Visualization with stylized line primitives

Carsten Stoll* Stefan Gumhold† Hans-Peter Seidel‡

Max Planck Institut fuer Informatik
Saarbruecken

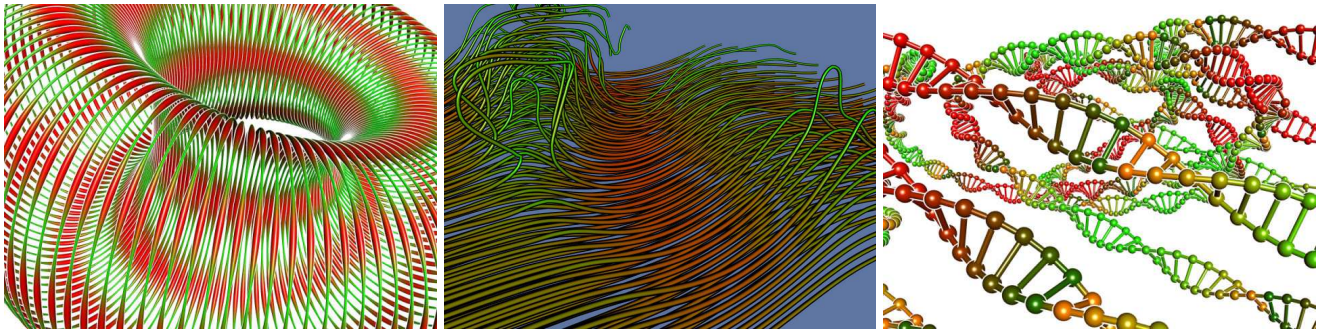

Figure 1: Several examples of objects and streamlines rendered with our algorithm.

ABSTRACT

Line primitives are a very powerful visual attribute used for scientific visualization and in particular for 3D vector-field visualization. We extend the basic line primitives with additional visual attributes including color, line width, texture and orientation. To implement the visual attributes we represent the stylized line primitives as generalized cylinders. One important contribution of our work is an efficient rendering algorithm for stylized lines, which is hybrid in the sense that it uses both CPU and GPU based rendering. We improve the depth perception with a shadow algorithm. We present several applications for the visualization with stylized lines among which are the visualizations of 3D vector fields and molecular structures.

CR Categories: I.3.7 [Computer Graphics]: Rendering and shading—Visualization of vector-/tensor-fields

Keywords: rendering, vector fields, streamlines

1 INTRODUCTION

The visualization of dense 3D data sets such as 3D vector fields is one of the most difficult problems in scientific visualization. The central problem is occlusion, what rules out the use of a dense visual representation of the data set. Techniques like 3D LIC can only be used in conjunction with slicing techniques that restrict the visualized data again to a 2D subset. For an actual 3D visualization one therefore has to fall back to sparse visual representations. Among these are iconic representations and line primitives. The latter has the additional advantage that it not only shows local but also global features of the underlying data. Although these sparse representations allow a better view into the data, it is quite difficult to achieve good depth perception. Thus one has to use as many of the visual cues for depth perception as possible, i.e. occlusion, perspective shortening, illumination and shadow effects.

In this paper we extend line primitives, which are commonly used for visualization in 3D, by several additional visual attributes and a visual representation that improves the depth perception significantly. As geometric and visual representation we propose to actually blow up the line primitives to real 3D primitives, which we denote *stylized line primitives*. Similar to stream tubes we chose to use generalized cylinders with a circular profile. The illuminated rendering of generalized cylinders gives a very good depth perception as can be seen in Figure 1. Furthermore, it allows the use of additional visual attributes. Here we investigate radius, color as well as textures of colors and normals. Rotation of the textures along the line primitives gives another well perceivable visual attribute – the orientation. Finally, we add a halo to improve depth perception even further.

A visualization technique is only valuable if it allows for real-time navigation in a potentially time-dependent data set. Therefore, the major contribution of this work is an efficient rendering algorithm for stylized line primitives. Although rendering algorithms exist for generalized cylinders, non of the existing algorithms allows for real-time navigation of large data sets with high image quality. Our new algorithm combines the speed of splatting based algorithms with high image quality, which is achieved by a hybrid approach that uses fast GPU supported splatting for most stylized line primitives and a CPU based tessellation in regions where the simple splatting approach leads to noticeable artifacts. We achieve a speed up of three in comparison to a purely tessellation based approach for static data sets and a speedup of a factor of up to 6 for time-dependent data sets. These speed ups are mainly due to the reduced amount of geometry data that has to be transfered to the GPU and processed there.

In the remainder of the paper we first discuss related work, introduce the stylized line primitive in section 3, detail the hybrid rendering approach in section 4, measure the performance and performance gains in 5 and apply the new visualization technique to several applications in section 6.

*e-mail:stoll@mpi-sb.mpg.de
†e-mail:sgumhold@mpi-sb.mpg.de
‡e-mail:hpseidel@mpi-sb.mpg.de

IEEE Visualization 2005
October 23-28, Minneapolis, MN, USA
0-7803-9462-3/05/$20.00 ©2005 IEEE.

2 Previous work

The visualization of 3D flows using streamlines as a sparse and fast representation method has been researched for a long time now. An important factor here is a good distribution of samples across the flow and a good path-tracing algorithm to integrate streamlines in a vector- or tensor-field. A comprehensive overview of such techniques can be found in [13].

Several methods have been introduced to visualize streamlines in the past years. Ueng et al. [15] use stream-ribbons and stream-tubes to visualize the flow, the first being a quad strip generated by the streamline and a normal vector extracted with it and the second being a generalized cylinder with a circular profile where the radius depends on the flow velocity. No efficient rendering algorithm for the stream-tubes was proposed though. Fuhrmann and Gröller [6] use generalized cylinders with opacity textures to represent dashed stream-tubes. They propose a simple tessellation scheme with 6 to 8 subdivisions along the cylinder. Zöckler et al. [16] introduced the simple to render illuminated streamlines, which is a lighting model for line primitives based on the Phong lighting. This enhances depth perception to some extent only and does not give a ductile impression of the stream-lines especially when using varying thickness. Mattausch et al. [11] later presented a comprehensible overview of methods for visualizing flows using illuminated streamlines, including levels of detail, magic volumes and halos.

Schussman and Ma [14] introduced a method for rendering huge amounts of streamlines using volume visualization, but their method aims for extremely dense fields and is not able to provide interactive frame-rates.

General cylinders where introduced by Agin and Bindford [1] in 1976 as a method to describe and model curved objects. Different methods for visualizing these have been proposed later, including ray tracing ([5]) and different polygonalization methods ([4], [2], [7]).

Some special attention has also been paid to the rendering of generalized cylinders with circular profile. Main contributions here are the *Optimal Tubes* [3] by Blinn, who adaptively tessellates a tube at the silhouette edges and lighting dependent positions and *Paintstrokes* by Neulander and van de Panne [12], who present different tessellation methods for rendering circular generalized cylinders. All the tessellation methods lead either to a huge number of rendering primitives or a bad image quality with artifacts. Our approach on the other hand optimally combines GPU based rendering exploiting fragment shaders with CPU based adaptive tessellation. One very important design criterion for the fragment shaders was to keep them as simple as possible in order to achieve high fill rates.

3 Stylized Line Primitives

Stylized line primitives are defined as generalized cylinders. A generalized cylinder is composed of a continuous path $\boldsymbol{P}(\tau)$ in 3d space and a closed two dimensional profile $\Pi(\phi) = \big(p_x(\phi), p_y(\phi)\big)$. We assume that the curve defining the path is regular, i.e. the path tangent $\vec{\boldsymbol{T}}(\tau)$ does not vanish:

$$\vec{\boldsymbol{T}}(t) = \partial_t \boldsymbol{P}(t) \neq \vec{\boldsymbol{0}}$$

Furthermore, we restricted ourselves to circular profiles of radius $R(\tau)$, so

$$\Pi(\phi) = R(\tau) \cdot \begin{pmatrix} sin(\phi) \\ cos(\phi) \end{pmatrix} \quad 0 \leq \phi < 2\pi. \tag{1}$$

The surface of the general cylinder can thus be described by

$$\boldsymbol{S}(\tau,\phi) = \boldsymbol{P}(\tau) + \Pi_x(\phi) \cdot \hat{\boldsymbol{x}}(\tau) + \Pi_y(\phi) \cdot \hat{\boldsymbol{y}}(\tau) \tag{2}$$

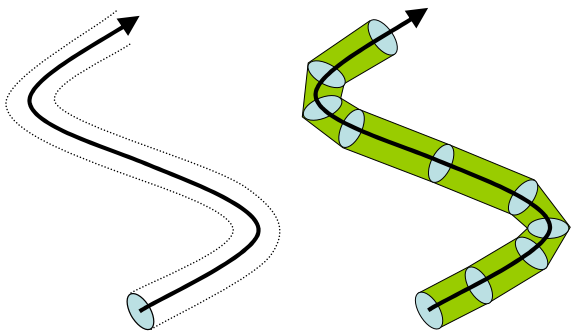

Figure 2: Schematic of a continuous general cylinder with circular profile and a discretization.

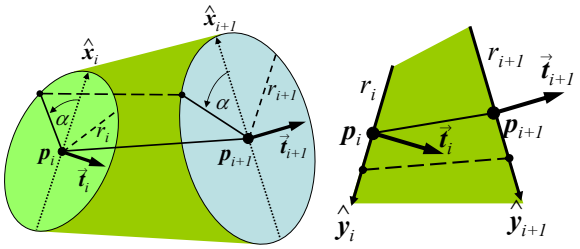

Figure 3: One segment of the discretized stylized line is spanned by two tangential circles parameterized synchronously over ϕ. *Left:* 3d view, *Right:* viewed along the $\hat{m}_i = \hat{m}_{i+1}$ direction.

where $\hat{\boldsymbol{x}}(\tau)$ and $\hat{\boldsymbol{y}}(\tau)$ are two orthonormal vectors that span the plane orthogonal to the tangent vector $\vec{\boldsymbol{T}}(\tau)$.

Generalized cylinders can be visualized in different ways. As our goal is the interactive visualization of a large number of stylized lines ray tracing based approaches drop out. In order to be able to exploit modern graphics hardware we chose a splatting based approach. For this the continuous path of a generalized cylinder $\boldsymbol{P}(\tau)$ is sampled at discrete path parameters τ_i, resulting in a set of points $\boldsymbol{p}_i = \boldsymbol{P}(\tau_i)$ with $i = 0 \ldots n$. The discretization is done in advance or during construction of the path with a flow integration method. It is advantageous to adapt the sampling resolution to the curvature such that a higher resolution is chosen in high curvature regions and a smaller resolution in low curvature regions. In our work we followed the approach of Gumhold [9] for adaptive subdivision, which is based on a maximal discretization error $\varepsilon_{\max}$.

To discretize the generalized cylinder each point $\boldsymbol{p}_i$ is attributed with the circle describing the profile at path parameter τ_i. The profile circle is uniquely defined by its radius $r_i = R(\tau_i)$ and the tangent vector $\vec{\boldsymbol{t}}_i = \vec{\boldsymbol{T}}(\tau_i)$ or its normalized version $\hat{\boldsymbol{t}}_i$, respectively. All the necessary information is gathered in a vertex $V_i = (\boldsymbol{p}_i, \hat{\boldsymbol{t}}_i, r_i, \ldots)$, which is later extended by color and texture coordinates.

In this way the discretization splits the stylized line primitive into segments, where each segment is defined by two consecutive vertices V_i and V_{i+1}, i.e. profile circles. Figure 3 a) illustrates the two circles defined by the vertex data. One way of defining the geometry of each segment from the vertex data is to use the convex hull of the two circles. This has the advantage that the convex hull of the 2d projection of the 3d segment could be computed from the 2d convex hull of the projected circles and that the hull is spanned by straight line segments. On the other hand is it quite complicated to find these line segments and to correctly illuminate the convex hull. Instead we define the geometry by using the ϕ-parameterization of the circular profiles. We simply connect points of equal ϕ on the two circles with straight line segments as illustrated for $\phi = \alpha$

in Figure 3 a). For this to work one has to synchronize the local frames $\{\hat{\boldsymbol{x}}(\tau_i),\hat{\boldsymbol{y}}(\tau_i)\}$ and $\{\hat{\boldsymbol{x}}(\tau_{i+1}),\hat{\boldsymbol{y}}(\tau_{i+1})\}$ appropriately. The vectors $\hat{\boldsymbol{x}}(\tau_i/\tau_{i+1})$ correspond to the $\phi = 0$ angles and uniquely define the correspondence if we assume right handed coordinate systems $\{\hat{\boldsymbol{x}},\hat{\boldsymbol{y}},\hat{\boldsymbol{t}}\}_{i/i+1}$. Furthermore it is not important to use a single $\hat{\boldsymbol{x}}(\tau_i)$ for the two segments (V_{i-1},V_i) and (V_i,V_{i+1}). Therefore we can define the correspondence for each segment independently in the following way. In the case when the two tangent vectors $\vec{\boldsymbol{t}}_i$ and $\vec{\boldsymbol{t}}_{i+1}$ are parallel, the two circles lie in parallel planes and any choice of parallel $\hat{\boldsymbol{x}}_{i/i+1}$ in this plane can be used. In the case of differing tangents, we chose $\hat{\boldsymbol{x}}_{i/i+1}$ in the direction of the intersection line of the two circle planes, i.e. parallel to $\vec{\boldsymbol{t}}_i \times \vec{\boldsymbol{t}}_{i+1}$. The intersection line is drawn in dotted style in Figure 3 a). Figure 3 b) shows the situation in 2d, viewed along the direction of the intersection line, i.e. along the synchronized directions $\hat{\boldsymbol{x}}_{i/i+1}$.

In order to illuminate the discretized stylized lines we interpolate the normal vectors along the connection lines of equal ϕ parameter values. At the end points on the two profile circles we chose the normal in the plane of the circle. This does not reflect the illumination of the actual geometry of each segment, what would lead to uncontinuous normals where two segments meet. Our choice on the other hand synchronizes the normals of adjacent segments at their common circle resulting in continuous normals and much better lighting results.

The brute force splatting approach simply tessellates each segment with a quad strip. For medium viewing distances a tessellation with eight quads per segment resulted in an acceptable quality. This approach is however extremely expensive since a large amount of geometry data is created and has to be transfered over the graphics card bus. A segment extended by per vertex color information polygonized into a quad strip of length eight results in 18 render vertices, each consisting of position, normal and color, summing up to 180 floats or 720 bytes of data per segment. While this is no problem for short stylized lines we quickly reach the transfer bandwidth or vertex processing limit even of modern graphics cards when rendering a large number of segments.

4 Hybrid Rendering of Stylized Line Primitives

In this section we describe a hybrid rendering algorithm for stylized line primitives that exploits the CPU and GPU in an efficient way. There are four possible bottlenecks in the design of such hybrid algorithms: the CPU processing rate, the data transfer rate from CPU to GPU, the vertex processing rate and the fill rate, i.e. the fragment processing rate. The simple tesselation strategy stresses CPU processing, data transfer from CPU to GPU and vertex processing. On the otherhand it achieves very large fill rates. In applications like ours there are a huge number of render primitives that project to a very small area in the frame buffer, such that also relatively low fill rates do not lead to a bottleneck. In this way the simple strategy is quite bad for our application.

To better balance the different processing and transfer rates we propose a strategy with slightly more complicated fragment processing that allows to process coarser geometric approximations which can be achieved with a much lower count of to be rendered vertices. In this way the first three bottlenecks are significantly reduced. The next two sections describe the straight forward mostly GPU based implementation of this approach. In section 4.3 we discuss severe artifacts that arise where the segments become parallel to the viewing direction. We worked a lot on ways to avoid these artifacts in the primarily GPU based approach but did not find any approach with acceptably simple fragment processing, which is necessary to avoid a new bottleneck in the fill rate.

The main observation leading to the solution proposed in section 4.4 was that these artifacts arise only in a small number of segments which can be efficiently sorted out. As the artifacts can be cleaned up afterwards, we extended the GPU based approach with a CPU stage that view-dependently detects the problematic segments and corrects the artifacts by overdrawing them with tesselated geometry. The additional geometry necessary per frame was small enough to not decrease the frame rates significantly. In this way we could very effenciently balance the work load on CPU and GPU exploiting the strengths of both architectures.

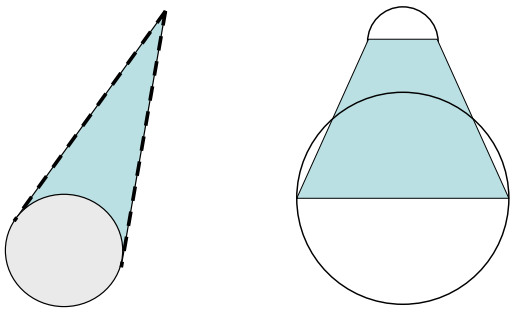

Figure 4: *Left:* Sketch of an infinite cylinder with silhouette lines dashed. *Right:* The simplified quad fails to cover the complete area of the GC segment.

4.1 Splatting the Geometry

The cannonic approach to splat a segment of the discretized generalized cylinder is to cover it completely with a render primitive, i.e. a quad, and then determine the actually covered fragments and their illumination in the fragment shader. The fragment shader would have to solve the ray-segment intersection incrementally. Especially complicated is the ray-segment intersection where the segment ends in the circles of the vertices. This makes the fragment shader far too complicated for an acceptable fill rate.

To simplify the splatting proceedure we exploit the fact that a sequence of segments is splatted and that both terminating circles of a segment are matched up with adjacent segments. At each circle in the discretization we approximate the generalized cylinder by an infinite cylinder, which results from extrusion of the circle along the tangent vector. From this approximation we compute two silhoutte points $\boldsymbol{s}_i^1$ and $\boldsymbol{s}_i^2$ for each vertex V_i. Each segment is finally covered by a quad connecting the four silhoutte points of its two vertices. This approach is similar to Neulanders [12] quality 0 paintstrokes and Blinns Optimal Tubes [3] but gives much better result through the use of a better shading in the fragment shader.

On the left of Figure 4 the perspective view of a half infinite cylinder is shown. The two silhouette lines are illustrated by the dashed bold lines. From these two lines we select the two silhouette points that lay in the same plane as the circle of the vertex. If C_i denotes all points on the infinite cylinder and $\boldsymbol{v}$ the view point, the silhouette points for vertex $V_i = (\boldsymbol{p}_i,\hat{\boldsymbol{t}}_i,r_i)$ can be defined as:

$$\boldsymbol{s}_i^{1,2} = \{\boldsymbol{p} \in C_i \mid \hat{\boldsymbol{n}}^T(\boldsymbol{p}-\boldsymbol{v}) = 0, \boldsymbol{p}\cdot\hat{\boldsymbol{t}}_i = \boldsymbol{p}_i\cdot\hat{\boldsymbol{t}}_i\}, \tag{3}$$

where $\hat{\boldsymbol{n}}$ is the normal vector of the cylinder at the point $\boldsymbol{p}$.

The computation of $\boldsymbol{s}_i^{1,2}$ can be reduced to a simple two dimensional problem since the relative position of the silhouette points to the point of the main axis in the same perpendicular plane is the same for all points of this axis. This is true because all lines on the cylinder surface parallel to $\hat{\boldsymbol{t}}_i$ have the same vanishing point and therefore can never cross (see figure 4 left), which means that if a point $\boldsymbol{s}$ is part of the silhouette, all points $\boldsymbol{s}_t = \boldsymbol{s} + t\cdot\hat{\boldsymbol{t}}_i$ are silhouette points of the infinite cylinder too.

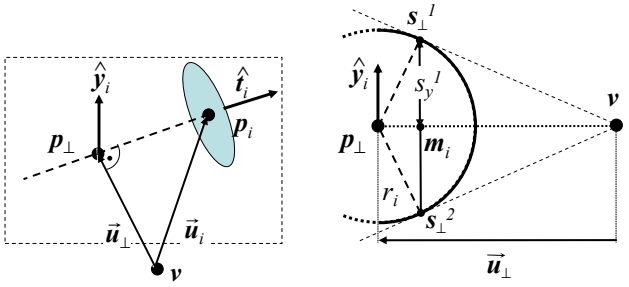

Figure 5: *Left:* Reduction of silhouette point calculation to 2D. *Right:* Determining the silhouette points of a streamline segment.

Only if the angle between the tangent $\hat{t}_i$ and the view vector $\vec{u}_i = p_i - v$ becomes too small (i.e. the viewpoint lies inside the infinite cylinder C_i), no reduction to 2D is possible and $s_i^{1,2}$ cannot be found, which leads to problems which we address in section 4.3.

This in turn means that, as long as $s_i^{1,2}$ exist, we can determine the relative positions of the silhouette for the point $p_\perp$ on the main axis of the cylinder, which is closest to the view point v. See the left of Figure 5 for an illustration. The view vector to $p_\perp$ is denoted by $\vec{u}_\perp$. At $p_\perp$ we define again a two dimensional local coordinate system $\hat{x}_i$ and $\hat{y}_i$ that spans the plane of the circle. The local coordinate direction $\hat{x}_i$ is chosen perpendicular to $\vec{u}_\perp$ and the tangent direction $\hat{t}_i$, i.e.

$$\hat{y}_i = \frac{\hat{t}_i \times \vec{u}_\perp}{\|\hat{t}_i \times \vec{u}_\perp\|}. \tag{4}$$

$\hat{y}_i$ is computed orthogonal to $\hat{x}_i$ and $\hat{t}_i$:

$$\hat{x}_i = \frac{\hat{t}_i \times \hat{y}_i}{\|\hat{t}_i \times \hat{y}_i\|}. \tag{5}$$

Figure 5 shows the problem reduced to 2D. To compute the local components $s_{x|y}^{1|2}$ of the silhouette points $s_\perp^{1|2}$ at $p_\perp$ we can apply Pythagoras rule to find that the point m_i is a distance of

$$s_x^{1|2} = r_i \frac{r_i}{|\vec{u}_\perp\|}$$

away from $p_\perp$ and that the $y-$components result in

$$s_y^{1|2} = \pm r_i \cdot f_i, \text{ with } f_i = \sqrt{1 - \left(\frac{r_i}{\|\vec{u}_\perp\|}\right)^2}.$$

Using this we can define two extrusion vectors

$$\vec{e}_i^{1|2} = s_\perp^{1|2} - p_\perp = r_i \left(\frac{r_i}{|\vec{u}_\perp\|}\hat{x}_i \pm f_i \cdot \hat{y}_i\right), \tag{6}$$

with which we can compute the silhouette points s_i^1 and s_i^2 to

$$s_i^{1|2} = p_i + \vec{e}_i^{1|2}. \tag{7}$$

In practice it can be observed that f_i is usually near equal to one for all segments, except for the cases where we are very close to the surface or the angle between $\hat{t}_i$ and $\vec{u}_i$ becomes very small ($\vec{u}_\perp$ becomes very small), which means that we are in an instable region. Due to this it is possible to ignore this term and estimate $\vec{e}_i^{1|2}$ as

$$\vec{e}_i^{1|2} = \pm r_i \cdot \hat{y}_i \tag{8}$$

without any noticeable visual impact on the result but saving several costly operations on the GPU.

The cylinder normals can also be expressed in terms of the local coordinate axes $\hat{x}_i$ and $\hat{y}_i$. The normals at the silhouette points $s_i^{1|2}$ are given as

$$\hat{n}_i^{1|2} = \sqrt{1 - f_i{}^2} \cdot \hat{x}_i \pm f_i \cdot \hat{y}_i. \tag{9}$$

By transmitting the two basis vectors $\hat{x}_i$ and $\hat{y}_i$ as well as all other vertex data necessary for lighting calculations to the fragment shader as texture coordinates, they will automatically be interpolated in a perspectively correct manner across the quad by the graphics hardware. In addition we will also transmit $\pm f_i$ in one of the texture coordinates tex_0 and interpolate it along the segment.

The calculation of $s_i^{1|2}$ can easily be performed in the vertex shader of the GPU in a very short time. Due to this we do not need to perform any operations on the CPU except for transfering data to the graphics card. A simple C-like pseudocode for rendering a single splatted segment on the CPU looks like this :

```
beginQuad();
  renderNormal(tangent[i]);
  renderTexCoords(-1, r[i]);
  renderVertex(position[i]);
  renderTexCoords(1, r[i]);
  renderVertex(position[i]);
  renderNormal(tangent[i+1]);
  renderTexCoords(1, r[i+1]);
  renderVertex(position[i+1]);
  renderTexCoords(-1, r[i+1]);
  renderVertex(position[i+1]);
endQuad();
```

The tangent vectors are transfered as normals to the graphics card while the 2D texture coordinates are used to denote the sign of the silhouette point and the radius of the circle. If more information is needed for rendering it can be transfered using additional texture coordinates. It is possible to render longer streamlines as quadstrips and since all data is view independant we can store the whole model into acceleration structures, such as vertex buffer objects or display lists, which enables us to render the complete model with a single draw call.

4.2 Fragment processing, Halos, Texture Mapping and Shadows

The actual shading in the fragment program is similar to a standard Phong shader, except that we need to calculate the current normal from the two transfered vectors upfront. The per fragment normal $\hat{n}$ can be calculated similar to 9 by normalizing

$$\vec{n} = \sqrt{1 - tex_0{}^2} \cdot \hat{x}_i \pm tex_0 \cdot \hat{y}_i \tag{10}$$

A simple extension to the proposed fragment processing is the support for a halo. A halo is similar to a single colored silhouette drawn around the stylized line (usually the background color) and is used to increase the depth perception of the rendering. By adding a second interpolation parameter tex_1 ranging from $-1 - h_{size}$ to $1 + h_{size}$ and multiplying tex_0 by $1 + h_{size}$ in the vertex shader we define a percentage of the radius of our stylized line to be a halo region. A fragment is in the halo region if tex_1 is smaller than -1 or larger than 1. This means that, unlike in methods like [11] our halo is defined in object space and not in screen space, and therefore scales with its distance to the viewpoint.

In addition to this we can also texture the stylized lines by providing additional texture coordinates. While it would be too difficult to distort the texture in such a way that it fits the actual approximated geometry and is oriented correctly, we still can use this to

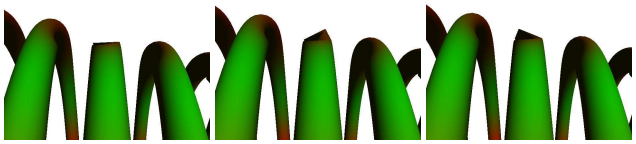

Figure 6: Flipping artifact on the simple shader. Also note the missing part of the silhouette on the center streamline.

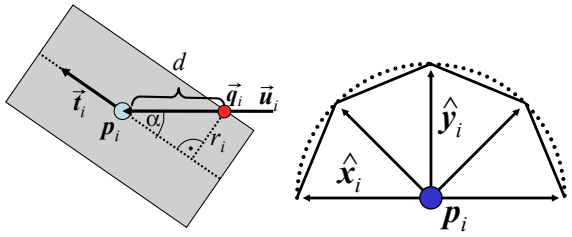

Figure 7: *Left:* Calculating the depth value of a cylinder segment. *Right:* Extrusion scheme for tessellation shader.

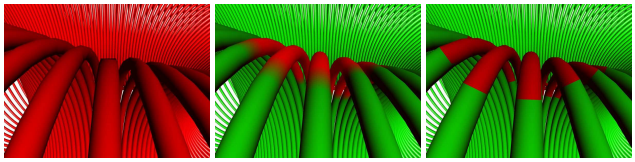

Figure 8: *Left:* Streamlines rendered as simple quads. *Center and Right:* Hybrid rendering with method 1 and 2, colorized for better visibility.

map additional data to the GC segment, like for example torsion by translating the texture to the left or right at the sample points.

Another extension is to use the shader for shadow rendering. Shadow volumes as well as shadow maps can be implemented using a fragment shader which only writes depth values by not extruding the segments with respect to the viewpoint but with respect to the light source instead. Shadow maps are suited better for calculating shadows in most cases though, since with shadow volumes a huge amount of additional geometry is created (at least 2 shadow quads per segment) and the amount of overdraw is very excessive. Depending on the type of data visualized, shadows can add another layer of depth perception to the rendering as illustrated in Figure 10 on the left.

4.3 Evaluation

While the previously proposed rendering method here is extremely fast since the segments can be rendered using quad strips and thus only a very small amount of data is needed, it suffers from large inaccuracies if the angle between tangent- and view vector becomes too small. This is due to the fact that in these areas the main visible part of a segment in the screen projection is provided by the profile area which is not covered by the silhouette quad we calculate (see Figure 4 on the right).

The calculation of the extrusion vector $\hat{\boldsymbol{y}}_i$ has a singularity once $\hat{\boldsymbol{t}}_i$ and $\hat{\boldsymbol{u}}_i$ are parallel or anti-parallel. Near this singularity the resulting extrusion vector fluctuates strongly even when only changing one of the input vectors slightly. This manifests in a so called flipping artifact, which can be seen in figure 6. The changes in view direction are only minimal, but have a huge visual impact.

A second problem is that the depth values created by this algorithm only represent the flat quad stripes and not the geometry of a true cylinder. While this only becomes noticeable when a stylized line intersects another object it still is a detrimental effect.

The latter problem can be partially solved for near perpendicular areas by adjusting the depth value per fragment. It is possible to approximate a second depth value per vertex by determining the thickness of each segment in direction of the view vector and interpolating between the quad depth and this second depth value in the fragment shader using the same method used for determining the per-fragment normal.

Using the infinite cylinder C_i we can determine the intersection point $\boldsymbol{q}_i$ of $\vec{\boldsymbol{u}}_i$ with the cylinder as seen in figure 7. Using the sine rule we can determine that

$$\frac{r_i}{sin(\alpha)} = \frac{d}{sin(\frac{\pi}{2})} \Longrightarrow d = \frac{r_i}{\sqrt{1 - \frac{\vec{\boldsymbol{u}}_i \cdot \hat{\boldsymbol{t}}_i}{\|\vec{\boldsymbol{u}}_i \cdot \hat{\boldsymbol{t}}_i\|}^2}} \tag{11}$$

Therefore we can determine the front intersection point $\boldsymbol{q}_i$ as

$$\boldsymbol{q}_i = \boldsymbol{p}_i - \vec{\boldsymbol{u}}_i \cdot d. \tag{12}$$

Since the depth buffer is non-linear it is not sufficient to interpolate only between the two depth values in the fragment shader. We have to transfer the z and w values of both the transformed $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$ to the fragment shader and interpolate between them based on the same interpolation values used for the normals. After that we can calculate the depth value by dividing z by w and normalizing to $[0,1]$. However, the smaller the angle between $\hat{\boldsymbol{t}}_i$ and $\vec{\boldsymbol{u}}_i$ becomes, the more inaccurate this depth approximation becomes. Thus, a way to deal with the singularity situations has to be found.

4.4 Hybrid Rendering

Since all visual problems arise around the area of the singularity we need to find another method to display those problem zones. A simple yet efficient way is to tessellate the segments as in the brute force method only in those areas. In most of the models and views only a small fraction of the segments lead to the problematic case, so we still gain a huge speed advantage over using the brute force method everywhere. This means we will render the accelerated quad stripes as before in a first pass and then check for problem areas and render tessellated geometry only there.

It is not yet possible in current state of the art graphics processors to generate additional geometry in a shader, and rendering the whole geometry and simply discarding unneeded segments by transforming them behind one of the clipping planes proved to be very inefficient, so we fell back on a software solution.

The problem areas are identified the same way as in the shader by calculating the dot product f_i between view-vector and tangent. If $|f_i| > 1 - \varepsilon$ we found a problem area and need to re-render the segments adjacent to $\boldsymbol{p}_i$. The selection of ε is dependent on the geometry and its sampling resolution, but we found values of about $0.02 - 0.03$ (corresponding to an angle of about 12-14 degrees) to be good choices.

To make the tessellated geometry fit correctly to the segments splatted with one quad, it is required to blend the tessellated segment over into a quad before and after the problem area, else there might be irregularities in the shading. There are two main approaches we used here. In the first we render tessellated segments and alpha-blended them at the start and end. This has the advantage that the depth values of the completed picture is correct even around the problematic areas but the downside is that the lighting might look different on the cylinder due to different kind of shaders used.

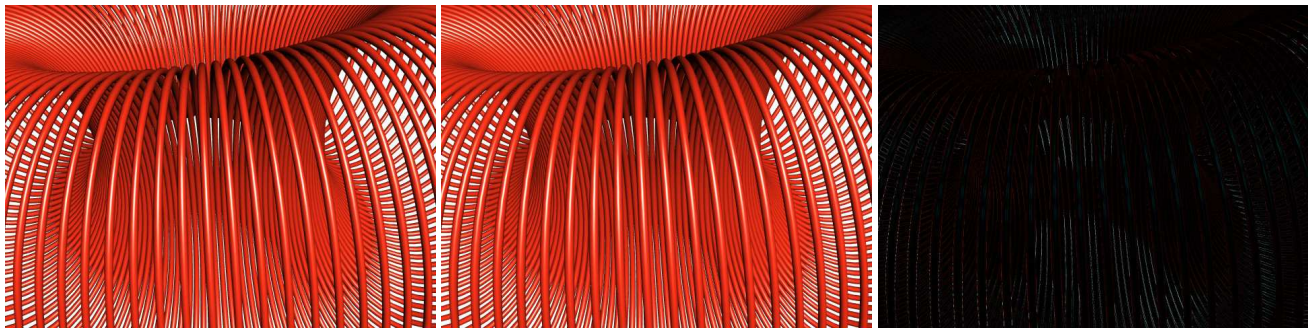

Figure 9: 32768 Phong shaded streamline segments rendered as static geometry (23.4 fps), with our method (54.8 fps) and difference image.

A second possibility is to create a tessellation shader, which is basically an extension of the simple rendering mentioned in the last section where we render not only one quad per segment but four, which is similar to Neulanders quality 2 Paintstrokes (see [12]). The non-silhouette vertices are additionally extruded with $\hat{\boldsymbol{x}}_i$ to approximate the real geometry. The advantage here is that we can physically transform the vertices to meet up with the quad in the shader without a problem and can use exactly the same normal calculation method as before, but we loose continuity of the depth buffer where the patches meet when rendering with depth correction.

Since transmitting geometry of any kind from CPU to GPU is still a big bottleneck we can accelerate both methods by precalculating batches (for example using vertex buffer objects) of geometry segments which we can render with a single draw call when one of the segments of the batch is in a problem area instead of sending each segment separately. If we find one segment in a batch where $|\phi_i| > \varepsilon$ we can skip testing the other segments and draw the batch.

Important factors for performance here are the length of a batch and the maximal tangent deviation of a batch. The maximal angle between all tangents in a batch should be smaller than 10-15 degrees. This allows us to greatly reduce load on the bus between CPU and GPU and therefore accelerating calculations by quite a bit.

5 Results & Benchmarks

We implemented our algorithm using OpenGL and GLSL-shader programs. Some benchmark results for our implementation can be found in table 1. On our GeForce FX5900 the fill rate of the shader is only 20 percent slower than a standard Phong shader, enabling halos takes up another portion of the fill rate. The fill rates have been measured using NVidias *nvshaderperf* [10]. The static geometry used for comparison was tessellated with 8 quads per GC segment. Segment throughput was measured on a 2x2 pixel view port with the model shown in figure 1 in the center, which has 256 stylized lines and 145374 segments. Performance can vary depending on the ratio of segments per stylized line. In average it is about a factor of three higher for the accelerated methods than for the geometry. The measurements denoted as DL were performed using a display list/vertex buffer objects (the methods are equally fast), the rates denoted with IM were performed in immediate mode (with direct open gl draw calls for every primitive) and the rates denoted with VA were performed with vertex arrays (excluding the time for computing the vertex arrays).

When enabling the hybrid rendering algorithm the segment throughput drops only slightly and is still 2.5 times faster than the brute-force approach, while the resulting image is near indistinguishable from the geometry approach, and at certain places even looks smoother than the tessellation with eight quads per segment.

Type	Fill rate	DL seg/s	VA seg/s	IM seg/s
Geometry	150.0 MP/s	1.9 M	0.77 M	56.5 K
Shader	120.0 MP/s	6.9 M	4.2 M	1.9 M
+Halo	94.7 MP/s	6.0 M	4.0 M	1.9 M

Table 1: Throughput measurements. Fill rate measured with NVidias nvshaderperf [10] software. Segments per second measured with display lists/vertex buffer objects, vertex arrays and immediate mode respectively on a 2x2 view port.

Rendering Type	FPS
Geometry	11.4
Geometry + Halo	9.7
Shader	28.2
Shader with halo	23.4
Hybrid	27.0
Hybrid with halo	22.5

Table 2: Frames per second measurement of a 65536 segment toroid model as seen in figure figure 1.

Table 2 shows benchmark results in frames per second on a scene with the Toroid like model shown in figure 1 on the left, consisting of 65536 stylized line segments. The halo for the geometry approach was created by rendering the segments a second time with larger scale, inverted normals and disabled lighting.

It is interesting to note that even when rendering all segments in immediate mode (meaning we transfer the geometry from CPU to GPU every frame anew, enabling us to modify the geometry at will, for example for highly dynamic data sets) we still achieve a throughput of about 1.9 million segments per second with just the simple shader enabled. This means that it is possible to render a model consisting of up to about 60k streamline segments still in real time with 30 frames per second, so we can handle dynamic models a lot better than with pure geometry where we have to calculate a new tessellation for every frame.

If we take a look at the size of the geometry it can be noted that to draw a single shaded segment we need only 110 bytes of data (4 vertices, 2 tangents, 2 colors and radius/interpolator information) compared to the 720 bytes needed to render a stripped geometry cylinder segment. This value can be halved for most of the segments since it is possible to render them as a strip instead of single quads.

Using our rendering technique we have moved the main stress from CPU processing rate and data transfer between CPU and GPU to the vertex and fragment processing on the GPU.

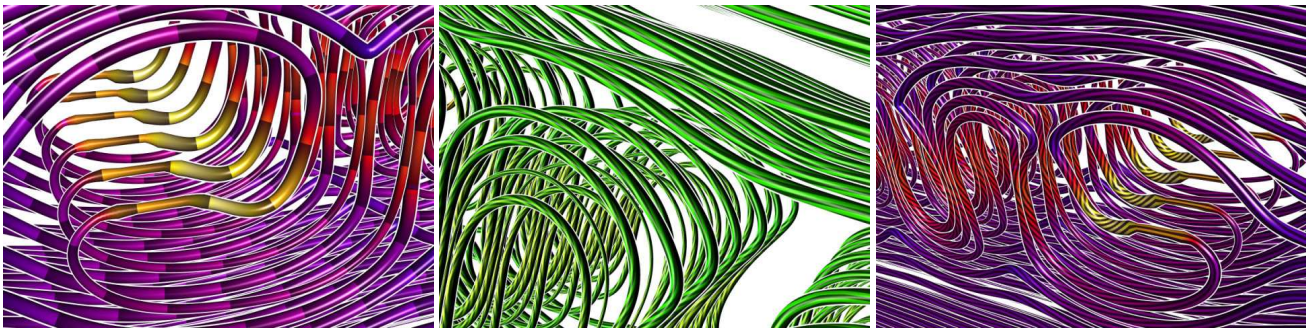

Figure 11: Mapping several scalar values of a vector field to streamlines. The left picture shows streamlines with velocity encoded in color (purple to red to yellow), the integrated divergence is mapped to the radius and the flow direction is visualized by a saturation modulation (flow is from low saturation to high saturation). The two pictures on the right use a similar encoding except that the texture encodes the rotation in direction of the flow.

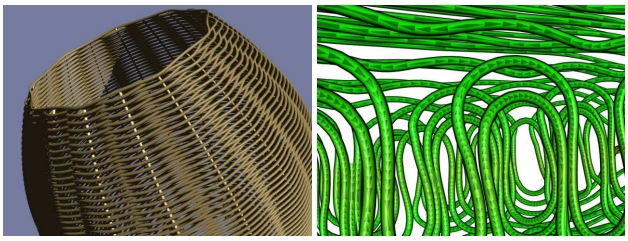

Figure 10: A basket with shadows enabled and a set of streamlines where direction of rotation compared to the tangent is mapped to an arrow texture with shorter arrows depicting areas of higher rotation.

6 Applications

Our algorithm has several interesting applications, of which the rendering of streamlines is one of the most important ones. Previously streamlines were mostly rendered with real geometry like streamtubes or -ribbons [15], or as illuminated lines [16].

Illuminated Streamlines have the disadvantage that they don't perform lighting across a line, so while providing an improvement to rendering simple lines without lighting they still tend to look somewhat flat. This can especially be noticed for the specular illumination component. Another aspect is that it is not possible to render lines of arbitrary radius, limiting the use to thin lines. If the visual quality of the result is important, our method provides better results, as we are approximating real cylindrical geometry. Our method trades rendering speed for better quality.

Our method also allows us to map more scalar values to visual components like color, radius and texture in a more intuitive way than it would have been possible using simple line primitives. Values like divergence or rotation of a vector field map very well to radius and texture (as demonstrated in figure 11). In the left two images we mapped the velocity to a color scale ranging from purple over red to yellow. The divergence of the vector-field was integrated along the stylized line primitives and the integral was mapped to the radius, such that the radius increases where divergence is positive and decreases where it is negative. The flow direction was visualized by a saturation modulation of the color from low saturation to high saturation. In the two images on the right of Figure 11 we mapped the integrated rotation to the orientation of a tangential stripe texture. For this we integrated the projection of the 3d rotation vector onto the flow direction. If this component is zero, one can integrate surfaces that are orthogonal to the flow-field. If the component is not zero no surface can be integrated, which hints at a swirl. The mapping to a rotating texture as shown in the two images on the right of Figure 11 yields an intuitive understanding of the tangential component of the rotation vector.

Another application of the fast rendering of stylized line primitives outside of flow visualization is the rendering of huge molecular structures. Molecules often consists of a huge amount of atoms visualized as spheres, which can be splatted as ellipsoids as proposed by Gumhold in [8], and interconnections which can be rendered using our method. With this it is possible to visualize huge molecules in real time with high visual quality. An example of this can be seen in figure 1.

7 Conclusions

We presented a method to visualize stylized line primitives with generalized cylinders. Our fast rendering approach is visually near undistinguishable from a tessellation of the stylized lines. In some cases the proposed rendering approach even looks smoother. Our hybrid approach overcomes the problems which arise when only using flat quads to approximate the geometry. It optimizes rendering speed by on the one hand keeping the vertex data small and on the other hand the fragment shader simple. In this way we can achieve high fill rates and high segment counts.

With new, more powerful GPUs coming soon it will be possible to completely generate all needed data on the GPU, without having to rely on the CPU, which would greatly increase the effectiveness of our approach.

As a main application we had a look at the visualization of 3D flows from vector- or tensor-fields with streamlines. Compared to methods that tessellate the geometry our method is faster and more flexible, while even allowing to easily incorporate halos. Rendering streamlines as shaded line primitives is still faster than our method by the order of a magnitude but delivers visually less qualitative results, failing to provide the amount of depth perception created by splatting real geometry. Further useful applications, like rendering of molecular structures have been presented.

References

[1] G. Agin and T. Bindford. Computer descriptions of curved objects. *IEEE Trans. Computers*, C-25(4):439–449, 1976.

[2] Alberto S. Aguado, Eugenia Montiel, and Ed Zaluska. Modeling generalized cylinders via fourier morphing. *ACM Trans. Graph.*, 18(4):293–315, 1999.

[3] James F. Blinn. Jim Blinn's corner — optimal tubes. *IEEE Computer Graphics and Applications*, 9(5):8–13, September 1989.

[4] W.F. Bronsvoort, P.R. van Nieuwenhuizen, and F.H. Post. Display of profiled sweep objects. *The Visual Computer*, 5:147–157, 1989.

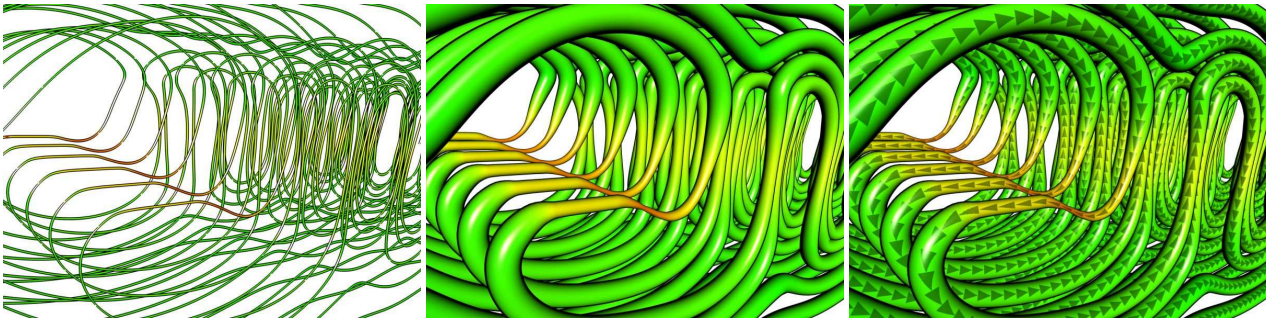

Figure 12: Comparing a dataset rendered with illuminated streamlines [16] on the left (flow velocity is mapped to color) to our rendering method in the center (flow velocity also mapped to streamline thickness) and enhanced even further with a texture marking flow direction on the right.

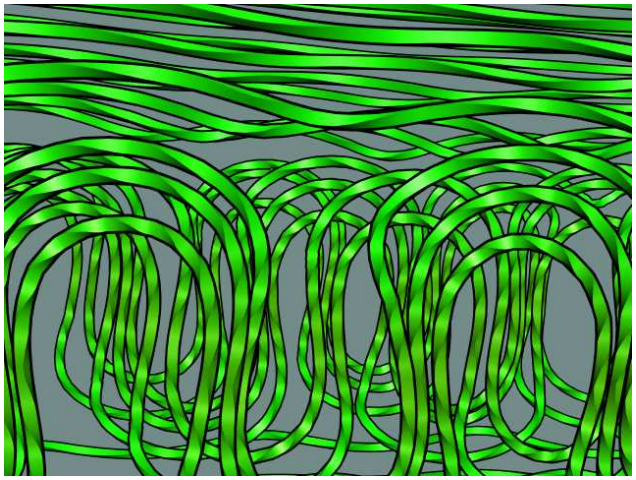

Figure 13: A streamline with rotation mapped to a normal map texture.

[5] Willem F. Bronsvoort and Fopke Klok. Ray tracing generalized cylinders. *ACM Trans. Graph.*, 4(4):291–303, 1985.

[6] Anton Fuhrmann and Eduard Gröller. Real-time techniques for 3d flow visualization. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 305–312, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[7] Laurent Grisoni and Damien Marchal. High performance generalized cylinders visualization. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 257, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Stefan Gumhold. Splatting illuminated ellipsoids with depth correction. In *Proceedings of 8th International Fall Workshop on Vision, Modelling and Visualization 2003*, pages 245–252, nov 2003.

[9] Stefan Gumhold. Designing optimal curves in 2d. In *Proceedings of CEIG 2004*, pages 61–76, Sevilla, Spain, July 2004.

[10] Nvidia developer pages, 2005.

[11] Oliver Mattausch, Thomas Theussl, Helwig Hauser, and Eduard Gröller. Strategies for interactive exploration of 3d flow using evenly-spaced illuminated streamlines. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 213–222, New York, NY, USA, 2003. ACM Press.

[12] Ivan Neulander and Michiel van de Panne. Rendering generalized cylinders with paintstrokes. In *Graphics Interface*, pages 233–242, June 1998.

[13] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. Feature extraction and visualization of flow fields. In *Eurographics 2002 State-of-the-Art Reports*, pages 69–100, Saarbrücken Germany, 2–6 September 2002. European Association for Computer Graphics, The Eurographics Association.

[14] Greg Schussman and Kwan-Liu Ma. Anisotropic volume rendering for extremely dense, thin line data. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 107–114, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):100–110, 1996.

[16] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive visualiztion of 3d-vector fields using illuminated streamlines. In *IEEE Visualization*, pages 107–113, 1996.