Advanced Rendering of Line Data with Ambient Occlusion and Transparency

David Groß, *Member, IEEE* and Stefan Gumhold

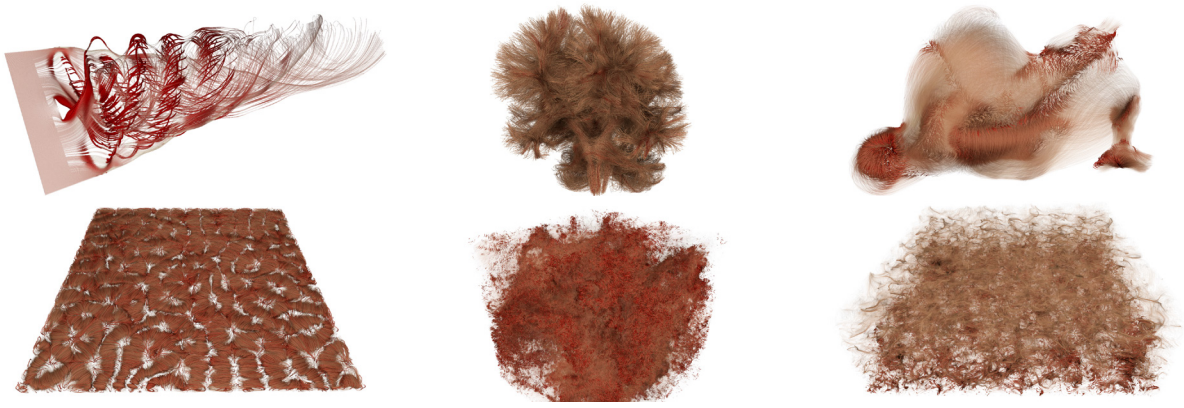

Fig. 1. Example renderings of the six data sets used during evaluation of our method, each rendered with local lighting, transparency and ambient occlusion. Data size ranges from approximately 100 thousand to 40 million segments. From top left to bottom right: *Propeller*, *Brain*, *Aneurysm*, *Convection*, *Turbulence* and *Clouds*.

Abstract—3D Lines are a widespread rendering primitive for the visualization of data from research fields like fluid dynamics or fiber tractography. Global illumination effects and transparent rendering improve the perception of three-dimensional features and decrease occlusion within the data set, thus enabling better understanding of complex line data. We present an efficient approach for high quality GPU-based rendering of line data with ambient occlusion and transparency effects. Our approach builds on GPU-based raycasting of rounded cones, which are geometric primitives similar to truncated cones, but with spherical endcaps. Object space ambient occlusion is provided by an efficient voxel cone tracing approach. Our core contribution is a new fragment visibility sorting strategy that allows for interactive visualization of line data sets with millions of line segments. We improve performance further by exploiting hierarchical opacity maps.

Index Terms—Scientific visualization, ray-casting, line rendering, transparency

1 Introduction

The efficient rendering of lines in 3D has become a valuable tool in scientific visualization. Simulations or the recording of data at higher resolutions generate increasingly large data sets. While local lighting is useful for perceiving the shape of individual lines, it is not sufficient to give an impression of large scale features like holes, rifts and cavities. Therefore, advanced effects from global illumination, like ambient occlusion or soft shadows, need to be applied to help discover these features. Furthermore, in densely sampled line sets inner structures of interest can be hidden by outside layers, hindering their perception and understanding. To overcome this problem the occlusion of structures needs to be reduced while still showing all of the data set, to preserve context. This can be achieved by applying transparency to the rendered geometry.

Several approaches exist for rendering lines. Most graphics APIs provide a line primitive type, which can be used to draw rasterized lines. This however, only allows to set a global line width given in pixels. Therefore all lines are displayed equally thick independent of perspective and distance to the camera. To overcome this issue, explicit geometry can be generated from the lines to render them as tubes. While feasible for smaller data sets, this approach can be impractical for large amounts of line segments, as a huge amount of geometry needs to be generated. This poses problems in memory consumption and rendering performance. An approach that does not suffer from the above mentioned problems is GPU-based ray casting [45, 47]. Here, first some proxy geometry - usually a camera facing billboard - is generated, which encompasses the primitive in screen space. In the fragment shader stage, for every generated fragment, a viewing ray can be calculated and tested for intersection with the primitive.

Incorporating global illumination effects in raster graphics is a challenging task, as only local geometry is accessible during the rendering procedure. Voxel-based global illumination [44] is one method to provide the necessary global scene information during rendering. A simplified representation of the scene is generated, by rasterizing geometry into a voxel grid in a pre-processing step. To allow for the computation of ambient occlusion, the density of the geometry inside each voxel needs to be accumulated. Based on this density the amount of blocked light can be determined by tracing rays and sampling the voxel grid. This can be efficiently done on the GPU using voxel cone tracing [5], which utilizes a hierarchical voxel structure.

The use of transparency in raster graphics requires blending of fragments in correct visibility order with respect to the camera position. Techniques purely based on object-order sorting cannot guarantee this

• *David Groß and Stefan Gumhold are with the Chair of Computer Graphics & Visualization, TU-Dresden, Dresden, Germany. E-mail: david.gross1—stefan.gumhold@tu-dresden.de.*

Manuscript received 30 Apr. 2020; revised 31 July 2020; accepted 14 Aug. 2020. Date of publication 7 Oct. 2020; date of current version 15 Jan. 2021. Digital Object Identifier no. 10.1109/TVCG.2020.3028954

for arbitrary geometry. Existing solutions overcome this problem either by accumulating all fragments in a large list [4, 46] and sorting them per-pixel in a second step, or approximating visibility using different techniques. Collecting all fragments is an unbounded memory problem, as the total number of generated list entries is not known beforehand. In practical terms, the storage has to be limited and can therefore not guarantee to hold all values. Approximate methods work well in practice, but cannot always deliver accurate results.

In this work, we present a method to efficiently visualize large line data sets, like the ones shown in Fig. 1, using GPU-based ray casting. We use truncated cones with spherical caps, which we call *rounded cones*, to render individual line segments. This shape enables the use of a varying radius along the line and ensures a smooth connection between two successive tube segments. Voxel cone tracing is used, to add ambient occlusion to the final result. For transparency, we combine order dependent and independent methods. We assume high resolution line sets for our method, with lines being composed of relatively short segments, compared to the overall extents of the data. When sorted on a per-segment basis, most of the incorrectly ordered fragments will be locally confined. Correct order is restored by using a temporary buffer in which local visibility conflicts are resolved.

Our main contributions are:

- A method to calculate tightly aligned billboards for rounded cone primitives.
- A novel approach to calculate fragment visibility order in a single geometry pass, which combines methods from order dependent and order independent transparency.
- An acceleration technique based on culling line segments to improve rendering performance on highly dense data sets, both for opaque and transparent rendering.

The remaining paper is organized as follows: In Sect. 2 we review work that is related to ours. We present an overview of our method in Sect. 3. Sect. 4 describes our approach to transparent rendering of rounded cones as well as the culling methods used to speed up rendering. In Sect. 5 we describe implementation details for the GPU-based rendering. Sect. 6 shows the results of our method being applied to six real world data sets, with a discussion on findings and drawbacks of our method. Conclusion and future work are presented in Sect. 7.

2 Related Work

In this section we present a selection of related work discussing ray casting on the GPU, advanced methods for visualizing line data sets and techniques for transparent rendering.

2.1 GPU-based Ray Casting

Traditional GPU-based rendering uses explicit geometry composed of triangle meshes. For simpler primitive shapes the approach of using an implicit representation of their geometry and performing ray casting on the GPU can be used to improve rendering efficiency and quality. While a bounding box is always applicable to produce the required fragments for ray casting, several methods have been presented to create much simpler or tighter proxy geometry.

A method for sphere primitives has been presented by Sigg et al. [40] and Reina [34]. Both show analytic formulations for finding the four corner points of a perspectively correct image-space bounding box. Gumhold [14] shows a technique for fast ray casting of ellipsoids on the GPU. The same is achieved by Klein and Ertl [21] by using different proxy geometry. Efficient quadrilateral billboards for cylinders are both covered by Wu et al. [47] and Toledo and Levy [45]. The latter work also uses optimized bounding geometry for cone sections and torus slices. They apply their ray casted primitives to produce high quality visualizations of industrial structures. While their cylinder billboard scheme could be adopted to work with rounded cones, we refrained from doing so. Using our method only requires a single quad to encompass all of the cones geometry and prevents the need to use additional proxy geometry for the caps. Most importantly to us, it also does not produce highly trapezoidal shapes for steep viewing angles, which is a prerequisite for our culling scheme presented in Sect. 4.3. In [11], Falk et al. present a method to transfer any bounding geometry to a screen-space bounding box. While this is not guaranteed to reduce the amount of generated fragments, it can dramatically simplify the required geometry.

2.2 Rendering of Line Sets

Illumination is a key aspect of helping in understanding a 3D scene. When using traditional line rasterization however, no 3D geometry is created and thus no normals are present for local lighting calculation. Zöckler et al. [48] overcome this issue by creating a definition for finding normal vectors on rasterized lines, which are coplanar with the light source direction. Precalculated textures are used to speed up rendering. An advancement to this method is presented by Mallo et al. [24], by defining a view dependent tangent frame for lines and a lighting model that averages reflectance over infinitesimally small cylinders. Preintegrated lighting information is stored in textures to enable efficient rendering supporting multiple light sources.

As simple line rasterization is limited, methods have been proposed that generate geometry from the line primitives. Kuhn et al. [22] use view aligned triangle strips to extend line width in image-space. They define a density on this strip based on the distance to the original line. Accumulation of multiple line strips emphasizes dense regions in the data set. Everts et al. [9] implement a stylized rendering of wide lines with depth dependent halos and depth cueing. They also apply their method to point sets. In a later work, Everts et al. [10] extend the stylized rendering to allow for mapping texture patterns like arrows onto lines. A parametrization based on data attributes is used to map local line features to styles. The usage of triangle strips to create the impression of tube geometry is shown by Stoll et al. [43]. They show a hybrid approach to render generalized cylinders with varying color and radius with simple geometry. On areas where planar primitives do not suffice, they tessellate geometry on the CPU. Similar approaches are shown by Schirski et al. [38] and [28]. Explicit geometry is used by [19], where tubes are created as triangle meshes.

To display lines as tubes without the need for generating explicit geometry, one can use ray tracing as shown in a recent work by Han et al. [17]. Their CPU-based implementation allows for high-performance visualization of generalized tube primitives and supports varying radii, bifurcations and transparency. An optimized GPU-based implementation of tube ray tracing is given by Kanzler et al. [18]. They encode line data into a voxel grid and encode the lines by discretizing their positions onto voxel boundaries. The voxel grid is used for accelerating the ray tracing process and to generate a density representation, which is further used for ambient occlusion and soft shadow effects. While the process of discretization reduces memory footprint, it also greatly influences visualization quality especially on low resolutions. Schussman and Ma [39] also use a voxel representation, which additionally captures directional anisotropy using spherical harmonics. Traditional volume rendering is used for visualization.

The usage of transparency in line rendering is both discussed in [15] and [29]. In the former work by Günther et al., an optimization process for determining view dependent transparency is presented. They allow for a user defined importance score to influence visibility calculation. Mishchenko and Crawfis [29] conduct a user study on the effectiveness of transparency in streamline rendering measuring the users ability to discern different flow directions. So far, a multitude of transparent rendering methods has been proposed. An overview of these methods as used in line rendering, as well as a performance evaluation, is given by Kern et al. [19].

Eichelbaum et al. [6] present LineAO, a screen-space solution to incorporate ambient occlusion in line rendering. They extend traditional screen-space ambient occlusion by sampling on multiple hemispheres and using distance based weighting. An object-space method based on voxel cone tracing [5] for particle data is shown by Staib et al. [41]. In addition they incorporate ambient illumination effects. Respectively, we apply voxel cone tracing to calculate ambient occlusion on a object-space level to our rounded cone rendering.

2.3 Transparency

Correct transparency in raster graphics involves visibility sorting on a per fragment basis. Carpenter [4] presents the concept of the a-buffer, which can be used as additional storage during rendering to accumulate fragment information. For transparency, one can store fragment color and depth and render them in sorted order in a second pass. Yang et al. [46] give an efficient way of implementing an a-buffer on modern GPUs, by using a global atomic counter and per-pixel linked lists. A similar method is proposed by Bavoil et al. [3] with their idea of a *k*-buffer, which serves as read-modify-write memory of *k* entries per pixel. This way, only the nearest *k* fragments get stored and blended correctly, with the remaining fragments being blended heuristically. We extend on this idea by using the *k*-buffer as a means of intermediate storage to reorder fragments as they are produced and blend them in correct visibility order. The process of depth peeling (DP), Everitt [8], involves the use of a dual depth buffer and multiple render passes. Each pass extracts the next closest fragment by comparing its depth with the previously stored value in the second depth buffer.

Several methods have been presented to achieve plausible results using order independent transparency. Hybrid transparency by Maule et al. [25] uses an exact compositing of a few foreground layers and a fast approximate method for the remaining fragments. By using a truncated a-buffer, memory needs stay bounded. McGuire and Bavoil [26] apply weights to the blending operation based on the fragments occlusion and distance to the camera. Salvi and Lefohn [35] employ an a-buffer implementation for their adaptive transparency technique, to build an approximate visibility function while accumulating fragments. In a second pass, fragments are blended in unsorted order using an additive blending operation. An approach using a per-pixel approximation of the transmittance function based on trigonometric or power moments is presented by Münstermann et al. [30]. The usage of additive blending is enabled by logarithmic scaling of the transmittance function. Salvi and Vaidyanathan [36] use a fixed-size storage per pixel to successively store layers and once full, blend the two most appropriate adjacent layers to keep memory needs bounded. Recently this method has been extended by Kern et al. [19] to reduce its dependency on the order in which fragments are generated. Kern et al. introduce depth buckets as a means to segment the scene into depth intervals and perform multi-layer alpha blending inside each bucket. The resulting values in each bucket are then blended in front-to-back order. Contrary to these methods, we rely on geometry being rendered in visibility order, although this may produce wrong fragment order at first. We provide a solution to restore the visibility order of fragments without the need for approximate blending functions.

Stochastic transparency by Enderton et al. [7] uses sub-pixel samples of transparent layers, with coverage relative to their opacity. The resulting image is created by averaging the samples. They provide a solution to reduce the noise inherent to this method. McGuire and Mara [27] also use a sample-based approach and show techniques for realistic rendering of translucent phenomena.

3 Method Overview

In this section we give an overview of our approach to render line sets with ambient occlusion and transparency. The whole process is shown in Fig. 2.

After loading the data and converting it to an internal format, we begin building the density volume. This is done by rasterizing the individual line segments as described in Sect. 5.3. This preprocessing step is performed only once per data set on the CPU. In order to render on the GPU, the prepared line data needs to be transferred into video memory. All of the proxy geometry needed for ray casting is generated on the GPU, so only the raw line data needs to be stored. Each line segment is defined by two positions, radii and RGBA color values. For opaque rendering these are stored in a vertex buffer object. A shader storage buffer object is used for transparent rendering instead (see Sect. 5.4).

During rendering the data is sorted according to the current view configuration. A fast GPU implementation is employed for this purpose, which directly uses the position buffer to calculate the distance values.

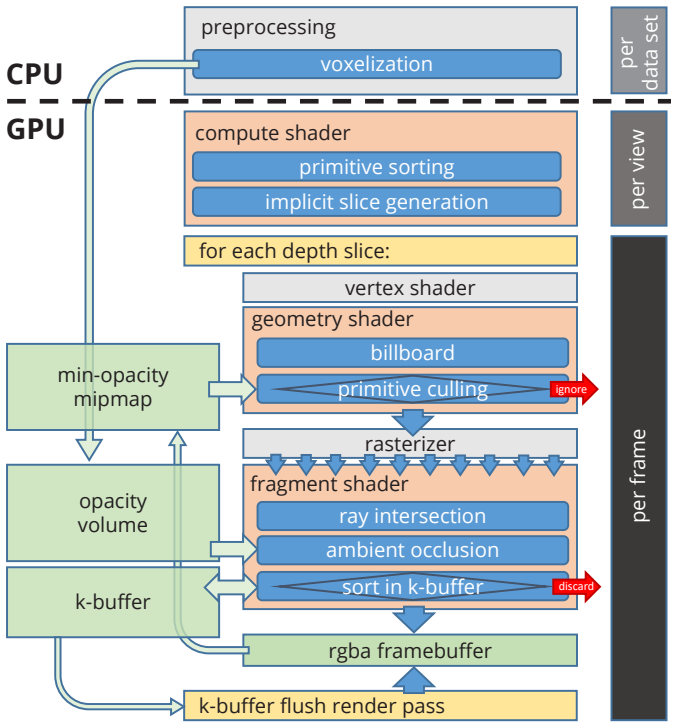

Fig. 2. Schematic illustration of the rendering procedure of our transparent rendering approach.

As a result of the sorting process detailed in Sect. 4.1, an index buffer can now be used to render line segments in per-object visibility order. By only drawing successive parts of the index buffer, the geometry is split into distinct depth slices. This is a perquisite to employ our primitive culling scheme as described in Sect. 4.3.

After each depth slice is drawn to a frame buffer object, the appropriate culling mipmap is created using a fast compute shader implementation. This can be used for the next slice, where it is utilized to speed up rendering. To render the rounded cone primitives in each depth slice, the geometry shader is used for generating view aligned billboards, which act as proxy geometry for the ray casting process. We show our approach to generate these billboards in Sect. 5.1. Using the culling mipmap we can determine if primitives shall be ignored for further processing, in turn saving GPU cycles.

An intersection test between the viewing ray and the rounded cone is performed in the fragment shader for each fragment produced after rasterization. If successful, we compute local lighting and evaluate the ambient occlusion term using the precomputed density volume. Using the depth information, we insert each fragment into the *k*-buffer, ensuring that the entries are kept in sorted order (see Sect. 4.2). Fragments are discarded until the intermediate buffer is completely occupied, at which point the overflowing fragments get blended to the screen. The process repeats until every slice is drawn. To complete the image a full screen pass blends the remaining fragments in the *k*-buffer to the screen.

In the case of rendering fully opaque, the evaluation of the shading and ambient occlusion term is omitted until every slice is drawn. Instead the color, position and normal values are stored in frame buffer attachments and a full screen deferred shading pass is performed to limit costly calculations to visible pixels only.

4 Visibility Sorting

Our approach to transparent rendering is based on object order. This requires the rounded cones to be rendered according to their visibility to the view point. We sort primitives in front to back order, i.e. by increasing depth, to support our hierarchical culling strategy.

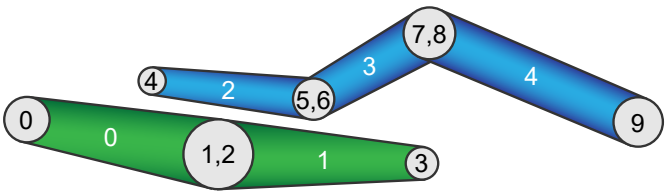

Fig. 3. Illustration of the indexing scheme used in our implementation. Vertex indices are given in black and segment indices in white.

4.1 Sorting of Rounded Cone Primitives

For visibility sorting of the rounded cone primitives we use the minimum distance to the view point over the line segment connecting the cone's end points. To prevent reordering the whole data an index buffer is used for rendering, which holds two indices per line segment. It is possible to sort the index buffer directly, by rearranging both indices as a pair. However, keeping the amount of memory per sort element low is key to high sorting performance. We therefore use another index buffer with a single index per segment, to sort the segments by distance to the camera. By carefully choosing our indexing scheme as shown in Fig. 3, we can compute the final vertex indices from the sorted segment indices as $L_0(S_i) = 2S_i$ and $L_1(S_i) = 2S_i + 1$. We finally render all line segments in a single indexed draw call with line primitives. While this method has the disadvantage of duplicated indices where two segments are connected, it allows for simple calculation of the index buffers without additional information on the GPU. Based on the sort criterion we use a GPU-based implementation of a prefix-sum radix sort similar to [16]. Sorting is recomputed only after changes to the viewing parameters.

4.2 Fragment Sorting

If segments are very close or even overlap, a pure per primitive sorting can produce an incorrect fragment order. It is therefore necessary to rearrange these fragments into their appropriate order during rendering. Using additional read-modify-write memory during the render process, allows us to temporarily store and manipulate fragment data. This can be achieved by employing a variant of the *k*-buffer [3], which is able to store *k* entries for each pixel. The buffer is initialized to be empty upon each frame.

Once the final color and depth value of a fragment is determined, it gets stored into the corresponding pixel buffer by following an insertion sort strategy, where all fragments with lower depth are moved by one slot in the list and the new one is inserted at the correct position. After *k* fragments have been generated for a specific pixel, the nearest will be pulled from the list, making room for the next fragment. Up to the *k* th fragment rendered per pixel, no output is written to the image, as all fragment values are stored to the intermediate buffer. After the whole geometry has been processed, each per-pixel buffer can have up to *k* entries, which have not yet been blended to the image. A second full-screen pass blends the fragment colors together, considering the correct visibility order, and writes them to the image.

When implemented on a GPU, two key problems to this approach arise. First, due to the parallel processing nature, more than one fragment falling into the same pixel can be generated at once, leading to race conditions when writing to the intermediate buffer. Second, the order in which geometry is processed cannot be guaranteed on the fragment level. Atomic operations can be used to overcome the first issue, as was shown by Liu et al. [23]. They effectively peel of the front-most *n* layers using atomic compare functions while looping over the buffer entries. Each entry is represented by a 32-bit unsigned integer, packing compressed depth and color information. As depth information is stored in the most-significant bits, the per-pixel lists are always sorted in visibility order. Modern OpenGL Hardware supports atomic operations on 64-bit unsigned integers using the NV_shader_atomic_int64 extension [32]. We therefore pack the 32-bit depth information in the most significant bits of a 64-bit unsigned integer. The lower bits are filled by color values with each RGBA component mapped to [0,255]. Using 64-bit instead of 32-bit unsigned integers allows us to keep full depth and sufficient color accuracy. Contrary to [23], we do not discard fragments overflowing from the per-pixel lists, but rather blend them to the screen.

After a fragment entry has been determined, it is stored in a temporary variable. We then loop over all *k* entries of the intermediate buffer, using an atomic max operation to compare the temporary variable to the one stored in the buffer. The process is stopped, when an empty slot is found or after looping over the whole buffer. Upon insertion of a new value into the buffer, the temporary variable will hold the entry with the smallest depth value and the entries will be sorted in depth-descending order. We restore the color information from the minimal entry and output the fragment.

OpenGL only guarantees to blend fragments in the order at which the objects they are generated from are drawn, but not how they are processed. Using this reordering strategy results in fragment shader invocations possibly writing fragments that originated from a different primitive. Solving this second problem can be accomplished by utilizing the ARB_fragment_shader_interlock [20] extension. The extension allows us to specify a critical section around the atomic loop used for sorting fragments. This ensures that, for two pairs of shader invocations overlapping the same pixel, only one is executing the atomic loop at a given time. Using this scheme, local ordering issues can be effectively overcome to produce correct transparency in a single geometry pass.

4.3 Culling of Primitives

The need for sorting objects using our approach to transparency enables the optimization of the rendering process. The general idea follows the one presented in Grottel et al. [12], where they render opaque particle data using a two-stage culling scheme. We adapt the vertex-level culling to work with rounded cones and our rendering method that supports transparent rendering. Drawing the primitives in a front-to-back manner allows to use information from geometry closer to the view point being used for successive objects. While rendering on the GPU, information from finished primitives is not readily available in the same render pass. We therefore split the scene into separate depth slices, drawing the front most slice first. Generating the slices is implicitly done by issuing multiple draw calls for successive parts of the index buffer, which prevents the need to split individual segments. It should be noted that the per-slice rendering is independent of the fragment sorting inside of the *k*-buffer. The buffer is not cleared between two slices and the final blend of the remaining fragments happens only after the last geometry slice is drawn. After each slice that is drawn, the frame buffer contains the according color and opacity information. Opacity values are then extracted and written into a separate 2D texture with filtering set to GL_NEAREST. A compute shader implementation is used to generate mipmap levels for this texture, by combining 4 neighboring texels and choosing the minimum opacity value. We omit the first level of the mipmap, with full image resolution, to avoid copying large amounts of data and instead directly write to the second level. The minimum-opacity mipmap is bound for rendering the next slice, where it can be sampled in the rounded cones shader program. After generating the image-space bounding quad, we transform this into a screen-aligned bounding box. Using the largest edge length of this bounding box, a mipmap level can be determined, whose texel size is greater or equal to this length. This way, the aligned bounding box overlaps a maximum of four texels of the mipmap level. Four texture samples are taken, each containing the minimum opacity value present in the pixel it covers. By taking the minimum of these samples, we can discard the primitive based on an opacity threshold, as further blending operations will only have a diminishing impact on the final color values. For our implementation we start culling at opacity values greater than 0.95. Primitives are discarded in the geometry shader by simply not writing any output. Through substitution of the minimum-opacity mipmap with a maximum-depth mipmap this optimization is also applicable to fully opaque rendering. The early-z test implemented on the GPU, is disabled when explicitly writing to the depth buffer, as done by the rounded cone ray casting approach. For opaque rendering a lot of

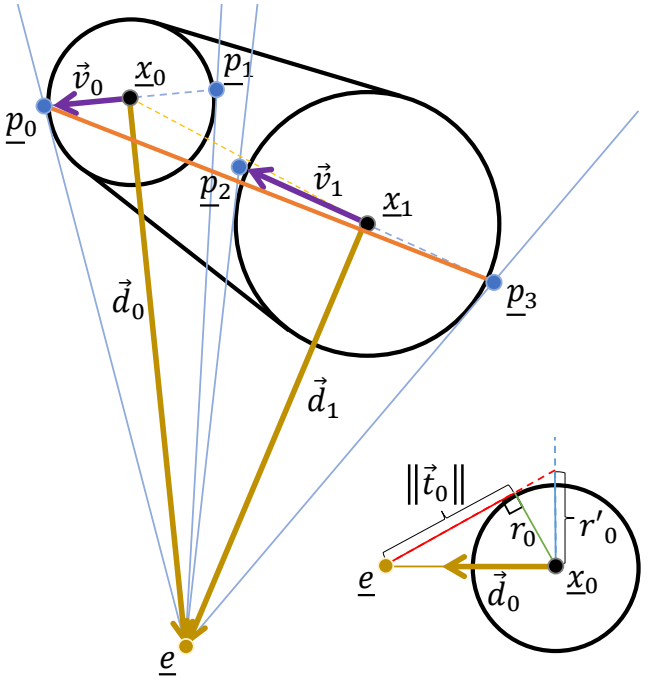

Fig. 4. Top view of the rounded cone showing the projection of points $\underline{p}_0$ to $\underline{p}_3$ and support directions. Bottom right shows the projection of a sphere silhouette to determine the corrected radius r'_0 on the example of the start cap sphere.

costly intersection tests are evaluated on fragments, that end up being occluded in the final image. A conservative depth estimate for each segment is made by choosing the closest of the two defining points and subtracting the maximum radius. This depth is then tested against the four texture samples. The primitive is discarded, when no sample has a depth value larger than the conservative estimate. In case of opaque rendering sorting is not strictly required, such that our culling technique introduces a notable overhead. We therefore only sort when larger changes in the camera position or view angle occur.

In addition to primitive-level culling we apply per-fragment culling using the minimum-opacity or maximum-depth mipmaps. This helps to remove invisible parts of primitives, which did not get removed in the previous step because of partial visibility. While this basically re-implements depth testing for opaque rendering, it also provides a way of discarding individual fragments when using transparency.

Through usage of the culling mipmaps, we can regain some of the lost performance in opaque rendering and speed up transparent rendering. The extension to culling whole primitives, in addition to only fragments, reduces the amount of geometry, that has to be produced by the geometry shader.

5 GPU-Based Ray Casting of Rounded Cones

Rendering the rounded cones is done using GL_LINE primitives. The vertex stage is a simple pass-through shader, which receives a position, radius and color as input. The proxy geometry needed for ray casting is built on the fly in the geometry shader, following the algorithm described in Sect. 5.1, and assembled as a triangle strip conststing of 4 vertices. The geometry shader is set up to receive lines, so that both the start and end point variables are accessible. The ray primitive intersection is computed as described in Sect. 5.2. The resulting depth is converted to window coordinates and passed on to the *k*-buffer insertion described in Sect. 4.2.

5.1 Billboards

For efficient ray casting on the GPU, in the context of raster graphics, it is necessary to perform the intersection test on as few fragments as possible. In order to generate these fragments, some geometry needs to be generated beforehand, which covers the whole silhouette of the rounded cone in image space. Using a bounding box for this purpose is both easy to implement and guaranteed to fulfill this requirement. However, for rounded cones the same can be achieve by using a single quad. Our goal is to generate a quad that is aligned with the cones main axis in image space, tightly encompassing its silhouette while remaining close to rectangular even at steep viewing angles. While our billboards might closely resemble screen-space bounding boxes, like discussed in [45], at first glance (see Fig. 5), they actually differ from them in that they rotate with the cone in screen-space. This prevents the generation of unnecessary fragments, especially for long thin tubes.

The general idea behind our approach can be described as follows: The spherical caps of the rounded cone primitive define the start and end point of the cone along its main axis. We can therefore implicitly define spheres at both of the caps center positions. For each of these spheres we can calculate two points located on the silhouette as seen from the camera, making sure they are aligned with respect to the main axis. From the outermost points we can construct the four corners of the billboard. The following explanation details this process and applies to cones that fulfill the condition $r_0 \leq r_1$, with r_0 being the start cap radius and r_1 the end cap radius. We later show how to modify this approach to work in the case of $r_0 > r_1$.

Let a cone be defined by start and end position $\underline{x}_0$ and $\underline{x}_1$. The direction of the cones main axis is defined as $\vec{d} = \underline{x}_1 - \underline{x}_0$. The directions from the start- and endpoint towards the view point $\underline{e}$ are given by $\vec{d}_0 = \underline{e} - \underline{x}_0$ and $\vec{d}_1 = \underline{e} - \underline{x}_1$. We can then define support unit vectors $\hat{v}_0$, $\hat{v}_1$ and $\hat{u}$ via

$$\begin{aligned} \hat{u} &= \vec{d} \times \vec{d}_0 \cdot \|\vec{d} \times \vec{d}_0\|^{-1} \\ \hat{v}_0 &= \vec{u} \times \vec{d}_0 \cdot \|\vec{u} \times \vec{d}_0\|^{-1} \\ \hat{v}_1 &= \vec{u} \times \vec{d}_1 \cdot \|\vec{u} \times \vec{d}_1\|^{-1}. \end{aligned} \tag{1}$$

This way, $\hat{u}$ is always orthogonal to the cones main axis and both $\hat{v}_0$ and $\hat{v}_1$ lie in the plane defined through $\underline{e}$ and $\vec{u}$. As such, they appear parallel to the cones main axis, when projected into image-space. Directions $\vec{d}_0$, $\hat{v}_0$ and $\hat{u}$ form a tangent space basis on the surface point $\underline{x}_0$ of a sphere centered at $\underline{e}$ with radius $\|\vec{d}_0\|$. The same applies for $\underline{x}_1$.

The tangent space directions can be used to define corner points of the camera facing billboard for each sphere by scaling them accordingly. Due to perspective distortion however, the given start and end radius for the cone cannot be used directly. Instead, the radii must be calculated as to take perspective distortion into account. Starting from the view point $\underline{e}$, a cone can be defined, which touches the sphere tangentially. Because of rotational symmetry around the axis defined by the line from $\underline{e}$ to $\underline{x}_0$, we can reduce this to a 2D problem. The process of calculating the corrected radius r'_0 is illustrated in Fig. 4 (bottom right). A line is defined, starting at $\underline{e}$, touching the sphere tangentially. Together with r_0 and the spheres center point $\underline{x}_0$, a triangle can be constructed. Using the rule of three we can solve for the new radius:

$$\begin{aligned} \frac{r'_0}{\|\vec{d}_0\|} &= \frac{r_0}{\|\vec{t}_0\|} \\ r'_0 &= \|\vec{d}_0\| \cdot \frac{r_0}{\|\vec{t}_0\|} = \|\vec{d}_0\| \cdot s_0 \end{aligned} \tag{2}$$

with

$$\|\vec{t}_0\| = \sqrt{\|\vec{d}_0\|^2 - {r_0}^2}. \tag{3}$$

This approach differs from the one presented in [34], as our formulation gives us a scaling factor s_0, and direct control over the distance at which we want to calculate r'_0. The corrected radius r'_1 can be determined likewise.

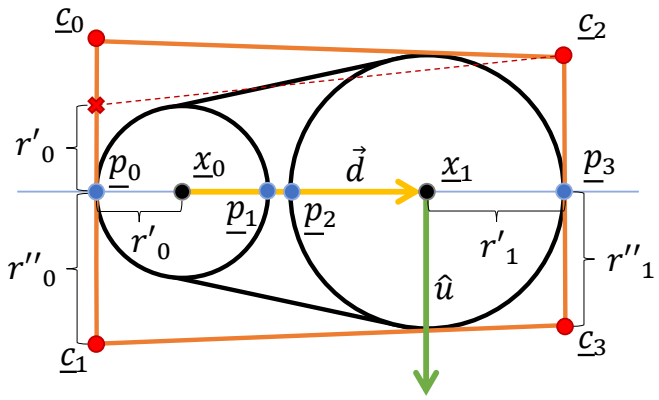

Fig. 5. Illustration of the rounded cone billboard as viewed from the camera perspective.

Now r'_0 and r'_1 can be used to scale $\hat{v}_0$ and $\hat{v}_1$ to fit to the spheres extents in direction of the cones main axis. We can now define four points as seen in Fig. 4, which delimit the cone start and end cap spheres on the plane defined by $\underline{e}$ and $\vec{u}$ in image space:

$$\begin{aligned} \underline{p}_0 &= x_0 + r'_0 \cdot \hat{v}_0, \quad \underline{p}_1 = x_0 - r'_0 \cdot \hat{v}_0, \\ \underline{p}_2 &= x_1 + r'_1 \cdot \hat{v}_1, \quad \underline{p}_3 = x_1 - r'_1 \cdot \hat{v}_1. \end{aligned} \tag{4}$$

The corner positions $\underline{c}_0$ to $\underline{c}_3$ shown in Fig. 5 will be calculated in the final step, by adding and subtracting the scaled direction $\hat{u}$. However, r'_0 and r'_1 cannot be used for this purpose as the start and end radius of the rounded cone can be different. If this were the case, $\underline{c}_0$ and $\underline{c}_1$ would be located too close to the main axis and the lines connecting $\underline{c}_0$ to $\underline{c}_2$ and $\underline{c}_1$ to $\underline{c}_3$ would intersect the larger sphere centered at $\underline{x}_1$, as depicted by the red dotted line in Fig. 5.

A conservative adjustment of the scaling factors of $\hat{u}$ has to be given. We do this by choosing the sphere with the largest image space projection using the scaling factors determined in Equation 2. By substituting $\|\vec{d}_0\|$ and $\|\vec{d}_1\|$ with 1 we can determine which sphere silhouette appears larger on screen, when both are at the same distance to the view point. The maximum of both scaling factors is defined as $s_m = \max(s_0, s_1)$. The scaled radii in direction $\vec{u}$ can then be determined as

$$r''_0 = \|\vec{d}_0\| \cdot s_m \quad \textit{and} \quad r''_1 = \|\vec{d}_1\| \cdot s_m. \tag{5}$$

All of the points $\underline{p}_0$ to $\underline{p}_3$ may be located on the silhouette of the rounded cone depending on view direction, though only two positions are going to fulfill this at any given time. Fig. 4 shows a view configuration for which $\underline{p}_0$ and $\underline{p}_3$ are the delimiting positions of the silhouette. Due to the way $\vec{v}_0$ is defined, $\underline{p}_0$ is always located to the left of $\underline{p}_1$ in image space. The same applies to $\underline{p}_2$ and $\underline{p}_3$. Using this property we can determine the leftmost position of the cones projection by choosing between $\underline{p}_0$ and $\underline{p}_2$ and the rightmost position is given by either $\underline{p}_1$ or $\underline{p}_3$. To chose $\underline{p}_0$ over $\underline{p}_2$ or vice versa we have to compare their distance to the cone center in image-space and choose the point with the largest distance. This is achieved by first generating a direction orthogonal to $\vec{u}$ and the camera view direction $\vec{v}$ via

$$\vec{w} = \hat{u} \times \hat{v}. \tag{6}$$

Then the scalar product is used to calculate the angles between $\underline{p}_0$ and $\underline{p}_2$, as seen from the view point $\underline{e}$, and $\vec{w}$:

$$\alpha_{0,2} = \langle \frac{\underline{p}_{0,2} - \underline{e}}{\|\underline{p}_{0,2} - \underline{e}\|}, \frac{\vec{w}}{\|\vec{w}\|} \rangle. \tag{7}$$

By comparing both angles we can find the position of the leftmost point on the rounded cone silhouette and the corresponding radius with

$$(\underline{p}_s, r_s) = \begin{cases} (\underline{p}_0, r''_0), & \text{if } \alpha_0 \leq \alpha_2 \\ (\underline{p}_2, r''_1), & \text{otherwise} \end{cases}. \tag{8}$$

The same procedure is applied for $\underline{p}_1$ and $\underline{p}_3$ by calculating the angles α_1 and α_3 using Equation 7. The rightmost silhouette position $\underline{p}_e$ and the corresponding radius r_e are the determined via

$$(\underline{p}_e, r_e) = \begin{cases} (\underline{p}_3, r''_1), & \text{if } \alpha_1 \leq \alpha_3 \\ (\underline{p}_1, r''_0), & \text{otherwise} \end{cases}. \tag{9}$$

The corner positions for the image-space bounding box of the rounded cone are calculated as

$$\begin{aligned} \underline{c}_0 &= \underline{p}_s - r_s \cdot \hat{u}, \quad \underline{c}_1 = \underline{p}_s + r_s \cdot \hat{u}, \\ \underline{c}_2 &= \underline{p}_e - r_e \cdot \hat{u}, \quad \underline{c}_3 = \underline{p}_e + r_e \cdot \hat{u}. \end{aligned} \tag{10}$$

As noted at the beginning of this section, this procedure works for rounded cones satisfying $r_0 \leq r_1$. To generalize the approach to arbitrary start and end radii, only requires to swap $\underline{x}_0$ and $\underline{x}_1$ as well as their radii.

5.2 Ray Intersection

After generating the view ray in the fragment shader the intersection with the rounded cone primitive needs to be determined. As the rounded cone is built from a truncated cone fitted with spherical caps, finding the intersection point of the combined primitive can be accomplished by testing against the simpler individual primitives. The final intersection can then be determined through a constructive solid geometry approach. Analytic formulations for ray-sphere and ray-cone intersections are well defined. This leaves the parameters for the truncated cone to be determined, such that the surface touches the spheres tangentially. The process is detailed in Han et al. [17]. We decided on an optimized formulation of this idea, which we found to be optimal for our purposes [33].

5.3 Ambient Occlusion

To incorporate ambient occlusion into the shading process, first a simplified representation of the scene needs to be created. Scene voxelization is done in a preprocessing step before rendering. As we only considered static data sets during our testing, we opted for a slower CPU-based implementation. The voxelized scene is stored in a single floating point component 3D texture of format GL_RED. A target resolution is chosen, which specifies the number of grid cells along the dimension with the largest extent of the data set. The other dimensions are chosen to fit the whole data set with uniform voxels along these axes. By using a 3D texture, quadrilinear interpolation by the GPU can be used to blend between successive levels of detail in the voxel cone tracing step.

Voxelization is done per line segment, by traversing the grid using the algorithm detailed by Amanatides and Woo [2]. Knowing the entry and exit point of the segment in each grid cell, we can determine the height and, through linear interpolation, the base and top radius of the corresponding truncated cone. The volume of this truncated cone divided by the voxel volume gives the approximate density contribution of each interval of the line segment. The contribution is further scaled by the opacity of the line at each intersection, which is interpolated from the start and end opacity. Density is accumulated at each voxel for all line segment intersections.

To apply voxel cone tracing similar to [5, 41], a hierarchical structure of the voxel density grid has to be generated. Coarser levels are created with half the resolution of previous levels by averaging density values. For our static test scenes, we decided on using the standard OpenGL function glGenerateMipmaps, as it generates a suitable mipmap-pyramid of the 3D texture, although faster solutions have been used in practice [41]. Evaluation of the occlusion term is performed by tracing 3 cones tightly aligned around the surface normal. With cone

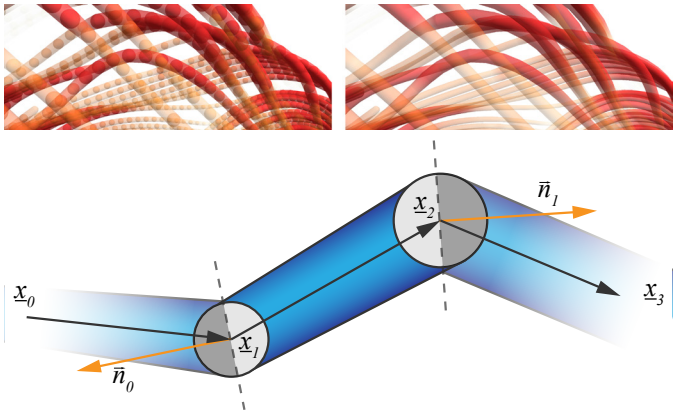

Fig. 6. Not clipping internal geometry for transparent lines leads to sphere artifacts (top left). By employing the clipping approach as seen in the bottom row, these can be successfully removed (top right).

opening angles ranging from 40° to 60°, a good approximation of the surrounding hemisphere can be achieved.

5.4 Fixing Overlapping Geometry

We chose rounded cones as our render primitive to depict line segments, because their spherical caps fill the gaps between two successive segments. This has the drawback of geometry overlapping at the point where two segments are connected, as can be seen in Fig. 6. Removing these overlaps is therefore necessary to give the impression of smoothly connected tubes.

The process is illustrated in Fig. 6. Let a segment be defined by two positions $\underline{x}_1$ to $\underline{x}_2$ and $\underline{x}_0$ the previous, respectively $\underline{x}_3$ the successive position in this line. We define clipping directions for the caps as

$$\vec{n}_0 = -\frac{\underline{x}_2 - \underline{x}_0}{\|\underline{x}_2 - \underline{x}_0\|} \quad \textit{and} \quad \vec{n}_1 = +\frac{\underline{x}_3 - \underline{x}_1}{\|\underline{x}_3 - \underline{x}_1\|}. \tag{11}$$

Through directions $\vec{n}_0$ and $\vec{n}_1$, normals for two clipping planes are defined. These planes are positioned at $\underline{x}_1$ and $\underline{x}_2$ respectively. When a ray-primitive intersection occurs during rendering, we check if the intersection point lies in between these two planes.

As clipping the spherical caps is undesirable for segments located at the start and beginning of a line, we have to encode at which points they should be removed. This is done by moving all of the line positions to the positive half of the coordinate system in direction of the x-axis, thus leaving all x-components greater or equal than zero. The sign bit can then be utilized as a flag, indicating if the segment should be clipped at the corresponding position. As access to the adjacent segments is necessary to retrieve their positions, a shader storage buffer object is used to store vertex data – alternatively the GL_LINES_ADJACENCY primitive type can be used[1]. Because we require the indices of successive line segments to be in strictly ascending order with increments of one (see Sect. 4.1), calculating the adjacent vertex indices is trivial. Special care must be taken before using the positional values in any computation, as the encoding has to be removed first by taking the absolute value of the x-component.

6 Results and Discussion

To test the effectiveness of our method, we created an implementation in C++ using the CGV-Framework [13] using OpenGL as the graphics API. Our test system was equipped with an Intel Core i9-9900K with 8×3.6 GHz, 64 GB RAM and an Nvidia GeForce RTX 2080 Ti with 11 GB VRAM. The RTX feature set was not utilized during the rendering process. All performance measurements were taken at a view port resolution of 1920×1080. We averaged the achieved frame time while orbiting the view point around the data set center for a predefined duration, making one full revolution in that time. Camera parameters were chosen so that each data set fills most of the view port, while filling the whole view port for the additional near performance tests.

[1]This requires to specify indices of adjacent vertices in the index buffer.

Table 1. The data sets used for evaluation, listed with their respective segment count and voxel grid resolution. Given in the last column are timings for the visibility sorting of the segments in ms.

data set	# segments	voxel resolution	sorting time
Propeller	121,840	256 x 64 x 64	0.23
Brain	1,118,841	214 x 223 x 256	1.19
Aneurysm	2,267,219	226 x 221 x 256	2.28
Convection	9,859,794	256 x 256 x 14	9.13
Turbulence	17,468,339	256 x 256 x 256	15.28
Clouds	39,600,000	256 x 91 x 256	34.95

Scene voxelization was performed with a target resolution of 256 voxels for all data sets, with resulting resolution given in Table 1. While, for smaller line sets, sufficient quality can be achieved with less resolution, we found this to be optimal in our testings, as a good compromise between memory consumption and detail is given. The difference in performance for other voxel resolutions is negligible. Unless stated otherwise, all measurements for transparent rendering were conducted with $k = 4$, using in total 64 MB of additional video memory. We used 10 depth slices for all measurements during primitive culling.

While our evaluation is limited to static data sets, our method is capable of handling animated data as well, as no geometry pre-processing is needed. Sorting is performed every frame on the GPU, effectively defining the depth slices for no additional cost. Our blending is independent of sliced rendering, so that no popping artifacts are produced as a result of primitives shifting between slices. When ambient occlusion is necessary, a faster method can be used, like in [41] to calculate the density volume interactively for each animation frame.

Six real world data sets were used to evaluate the rendering performance and accuracy of transparent rendering. They were chosen based on different levels of complexity and data size. The data sets, that can be seen in Fig. 1, are as follows:

- *Propeller*: 500 streamlines seeded on a straight line in the velocity field of a rotating ship propeller simulation. Angular velocity is mapped to transparency and cone radius.
- *Brain*: 23,419 short DTI fiber tracts generated from a scan of the whole human brain. Color and radius is inversely mapped from the local line density. Transparency is set to a constant value of 50%.
- *Aneurysm*: 9213 streamlines, which were randomly seeded in the interior of an aneurysm. Line transparency is derived from the vorticity of the underlying flow field.
- *Convection*: 99,605 streamlines, seeded uniformly in a Rayleigh-Bénard convection, simulating a cold top and heated bottom wall [37]. Transparency is mapped from line curvature.
- *Turbulence*: 80,000 long streamlines, filling the complete simulation domain. Streamlines were generated in a forced turbulence field as with a resolution of 1024^3. The process is described by Aluie et al. [1].
- *Clouds*: 400,000 seeded streamlines from a large eddy simulation (LES) of a cloud resolving boundary layer [42]. Transparency is derived from the magnitude of vorticity along each streamline.

6.1 Performance Analysis

Given the fact that our method to resolve transparency requires objects being rendered in visibility order, we need to sort the segments of each data set during rendering. The duration of one full sorting run, using our GPU implementation, depends on the number of segments in a

data set and is given in Table 1, along with the number of segments. As our method is not only applicable for transparent visualizations, we measured performance for fully opaque renderings to get insight about the influence of culling primitives and the impact of evaluating ambient occlusion. Table 2 (top) lists average render times in ms for an animation around the data sets, as described at the beginning of this section. In addition, a closeup test has been conducted. As stated in Sect. 4.3, sorting the segments while culling is active is only performed on large view changes, which results in 10 sorts for the used camera animation. For all data sets except *Propeller*, increased render speed can be measured, when applying the culling scheme presented in Sect. 4.3. The *Propeller* data set is sparsely populated and does not suffer from large amounts of occlusion, hence the extra calculations needed for the maximum-depth mipmap produce too much overhead, compared to the low number of culled primitives. Performance is increased by up to a factor of 2.8 (see *Aneurysm* D and E in Table 2) for the other data sets. We note, that the culling scheme is most applicable for dense and cube-like shaped data sets, while still delivering increased effectiveness in closeup exploration. Shading the opaque rendered geometry is deferred to a second render pass. This drastically reduces the amount of fragments, for which ambient occlusion needs to be determined. Evidence of the effectiveness can be seen in Table 2 (top). The evaluation of the ambient occlusion term reduces performance only by about 3.3% on average and has less impact on larger data sets, as most of the geometry is occluded.

The same measurements were performed for the transparent rendering case, with the results being shown in Table 2 (bottom). Performance is overall reduced, which is to be expected. The measured times include the run time of the sorting step for each frame. Our implementation achieved interactive frame rates for every data set except *Clouds*. Similar to the opaque rendering, our culling scheme proves helpful in increasing performance when data sets are densely sampled, as can be seen in *Turbulence* and *Clouds* data sets. However, with the addition of transparency there is another factor which influences the effectiveness of culling. This is especially noticeable on the *Aneurysm*, where large portions of the lines are very transparent with only small opaque features. In this scenario the accumulated opacity does not surpass the culling threshold and therefore only little geometry is culled, resulting in similar performance compared to rendering without culling. In contrast to opaque rendering, evaluating the ambient occlusion term has a larger impact on performance, because it needs to be calculated for every produced fragment, slowing down render time on average by 24%.

The accuracy of our approach can be controlled by the choice of parameter k, which determines the intermediate fragment list size for each pixel. This not only influences memory consumption but also render times. Measurements for different values of k on three selected data sets are shown in Table 3. We give render times with and, to isolate the impact of k on render times, without culling. Frame times increase for larger values of k, while the effect is less pronounced when culling is enabled. This is especially visible for dense data sets with moderate transparency with a potentially large amount of occlusion.

Table 2. Measured render time in ms for opaque (top) and transparent (bottom) rendering at two different zoom levels *A*: far, *B*: far with culling, *C*: far with culling and ambient occlusion, *D*: near, *E*: near with culling, *F*: near with culling and ambient occlusion

data set	A	B	C	D	E	F
Propel1er	1.2	2.1	2.3	1.8	2.7	2.8
Brain	6.3	3.8	4.1	14.8	6.5	6.7
Aneurysm	10.2	4.7	4.9	24.2	8.6	8.9
Convection	17.1	14.8	14.9	15.5	15.1	15.4
Turbulence	45.6	20.7	21.1	52.3	26.3	26.8
Clouds	60.3	37.1	37.7	90.2	41.7	42.3
Propel1er	4.6	5.7	6.6	4.8	6.3	7.4
Brain	14.9	12.3	17.9	26.8	13.2	19.6
Aneurysm	24.1	24.0	36.6	44.9	42.5	67.5
Convection	62.9	58.8	86.1	43.1	38.9	50.0
Turbulence	144.9	93.7	141.4	120.7	69.1	93.9
Clouds	3229.0	1442.8	1463.1	1974.1	540.6	573.0

Table 3. Render times in ms compared between different values for the fragment list size k when culling is applied (first) and without culling (second).

data set	2	4	8	16
Aneurysm	20.1 / 20.9	24.0 / 24.1	27.9 / 29.8	30.8 / 32.8
Convection	52.3 / 57.2	58.8 / 62.9	66.4 / 73.0	72.6 / 79.6
Turbulence	84.2 / 129.8	93.7 / 144.9	104.7 / 166.0	112.2 / 179.9

To give an overview of the performance of our method in the context of line rendering, we compared it to two recently presented techniques – Multi-Layer Alpha Blending with Dept Bucketing (MLABDB) from [19] and Voxel-based Ray Casting (VRC) [18]. We adopted an implementation of both approaches [31] to our needs to enable comparison with our method. Transfer functions and line radius for each data set were chosen to be identical across all implementations. Measured times, given in Table 4, show the average duration in ms to render one frame, averaged over a mixture of far and near camera movements. While we intend our method to be used with culling enabled, we have also given results without culling. These timings are independent of the used transfer function and represent worst case performances for highly transparent data sets. The results show that no method is clearly superior over the others. MLABDB is, like our method, highly dependent on overall fragment count, with frame times being higher for the larger data sets. The performance of VRC is heavily determined by the view port resolution and voxel grid resolution. VRC performs worst on the *Aneurysms* data set, partly due to the high grid resolution compared to data detail. The other factor is the highly transparent transfer function used during testing, which prevents early ray termination and therefore requires more computation. Similar, our method achieved the least speedup when enabling culling. The impact of culling was most noticeable on the *Turbulence* data set, where the performance of our implementation even surpassed VRC. For the *Convection* data set VRC produced the fastest render times, as the grid resolution is rather small in one dimension. However, this comes at the cost of decreased line accuracy.

6.2 Qualitative Analysis

To evaluate the quality of renderings produced with our approach, we again compared it to MLABDB, VRC and a ground thruth solution acquired from DP, with results being shown in Fig. 7. Overall the results appear very similar with each method giving an adequate impression of transparency. Zoomed in views for regions of interest reveal differences and shortcomings of each approach. The green region of Fig. 7 shows small imperfections of our approach compared to the ground truth solution. In comparison, MLABDB looks slightly blurry, however, all methods achieve good results. A zoomed in view of opaque features being covered by layers of transparent lines is shown in the blue region. Our method and VRC are very similar to the ground truth producing correct results for transparency. The result from MLABDB shows some of the opaque features more pronounced due to the approximate blending approach. Additionally, possibly as a result of the color compression, MLABDB produces a slight shift in color compared to the original transfer function. VRC delivers overall correct transparency but sacrifices line quality, which is a direct result of the discretization

Table 4. Performance comparison between MLABDB, VRC and our method on 4 chosen data sets. Frame time is given in ms.

data set	MLABDB	VRC	ours	ours + culling
Aneurysm	23.0	98.3	29.5	27.8
Convection	37.9	16.9	72.4	46.6
Turbulence	117.5	97.3	131.3	82.7
Clouds	225.9	118.6	2460.3	910.6

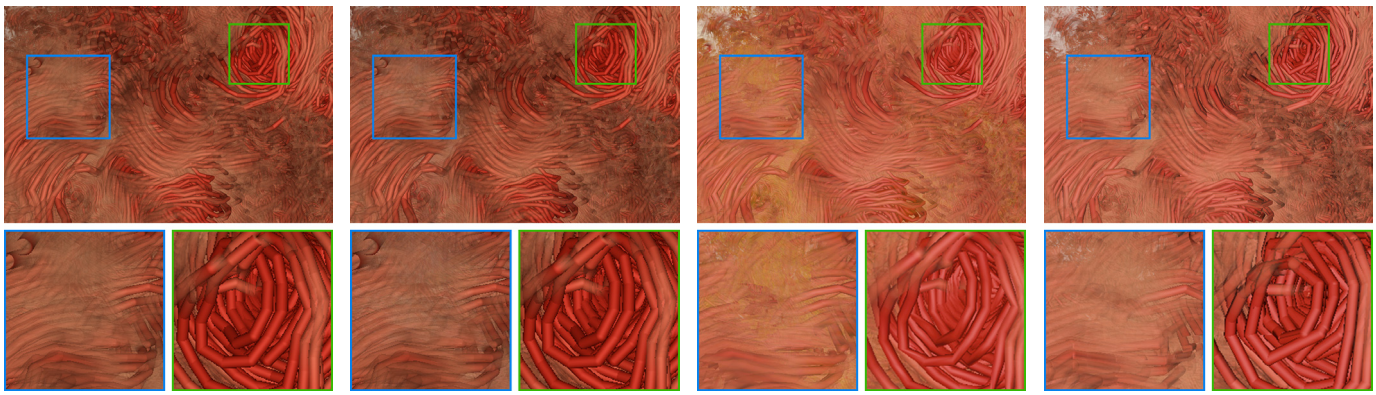

Fig. 7. Comparison of different transparency rendering methods for lines on the *Turbulence*, with zoomed in views for regions of interest. From left to right: Ground truth from DP, our method, MLABDB, VRC. Due to small differences in local lighting our renderings appear slightly darker.

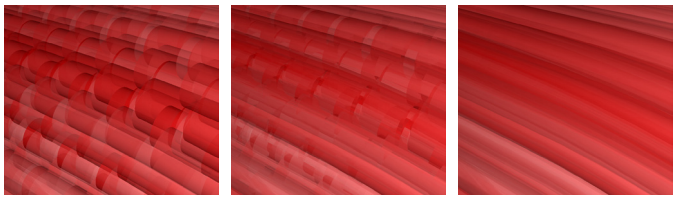

Fig. 8. Comparison between transparency based purely on object order (left), our method using $k = 4$ (middle) and a ground truth solution rendered with DP (right). A list size of 4 is not sufficient to fully resolve the wrong fragment order on highly overlapping tubes.

into a voxel grid. Our approach is superior in line accuracy and quality, although this comes at the cost of longer render times. An application of using a differing tube radius can be seen in Fig. 9.

Comparing our approach to the ground truth solution shows its effectiveness in restoring the correct fragment order. Overall a correct impression of the transparent layers is accomplished. Unwanted blending of internal geometry to outside layers cannot be observed and transparent layers are well defined without blurry edges. We attribute this to not using an approximate blending scheme to combine fragments. Upon further inspection minor artifacts can be seen in Fig. 7 (green square). While this is almost imperceptible in the final image, it shows a shortcoming of our approach which we further investigate.

Limitations of our intermediate sorting strategy can be seen in Fig. 8. We compare our method to a ground truth solution rendered with DP for densely sampled lines, that interpenetrate a lot in the *Brain* stem region, showing artifacts arising due to fragments being blended in wrong order. As reference, Fig. 8 (left) shows the result of blending purely based on object order, to illustrate the kind of local errors our method needs to resolve. With the used buffer size of $k = 4$ our method was not able to sort all fragments to their correct visibility order in this challenging example. Compared to other approaches using approximate blending methods [19, 26, 35, 36], which tend to blur out features or produce wrong transparency and color impression, our method produces sharp, more pronounced artifacts when unable to sort all fragments correctly, yet produces correct results in most cases. When orbiting the view these artifacts can suddenly appear or disappear, as primitives change their order – although this is barely noticeable on typical viewing distances. Effectively a buffer of size k can restore order for all fragments not more than k layers away from the correct sorting position. With lines packed too close this cannot be guaranteed for every pixel.

7 Conclusion and Future Work

In this work we introduced a method for interactive visualization of 3D line data sets with up to millions of segments, incorporating ambient occlusion and applying a novel approach to transparent rendering. We build our approach on GPU-based ray casting using optimized billboards. The rounded cone primitive allows for smooth connections between successive line segments and a simple culling method to remove overlapping internal structures in transparent rendering. Using GPU-based ray casting to draw the rounded cones allows for high image quality independent of zoom level. Transparency based on object order with intermediate fragment sorting produces correct results for lines with limited overlap.

Limitations of our approach are noticeable on densely packed lines, where a larger buffer size k is necessary to achieve correct visibility order. The need for explicit sorting introduces overhead, which is more noticeable on large data sets. By employing our culling scheme we can gain back performance lost to sorting. Culling has proven especially helpful for opaque rendering.

For future work we will focus on substituting rounded cones with other primitives, like generalized cylinders following quadratic bézier splines. We will investigate if this can be used to resample lines, reducing the number of segments and providing smoother curvature along the line. Additionally we are interested in incorporating more advanced lighting effects into the rendering, as well as defining a volumetric density model for the tubes to improve 3D perception.

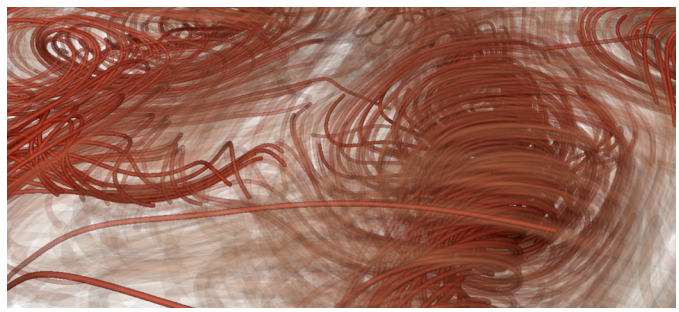

Fig. 9. Visualization of the *Convection* data set, with curvature inversely mapped to radius. Thinner tubes help to discern details in the dense regions, while the thicker transparent lines provide context.

Acknowledgments

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

References

[1] H. Aluie, G. Eyink, V. E., S. C. K. Kanov, R. Burns, C. Meneveau, and A. Szalay. Forced MHD turbulence data set. `http://turbulence.pha.jhu.edu/docs/README-MHD.pdf`. Accessed: 20-April-2020.

[2] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proc. of Eurographics Conference on Technical Papers*. Eurographics Association, 1987. doi: 10.2312/egtp.19871000

[3] L. Bavoil, S. P. Callahan, A. Lefohn, J. a. L. D. Comba, and C. T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proc. Symp. Interactive 3D Graph. and Games*, pp. 97–104. Association for Computing Machinery, New York, NY, USA, 2007. doi: 10.1145/1230100.1230117

[4] L. Carpenter. The a-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, Jan. 1984. doi: 10.1145/964965.808585

[5] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing: A preview. In *Proc. Symp. Interactive 3D Graph. and Games*, p. 207. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/1944745.1944787

[6] S. Eichelbaum, M. Hlawitschka, and G. Scheuermann. LineAO - improved three-dimensional line rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):433–445, Mar. 2013. doi: 10.1109/TVCG.2012.142

[7] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. Stochastic transparency. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1036–1047, Aug. 2011. doi: 10.1109/TVCG.2010.123

[8] C. Everitt. Interactive order-independent transparency. *NVIDIA Corporation*, 2, Oct. 2001.

[9] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg. Depth-dependent halos: Illustrative rendering of dense line data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, Nov./Dec. 2009. doi: 10.1109/TVCG.2009.138

[10] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg. Interactive illustrative line styles and line style transfer functions for flow visualization. *CoRR*, abs/1503.05787, Mar. 2015.

[11] M. Falk, S. Grottel, M. Krone, and G. Reina. *Interactive GPU-based Visualization of Large Dynamic Particle Data*. Morgan and Claypool, 2016. doi: 10.2200/S00731ED1V01Y201608VIS008

[12] S. Grottel, G. Reina, C. Dachsbacher, and T. Ertl. Coherent culling and shading for large molecular dynamics visualization. *Computer Graphics Forum*, 29(3):953–962, Aug. 2010. doi: 10.1111/j.1467-8659.2009.01698.x

[13] S. Gumhold. The computer graphics and visualization framework. `https://github.com/sgumhold/cgv`. Accessed: 20-April-2020.

[14] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In T. Ertl, ed., *Proc. VMV*, pp. 245–252. Aka GmbH, Nov. 2003.

[15] T. Günther, C. Rössl, and H. Theisel. Opacity optimization for 3d line fields. *ACM Trans. Graph.*, 32(4), July 2013. doi: 10.1145/2461912.2461930

[16] L. K. Ha, J. H. Krüger, and C. T. Silva. Fast four-way parallel radix sorting on GPUs. *Comput. Graph. Forum*, 28(8):2368–2378, Dec. 2009. doi: 10.1111/j.1467-8659.2009.01542.x

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

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

[19] M. Kern, C. Neuhauser, T. Maack, M. Han, W. Usher, and R. Westermann. A comparison of rendering techniques for 3D line sets with transparency. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, Feb. 2020. doi: 10.1109/TVCG.2020.2975795

[20] Khronos Group. ARB_fragment_shader_interlock. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_fragment_shader_interlock.txt`. Accessed: 15-April-2020.

[21] T. Klein and T. Ertl. Illustrating magnetic field lines using a discrete particle model. In *Proc. VMV*, pp. 387–394. Aka GmbH, 2004.

[22] A. Kuhn, N. Lindow, T. Günther, A. Wiebel, H. Theisel, and H.-C. Hege. Trajectory density projection for vector field visualization. In M. Hlawitschka and T. Weinkauf, eds., *Proc. EuroVis - Short Papers*. The Eurographics Association, 2013. doi: 10.2312/PE.EuroVisShort.EuroVisShort2013.031-035

[23] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. FreePipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proc. Symp. on Interactive 3D Graph. and Games*, pp. 75–82. Association for Computing Machinery, New York, NY, USA, 2010. doi: 10.1145/1730804.1730817

[24] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo. Illuminated lines revisited. In *Proc. IEEE Visualization*, pp. 19–26, 2005. doi: 10.1109/VISUAL.2005.1532772

[25] M. Maule, J. a. Comba, R. Torchelsen, and R. Bastos. Hybrid transparency. In *Proc. Symp. on Interactive 3D Graph. and Games*, pp. 103–118. Association for Computing Machinery, New York, NY, USA, 2013. doi: 10.1145/2448196.2448212

[26] M. McGuire and L. Bavoil. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, Dec. 2013.

[27] M. McGuire and M. Mara. Phenomenological transparency. *IEEE Transactions on Visualization and Computer Graphics*, 23(5):1465–1478, May 2017. doi: 10.1109/TVCG.2017.2656082

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

[29] O. Mishchenko and R. Crawfis. On perception of semi-transparent streamlines for three-dimensional flow visualization. *Comput. Graph. Forum*, 33(1):210–221, Feb. 2014. doi: 10.1111/cgf.12268

[30] C. Münstermann, S. Krumpen, R. Klein, and C. Peters. Moment-based order-independent transparency. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1), July 2018. doi: 10.1145/3203206

[31] C. Neuhauser and M. Kern. OIT rendering tool (PixelSyncOIT). `https://github.com/chrismile/PixelSyncOIT`. Accessed: 14-April-2020.

[32] NVIDIA Corporation. NV_shader_atomic_int64. `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_atomic_int64.txt`. Accessed: 16-April-2020.

[33] I. Quilez. Ray-rounded-cone intersection. `https://www.shadertoy.com/view/MlKfzm`. Accessed: 22-April-2020.

[34] G. Reina. *Visualization of uncorrelated point data*. PhD thesis, University Stuttgart, Jan. 2009. doi: 10.18419/opus-2649

[35] M. Salvi, J. Montgomery, and A. Lefohn. Adaptive transparency. In *Proc. of the ACM SIGGRAPH Symp. on HPG*, pp. 119–126. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/2018323.2018342

[36] M. Salvi and K. Vaidyanathan. Multi-layer alpha blending. In *Proc. Symp. Interactive 3D Graph. and Games*, pp. 151–158. Association for Computing Machinery, New York, NY, USA, 2014. doi: 10.1145/2556700.2556705

[37] J. D. Scheel, M. S. Emran, and J. Schumacher. Resolving the fine-scale structure in turbulent Rayleigh–Bénard convection. *New Journal of Physics*, 15(11):113063, Nov. 2013. doi: 10.1088/1367-2630/15/11/113063

[38] M. Schirski, T. Kuhlen, M. Hopp, P. Adomeit, S. Pischinger, and C. Bischof. Efficient visualization of large amounts of particle trajectories in virtual environments using virtual tubelets. In *Proc. ACM SIGGRAPH Int. Conf. VRCAI*, pp. 141–147. Association for Computing Machinery, New York, NY, USA, 2004. doi: 10.1145/1044588.1044615

[39] G. Schussman and K. . Ma. Anisotropic volume rendering for extremely dense, thin line data. In *IEEE Visualization 2004*, pp. 107–114, 2004. doi: 10.1109/VISUAL.2004.5

[40] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Proc. Eurographics / IEEE VGTC Conference on Point-Based Graphics*, pp. 59–65. Eurographics Association, Goslar, DEU, 2006.

[41] J. Staib, S. Grottel, and S. Gumhold. Visualization of particle-based data with transparency and ambient occlusion. *Computer Graphics Forum*, 34(3):151–160, July 2015. doi: 10.1111/cgf.12627

[42] B. Stevens. Introduction to UCLA-LES. `https://www.mpimet.mpg.de/fileadmin/atmosphaere/herz/les_doc.pdf`. Accessed: 18-July-2020.

[43] C. Stoll, S. Gumhold, and H. . Seidel. Visualization with stylized line primitives. In *Proc. IEEE Visualization*, pp. 695–702, 2005. doi: 10.1109/VISUAL.2005.1532859

[44] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller. Voxel-based global illumination. In *Proc. Symp. Interactive 3D Graph. and Games*, pp. 103–110. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/1944745.1944763

[45] R. Toledo and B. Levy. Visualization of industrial structures with implicit GPU primitives. In *Proc. Symp. Advances in Visual Computing*, vol. 5358. Springer, Berlin, Heidelberg, 12 2008. doi: 10.1007/978-3-540-89639

-5_14
[46] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the GPU. In *Proc. of the 21st Eurographics Conference on Rendering EGSR*, pp. 1297–1304. Eurographics Association, Goslar, DEU, 2010. doi: 10.1111/j.1467-8659.2010.01725.x
[47] W. Zhaocong, N. Wang, J. Shao, and G. Deng. GPU ray casting method for visualizing 3D pipelines in a virtual globe. *International Journal of Digital Earth*, 12:1–14, Jan. 2018. doi: 10.1080/17538947.2018.1429504
[48] M. Zockler, D. Stalling, and H. . Hege. Interactive visualization of 3D-vector fields using illuminated stream lines. In *Proc. IEEE Visualization*, pp. 107–113, 1996.