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

Abstract

Interactive visualizations of tube-based structures find application in various scientific fields like physics and medicine [?]. For tube-based scenes of high complexity, specialized rendering algorithms are needed to achieve interactive frame rates. This paper presents a tessellation-based tube rendering approach with dynamic geometry refinement as well as an approach that 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 [?] and blood vessels [?] over particle trajectories [?] to stream lines [?] 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 out 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 mask out unaffected 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 considered also.

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.

The following section first provides an overview of preceding work in the field of generalized cylinder rendering before the proposed methods are described afterwards.

2. Previous 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. The majority of approaches specialized in rendering generalized cylinders are tessellation and rasterization based. A common problem often associated with the tessellation of generalized cylinders is that of choosing appropriate *cross-section frames* along the tube axis. 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 [Blo]. 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* [?] 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 the scene geometry on the CPU side and sending the resulting vertices together with the required connectivity information to the GPU for rendering, while certainly functional, can quickly become bottlenecked by the high bandwidth demands this approach induces for complex scenes. To circumvent this, all techniques reviewed in the following attempt to transfer only the minimal amount of data necessary to construct the geometry at render time. This strategy effectively trades reduced bandwidth demand for processing time.

The approach described in [?] 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. 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 paintstrokes renderer uses an A-Buffer to implement antialiasing. It follows the original implementation introduced in [?] albeit with modifications regarding the generation and blending of fragments.

Another tessellation-based spline tube rendering approach is described in [?]. Unlike the paintstrokes algorithm, it does not perform the 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 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.

Considering that both, [?] and [?], apply dynamic tessellation strategies that maintain a topology that allows to identify conic sub-segments, it stands to reason that it might be more efficient to ray cast these instead of tessellating the cross-section.

[?], too, present a tessellation-based approach with curvature dependent adaptive sampling. 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, [?] use linear blend skinning [?] 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, [?] 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 [?] 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 rasterizer, and are rendered faster using ray casting as long as the viewport is kept small.

3. Spline tubes rendering

In this section a tessellation-based rendering approach and the new ray casting method are described followed by an evaluation of their performance characteristics for different data sets. Common concepts and implementation aspects both approaches share are outlined first.

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 derived from the respective values and derivatives stored at the nodes. 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 that is centered at the origin and of which the normal coincides with the tangent of a point of interest on a given tube axis. Any set of spline segments (consisting of a position, a radius, and a color spline segment) provides the mathematical description of an associated tube segment, which is therefore fully defined by a pair two subsequent nodes.

During rendering, per-segment and per-node data is stored in a GPU buffer called the *tubes buffer* (see figure **??**). Its particular type is API and implementation dependent. 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 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.

3.1. Tessellation-based approach

Below, the tessellation-based spline tube rendering approach is presented starting with a depiction of the procedure that generates the coarse base tessellation followed the description of the dynamic LOD scheme.

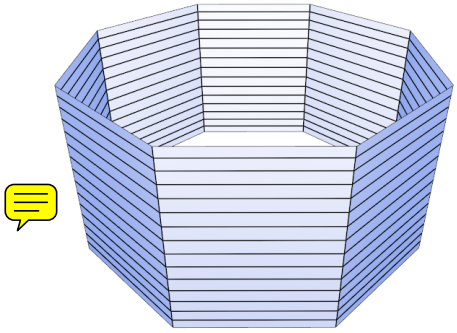

Figure 1: *text text text.*

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 1). The axis of the cylinder is aligned with the z-axis in object space. The z components of the object 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].

To transform the cylinder mesh to a curved tube segment with varying radius, the vertex shader procedure described in the following is applied. First, the position, tangent, and radius splines are derived and evaluated using the z coordinate of the given vertex 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 (see figure **??**). This offset vector is calculated from the local vector that describes the translation from axis position to object space position of the vertex. For the purpose of transforming this vector to the offset vector, two vectors that together with the curve tangent form an orthogonal cross-section frame are needed (see figure **??**).

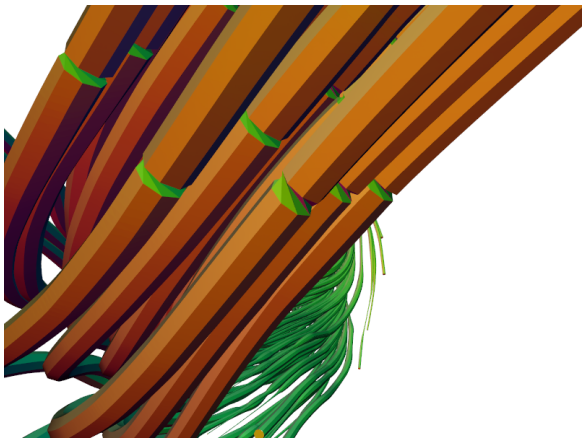

Figure 2: *text text text.*

The first of these can be determined by picking an arbitrary vector orthogonal to the tangent using one of numerous existing strategies [**?**] [**?**]. The second one 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 2). 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. This process is performed as a pre-processing 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 (see figure **??**). 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 out 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 (see figure **??**). 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 that describes 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 **??**). At last, a normal is computed for 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 **??**).

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. In practice, however, this did not pose a problem for the tested data.
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.
One problem that can arise due to varying tessellation levels

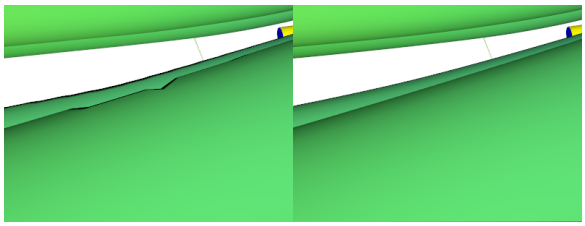

Figure 3: *text text text.*

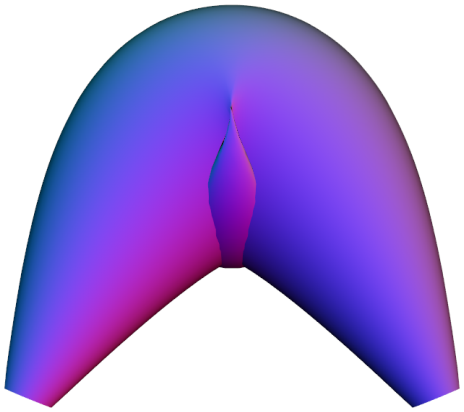

Figure 4: *text text text.*

are notchy silhouettes (see figure 3). 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 and divided by the w components of the screen space positions of the edge midpoints. This leads to an increasingly isotropic edge subdivision towards silhouettes. Furthermore, using a quad patch topology for the base tessellation, too, turned out to be essential in order to prevent this kind of artifacts. 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 resulting from self-intersections (see figure 4). 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.

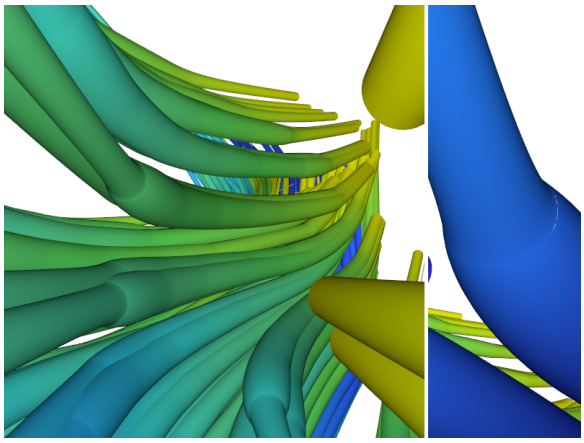

Figure 5: *text text text.*

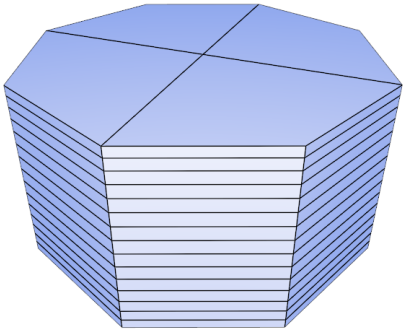

Figure 6: *text text text.*

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 (see figure 6). 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 $C1$ discontinuties without having to extent the framework (see figure 5 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 5 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.

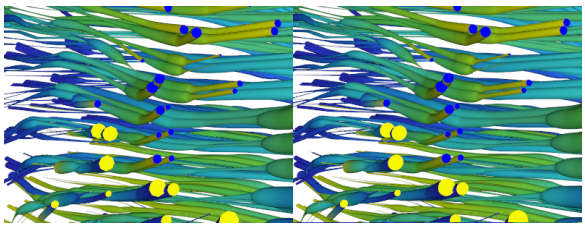

Figure 7: *text text text.*

In general, cap seam edges need to be finer tessellated then the body of the segment to provide a smooth appearance (see figure 7). 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.

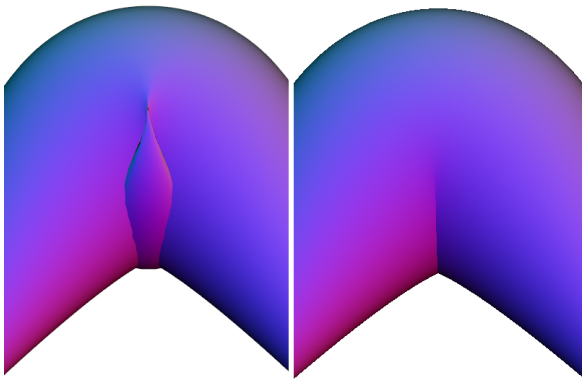

Figure 8: *text text text.*

3.2. Raycasting-based appproach

The rendering approach described in the following uses a ray-spline tube segment intersection routine to compute the intersections of the camera rays with the scene.

Deriving the intersection routine requires a parametrized ray-segment intersection 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 ray intersection with a sweep of scaled geometric primitives following a given space curve 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 (see figure 8). 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, rearranging the ray-disk tube intersection equation **??** results in polynomials of degree 6 and 10 for quadratic and cubic splines, respectively, making the root search too costly in both cases. In contrast, a sphere-based approach does not suffer from bulb artifacts (see figure 8), while, at least for quadratic splines, allows to find the roots with reasonable effort. Therefore, the cubic spline segments are approximated by two or more quadratic *sub-segments*. Even though this approximation introduces $C2$ discontinuities, the smooth shading applied to the tubes usually does not expose those.

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. 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:

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

This circle is then intersected with the x-axis:

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

Substituting R into s 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 real, local minima of f_t correspond to ray intersections 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:

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

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

Resulting in:

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

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

$$f_t'(l) = polyA_t'(l) - polyB_t'(l)/(2 * \sqrt{polyB_t(l)}) \tag{8}$$

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.

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, a closed from solution returning the roots exist [**?**]. However, due to the large amount of arithmetic operations involved in computing an analytical solution, the result can suffer from numerical inaccuracies as well as mediocre performance [**?**]. Therefore, the 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 [**?**] (see figure **??**). Strategies with better theoretical convergence like Newton-Raphson [**?**] 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 (see figure **??**). 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 two roots of the second most derived version of g analytically. 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 therefore 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'. Since f_t' is the derivative of f_t, of which the minima coincide with the front side ray-tube intersections, there are up to three real roots (see figure **??**) of which the outer two are of interest. Therefore, up to 5 binary searches per real interval are necessary to find the relevant roots of f_t' (see figure **??**). However, in most cases 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 is due the sphere sweep effectively forming a level set of the distant field of the tube segment...no sure (need to verify, might be only true for constant radius...) The interpolated color, in contrast to position and radius, is not computed from the quadratic approximations but by evaluating the original cubic spline. At last, 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 9). The x-axis of the bounding box equals the direction 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.

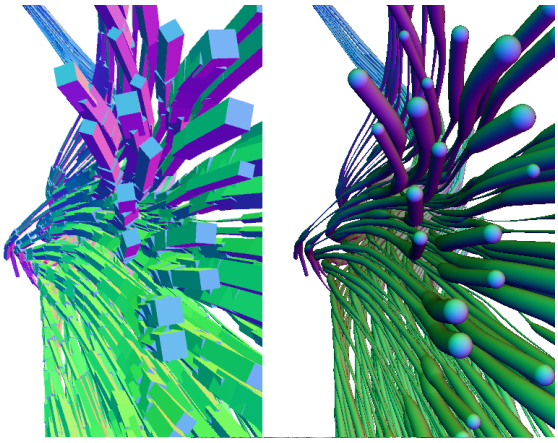

Figure 9: *text text text.*

3.3. Results and Comparison

Now look at profiling results of different data sets and compare performance characteristics. much fast

4. Future Work

- adaptive resampling of tubes, curvature stuff like the others - ray footprint for tess - cubic patches - sphere tessellation - better root finding - anlytical normals for disk model - analyt ortho vecs - less precomp ortho vecs - compare to interval/affine arithmetic or hybrid, diff root finder in gneral - raycasr bounding boxes on CPU side, culling stuffs - precision problem for looong segments, dynamic iteration count, precision bound - pathtracing - soft instancing - tighter bounding geo, screen space bounding geo - rendering with dynamic resolution for raycasting + supersampling with fixed cam - pp aa -> geometrical aa, reprojection aa - AO volumes

5. Conclusion

Splines tubes are very cool. It has been shown that one approaches is different from the other! :O

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

[Blo] BLOOMENTHAL: Calculation of reference frames along a space curve. http://webhome.cs.uvic.ca/~blob/courses/305/notes/pdf/ref-frames.pdf. 1