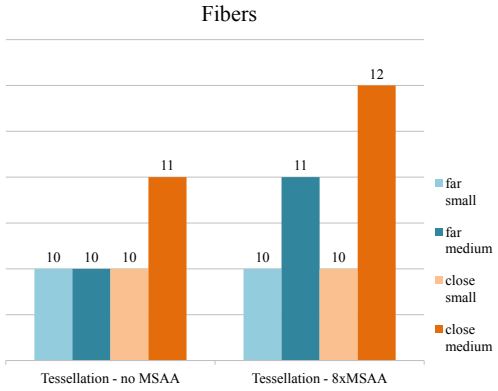

Figure 23: *Comparison of the frame times achieved by the tessellation-based approach for the Fibers data set using multisample antialiasing. The measurements are given in milliseconds.*

- far/close*: camera positioning relative to the scene*
- small/medium*: viewport size (800x600 px/1200x768 px)*

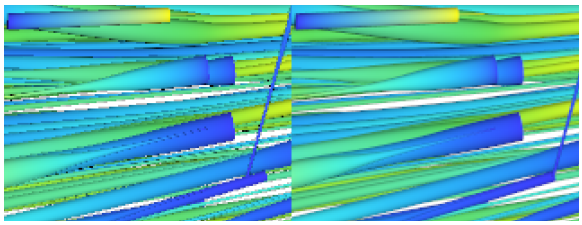

Figure 24: *Renderings of the Fibers data set using the tessellation-based approach in conjunction with no MSAA (left) and 8xMSAA (right).*

etry to the camera. A geometry-aware upsampling process could be applied to the low resolution renderings afterwards. Also, the performance problems arising from geometry clusters could be mitigated using a conservative depth output from the fragment shader. In that regard, both approaches still offer a lot of potential for optimizations. With respect to rendering quality, the ray casting method has the advantage to always produce smooth geometry while a tessellation inevitable generates sharp features at some scale, which can become apparent when rendering geometry from close up. Furthermore, the ray casting approach does not suffer from bulb artifacts like the tessellation-based approach does. However, the C^2 discontinuities resulting from the quadratic approximations can introduce artifacts for the applied shading. The tessellation-based method, on the other hand, does not suffer from this deficit. Extending the tessellation-based technique to support more general tube types with different cross-sections is comparably straightforward. In contrast, for the ray casting method, this is only possible when the desired cross-section can be composed from multiple circular ones, limiting its potential use cases.

4. Future Work

As indicated by the profiling results, extending the tessellation-based technique by a level-of-detail scheme to dynamically simplify the base geometry promises great potential for performance improvements. In that case, a curvature adaptive sampling would probably become necessary to maintain a consistent tube approximation quality for the low-poly LOD levels. The frame time of the ray casting method, on the other side, is dominated by the expensive root searches inside the fragment shader. Therefore, improving the efficiency of the root finding strategy, possibly by evaluating the roots of $polyB_t$ analytically, and reducing the number of performed root searches should decrease the frame time considerably. The latter could be achieved by using tighter bounding geometry and by ray casting at a lower resolution followed by a post-processing pass performing a geometry-aware upsampling. Also, using a conservative output from the fragment shader in conjunction with sorted draw calls would reduce overdraw and could consequently resolve the performance issues that arise when rendering clustered geometry. Reducing the size of the tubes buffer would also help to improve performance. Especially in the case of the tessellation-based approach choosing a more compact format for the orthogonal vectors would lead to a significant reduction in bandwidth demands. Some of the lost precision could then be recovered at render time by re-projecting the vectors onto the planes of their respective tangents followed by a re-normalization. Apart from that, it might also be worthwhile to explore algorithms that would allow to efficiently resolve bulb artifacts and to analytically compute normals for the disk-based tube model. A major disadvantage of the ray casting technique with respect to the render quality is the lack of antialiasing. However, implementing a post-processing antialiasing pass should be relatively straightforward. Essential information to derive an approximate per-fragment geometry coverage is already available in form of the roots of $polyB_t$, which correspond to points where the ray tangentially touches the tube segment. Signed distance field-based ambient occlusion might be another worthwhile extension for the ray casting technique. The implementation could be realized in a deferred shading fashion by rasterizing additional sub-segment bounding volumes of larger scales that subsequently serve as a sort of "negative" lights. On the downside, while potentially providing high-quality ambient occlusion, this approach might be computationally expensive since the signed distance field of each tube sub-segment would need to be evaluated numerically.

5. Conclusion

In this work a novel ray casting-based spline tube rendering technique, as well as tessellation-based technique, was presented. A discussion regarding the performance and quality characteristics of both methods was provided and potential directions for future work were suggested.

References

[AB76] AGIN G. J., BINFORD T. O.: Computer description of curved objects. *IEEE Trans. Comput. 25*, 4 (Apr. 1976), 439–449. URL: http://dx.doi.org/10.1109/TC.1976.1674626, doi:10.1109/TC.1976.1674626. 1, 2

[Blo] BLOOMENTHAL: Calculation of reference frames along a space curve. http://webhome.cs.uvic.ca/~blob/courses/305/notes/pdf/ref-frames.pdf. 2, 4

[Car84] CARPENTER L.: The a -buffer, an antialiased hidden surface method. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 103–108. URL: http://doi.acm.org/10.1145/800031.808585, doi:10.1145/800031.808585. 2

[GM03] GRISONI L., MARCHAL D.: High performance generalized cylinders visualization. In *Shape Modelling International* (2003), pp. 257–263. (ACM Siggraph, Eurographics, and IEEE sponsored). 2

[Har94] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12* (1994), 527–545. 3

[HE95] HERBISON-EVANS D.: Solving quartics and cubics for graphics. http://sydney.edu.au/engineering/it/research/tr/tr487.pdf, 1995. 8

[KHK*09] KNOLL A., HIJAZI Y., KENSLER A., SCHOTT M., HANSEN C., HAGEN H.: Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2008.01189.x. 3

[MSE*06] MERHOF D., SONNTAG M., ENDERS F., NIMSKY C., HASTREITER P., GREINER G.: Hybrid visualization for white matter tracts using triangle strips and point sprites. 1181–1188. doi:http://dx.doi.org/10.1109/TVCG.2006.151. 1

[NvdP98] NEULANDER I., VAN DE PANNE M.: Rendering generalized cylinders with paintstrokes, 1998. 2

[NVR*12] NUNES G. N., VALDETARO A., RAPOSO A., FEIJÓ B., DE TOLEDO R.: Rendering tubes from discrete curves using hardware tessellation. *J. Graphics Tools 16*, 3 (2012), 123–143. URL: http://dx.doi.org/10.1080/2165347X.2012.659610, doi:10.1080/2165347X.2012.659610. 2

[RBE*06] REINA G., BIDMON K., ENDERS F., HASTREITER P., ERTL T.: GPU-Based Hyperstreamlines for Diffusion Tensor Imaging. In *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization* (2006), Santos B. S., Ertl T., Joy K., (Eds.), The Eurographics Association. doi:10.2312/VisSym/EuroVis06/035-042. 3

[Sam] SAM: On picking an orthogonal vector (and combing coconuts). http://lolengine.net/blog/2013/09/21/picking-orthogonal-vector-combing-coconuts. 4

[SGpSP05] STOLL C., GUMHOLD S., PETER SEIDEL H., PLANCK M.: Visualization with stylized line primitives. In *In: Proceedings of IEEE visualization 2005* (2005), pp. 695–702. 3

[Sta] STACKOVERFLOW: Given a single arbitrary unit vector, what is the best method to compute an arbitrary orthogonal unit vector? http://stackoverflow.com/questions/19649452. 4

[Web00] WEBER J.: Run-time skin deformation. In *Proceedings of the Game Developers Conference* (2000). 2

[Wika] WIKI: Bisection method. https://en.wikipedia.org/wiki/Bisection_method. 8

[Wikb] WIKI: Frenet Serret formulas, Kinematics of the frame. https://en.wikipedia.org/wiki/Frenet-Serret_formulas#Kinematics_of_the_frame. 2, 4

[Wikc] WIKI: Halley's method. https://en.wikipedia.org/wiki/Halley's_method. 8

[Wikd] WIKI: Newton's method. https://en.wikipedia.org/wiki/Newton's_method. 8

[ZSH96] ZÖCKLER M., STALLING D., HEGE H.-C.: Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proceedings of the 7th Conference on Visualization '96* (Los Alamitos, CA, USA, 1996), VIS '96, IEEE Computer Society Press, pp. 107–ff. URL: http://dl.acm.org/citation.cfm?id=244979.245023. 1