Cone Tracing for Furry Object Rendering

Hao Qin, Menglei Chai, Qiming Hou, Zhong Ren and Kun Zhou, ***Senior Member, IEEE***

Abstract—We present a cone-based ray tracing algorithm for high-quality rendering of furry objects with reflection, refraction and defocus effects. By aggregating many sampling rays in a pixel as a single cone, we significantly reduce the high supersampling rate required by the thin geometry of fur fibers. To reduce the cost of intersecting fur fibers with cones, we construct a bounding volume hierarchy for the fiber geometry to find the fibers potentially intersecting with cones, and use a set of connected ribbons to approximate the projections of these fibers on the image plane. The computational cost of compositing and filtering transparent samples within each cone is effectively reduced by approximating away in-cone variations of shading, opacity and occlusion. The result is a highly efficient ray tracing algorithm for furry objects which is able to render images of quality comparable to those generated by alternative methods, while significantly reducing the rendering time. We demonstrate the rendering quality and performance of our algorithm using several examples and a user study.

Index Terms—ray tracing, fur rendering, depth of field, antialiasing, reflection, refraction, shadows, cone tracing

1 Introduction

FUR and hair are among the most important features of avatar personalization [1], and can be found on most virtual characters in digitally created contents such as movies and games. Researchers have been developing efficient approaches over the years for rendering realistic fur and hair, taking into account complex visual effects including transparency, self-shadowing and multiple-scattering. Despite the significant progress, cinematic-quality rendering of furry objects is still time consuming especially in the presence of ray tracing effects such as reflection and refraction and camera effects like depth of field (DOF).

A major challenge faced by any fur renderer is the thin geometry of fur fibers, which requires extremely high supersampling rates to produce an aliasing-free image, especially when rendering camera effects like DOF. For ray tracing based renderers, this means a vast amount of rays need to be traced to produce the antialiasing samples (or visibility samples). Furthermore, since fur fibers are often rendered as transparent strands, a significant number of ray-fur intersections may have to be composited to produce each antialiasing sample. The final pixel colors are computed by downsampling the colors and opacities of the antialiasing samples using a filter function. The combined computational cost of sampling, compositing and filtering makes high-quality ray tracing of furry objects highly expensive. As ray tracing attains greater significance in high quality rendering [2], [3], [4], it is of great interest to overcome these challenges and develop efficient ray tracing techniques for fur and hair.

• *The authors are with the State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou, China, 310058. Email: {qinneo, cmlatsim, hqm03ster, zren6ing}@gmail.com, kunzhou@acm.org.*

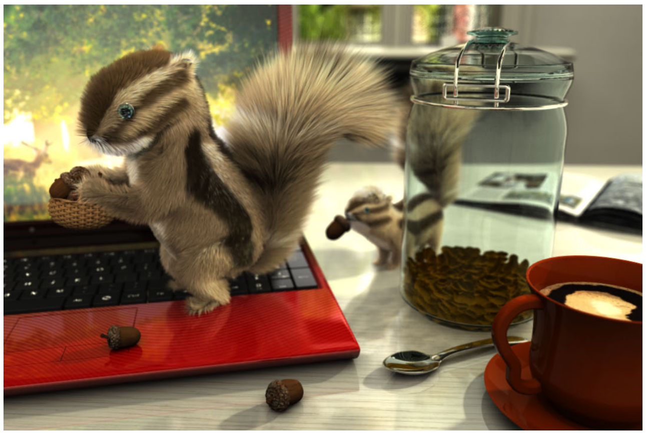


Fig. 1. A scene with two squirrels rendered with reflection and refraction effects at 1080× 720 resolution. The scene contains 368K fur fibers.The image is rendered in 1,081 seconds on an NVIDIA GTX 570 GPU with no supersampling for the viewing rays and 11×11 supersampling for the reflection and refraction rays. In contrast, the stochastic ray tracing algorithm takes 3,722 seconds to render an image of comparable quality under the supersampling rate of 21×21.

In this paper, we present a cone-based ray tracing approach for high-quality rendering of furry objects. By aggregating all sampling rays in a pixel as a single cone, we significantly reduce the high supersampling rate required by the thin geometry of fur fibers. The result is a highly efficient ray tracing algorithm which is able to render images of quality comparable to those generated by alternative methods, while significantly reducing rendering time. We demonstrate the rendering quality and performance of our algorithm using several examples and a user study.

1.1 Related Work

Our work is most related to fur/hair rendering and bundled ray tracing. In the following we only cover the most relevant references, as the literature covering these topics is vast.

Fur Rendering The light scattering model for a single fur/hair fiber has been well studied [5], [6]. Most recent research focuses on rendering hair with complex visual effects, such as transparency and self shadowing [7], [8], [9], multiple scattering [10], [11] and natural illumination [12], [13].

Fur fibers can be densely diced into general primitives, such as micropolygons, and rendered by ray tracing [14]. Specific algorithms [15] have also been developed for directly ray tracing curves representing fur fibers. Nonetheless, the fine geometry of fur fibers poses significant difficulties for sampling and antialiasing, and the cost of obtaining a noise-free ray traced image is high. Fur fibers can also be voxelized for efficient ray tracing [12], [16]. This coarse approximation suitable for evaluating irradiance due to multiple scattering, however, is not precise enough for tracing view or shadow rays.

Bundled Ray Tracing Bundled ray tracing aims to solve the sampling and aliasing problems that plague conventional ray tracing approaches. The basic idea is to trace coherent bundles of rays as beams [17], cones [18], [19] or hypercubes [20]. Igehy [21] developed a general and robust model for the interactions between the ray bundles and scene, where the ray footprints are estimated according to the differentials of the ray properties with respect to the screen coordinates. A chain rule is also developed to handle multiple surface interactions. These methods effectively calculate the path of every possible ray within each bundle and are therefore not prone to under-sampling or over-sampling. This alleviates the sampling and aliasing problems faced by ray tracing. The computational complexity associated with the formation of ray bundles and intersecting them with scene primitives, however, is often much higher than that of individual rays. Specific algorithms are often needed for different kinds of scene primitives.

Our approach is based on cone tracing for a specific type of scene primitive, i.e., fur fibers represented as a series of connected line segments with linearly interpolated per-vertex widths. We choose cones instead of other bundle representations as its shape more closely represents the image filter used in downsampling antialiasing samples. Crassin et al. [19] use cone tracing for interactive indirect illumination. Wand and Straßer [22] propose to intersect anisotropic ray cones with prefiltered and oriented surface sample points from a multi-resolution point hierarchy. However, their method cannot be directly applied to fur tracing. To avoid unintended blurring between thin fibers, the distance between sample points has to be well below the average distance between fibers, resulting in an impractically large set of points. Lacewell et al. [23] extend Wand and Straßer's idea to prefilter occlusion of aggregate geometry, e.g., foliage or hair, and store the directional opacity in a bounding volume hierarchy. At runtime, the prefiltered occlusion is used for efficient rendering of soft shadows and ambient occlusion effects. This method, however, cannot be used to handle view and reflection/refraction rays, which requires more accurate ray-fur intersection computation.

1.2 Contributions

Our main contribution is an efficient cone-based ray tracing algorithm for high-quality furry object rendering. As mentioned above, in our algorithm fur fibers are represented as a series of connected line segments with linearly interpolated per-vertex widths, which means the geometry of each fur fiber is a generalized cylinder with the connected line segment as its axis. We focus on tackling two challenges caused by this special geometry of fur fibers, which has not been addressed by previous cone tracing techniques.

The first challenge is the high cost of intersecting fur fibers with cones. Computing such intersections precisely would negate the benefit of the reduced supersampling rates. Our algorithm first constructs a bounding volume hierarchy (BVH) for the fiber geometry and traverses the BVH to find all fibers that may intersect each cone. The projections of these fibers on the image plane are then approximated as a set of *ribbons* (or quadrilaterals), each of which corresponds to a line segment of a fiber. Finally, instead of computing the intersections between the cone and ribbons, we evaluate the intersection area of each ribbon with the cone, which suffices for further compositing and filtering computations. The second challenge is to handle transparency within each cone. Complex fur geometry may generate a considerable amount of transparent cone intersections, resulting in expensive compositing computation. We solve this problem by approximating away in-cone variations of shading, opacity and occlusion. Specifically, we assume the depth order required for compositing transparent samples does not change within each cone and perform the composition on a per-cone basis. To facilitate such a compositing order, we convert each cone-ribbon intersection into a single *effective opacity* according to the intersection area and an aggregated shading by further assuming shading and opacity are smooth within each cone.

Compared to alternative ray tracing methods, our algorithm is able to generate images of comparable quality in significantly less rendering time according to our experiments (see Fig. 8) as well as a simple user study. Furthermore, image errors caused by the approximations made in our algorithm can be reduced by increasing the supersampling rates and decreasing the cone size (see Fig. 12).

Our fur rendering algorithm can be easily implemented on the GPU, and integrated into a ray tracing framework. As exemplified in Fig. 1, a moderately complex scene with a refractive glass bottle, a reflective laptop pane and two furry squirrels is rendered with ray tracing effects. Our algorithm is able to render the image in 1081 seconds – 3.4× faster than the results produced by brute force supersampling with similar image quality.

Note that our cone tracing algorithm handles view rays, reflection, refraction and shadow rays, but does not deal with the multiple scattering among fur fibers.

2 Cone Tracing Fur Fibers

For an image rendered at the resolution of n pixels with $m \times m$ supersampling, our algorithm needs $n \times m^2$ cones, each of which is traced for each pixel (or subpixel if $m > 1$). As illustrated in Fig. 2, we represent a cone by its apex $\boldsymbol{o}$, the ray direction $\boldsymbol{v}$ and two 2D vectors $\boldsymbol{R}_x$ and $\boldsymbol{R}_y$ on a reference plane Π perpendicular to $\boldsymbol{v}$ and offset by a unit distance from $\boldsymbol{o}$ in the ray direction. The two vectors are determined by the major and minor axes of the sheared ellipse formed by the intersection of Π with the cone. For the simplest case of cone formation shown in Fig. 2, the viewpoint is taken as the ray cone apex and connected with the circumcircle of a pixel on the image plane to form a cone for the pixel. More complicated cases of cone formation for DOF, reflection and refraction effects are similar to that of [22] and explained in detail in Section 2.3.

In the following we first describe how to compute the shading value for each cone assuming the ribbons intersecting the cone are known, and then explain how to generate these ribbons.

2.1 Compositing and Filtering Within a Cone

To compute the shading value L for each cone, suppose we can generate a set of ray samples Υ in the cone to intersect the potentially intersected ribbons Ω, yielding a set of sample points, each of which is associated with a shading value ϕ and an opacity value α. The shading value of the cone can be computed by averaging the composited shading values of all ray samples, or more precisely:

$$L = \frac{\sum_{i\in\Upsilon}\sum_{j\in\Omega_i}\left(\alpha_{i,j}\phi_{i,j}\prod_{k\in\Omega_{i,j}}(1-\alpha_{i,k})\right)}{|\Upsilon|}, \quad (1)$$

where Ω_i is the set of ribbons hit by ray i. The pair (i, j) specifies a sample point generated by ray i and ribbon j. $\Omega_{i,j}$ is the set of ribbons which are hit by ray i and are located in front of the current sample (i, j).

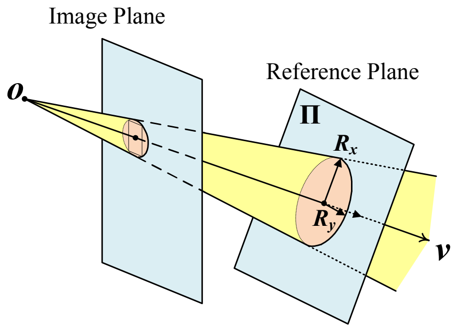


Fig. 2. Illustration of our cone representation.

Eq. (1) can be rewritten in the form of summing over the ribbons by exchanging the summation order,

$$L = \frac{\sum_{j\in\Omega}\sum_{i\in\Upsilon_j}\left(\alpha_{i,j}\phi_{i,j}\prod_{k\in\Omega_{i,j}}(1-\alpha_{i,k})\right)}{|\Upsilon|}, \quad (2)$$

where Υ_j is the set of ray samples that hits ribbon j.

Since in high-quality rendering fibers are diced densely to ensure enough shading precision and a smooth curve representation [14], [15], the size of each ribbon is often very small. Hence we choose to assume the opacity α, shading ϕ and occluding ribbon set $\Omega_{i,j}$ do not change over the entire ribbon, and approximate the shading by

$$\begin{aligned} L &\approx \textstyle\sum_{j\in\Omega}\left(|\Upsilon_j|\alpha_j\phi_j\prod_{k\in\Omega_j}(1-\alpha'_k)\right)/|\Upsilon| \\ &= \textstyle\sum_{j\in\Omega}\left(\alpha'_j\phi_j\prod_{k\in\Omega_j}(1-\alpha'_k)\right), \end{aligned} \quad (3)$$

where ϕ_j is the average shading value of ribbon j, and the effective occluding ribbon set Ω_j of ribbon j can be determined by comparing the average depth values of ribbons. The fraction $|\Upsilon_j|/|\Upsilon|$ converges to the fraction of the cone area covered by ribbon j. The effective opacities $\alpha'_k = \alpha_k|\Upsilon_k|/|\Upsilon|$, $\alpha'_j = \alpha_j|\Upsilon_j|/|\Upsilon|$ take into account the original opacity of ribbon k (or j) and its intersection area with the cone, or the fraction of the cone area covered by the ribbon. In doing this, we ignore the actual overlapping relationships among ribbons, and approximate the occlusion of ribbon k using the fraction of the cone area covered by the ribbon. We will analyze the shading error caused by these approximations in Section 4.

In short, to compute the shading value for a cone, we loop over the set of potentially intersected ribbons of the cone. For each ribbon, we compute an average shading value and effective opacity value and yield a sample. These samples are then composited according to the order of the average depth value of each ribbon to compute the shading value of the cone. In computing the effective opacity α'_k of each ribbon k, we need to evaluate the intersection area of each ribbon with the cone.

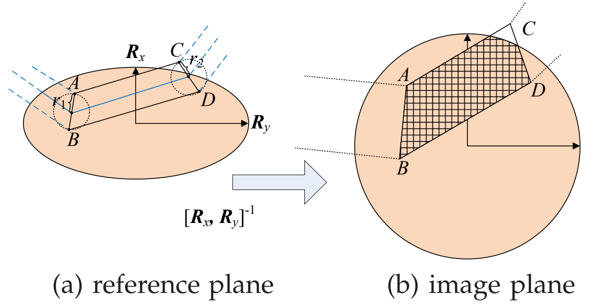


Fig. 3. A fiber is projected to the reference plane of the cone. For each line segment of the fiber, a quad is constructed to approximate the projection (a). The quad is transformed to the image plane to produce a ribbon corresponding to the line segment (b).

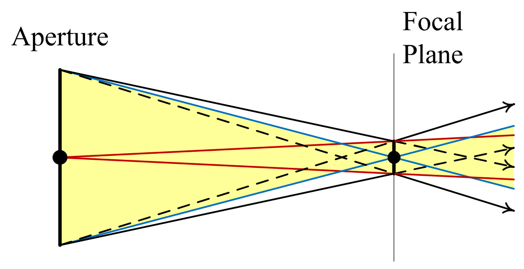


Fig. 4. Cone formation for DOF.

2.2 Cone-Fiber Intersections

Now we describe how to generate the potentially intersected ribbons for each cone and compute the intersection area of each ribbon with the cone.

We first construct a BVH for the fibers using the surface area heuristic (SAH) [24]. As aforementioned, the fiber geometry is a generalized cylinder with the connected line segments as its axis. For each line segment, we construct an axis aligned bounding box (AABB) for the two spheres centered at the two ending vertices of the segment, each of which has a radius equal to the width value at the vertex. This AABB is regarded as the basic geometric primitive when building the BVH. It can be proved that the combination of all the AABBs bounds the fiber geometry.

During ray tracing, for each cone, the BVH nodes are first checked against the ray cone using a fast separating axis theorem test, which conservatively excludes many non-intersected nodes. For each AABB passing the test, we construct a ribbon for its corresponding line segment to approximate the projection of the fiber geometry bounded by the AABB to the image plane.

We first project the two vertices of the line segment to the reference plane Π of the cone. The projected fiber width at each vertex is determined by dividing the vertex's fiber width by the vertex's depth value z with respect to the cone apex $\boldsymbol{o}$. The cross-section formed by the cone and the reference plane is a sheared ellipse determined by $\boldsymbol{R}_x$ and $\boldsymbol{R}_y$ (Fig. 3(a)).

On the reference plane, we approximate the projection of the fiber geometry bounded by the AABB as a quadrilateral. At each vertex of the projected line segment, the bisector of the angle formed by the two connected line segments sharing the vertex is intersected with a circle whose radius equals the projected fiber width at the vertex, yielding a pair of points. If the vertex is the end of a fiber and there is only one segment sharing the vertex, we use the line perpendicular to the projected line segment to intersect the circle. These points are then connected in turn to form a quadrilateral (see the quadrilateral $ABDC$ in Fig. 3(a) for example).

Note that the transformation from the image plane to the cone's reference plane is described by $T = [\boldsymbol{R}_x, \boldsymbol{R}_y]$. Therefore, the quadrilateral can be transformed back to the image plane by T^{-1} to obtain a ribbon, which is also a quadrilateral (Fig. 3(b)). The intersection area of the ribbon with the cone (i.e., a circular disk) can be efficiently computed on the GPU (see details in Section 3).

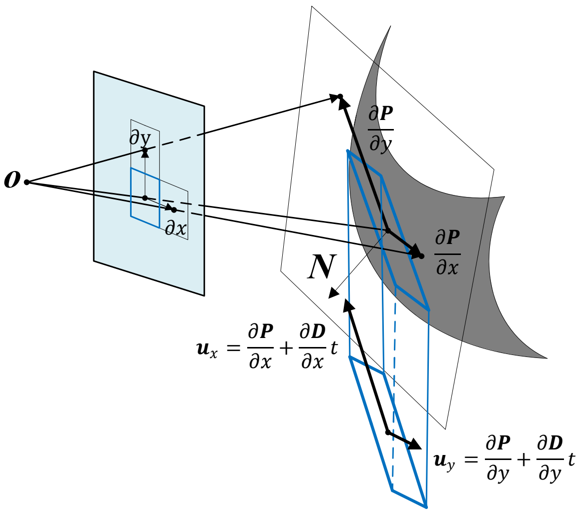
2.3 Cone Formation for DOF, Reflection and Refraction

Here we discuss how cones are formed to represent the ray bundles for tracing more complicated effects, including DOF, reflection and refraction. Our method is similar to that used by [22]. For DOF, we use the envelope of two cones, one for the ray samples over the aperture and the other for those over the pixel. Cones for the reflection and refraction ray samples are formed according to ray differentials which describe the evolution of the ray footprint along the rays.

Depth of Field To correctly model the DOF effect, a ray tracer needs to sample the rays connecting a point on the aperture and a point on the projection of a pixel on the focal plane [25]. The envelope of these ray samples can be approximated by two cones, as illustrated in Fig. 4, where one is formed by connecting the aperture center and the circumcircle of the focal plane projection of the pixel (marked in red), and the other one is formed by connecting the projection of the projected pixel center on the focal plane and the aperture circle (marked in blue).

When a fiber segment is projected onto the reference plane of the DOF cones, the average depth value of the segment is used to compute the sizes of the cross-sections of the DOF cones. And the segment is projected to the reference plane of the cone with larger cross-section size for intersection computation. The depth values are computed with respect to the view point and the samples are stored in a single buffer and composited to yield the shading of the pixel, as described in detail in Section 2.1.

Reflection/Refraction For reflection and refraction rays, cones are formed according to ray differen-



O
∂y
∂x
$\frac{\partial P}{\partial y}$
$\frac{\partial P}{\partial x}$
N
$u_x = \frac{\partial P}{\partial x} + \frac{\partial D}{\partial x} t$
$u_y = \frac{\partial P}{\partial y} + \frac{\partial D}{\partial y} t$

Fig. 5. Cone formation for reflected and refracted rays based on ray differentials.

tials [21]. Note that scene geometries other than fur are not necessarily traced by cones and we only assume that the ray tracer used for them can provide a set of reflected/refracted rays, along with the ray differentials.

More specifically, for a ray $\boldsymbol{R} = \langle \boldsymbol{P}, \boldsymbol{D} \rangle$, the ray tracer used for scene geometry provides four partial derivative vectors of the ray: $\frac{\partial \boldsymbol{P}}{\partial x}, \frac{\partial \boldsymbol{P}}{\partial y}, \frac{\partial \boldsymbol{D}}{\partial x}$ and $\frac{\partial \boldsymbol{D}}{\partial y}$. Here $\boldsymbol{P}$ is a position on the ray and $\boldsymbol{D}$ is the ray direction. The derivatives for $\boldsymbol{P}$ and $\boldsymbol{D}$ describe the differential offsets of the position and direction with respect to the image space coordinates [21].

At the starting point of the ray where reflection or refraction takes place, the ray footprint is a deformed pixel determined by the two derivatives of $\boldsymbol{P}$ (see Fig. 5). As the ray proceeds in its direction, the two spanning vectors of the ray footprint are given by:

$$\boldsymbol{u}_x = \frac{\partial \boldsymbol{P}}{\partial x} + \frac{\partial \boldsymbol{D}}{\partial x} t, \quad \boldsymbol{u}_y = \frac{\partial \boldsymbol{P}}{\partial y} + \frac{\partial \boldsymbol{D}}{\partial y} t,$$

where t is the distance to where the ray starts. The envelope of this ray footprint is again a complex shape which cannot be easily represented by a cone. But we can study the area of the footprint to get an idea of how this envelope converges.

The square of the ray footprint area is proportional to $\sigma(t) = \|\boldsymbol{u}_x \times \boldsymbol{u}_y\|^2$, which is a quadratic function of t. Solving $\sigma'(t) = 0$ gives us up to three extrema. If only one extremum $t = t_0$ exists, we take the point corresponding to t_0 as the apex of the cone. Otherwise there are three extrema, and we take the point corresponding to the average of the smallest and the largest t as the apex.

We then project the derivatives of $\boldsymbol{D}$ to the reference plane Π and multiply them by a diagonal factor of $\frac{\sqrt{2}}{2}$ to yield the tangential vectors $\boldsymbol{R}_x$ and $\boldsymbol{R}_y$ of the cone.

3 Algorithm Implementation

In this section we discuss several non-trivial implementation details of our algorithm.

Fur Shading We support any curve representation of fur fibers. During rendering, each fiber is first view-dependently diced into a series of connected line segments each of which is no longer than three pixels, and shading is computed at the vertices of line segments. We use the shading model proposed by Marschner et al. [6] to compute the reflectance.

We use the same cone tracing algorithm to handle shadow rays. Cones are formed from the shading point and the lighting information - point lights are modeled as spheres and directional lights as disks spreading a small solid angle, and they are connected with the shading point to form the cones. More complicated lighting/shadow conditions like area light sources can also be approximated with cones. For example, Fig. 1 is rendered with SRBF-approximated environment lighting [12] with cone-traced shadows.

Composition Optimization Our ray tracer is implemented on the GPU using CUDA [26]. To bound the GPU memory consumption, we use the adaptive transparency method proposed by Salvi et al. [8]. Specifically, a sample buffer of fixed size is maintained for each ray cone. New samples are inserted into the buffer according to their depth values. In the case that the buffer is full and a new sample needs to be inserted, current samples in the buffer would be looped over and an optimal candidate is selected for replacement so as to minimize the error to the visibility function integration. We refer the readers to [8] for more algorithmic details. Note that unlike the original implementation of [8], we can completely avoid data races and a fixed memory bound can be assured, as our samples are generated by ray tracing instead of rasterization.

Reflection/Refraction Cones Reflection and refraction cannot be directly applied to cones as rays within each individual cone may hit different reflective/refractive objects and diverge. Therefore, we use the shading reuse metric described in [4] to cluster supersampled reflection/refraction rays that happen to be coherent into groups and generate an aggregated cone for each group. Specifically, we create one group for all reflection/refraction rays from the same pixel with hit points sharing the same shading value. The origin and direction of the cone is determined by averaging the origins and directions of the rays in the group. The ray differentials [21] required in forming the cone are also determined by averaging the ray differentials in the group, scaled by $\sqrt{n_G}$ where n_G is the number of rays in the group. This is to ensure that the aggregated cone roughly covers the same area on the reference plane as the sum of all cones in the corresponding group. Note that by taking the intersection set of pixels and shading reuse clusters,

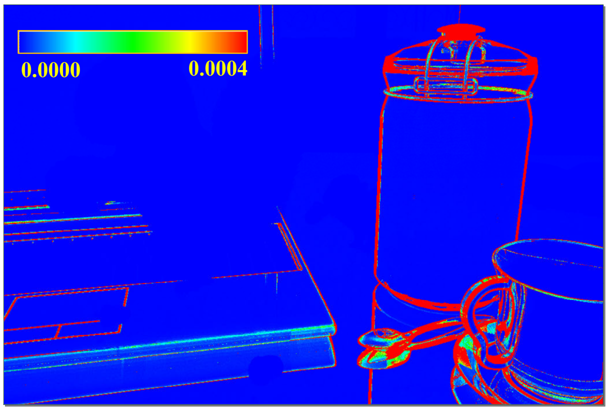


Fig. 6. Visualization of the cone sizes of reflection/refraction rays for the scene shown in Fig. 1. Cone sizes are measured as solid angles.

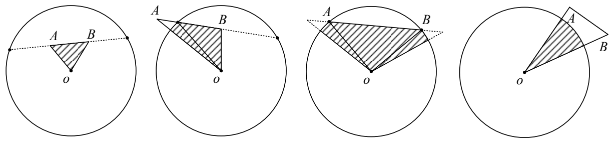


Fig. 7. Computing the signed area of the triangle formed by a line segment and the disk center. The segment can have two outer intersections (a), one inner and one outer intersection (b), two inner intersections (c) with the disk boundary or lie completely outside of the disk (d).

Algorithm 1 Pseudo code of the ray tracing system

```
1:  image = EmptyImage()
2:  rays = GeneratePrimaryRays()
3:  while rays.isNotEmpty() do
4:      hits = TraceAndShadeScene(rays)
5:      cones = ClusterIntoCones(rays)
6:      AT_buffer = ConeTraceFur(cones)
7:      fur_rgba = AT_buffer.QueryRgbaAt(hits.Depths())
8:      final_rgba = AlphaBlend(hits.Colors(), fur_rgba)
9:      image += DownSample(final_rgba * rays.Contribution())
10:     rays = NextBounce(rays)
11: end while
```

we ensure rays within each individual group are reasonably coherent at both image plane intersections (where pixels are defined) and final hit points (where shadings are defined). Therefore, the generated ray cone can be expected to be compact throughout the entire traversal, assuming the ray derivative values are within a reasonable bound.

Reflection/refraction rays generated around object boundaries could have very large derivative values, resulting in very large cones (see Fig. 6 for an example). The cones reflected at the boundary of the iron wires of the bottle are so large that they intersect with most of the fur in the scene, which severely affects the workload balance among GPU threads and is problematic for parallel tracing. In our algorithm, we revert to trace all original supersampling rays within a cone if the solid angle of the cone is greater than a threshold (0.0001 in our implementation). This simple scheme works well for all of our scenes. For example, in Fig. 1, we revert to ray tracing for about one tenth of the cones.

Quadrilateral-Disk Intersection The quadrilateral-disk intersection is executed on the GPU. To maximize performance, we need to store all intermediate results in registers and minimize register usage, which prevents us from storing an explicit representation of the intersection.

We iterate the four edges of the quadrilateral in turn. Each edge forms a triangle with the center of the disk. We then intersect the disk with each of the triangles and calculate a *signed area* for each intersection. All signed areas are then summed, and the absolute value of the total signed area is the intersection area of the quadrilateral and the disk.

To compute the signed area, the edge is extended to a line and intersected with the disk, and the signed area is computed for different cases of intersections (see Fig. 7). If the edge AB has two intersections with the disk and all intersections are outside of the edge (Fig. 7(a)), the signed area is computed as the area of the triangle formed by the edge and the disk center ABO. If one of the intersections of AB and the disk is inside AB and the other is outside (Fig. 7(b)), the signed area is computed by adding a triangle and a sector of the disk. If two intersections are both inside AB (Fig. 7(c)), the signed area is computed by adding a triangle and two sectors of the disk. If the edge is completely outside of the disk (Fig. 7(d)), the signed area is the sector of the disk covered by the triangle. Note that only a fixed amount of temporary storage is required for each edge. This enables us to perform all computations in GPU registers.

Note that in [27] a coverage algorithm was proposed to compute the intersection of hard shadow quads with light source quads by looking-up into a precomputed 4D coverage texture. Our algorithm does not need any precomputed texture but computes the quadrilateral-disk intersection analytically, and is carefully optimized to minimize register usage and make sure that all computations can be performed in GPU registers.

Integration with Scene Geometry Algorithm 1 shows how our cone tracing algorithm is integrated with the ray tracing of scene geometry. Mutual occlusions between fur and scene geometry need to be taken into account. We first trace the scene geometry to obtain a hit sample for each ray (line 4). Then we trace the fur using cones and generate the adaptive transparency (AT) buffer [8] (line 6). For each cone, the maximum depth of the scene geometry samples covered by the cone are obtained to cull the fur during the tracing (line 7). The scene geometry samples are updated using the opacity obtained from the fur AT

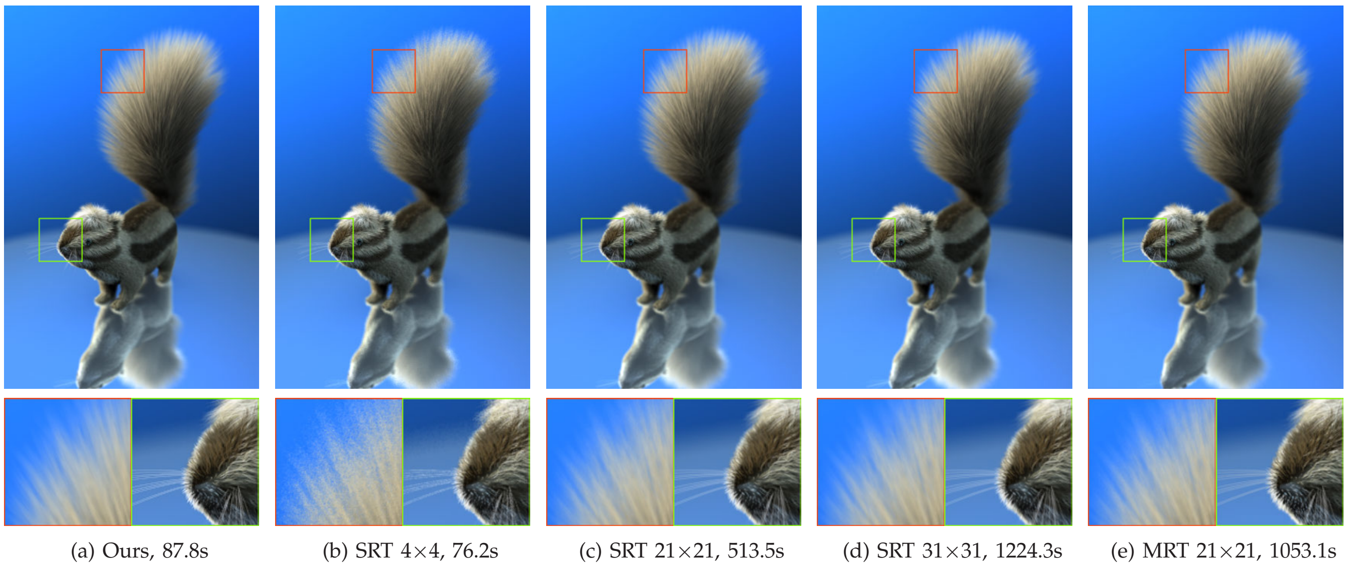

(a) Ours, 87.8s (b) SRT 4×4, 76.2s (c) SRT 21×21, 513.5s (d) SRT 31×31, 1224.3s (e) MRT 21×21, 1053.1s

Fig. 8. A squirrel rendered with DOF and reflection effects. Our algorithm is able to achieve a high-quality result (a) comparable to that of stochastic ray tracing (SRT) at a 31×31 supersampling rate (d) with significantly less rendering time. Stochastic ray tracing with 4×4 supersampling takes similar rendering time but suffers from severe noise artifacts (b), which are visible until the supersampling rate is increased to 21×21 (c). A similar supersampling rate is required for the recent micropolygon ray tracing (MRT) method [14] (e) to achieve a result of comparable quality. This furry object consists of 129K fibers, diced into 931K line segments.

buffer, and the fur color is finally added to the sample (line 8).

4 Experimental Results

Our cone tracing algorithm and other ray tracing algorithms used for comparison in this paper have been implemented and tested on a 2.33GHz dual-core PC with 4GB of memory and an NVIDIA GTX 570 GPU. We use a Lanczos filter [28] of three-pixel diameter for the antialiasing of all results. No supersampling is used for fur rendering (i.e., $m = 1$), unless stated otherwise. The supersampling rate used for tracing scene geometry is always set to 9×9 to eliminate the noisy artifacts of ray traced DOF, reflection and refraction effects for scene geometry, unless otherwise stated.

Comparisons We compare our algorithm with alternative techniques including stochastic ray tracing (with/without adaptive sampling) and the recent micropolygon ray tracing [14] in both rendering performance and quality. All implementations are based on the GPU.

When implementing stochastic ray tracing, we use the same dicing, shading, BVH construction and BVH traversal loop as in our method, and only replace the ray-box and ray-ribbon intersection routines with the corresponding cone versions. The persistent while-while traversal algorithm [29] is used for efficient work distribution on the GPU. Packet ray tracing is not used because it has been tried in GPU ray tracing [29] and there is no evidence that it brings any benefit in performance. The ray-ribbon intersection is implemented by computing the shortest distance between the ray and the ribbon's line segment, and generating a sample if the distance is less than the width at the segment point having the shortest distance. The width value at an arbitrary point is obtained by interpolating the input width values at the line segment vertices. The tracing performance of primary rays for the scene shown in Fig. 1 is 5.81 Mrays/s. Note that this appears to be much lower than the surface tracing performance reported in literature [29]. We would like to point out that fur tracing is much more computationally expensive than surface tracing. It is thus inappropriate to directly compare fur tracing performance with surface tracing performance. The distribution of fur fibers are considerably different from that of surface triangles, resulting in different traversal/intersection behaviors. For example, for the scene in Fig. 1, the average numbers of traversed BVH nodes and intersection tests for each ray are 266.46 and 44.28 in fur tracing, while those in scene geometry tracing are much less (52.23 and 9.21). Additionally, fur tracing requires generating all hit points and adding them to an AT buffer, while surface tracing typically only needs to store one. This results in a significant higher intersection cost in the fur tracer.

As shown in Fig. 8, our result is comparable to the result generated by stochastic ray tracing at a very high supersampling rate (31×31) but is produced in about 10× less time. Under the condition of comparable rendering performance, the result of stochastic ray tracing at 4×4 supersampling suffers from severe noise (Fig. 8(b)). The micropolygon ray

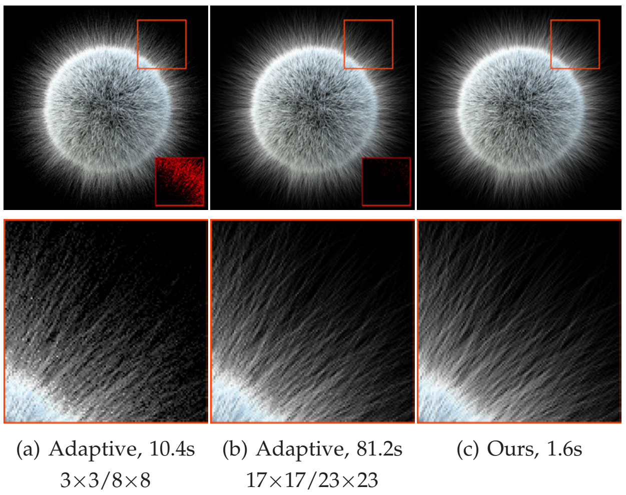


Fig. 9. Comparison between adaptive sampling and our method. Timings given below each column are for tracing view rays (cones). (a) using 3×3 supersampling in the coarse phase and 8×8 in the refining phase produces noisy results. Some pixels, marked in red in the insets in the top row, are missed in the coarse phase and ignored in the refining phase. (b) this problem can only be alleviated with a 17×17 coarse phase supersampling rate and such an adaptive sampling scheme hardly brings any performance gain. (c) our rendering result.

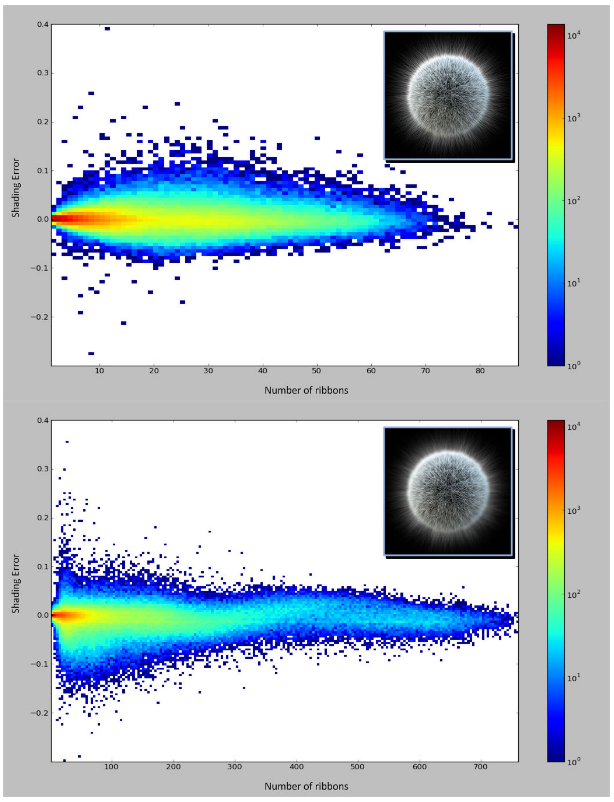


Fig. 10. Shading error distribution. The top row is generated for the fur ball scene in Fig. 9, and the bottom row is generated for the same scene with DOF effects.

tracing algorithm also needs a 21×21 supersampling rate to achieve a satisfactory result. Moreover, it dices fur fibers into micropolygons facing the viewpoint and causes visible artifacts in reflection, e.g., the reflection of the squirrel whiskers is mostly lost in Fig. 8(e). Another comparison between our result and that obtained by stochastic ray tracing with 21 × 21 supersampling is shown in Fig. 11. The scene is taken from an animation with a running furry character.

Adaptive sampling techniques [30], [31], [32] strive to resolve the tension between sampling expense and image fidelity by investing more samples in regions of rapid radiance changes. A coarse phase is often employed to evaluate the local radiance change rates, which are then used to guide the distribution of additional samples in the following refining phases. This strategy, however, can "miss minute isolated features" [31] and does not work well in the case of fur ray tracing. The main reason is that a region where thin fur fibers are missed completely by the coarse phase would be mistakenly interpreted as having smoothly changing radiance, ending up with getting less than ideal samples in the subsequent refining phases. Capturing these high frequency details in the coarse phase, on the other hand, requires very high initial supersampling rates, negating the benefit of the adaptive sampling.

We implemented the adaptive sampling algorithm proposed by Mitchell [31] and compared the results with ours, as shown in Fig. 9. For the pixels marked in red in the inset of Fig. 9(a) (top row), none of the samples in the coarse phase intersects with the fur strands that actually pass through the corresponding pixel. Therefore no additional samples are invested on these pixels, resulting in small dark spots in the final result (bottom row). Increasing the supersampling rate of the coarse phase alleviates the problem (Fig. 9(b)), but cancels out most of the performance gain as well. In contrast, our method is able to obtain smooth results with superior performance.

Error Analysis Our algorithm makes three approximations to achieve efficient high-quality rendering of furry objects. First, we approximate away in-cone variations of shading, opacity and occlusion over each ribbon by assuming that the ribbon is sufficiently thin and the shading and opacity are smooth within the ribbon. Second, we approximate the projection of a fiber on the image plane as a set of connected ribbons (or quadrilaterals). Finally, we assume uncorrelated ribbon coverage, and approximate the occlusion of a ribbon using the fraction of the cone area covered by the ribbon.

In Fig. 10, we visualize the shading error distribution for the fur ball scene, both with and without DOF effects. Error is computed as the luminance difference. Cones are grouped into bins according to the number

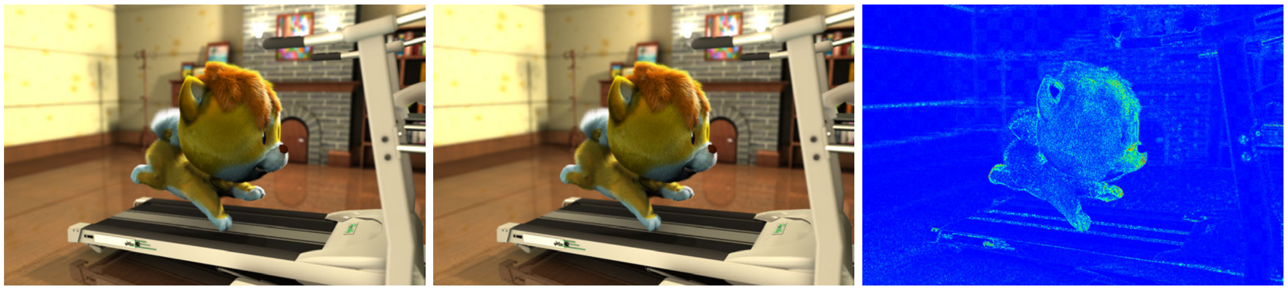


Fig. 11. A dynamic scene with an animated furry character, rendered in 340.6 seconds at the resolution of 1080 × 720 with DOF and reflection effects (a). The SRT algorithm takes 1329.3 seconds to render an image (b) of comparable quality. The difference between the two images is visualized as color temperature (c). The RMS error is 3.38%.

of intersected ribbons and the error value. The log of the number of the cones belonging to each bin is visualized as color temperature. Note that a logarithmic scale is used here to better display cones with large errors. As seen from the plots, most cones are distributed in low-error regions, and statistically cones having more intersected ribbons tend to have smaller errors. This is a reflection of the fact that cones having more intersected ribbons usually have uncorrelated ribbon coverage, resulting in small approximation errors. Similar error distributions can be observed for all scenes we tested (see the supplementary material for more results).

Errors introduced by our approximations can be effectively reduced by decreasing the cone size, i.e., increasing the supersampling rate m. Fig. 12 shows the rendering errors of our algorithm with different supersampling rates, using the stochastic ray tracing result (31×31 supersampling) as the reference. This clearly demonstrates that our result can converge to the reference when increasing the supersampling rate. Note that our result with 7×7 supersampling already has a very low RMS error (1.53%) and is visually indistinguishable from the reference, while our rendering time is still less (2.5×).

In practice, we found that our algorithm can generate visually pleasing results even without supersampling. The reason is that our errors consistently exaggerate depth-based occlusion, which results in a visually pleasant enhancement of the least occluded fur layer. To evaluate the rendering quality of our result, we performed a simple user study involving 35 subjects, including 15 participants from game/animation studios, 10 graduate students majoring in computer graphics and 10 non-graphics students. We showed each subject our result (Fig. 8(a)) and the reference (Fig. 8(d)), and asked the subject to identify the most realistic image. The subjects did not have prior knowledge on how either image was produced, and had unlimited time to finish this task. In the study, 18 participants favored our result, 9 favored the reference, and 8 participants felt the differences to be very

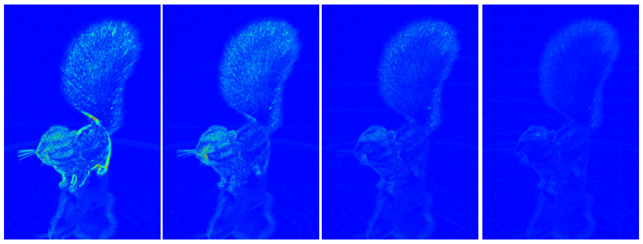


Fig. 12. Supersampling effectively reduces the approximation error. The relative errors are visualized as color temperature images, with red corresponding to a relative error of 10%, and blue 0%. The supersampling rates used for fur/scene geometry are shown under each figure. From left to right, the quantitative RMS errors are 3.38%, 2.68%, 1.53% and 1.12%, respectively. And the rendering time is 87.8s, 124.9s, 207.3s and 425.1s, respectively.

subtle with neither image more realistic than the other. Participants choosing the reference commented that the reference is “less aliased in the tail” or “has alpha falloff along the length of the whiskers”. Participants choosing our result commented that our result has “better definition on edge of core of the tail; looks more solid” or “better (more realistic) handling of whisker edges”. Although this user study is simple, the results reflect the high rendering quality of our algorithm to a certain extent.

Performance A timing breakdown of different stages of our fur rendering algorithm and the stochastic ray tracing method (SRT) is provided in Table 1. In this table, “dice” stands for the time used to dice fur fibers into line segments and “bvh” stands for the BVH construction time. The dicing, BVH construction and shading time are the same for both methods. Note that when computing the shading at the fiber vertices, we use the same shadow algorithm described in Section 3 for both methods. As we focus on fur rendering, the timings for rendering scene geometries are excluded from the numbers in the table, except for

TABLE 1
Timings (in seconds) of our algorithm and SRT (21×21 supersampling) for the test scenes.

scene	#triangles	#fibers	resolution	dice	bvh	shade	view rays		refl./refr. rays		total	
							ours	SRT	ours	SRT	ours	SRT
Fig. 1	819K	368K	1080×720	0.51	9.78	84.3	1.97	56.3	91.8	142.7	1081.4	3722.9
Fig. 8	25.4K	129K	720×1080	0.38	4.27	15.6	8.05	87.6	17.6	38.8	87.2	513.5
Fig. 11	235K	801K	1080×720	0.62	16.1	143.1	5.14	88.7	18.6	22.1	340.6	1329.3
Fig. 14(a)	0	10K	1024×1024	0.04	4.6	24.6	4.7	128.1	-	-	47.5	428.4

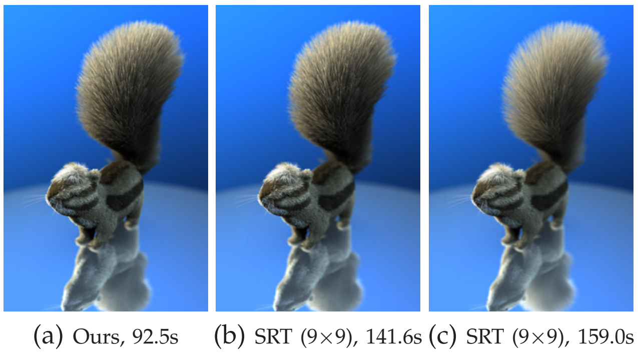

(a) Ours, 92.5s (b) SRT (9×9), 141.6s (c) SRT (9×9), 159.0s

Fig. 13. Rendering the squirrel in Fig. 8 with doubled fiber width. In (c) the fiber opacity is reduced by half.

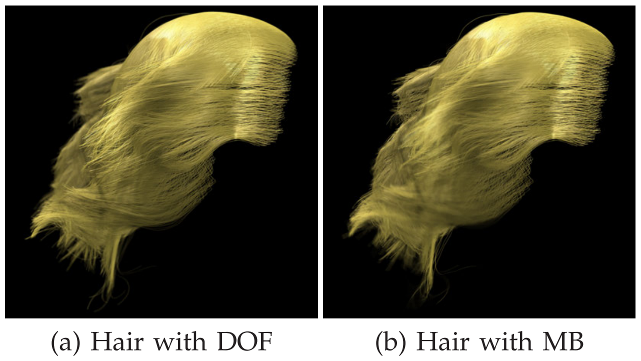

(a) Hair with DOF (b) Hair with MB

Fig. 14. Rendering hair with DOF (a) and motion blur (b) effects. The rendering time of our method is 42.5s and 173.2s respectively. For the motion blur result, we use 13×13 supersampling of the shutter open time. The stochastic ray tracing method takes 223.6s and 368.5s to render results of comparable quality. The hair consists of 10K fibers.

the two right-most columns, where the total rendering time is reported.

As illustrated, our algorithm is very effective in accelerating the tracing of viewing rays, thanks to the reduced supersampling rate. For reflection and refraction rays, the speedups are less significant due to the finer grain cone generation. We use cone tracing to compute shadows of both fur and geometry, which takes a large portion of the shading time and reduces the overall speedups (especially for the scenes shown in Fig. 1 and Fig. 11). It is also possible to use alternative shadowing techniques such as deep shadow maps to reduce this cost.

The memory consumption of our algorithm can be divided into two parts. For the BVH of fiber geometry, we pre-allocate a buffer no larger than a user-specified upper bound (256MB in our implementation) in the GPU memory. When the actual BVH size is larger than the upper bound, we swap in and out chunks of the BVH data and fiber geometry from the memory during the traversal in a way similar to the out-of-core GPU ray tracing algorithm described in [4]. We also maintain a buffer to store the adaptive transparency data, in which 512 bytes (32 samples) are allocated for each cone. We use up all remaining GPU memory, deducting the memory required for shading and scene geometry rendering, for this sample buffer to trace as many cones as possible in parallel. Based on this memory management scheme, our algorithm is scalable to large scenes. In the accompanying video[1], we show a test scene of six squirrels with 2,584K fibers, diced into 5,958K line segments and rendered with DOF effects.

Discussion and Limitations The performance benefit of our algorithm over stochastic ray tracing is more significant for thin fibers than for wide fibers. In Fig. 13, we show the rendering results of the squirrel of Fig. 8 with doubled fiber width. For this scene, our algorithm (a) is only 1.5× faster than stochastic ray tracing (b) for which 9×9 super-sampling is sufficient to produce a satisfactory image. This increased fiber width, of course, gives the furry object a very different look. Reducing the fiber opacity by half would restore the overall occlusion, but still generates an image (c) quite different from the original thin fiber result.

One limitation of our algorithm is that we cannot accelerate motion blur rendering due to the difficulty of formulating the 4D problem as 3D cones. On the other hand, we can still render motion blur effects by directly combining our algorithm with shutter time supersampling. Fig. 14(b) shows the rendering of long hair with motion blur effects. Another problem is that the approximation we used for the projected fiber geometry could generate self-intersecting ribbons if the fiber width is comparable to or larger than the line segment length, making the computation of fiber-ribbon intersection areas incorrect. In our experiments, however, we did not observe any annoying

1. http://gaps-zju.org/publication/2012/fur-divx.avi

artifacts caused by this problem. The cone footprints can also become very large with the propagation of rays in the scene and a large amount of ray fibers may be covered by a single cone. We are interested in developing level-of-detail techniques to achieve further performance acceleration.

Note that we did not choose rasterization for primary ray effects due to the integration difficulty with a ray tracer. Current rasterization based OIT (order-independent transparency) methods require allocating large render buffers. However, on current GPUs such buffers cannot be easily reused for other purposes (like storing scene BVH or rays) and dynamic allocation and destruction of large render buffers can be very costly. Moreover, to get high-quality results, rasterization still needs high supersampling rates and does not reduce the cost of compositing, while our algorithm is able to reduce both the sampling and compositing cost (see the supplementary material for details). Actually we initially tried a rasterization-based solution but we found that we must either face degraded scene rendering performance from memory stress, or endure the render buffer destruction and reallocation cost for every bucket.

5 Conclusion

We have presented an efficient cone tracing algorithm for high-quality rendering of furry objects with reflection, refraction and DOF effects. Compared with alternative ray tracing methods, our algorithm can generate images of comparable quality but is significantly faster. According to a simple user study, our algorithm can generate visually pleasing results without any supersampling. Moreover, errors introduced by the approximations made in our algorithm can be effectively reduced by increasing the cone supersampling rate.

Acknowledgments

The work is partially supported by the NSF of China (No. 61103102, No. 61272305 and No. 61379070).

References

[1] N. Ducheneaut, M.-H. Wen, N. Yee, and G. Wadley, “Body and mind: a study of avatar personalization in three virtual worlds,” in *Proceeding of CHI*, 2009, pp. 1151–1160. [Online]. Available: http://doi.acm.org/10.1145/1518701.1518877

[2] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, “Optix: a general purpose ray tracing engine,” *ACM Trans. Graph.*, vol. 29, pp. 66:1–66:13, July 2010.

[3] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, “Razor: An architecture for dynamic multiresolution ray tracing,” *ACM Trans. Graph.*, vol. 30, pp. 115:1–115:26, October 2011.

[4] Q. Hou and K. Zhou, “A shading reuse method for efficient micropolygon ray tracing,” *ACM Trans. Graph.*, vol. 30, pp. 151:1–151:8, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2070781.2024185

[5] J. T. Kajiya and T. L. Kay, “Rendering fur with three dimensional textures,” in *Proceedings of ACM SIGGRAPH'89*, 1989, pp. 271–280.

[6] S. R. Marschner, H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan, “Light scattering from human hair fibers,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 780–791, 2003.

[7] E. Sintorn and U. Assarsson, “Hair self shadowing and transparency depth ordering using occupancy maps,” in *Proceedings of I3D*. ACM, 2009, pp. 67–74. [Online]. Available: http://doi.acm.org/10.1145/1507149.1507160

[8] M. Salvi, J. Montgomery, and A. E. Lefohn, “Adaptive transparency,” in *Proceedings of HPG*, 2011, pp. 119–126.

[9] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, “Stochastic transparency,” in *Proceedings of I3D*, 2010, pp. 157–164.

[10] J. T. Moon, B. Walter, and S. Marschner, “Efficient multiple scattering in hair using spherical harmonics,” *ACM Trans. Graph.*, vol. 27, no. 3, pp. 31:1–7, 2008.

[11] A. Zinke, C. Yuksel, A. Weber, and J. Keyser, “Dual scattering approximation for fast multiple scattering in hair,” *ACM Trans. Graph.*, vol. 27, no. 3, pp. 32:1–10, 2008.

[12] Z. Ren, K. Zhou, T. Li, W. Hua, and B. Guo, “Interactive hair rendering under environment lighting,” *ACM Trans. Graph.*, vol. 29, no. 4, pp. 55:1–8, 2010, sIGGRAPH 2010.

[13] K. Xu, L.-Q. Ma, B. Ren, R. Wang, and S.-M. Hu, “Interactive hair rendering and appearance editing under environment lighting,” *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 173:1–173:10, 2011.

[14] Q. Hou, H. Qin, W. Li, B. Guo, and K. Zhou, “Micropolygon ray tracing with defocus and motion blur,” *ACM Trans. Graph.*, vol. 29, pp. 64:1–64:10, July 2010. [Online]. Available: http://doi.acm.org/10.1145/1778765.1778801

[15] K. Nakamaru and Y. Ohno, “Ray tracing for curves primitive.” in *WSCG*, 2002, pp. 311–316.

[16] J. T. Moon and S. R. Marschner, “Simulating multiple scattering in hair using a photon mapping approach,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 1067–1074, 2006.

[17] P. S. Heckbert and P. Hanrahan, “Beam tracing polygonal objects,” *SIGGRAPH Comput. Graph.*, vol. 18, pp. 119–127, January 1984.

[18] J. Amanatides, “Ray tracing with cones,” *SIGGRAPH Comput. Graph.*, vol. 18, pp. 129–135, January 1984.

[19] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, “Interactive indirect illumination using voxel-based cone tracing: an insight,” in *ACM SIGGRAPH 2011 Talks*, ser. SIGGRAPH '11. New York, NY, USA: ACM, 2011, pp. 20:1–20:1. [Online]. Available: http://doi.acm.org/10.1145/2037826.2037853

[20] J. Arvo and D. Kirk, “Fast ray tracing by ray classification,” *SIGGRAPH Comput. Graph.*, vol. 21, pp. 55–64, August 1987.

[21] H. Igehy, “Tracing ray differentials,” in *ACM SIGGRAPH*, 1999, pp. 179–186.

[22] M. Wand and W. Straßer, “Multi-resolution point-sample ray-tracing,” in *Graphics Interface*, 2003, pp. 139–148.

[23] D. Lacewell, B. Burley, S. Boulos, and P. Shirley, “Raytracing prefiltered occlusion for aggregate geometry,” in *IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 19–26.

[24] J. Goldsmith and J. Salmon, “Automatic creation of object hierarchies for ray tracing,” *IEEE CG&A*, vol. 7, no. 5, pp. 14–20, 1987.

[25] R. L. Cook, T. Porter, and L. Carpenter, “Distributed ray tracing,” *SIGGRAPH Comput. Graph.*, vol. 18, pp. 137–145, January 1984. [Online]. Available: http://doi.acm.org/10.1145/964965.808590

[26] NVIDIA, “CUDA downloads page,” http://developer.nvidia.com/cuda/cuda-downloads.

[27] U. Assarsson and T. Akenine-Möller, “A geometry-based soft shadow volume algorithm using graphics hardware,” *ACM Trans. Graph.*, vol. 22, no. 3, pp. 511–520, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882300

[28] C. E. Duchon, “Lanczos filtering in one and two dimensions,” *Journal of Applied Meteorology*, vol. 18, pp. 1016–1022, 1979.

[29] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on gpus,” in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 145–149. [Online]. Available: http://doi.acm.org/10.1145/1572769.1572792

[30] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980. [Online]. Available: http://doi.acm.org/10.1145/358876.358882
[31] D. P. Mitchell, "Generating antialiased images at low sampling densities," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 65–72, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37410
[32] T. Hachisuka, W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. W. Jensen, "Multidimensional adaptive sampling and reconstruction for ray tracing," *ACM Trans. Graph.*, vol. 27, pp. 33:1–33:10, August 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360632