High-Quality Curve Rendering using Line Sampled Visibility

Rasmus Barringer
Lund University

Carl Johan Gribel
Lund University

Tomas Akenine-Möller
Lund University and Intel Corporation

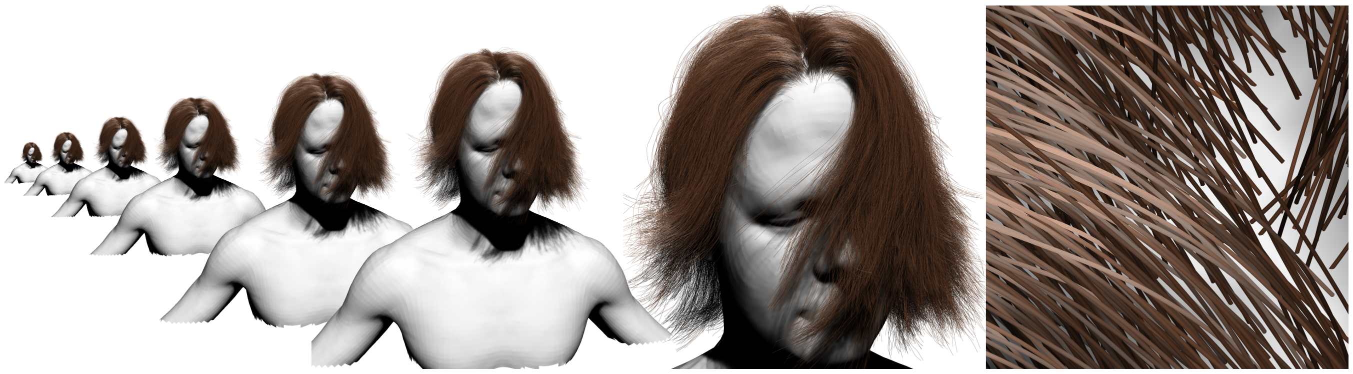


Figure 1: *Our novel thin curve rendering algorithm used on a test production model to compute accurate visibility. The model has 32,000 unique hair strands, which consists of over one million Bézier curves with varying thickness. As can be seen, our algorithm works at all different scales, from cases where there are hundreds of hair strands per pixel to zooming in on the hair strands. All images were rendered at* 1024×1024 *pixels with our GPU implementation. The leftmost image took 109 ms to render, while the close-up on the face took 468 ms. The rightmost image showcases our ability to handle thick curves. Hair model courtesy of Weta Digital.*

Abstract

Computing accurate visibility for thin primitives, such as hair strands, fur, grass, at all scales remains difficult or expensive. To that end, we present an efficient visibility algorithm based on spatial line sampling, and a novel intersection algorithm between line sample planes and Bézier splines with varying thickness. Our algorithm produces accurate visibility both when the projected width of the curve is a tiny fraction of a pixel, and when the projected width is tens of pixels. In addition, we present a rapid resolve procedure that computes final visibility. Using an optimized implementation running on graphics processors, we can render tens of thousands long hair strands with noise-free visibility at near-interactive rates.

CR Categories: I.3.3 [Picture/Image Generation]: Antialiasing; I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture;

Keywords: analytical visibility, anti-aliasing, curve rendering

Links: DL PDF

1 Introduction

High quality rendering of thin, curved primitives, e.g., hair, fibers, fur, and grass, is an important ingredient in today's computer generated imagery. This is particularly true for offline rendering for feature films, but also increasingly so for real-time rendering in games. One approach is to model such thin curves as ribbons with varying width, e.g., using RenderMan's `riCurves` primitive, and then sample visibility using point sampling. A similar modeling and rendering technique was used by Marschner et al. [2003] when developing an accurate appearance model for hair. Another common approach is to rasterize lines with alpha blending to simulate line widths smaller than one pixel [Leblanc et al. 1991; Sintorn and Assarsson 2008]. A third approach is to model and render hair with volumetric textures using ray marching [Kajiya and Kay 1989].

While shading for some types of thin primitives, in particular hair [Marschner et al. 2003; Moon et al. 2008; Zinke et al. 2008; Zinke 2008; Hery and Ramamoorthi 2012], is well understood, computing accurate visibility rapidly for a large number of curves remains a challenge. A major problem is that when point sampling is used, noise is inevitable unless a very large number of samples per pixel is used. This is especially true at a macro-scale, when the viewer is relatively far away from the curves, and the projected width of the curve is only, say, 10% or less, of the pixel width. In such cases, hundreds of samples per pixel may be needed for accurate visibility. For comparison, the diameter of a hair strand is about 0.1 mm [Hadap et al. 2007]. Another problem is that the ribbon model breaks down at the microscale, i.e., when a curve's width project to relatively large number of pixels. In those cases, a hair strand, for example, does not appear as a cylinder as expected.

As a solution to this challenge, we present a visibility engine based on line sampling [Jones and Perry 2000] in the spatial domain. Our curves are modeled as Bézier splines with varying thickness. We develop a novel intersection algorithm between such curves and line samples and present a new interval resolve procedure. As can be seen in Figure 1, our approach renders practically noise-free images at large spectrum of scales. For rapid rendering, we have also implemented our visibility engine on a graphics processor running in parallel.

2 Previous Work

There is a wealth of literature on the topic of simulation, animation, and rendering of thin curves, e.g., hair strands, fur, and grass. Here, we will review the most important references to our work, and refer to the SIGGRAPH course [Hadap et al. 2007] on hair and strands if more information is needed.

Appearance Models Kajiya and Kay [1989] presented an illumination model for single-scattering in fur and hair. The fine geometric detail of fur and hair is represented by volumetric three-dimensional textures. Illumination is calculated by integrating diffuse and specular contributions, using a modified Phong model, by ray marching in the texture. Marschner et al. [2003] present a rich shader model for human hair that captures more distinguished features such as inner reflection, eccentricity and surface scales. Fibers are represented geometrically as procedurally generated flat ribbons and as transparent elliptic cylinders within the shader model. This work was extended to include multiple scattering of light using photon mapping [Moon and Marschner 2006]. Rendering was made using two passes: first particles were traced through the hair to create a photon map, and then the hair was ray-traced to compute direct illumination and indirect radiance gathered from the photon map. Moon et al. [2008] voxelize the individiual fibers into a rectilinear grid, and populate it with aggregate data of nearby fibers using sampling of density, median direction, and variance. By taking advantage of the smoothness of the distributed scattering function, this approach proved faster and less memory-consuming than approaches based on photon mapping. Zinke and Weber [2007] present a shading framework for fibers that have both near and far field solutions. Recently, Hery and Ramamoorthi have presented an importance sampling algorithm for hair fibers [2012]. Fast, but accurate models for shading hair using a dual scattering approximation, targeting real-time rendering, have also been presented [Zinke et al. 2008; Zinke 2008].

Visibility and Shadows The deep shadow mapping algorithm [Lokovic and Veach 2000] generates a monotonic function per shadow map texel, which represents the fractional visibility for each depth from the light source. This works particularly well for hair, fur, and smoke. Jones and Perry [2000] presented line sampling in the spatial domain for efficient anti-aliasing. Gribel et al. [2010] used a similar approach applied to analytical motion blur with spatial point samples, instead of using it for spatial line sampling with no motion blur. Recently, that method has been combined with spatial line samples, for high-quality spatio-temporal anti-aliasing [Gribel et al. 2011]. Tzeng et al. [2012] used line samples over the lens domain to create depth-of-field. A common rendering technique using graphics processors, is to render the hair strands as lines with alpha blending [Leblanc et al. 1991], and composite the fragments in back-to-front order [Sintorn and Assarsson 2008]. An alternative is to use stochastic transparency, where multi-sampled anti-aliasing is used to represent transparency in a pixel [Enderton et al. 2011].

Offsets and Subdivision Our intersection algorithm is based on offsets for Bézier curves and subdivision. The offset to a polynomial curve is *not* polynomial in general [Elber et al. 1997]. A standard approach is subdivide the curve until the offset of each subsegment can be represented by a simpler primitive such as a line or curve. This can be done recursively [Catmull 1974] (recursive subdivision) or more sophisticated rules can be used in order to reduce the number of subdivisions [Hain et al. 2005]. A comparison of different curve offset methods is presented by Elber et al. [1997]. Tiller and Hansen use quadratic Bézier curves to construct the offset curve, where the edges of the control polygon are offset [1984]. A method suited for non-constant curve radius, is to offset each control point in their respective normal direction [Cobb 1984]. Recently, Ruf [2011] presented a method for fast creation of quadratic bounding-Bézier offsets of quadratic Bézier curves.

3 Algorithm Overview

In this section, we present a high-level overview and motivation of some design choices of our visibility algorithm for high-quality rendering of thin primitives, and we also introduce key terminology used throughout this paper.

To sample visibility, we rely on *line sampling* [Jones and Perry 2000] rather than the commonly used point sampling. There are several appealing aspects of the utilization of line samples when rendering a scene. As noted by Gribel et al. [2011], compute power rather than memory bandwidth is used to a greater extent, which meshes well with recent trends in hardware development. Furthermore, sampling using lines effectively means that scene geometry is considered through an entire dimension instead of at discrete points, which means that there is a smaller risk of missing thin primitives with line samples. As illustrated to the left in Figure 2, a line sample can be thought of as the triangle defined by the location of the camera and by a line in screen space, where the triangle extends towards infinity beyond the image plane. We will refer to these as *line sample planes* or simply as *line samples*, where each line sample is parametrized by $l \in [0, 1]$. Usually, a small number of line samples per pixel is required to faithfully sample visibility.

In general, a thin primitive is defined by a three-dimensional Bézier spline, which represent the "core" of, say, a hair strand. In addition, a two-dimensional profile is swept along the curve, and this traces out the geometry that we need to compute visibility for. We call the generated geometry a *thin curve*, for the lack of a better term. A circular profile generates a shape similar to sweeping a sphere along a curve [van Wijk 1985], and it is also related to computing offset curves. See, for example, the recent work by Ruf [2011]. In all our examples, we will sweep a circular profile whose radius varies along the curve, but this can be extended to other profiles.

In broad terms, our algorithm works as follows. For parallel execution of our visibility algorithm, we employ a *sort-middle* approach [Molnar et al. 1994], where the geometry is binned to rectangular pixel tiles. The geometrical content within each tile is binned once more to line samples in the tile. Next, the actual intersections between the curves and their associated line samples are computed. This is illustrated in the middle in Figure 2. The intersection is approximated by a set of *intervals* defined by a start point and an end point in lz-space, where z is depth. When all intervals for a line sample in a tile have been identified, visibility has to be resolved by finding the nearest (in depth) interval segments along the entire line sample. To that end, we present a simple, yet very efficient resolve procedure, which is substantially faster than the resolve presented by Gribel et al. [2010]. An example is shown to the right in Figure 2. We store intervals in a binary heap ordered according to starting points, and resolve visibility in one pass without extra sorting and no extra storage. Shading is computed by taking point samples over the intervals. In the end, the colors from all line samples of a pixel are weighted to get the final color of the pixel.

Next, we present our visibility engine, which includes a description of our geometry representation, intersection computations, and our new resolve procedure.

4 Visibility Engine

In this section, we describe all the involved components and algorithms in our visibility engine for high-quality thin curve rendering.

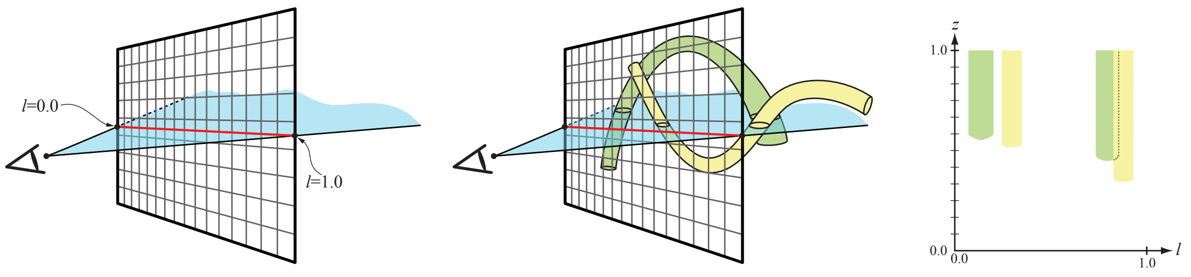


Figure 2: *Left: a line sample is defined by the camera location and a line (red) in screen space, and parameterized by l. Middle: two curves with thickness intersect the line sample. Right: visibility is resolved in the lz-plane by finding the closest surfaces in depth, z, along l. As can be seen, the yellow thin curve occludes part of the green thin curve over this line sample to the right, but not to the left.*

First, our representation of thin curves is presented, and then follows our intersection algorithm between a line sample and a thin curve in Section 4.2, which is extended in Section 4.3 to handle situations where the projected width of a curve covers many pixels. Finally, our novel resolve procedure is presented in Section 4.4.

4.1 Thin Curve Representation

We represent a thin curve by a quadratic Bézier spline, which is a series of connected quadratic Bézier curves. In addition, the "thickness", or radius, varies along our thin curves. A quadratic Bézier curve segment is defined using three control points [Farin 2002]. We use both spatial control points, $\mathbf{p}_i$, as well as radii, r_i, $i \in \{0,1,2\}$, to define the curve's position, $\mathbf{p}(t)$, $t \in [0,1]$, and interpolated radius, $r(t)$:

$$\mathbf{p}(t) = \sum_{i=0}^{2} B_{i,2}(t)\mathbf{p}_i, \quad r(t) = \sum_{i=0}^{2} B_{i,2}(t) r_i, \tag{1}$$

where $B_{i,2}(t)$ are Bernstein basis polynomials. The varying radius makes it possible for, e.g., a piece of fur to be thicker at the animal body and thinner towards the other end. Note that the degree two of the Bézier curves was chosen to make the intersection algorithm simpler, but in principle, a higher degree can be used at the cost of a more involved intersection algorithm.

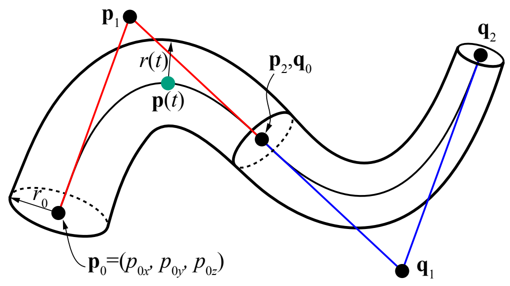
In general, the radius controls the size of a two-dimensional profile that is used to define the character of the thin curve. In all our examples, we use circular profiles. The geometry of the thin curve is defined by sweeping the two-dimensional profile along the Bézier curve while varying the size of the profile according to the interpolated radius, $r(t)$. This is illustrated in Figure 3. Instead of a circular profile, one can use two connected straight lines, e.g., a V, which could be used to model grass, for example.[1] Exploring different profiles and developing their corresponding intersection tests is left for future work. It is important to ensure (at least) G^1 continuity when connecting neighboring Bézier curves. Any global optimization methods [Farin 2002] can be used for this, or simpler heuristics as desired by the user.

In Appendix A, a memory-efficient representation is presented. Next, we present our intersection method between a line sample and a thin curve.

4.2 Thin Curve/Line Sample Intersection

Our visibility engine requires us to compute the intersection between thin curves and line sample planes. Our approach to solving

[1]This would require extending each control point with a binormal as well in order to establish a coordinate system for each t. This is not needed for circular profiles since they are fully symmetric.



$\mathbf{p}_1$
$r(t)$
$\mathbf{p}(t)$
$\mathbf{p}_2$,$\mathbf{q}_0$
$\mathbf{q}_2$
r_0
$\mathbf{p}_0$=(p_{0x}, p_{0y}, p_{0z})
$\mathbf{q}_1$

Figure 3: *A thin curve composed of two quadratic Bézier curves. A circular profile is used in this example, and each control point consists of a three-dimensional position, and a radius of the circular profile. The spatial Bézier curve to the left is defined by $\{\mathbf{p}_i\}$, while the one to the right is defined by $\{\mathbf{q}_i\}$, $i \in \{0,1,2\}$. Note that to ensure G^1 continuity, we set $\mathbf{p}_2 = \mathbf{q}_0$, and also make sure that the tangent directions at that shared position is the same on both sides. C^1-continuity is obtained if the tangents also have the same length.*

this efficiently is based on using *offsets* [Elber et al. 1997; Tiller and Hanson 1984; Cobb 1984; Ruf 2011]. We will use *approximate* offsets, and intersect these with the line sample planes using a subdivision technique to reach a certain error tolerance. The offset intersections are then projected to screen space (along the sample line), connected into intervals, and fed to the visibility engine.

4.2.1 Offsets

The offset, $\mathbf{o}(t)$, of a curve can be said to represent all points being located at a distance, $r(t)$, in the normal direction of $\mathbf{p}(t)$. The normal is defined as $\mathbf{n}(t) = \frac{\mathbf{p}'(t)\times\mathbf{v}(t)}{|\mathbf{p}'(t)\times\mathbf{v}(t)|}$, where $\mathbf{p}'(t)$ and $\mathbf{v}(t)$ are the curve tangent and view-vector, respectively. In such a setting, the two offset curves of $\mathbf{p}(t)$ can be described as:

$$\mathbf{o}^{\pm}(t) = \mathbf{p}(t) \pm r(t)\mathbf{n}(t). \tag{2}$$

A consequence of the presence of the square root in the expression for $\mathbf{n}(t)$ is that, even though $\mathbf{p}(t)$ is in polynomial form, its offsets $\mathbf{o}^{\pm}(t)$ are, in general, not [Elber et al. 1997]. Hence, to find the intersections efficiently, we use approximations, $\mathbf{o}_a^{\pm}(t)$, of the real offsets $\mathbf{o}^{\pm}(t)$. The offset approximation by Cobb [1984] is defined by the control points of the original curve, offset by a constant amount, r, in their respective normal direction, $\mathbf{n}_i$. Since we have varying radius along the curve, we use the following, slightly modified version of that offset approximation:

$$\mathbf{o}_a^{\pm}(t) = \sum_{i=0}^{2} B_{i,2}(t)(\mathbf{p}_i \pm r_i\mathbf{n}_i), \tag{3}$$

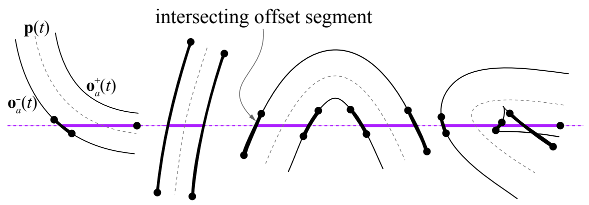


Figure 4: *A horizontal line sample (purple) is shown together with a number of quadratic Bézier curves, $\mathbf{p}(t)$, with thickness. The two offset curves, $\mathbf{o}_a^+(t)$ and $\mathbf{o}_a^-(t)$, of $\mathbf{p}(t)$ are also shown. Since the offset curves are approximate, we use a subdivision approach to refine the offsets until the error is sufficiently small. First, an offset is created from $\mathbf{p}(t)$, and then, depending on curvature, it is subdivided in order to reach the desired error tolerance. The second curve from the left has modest curvature and requires no subdivision at all, while the rightmost, which is strongly curved and even contains a self-intersecting loop, requires several subdivisions. Such cases are very rare in practice. Once one or multiple satisfactory offset segments (fat black curves) are found, they are intersected with the line sample to produce roots outlining the silhouette of the intersection (solid horizontal lines).*

where $\mathbf{n}_0 = \mathbf{n}(0)$, $\mathbf{n}_1 = \mathbf{n}(0.5)$, $\mathbf{n}_2 = \mathbf{n}(1)$, $r_0 = r(0)$, $r_1 = r(0.5)$, and $r_2 = r(1)$. The two offset curves, $\mathbf{o}_a^+(t)$ and $\mathbf{o}_a^-(t)$, are illustrated in Figure 4 for several examples. Intersecting Equation 3 with a plane yields a second degree polynomial, and it is therefore fast and simple to compute these intersections. However, similar to Cobb's method, it will produce approximate offsets that, in general, underestimates the real offset. Given a relative error tolerance, ϵ, the offset approximation, $\mathbf{o}_a(t)$, will meet this tolerance at a certain t, if the following expression holds:

$$r(t)^2(1-\epsilon)^2 \le (\mathbf{o}_a(t) - \mathbf{p}(t))^2 \le r(t)^2(1+\epsilon)^2, \quad (4)$$

which simply is a more efficient calculation of the relative error test: $\|1 - \|\mathbf{o}_a(t) - \mathbf{p}(t)\|/r(t)\| \le \epsilon$. To obtain arbitrary precision, we use a subdivision process, which is described next.

4.2.2 Offset Creation by Subdivision

The approximate offsets, $\mathbf{o}_a^\pm(t)$, are created by splitting $\mathbf{p}(t)$ through subdivision into shorter segments. As the segments get shorter, the offset approximations become more accurate. Hence, accuracy is traded for increased cost of subdivision and offset generation. The nature of the approximation in Equation 3 implies that the end-points of the offset approximation will correctly match the real offset, and subdivision will therefore ensure convergence towards the real offset due to the end-point interpolation property of Bézier curves [Farin 2002].

To avoid unnecessary work, we cull away and abort subdivision for offset segments that will not end up intersecting the line sample. Since the control point triangles of $\mathbf{o}_a^\pm(t)$ do not necessarily bound the real offset, these control point triangles cannot be used for this purpose. Instead, we cull against the *offset bound* of $\mathbf{p}(t)$, which we define as the convex hull of circles of radius, r_i, located at the control points, $\mathbf{p}_i$. Once an offset bound is above or below a line sample, the corresponding offset curve can be culled safely.

To be as memory efficient as possible, the algorithm subdivides successively by maintaining a stack of t-intervals that are eligible for intersection. The starting element of the t-stack is set to $\Delta t = [0,1]$, meaning that the entire offset is eligible initially. We will use subdivision [Farin 2002] of the Bézier curve in our algorithm, and will use the notation $\mathbf{p}_i^{\Delta t}$ for the control points of a subdivided Bézier curve, derived from $\mathbf{p}_i$, which is valid over Δt. Given a curve $\mathbf{p}(t)$, defined by $\mathbf{p}_i$, and a line sample, L, the algorithm has the following steps, applied to one side (either $\mathbf{o}_a^+(t)$ or $\mathbf{o}_a^-(t)$) of the offset at a time:

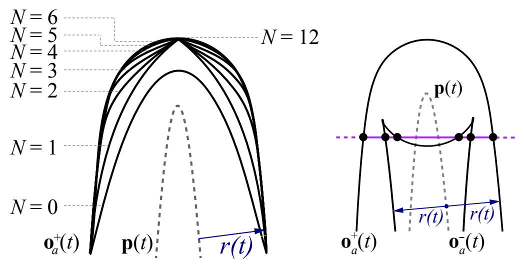


Figure 5: *Left: evolution of curve offset through subdivision. An offset curve, $\mathbf{o}_a^+(t)$ is created by displacing the control points of the original curve, $\mathbf{p}(t)$, in the normal direction. With no subdivision, $N = 0$, the offset always underestimates the true offset. As N increases, the core curve is split into smaller and smaller segments, and the accuracy of the offset increases. Right: Up to six intersections may arise along a line sample in cases with self-intersection. Here, only four are kept: two from the outer offset, and the outermost two from the inner offset.*

1. Pop the top element, Δt, from the t-stack, and subdivide the curve using De Casteljau's algorithm to obtain $\mathbf{p}_i^{\Delta t}$. If the t-stack is empty, then exit algorithm.
2. If L does not intersect the offset bound of $\mathbf{p}_i^{\Delta t}$ then goto 1.
3. Create approximate offset to $\mathbf{p}_i^{\Delta t}$ according to Equation 3.
4. If the error test, according to Equation 4, of the approximation is fulfilled: compute intersections between the approximate offset and L and project them to screen space. Else, split Δt into two halves and push to the t-stack. Goto step 1.

In a final step, intersections between the end-cap lines of the thin curve and L are computed. The output from the intersection algorithm for a complete curve, including both offset curves and the end-caps, is up to four intersection points. See Figure 4.

The termination criteria in step 4 with the user-defined error tolerance, ϵ, is preferably complemented with a maximum subdivision depth, N_{max}. The t-stack will then contain at most $2N_{max}$ elements, and each element Δt can be stored using $2 \cdot 16$ bits of memory. In addition to the storage for the stack, storage is needed for the current subdivided control points, $\mathbf{p}_i^{\Delta t}$, its offset approximation, and the roots in t, generated from solving the second degree polynomial. The memory requirements are thus bounded and relatively small. For practical reasons, we evaluate the error function, $e(t)$, in Equation 4 at two discrete positions spread uniformly over the interval Δt. The end points of Δt are not included since the error is zero there. As a heuristic, we require that the error tolerance is to be met at these two locations before terminating the subdivision. For thin curves, we have observed that tolerances of $\epsilon = 0.5\% - 1\%$ are typically sufficient. The left part of Figure 5 illustrates convergence of the offset by subdivision for a heavily bent curve.

Next, the intersection points are connected into intervals using a few simple heuristics. If only two intersections are present, they are obviously connected and we are done. For four intersections, there will be two intervals that are either disjoint or overlapping, depending on the curve parameter, $t \in [0,1]$. See Figure 4. The intervals are formed in a disjoint manner if the first two and last two intersections along L are disjoint with respect to t (third example in Figure 4). However, if the t-spans of the first pair and last pair overlap, the intervals are consequentially formed to overlap, simply

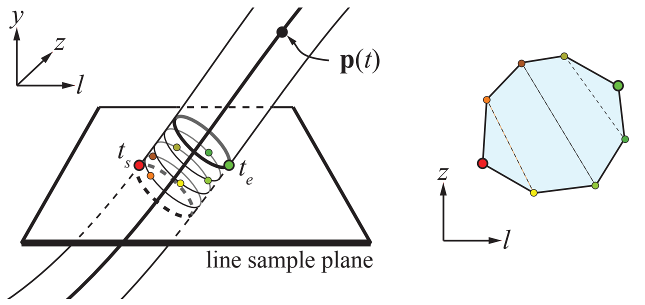


Figure 6: *To compute the intersection between a curve whose width projects to many pixels, and a line sample plane, the time t_s and t_e are first computed. These are the times when the moving circle (as a function of t) first intersects the line sample plane, and when it exits. In this case, we generate three extra circles between t_s and t_e, and all circles are intersected against the line sample plane. Together these form a tessellated approximation (right) to the true intersection.*

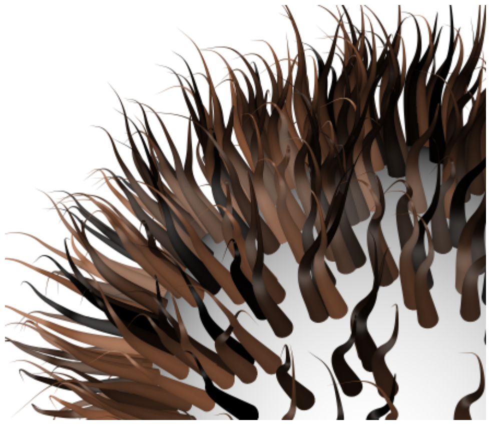


Figure 7: *An extreme example of thin curves whose widths project to a large number of pixels (about 20 at the base). Note that each hair strand's width transitions to subpixel size towards their outer end points.*

by connecting the first intersection point with the third, and the second with the fourth (rightmost example in Figure 4). Overlapping intervals is a result of curves that undergo self-intersection. Self-intersecting curves, such as the right curve in Figure 5, may in fact give rise to as many as six intersections along the sample line. For simplicity, our algorithm keeps only the outermost two, limiting the number of intersections to four for the entire curve.

Though self-intersection may seem somewhat abstract and unintuitive since real-world objects usually do not self-penetrate [Tiller and Hanson 1984], it is nevertheless commonplace in our setting as curves are projected from 3D to 2D. In other words: a curve with no self-intersection in 3D-space may self-intersect when projected to 2D from a certain point of view. It is for this reason necessary for our algorithm to robustly handle all kinds of curve behavior.

4.3 Visibility for Large Projected Curve Widths

When the projected width of a curve covers many pixels, it should become possible to see the geometrical shape of the curve. For example, a hair strand should look like a curved cylinder. The algorithm in Section 4.2 only gives the endpoints of an intersection between a thin curve and a line sample. When the projection is rather large, this is not sufficient. Here, we present a simple extension which solves this case.

Recall that the curve is described by a Bézier curve, $\mathbf{p}(t)$, and a radius, $r(t)$. For circular profiles, this can be interpreted as a circle with varying radius that moves along the curve, $\mathbf{p}(t)$, as a function of t, and we need the intersection between the thin curve and the line sample plane. This intersection consists of one or more closed curves.

To this end, the normal used for thin curves in the previous section is replaced by an expression that generate offset curves above and below the thin curve along the y-axis:

$$\mathbf{n}(t) = \frac{\mathbf{p}'(t) \times \big(\mathbf{p}'(t) \times (0,1,0)\big)}{|\mathbf{p}'(t) \times \big(\mathbf{p}'(t) \times (0,1,0)\big)|}. \qquad (5)$$

Next, we use the method from the previous section to compute the first time, t_s, where the moving circle touches the line sample plane, and the time, t_e, when the moving circle exits the line sample plane. This is illustrated in Figure 6. Given the time interval, we essentially use a tessellation procedure to compute an approximation of the true intersection curve, which, in general, is a complex high-order curve. First, n uniform t-values between t_s and t_e, are computed, i.e., $t_i = (1-\alpha_i)t_s + \alpha_i t_e$, $\alpha_i = i/(n-1)$, where $i \in \{0, 1, \ldots, n-1\}$. In our implementation, we use $n = 32$. However, any number can be used, and it may be beneficial to calculate a suitable number based on the thickness of the curve. Next, n circles are generated centered at $\mathbf{p}(t_i)$, with radius $r(t_i)$, and the normal of the plane equation in which the circle lies is $\mathbf{p}'(t)$, i.e., the tangent of the curve. These circles are intersected with the line sample plane, and the points connected to form a closed tessellated curve. This is shown to the right in Figure 6. The lines of this tessellation are inserted (as usual) as intervals into our visibility engine. An example of possible results using this technique is shown in Figure 7.

As discussed previously for thin curves, a single curve can create multiple disjoint intersections with a line sample. One such example is the third curve in Figure 4. Self-intersecting curves, such as the right curve in Figure 5, exhibits a similar behavior except that the two intervals overlap. For thick hairs, each such interval is tessellated independently of others.

4.4 Interval Resolve Procedure

As described in the previous subsections, the intersection computations between thin curves and a line sample generate a list of intervals. In order to determine the visibility along a line sample, and ultimately determine the color of the pixels overlapping the line sample, we need to find the (clipped) intervals closest in depth to the camera. By replacing t (time) for l (the parameter along the line sample), it would be possible to use the resolve procedure by Gribel et al. [2010]. However, we have devised a novel resolve algorithm, specialized for opaque geometry, which is simpler, uses less memory, and is therefore faster. Our algorithm is described below.

All intervals are stored in a binary heap [Cormen et al. 2009], ordered by their starting interval point. Note that the intervals are not sorted and we simply perform a build heap operation which takes $O(n)$ time. This enables us to do efficient insertions, and removals, while avoiding the dependent memory accesses inherent in any kind of self-balancing search tree. Our algorithm performs a single sweep over the line sample and process two intervals at a time. For each pair, the space that separates them can be resolved immediately and accumulated to the exposed pixels. The sweep continues with the interval closest to the camera. If the occluded interval spans past the point of occlusion, it is reinserted into the heap at a point where it may be visible again. The details are shown in Algorithm 1. An illustration of how our resolve works in a simple situation is shown in Figure 8.

The most expensive parts in the resolve procedure are the removal from and the insertion to the heap. As such, it is very important to realize that these operations can be performed simultaneously. We can read the minimum element from the heap by inspecting the first value in its array. Removal moves the last element in the array to the first position and then restores heap order. Insertion puts the new element last and restores heap order. By delaying the removal of the minimum element until the very end of the loop, we can choose whether to replace the minimum element by the interval that is to be reinserted, or by the last element of the heap. We can therefore get away with a single restore of heap order. This also ensures that instruction divergence is kept to a minimum since we only need a small branch deciding what element to put first in the heap. This property is very important for GPUs.

Note that each interval is stored only once in the heap which means that given n intervals, we need to store only n items. The memory requirements are thus predictable as they do not depend on the geometric relationship between intervals. Given n intervals, the previous resolve [Gribel et al. 2010] needs to store at least $2n$ items, and in the worst case, $\frac{1}{2}(3n + n^2)$ items depending on the number of intersections. Another benefit of our algorithm is that we only intersect intervals against the interval closest to the camera, resulting in far fewer intersections than when finding all intersections between all intervals. The number of intersections are kept to a minimum since our algorithm includes a form of occlusion culling. When an interval is (partially) occluded, it will be discarded from all calculations until a point where it may be visible again (if such a point exists). When the interval is no longer occluded by the previous occluder, it is checked again against the interval closest to the camera. If it is still occluded, it will be discarded again. An interval can thus be efficiently occlusion culled even if it is only occluded collectively by multiple intervals.

The algorithm shown here includes intersections between intervals for completeness only. In our current implementation, all intervals are treated as flat, i.e., after projection, the interval gets the average depth of the two end points. This approximation is acceptable because our intervals are very thin (even for thick curves as they are tessellated into multiple thin intervals). This makes the performance improvement with respect to intersections less important for our purposes. Disregarding intersections, our resolve is still more efficient than the previously described algorithm as they first need to sort all intervals and then maintain an additional sorted list of active intervals during the sweep. In contrast, we only need a single data structure that is manipulated very efficiently.

5 GPU Implementation

Our GPU pipeline resembles a recent software sort-middle pipeline [Laine and Karras 2011]. The major difference is that our pipeline is used for rendering thin curves efficiently using line samples, rather than rendering triangles using point samples. In this section, we describe the different stages, namely, setup, binning, rasterization and sample blend, of the pipeline. The discussion in this section refers to pixels in the context of two line samples per pixel. A straightforward way to increase the number of line samples per pixel is to render the image in higher resolution, and then down sample the resulting image to the desired resolution. Our current implementation uses either two (no down sampling) or four (2× down sampling) axis-aligned line samples per pixel.

Algorithm 1 Resolve

```
insert all intervals into $heap$ ordered by $start$
$ival_a \leftarrow removeMin(heap)$
$l_s \leftarrow start(ival_a)$
**while** $heap \neq \emptyset$ **do**
    $ival_b \leftarrow removeMin(heap)$
    $l_e \leftarrow start(ival_b)$
    accumulate $ival_a \in [l_s, l_e]$ to current pixel
    $l_s \leftarrow l_e$
    **if** $end(ival_a) \leq l_s$ or $ival_a$ occluded by $ival_b$ after $l_s$ **then**
        swap($ival_a, ival_b$)
    **end if**
    $i \leftarrow intersect(ival_a, ival_b)$
    **if** $i \neq nil$ and $i > l_s$ **then**
        clip part of $ival_b$ before $i$
        reinsert $ival_b$ in $heap$
    **else if** $end(ival_b) > end(ival_a)$ **then**
        clip part of $ival_b$ before $end(ival_a)$
        reinsert $ival_b$ in $heap$
    **end if**
**end while**
$l_e \leftarrow end(ival_a)$
accumulate $ival_a \in [l_s, l_e]$ to current pixel
```

5.1 Setup

The input to the pipeline is a set of thin curves, defined by quadratic Bézier curves (BCs) with varying thickness (Equation 1). Our pipeline can either use interpolation of thin curves using the method in Appendix A, or directly render the curves without interpolation. For interpolation, we use thin *control* curves. Each thin control curve consists of a series of *control* Bézier curves (CBCs).

The purpose of this stage is to generate a record for each BC of each thin curve. A BC is identified by a 32-bit *BC identifier*, where the high 24 bits identify the thin curve index, and the low 8 bits identify the BC index. When interpolating a BC from the CBCs, the weights and base CBC indices are fetched using the thin curve index. To get the actual CBC indices, the base CBC indices are offset by the BC index. Each final record stores the projected bounding box as well as BC identifier. In the current implementation, each such record requires 16 bytes, where the last 4 bytes are used as padding to preserve alignment. If we instead opt to recompute the bounding box when needed, we can get away with as little as 4 bytes, which might be preferred in cases where memory is scarce. In addition, each BC is culled against the view frustum. If a BC is culled, it is only flagged as such to maintain a fixed input-to-output mapping. The flagged BCs are removed in the next stage.

In order to handle thick curves, the width of a curve is estimated by calculating the projected radius of each control point. If this radius is larger than a pixel, the curve is flagged as thick. A parallel partition is used to make sure that all thick curves appear before any thin curves. This order is conserved throughout the pipeline.

5.2 Binning

This stage subdivides the screen into tiles of 128 × 128 pixels. The BCs are binned to per-tile lists, called *bin lists*. In order to avoid excessive synchronization, a large thread block is allocated per streaming multiprocessor (SM), so that only a single thread block is active in each SM at any given time. These thread blocks are kept running until all input data have been processed, similar to persistent threads [Aila and Laine 2009]. We use thread blocks with 16 warps, i.e., 512 threads. For each tile, a separate bin list is

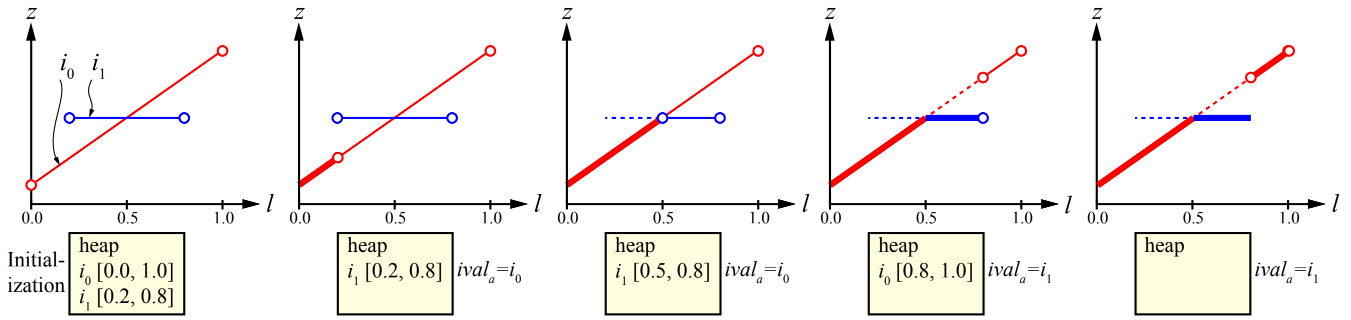


Figure 8: *Illustration of our resolve procedure (from left to right). We have two intervals, one red and one blue, and desire to find the closest interval segments along l. The fat lines indicate resolved color, which will be accumulated to the final pixel color. First, all intervals are ordered in a binary heap by their starting position along l (first diagram). The algorithms then works by sweeping l, keeping track of the interval currently being closest to the camera, that is, with the smallest z-value. At each encountered starting point, the closest interval is selected while the occluded interval is moved to the next-coming starting point (diagram 2). At intersections, the interval being occluded initially is clipped into a (potentially) visible part (diagram 3). This way, only two intervals needs to be processed at a time.*

kept for each SM, which allows for parallel insertion without inter-SM synchronization. The number of SMs on an NVIDIA GTX 580 GPU, for example, is 16, which makes this a reasonable approach. Each thread block reads chunks of BCs, and may compact them due to culled segments. More chunks are read until we have 512 segments within the view frustum to process (or until all segments have been fetched). This ensures that all threads within the thread block have work to do. Shared memory is used to efficiently coordinate bin list insertion between threads [Laine and Karras 2011].

5.3 Rasterization

This pipeline stage is divided into two separate kernels, *coarse* and *fine* rasterization. First, coarse rasterization is performed using the bin list of the tile, and then fine rasterization is performed to calculate the intersections with the line samples.

Coarse Rasterization Here, the BCs in each 128×128 tile is binned to the line samples within each tile. To determine whether a BC overlaps with a line sample, the BC's bounding box is simply tested against plane of the line sample.

Fine Rasterization During fine rasterization, each warp processes a single line sample. The line sample is divided into 32 4-pixel regions associated with memory for storing intervals overlapping that region. The current size of each list is kept in shared memory. This memory is allocated for each thread of a warp when the kernel is launched and is reused for each line sample processed by a warp. Typical memory requirements for these lists can be found in Section 6.

First, all BCs corresponding to thick curves are processed for the line sample. Each thread calculates the tessellated intersections for a single BC. The resulting intervals are shaded and appended to overlapping interval lists using shared memory atomics. Then, all remaining BCs are processed. Each thread calculates the intersection between the line sample and a BC. The resulting intervals are once again shaded and appended to overlapping interval lists using shared memory atomics. Once all BCs have been processed, each thread becomes responsible for a 4-pixel region and its associated interval list. Each interval list is then resolved according to the algorithm in Section 4.4 in parallel. Occlusion culling is performed both using the bounding box of each BC as well as for each interval before it is put into an interval list.

5.4 Sample Blend

After rasterization is complete, we essentially have two copies of the frame buffer; one for horizontal line samples and one for vertical line samples. The purpose of this stage is to combine the result of these sampling directions in a smart way. For each pixel, we blend between two line samples using a similar heuristic for curves that Jones and Perry [2000] use for triangle edges. As an extension, the contribution by a curve is multiplied by its visible length along the line sample. The purpose of this extension is that misaligned occluded curves should not change how the weighting is performed. Also, we attempt to punish misaligned curves by providing a *negative* weight for them. Below we give the details of this weighting for the reproducibility of our work.

If the curve tangent is projected to screen space, the angle, α, from the line sample to the tangent influences the weight. In particular, we use the weight:

$$w = \begin{cases} +\left(\frac{\sin\alpha - \sin\beta}{1-\sin\beta}\right)^2, & \text{if } \alpha > \beta \\ -\left(\frac{\sin\beta - \sin\alpha}{\sin\beta}\right)^2, & \text{otherwise} \end{cases}, \tag{6}$$

where β is the threshold were the curve is assigned a negative weight. We use $\beta = 15°$. This weight is integrated during the interval resolve procedure along with the color of the interval. During sample blend, we have two weights, w_h and w_v, and two colors, $\mathbf{c}_h$ and $\mathbf{c}_v$, that represent horizontal and vertical line samples respectively. If any of the line samples have a negative weight, the sample with the largest weight will be picked. Otherwise, the following formula, using a smoothstep function, is used to blend between the samples:

$$\mathbf{c}_p = \mathbf{c}_h + (\mathbf{c}_v - \mathbf{c}_h) \cdot s^2(3 - 2s), \tag{7}$$

where $s = \frac{w_v}{w_h + w_v}$ and $\mathbf{c}_p$ is the final color of the pixel.

Sample blending is not perfect, e.g., when a single near horizontal curve is in front of multiple near vertical curves. In this case, both sampling directions may contain significant error. A simple way to deal with those cases is to use more than two line samples per pixel.

6 Results

Our implementation runs on an NVIDIA GTX 580, and all images in this paper were rendered at 1024×1024 pixels. For all our timings, we report how much time it took to render primary visibility and local shading. The shadows were baked into the control points

HAIRY GUY	Zoom 1 (furthest)		Zoom 2		Zoom 3		Zoom 4		Zoom 5		Zoom 6 (nearest)	
Our	110 ms	*65.33* dB	130 ms	*60.83* dB	140 ms	*56.23* dB	140 ms	52.05 dB	220 ms	48.65 dB	250 ms	47.39 dB
CudaRaster	1700 ms	56.62 dB	1150 ms	53.24 dB	660 ms	49.39 dB	550 ms	45.40 dB	670 ms	42.05 dB	800 ms	40.38 dB
OpenGL	*20* ms	61.33 dB	*30* ms	58.51 dB	*90* ms	56.22 dB	*80* ms	*54.75* dB	*140* ms	*52.26* dB	*160* ms	*51.13* dB

Table 1: *Rendering times for the teaser scene compared to other algorithms. Each of the six leftmost zoom-levels in the image are included here. Higher PSNR indicates closer resemblance to ground truth and is thus better. OpenGL is consistently faster, but overall only by a modest factor. At far distances, our algorithm achieves better image quality than OpenGL does. CudaRaster requires* $3 - 15\times$ *more rendering time than our algorithm, while also exhibiting lower PSNR.*

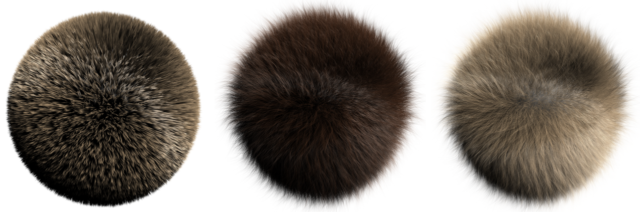


Figure 9: *Some different types of fur rendered with our algorithm. From left to right, these models contain 50k, 100k, and 150k fur strands, where each fur strand consist of 8, 16, and 16 Bézier curves, respectively. The rendering times were (left to right) 78 ms, 358 ms, and 531 ms. Zoom in the pdf to explore the quality of the visibility.*

of the Bézier curves using opacity shadow mapping [Kim and Neumann 2001] with line sampling. If n layers are used, we have seen that the shadow map generation takes approximately $n \cdot t$ ms, where t is the time for computing primary visibility. For simplicity, all our images were rendered with Kajiya and Kay's phenomenological appearance model for hair [1989]. In the case of thick curves, we apply a simple normal variation that integrates to the same color as the corresponding thin curves. Our focus in this research is on accurate visibility, but we note that other appearance models, such as the one by Marschner et al. [2003], should be possible to use as well. In all our images, we use two horizontal and two vertical line samples per pixel.

To make comparisons, we use two additional algorithms, namely, a modified version of Laine and Karras' CudaRaster [2011] and OpenGL. CudaRaster is modified to support 64 samples per pixel and, since it does not support tessellation, the thin curves were tessellated in a separate stage and fed to the pipeline as triangles. In this separate stage, we tessellate until the screen-space area of the triangle formed by the control points of a curve is smaller than a threshold of 0.5 or 1, depending on the scene complexity. We take care to only tessellate curves within the view frustum in order to make the comparison as fair as possible. The time required for this tessellation stage have been excluded from all measurements of the performance of CudaRaster. The OpenGL implementation uses multi-sampling anti-aliasing (MSAA) and hardware tessellation with pre-tessellation frustum culling to achieve maximum performance. An MSAA rate of 8 was used in combination with downsampling from a render target enlarged 4×4, for an effective rate of 128 samples per pixel. It should be noted that 128 is the maximum, single-pass sampling rate available for OpenGL, due to size restrictions of the render target. To reach a comparable curve quality, we implemented adaptive tessellation to sub-pixel size (0.25 pixel). We measure the quality of the algorithms by calculating peak-signal-to-noise (PSNR) figures between ground truth images and images produced by each of the algorithms. For ground truth, we use OpenGL with tiling to increase the sampling rate to 8192 samples per pixel.

In Figure 1, we show that our visibility algorithm can render practically noise-free visibility of a complex model from many different distances. The hair strands in this model are each unique, i.e., the thin control curve interpolation method in Appendix A was *not* used. As mentioned in the caption, the image to the left (where the camera is farthest away) renders in 110 ms, and the most detailed close-up of the face renders in 470 ms. Note that the leftmost image is relatively difficult to process quickly since there are many hair strands per pixel, which increases intersection computations. The rightmost image showcases our ability to render thick curves. Rendering times and PSNR figures for the different algorithms for this scene is shown in Table 1. It is clear that our algorithm outperforms CudaRaster in every case, both concerning PSNR and rendering time. OpenGL is usually better than our algorithm, but it is worth noting that we gain performance as we zoom in to the image. Our quality is also better when looking at the model from far away. It should be noted that when the model is far away, PSNR increases by default due to the presence of more white pixels.

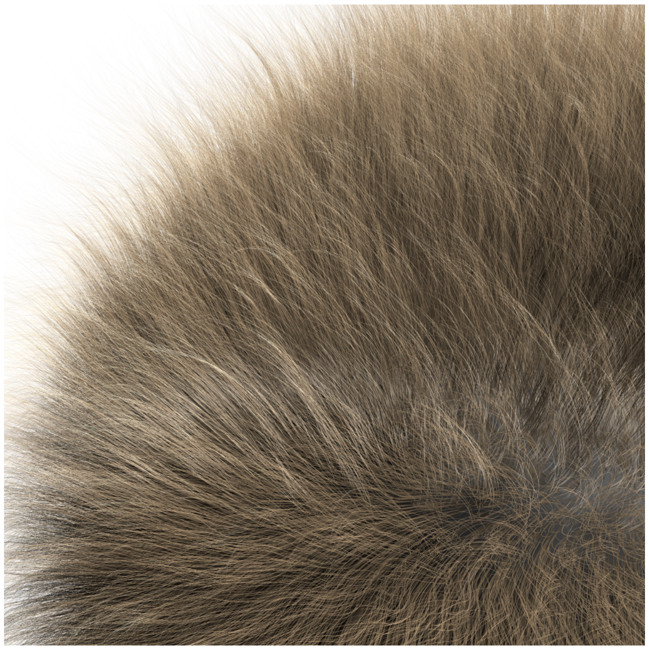


Figure 10: *Closer rendering of the fur shown to the right in Figure 9.*

In Figure 9 and 10, we show some renderings of different fur with different shader parameters with many fur strands per object. These models use the interpolation technique from Appendix A with 5,762 thin control curves. The fur ball to the right can be seen as a stress test for our algorithm, since it consists of $16 \cdot 150\text{k} = 2{,}400\text{k}$ Bézier curves. We show a detailed performance comparison of the ball in the middle in Table 2 at different zoom levels. From far away, we have better PSNR than OpenGL, but OpenGL is faster. In the close up, OpenGL is slower, but have better PSNR. CudaRaster performs worse than the other algorithms in all cases.

It is clear that OpenGL outperforms our algorithm in almost every scene. We are, however, very close and would like to stress that it

MINK			
Our	140 ms *56.24* dB	360 ms 41.37 dB	*130* ms 44.19 dB
CudaRaster	1420 ms 48.70 dB	1200 ms 34.91 dB	690 ms 38.58 dB
OpenGL	*60* ms 53.49 dB	*230* ms *43.30* dB	140 ms *50.63* dB

Table 2: *Rendering time comparisons for the middle ball of Figure 9 at various distances. Up close (right), our algorithm is able to cull large parts of occluded geometry and performs faster than OpenGL, even though PSNR is lower. At a distance (left), however, while being slower than OpenGL, our rendering has better PSNR.*

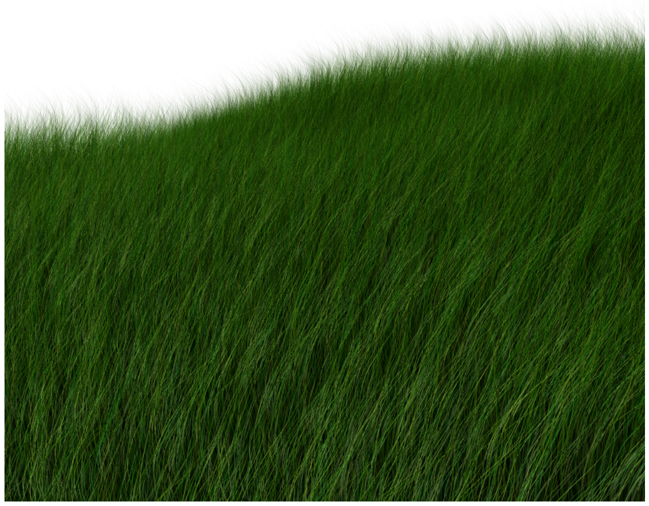


Figure 11: *Our algorithm renders the* GRASS *scene, containing 150k straws with a total of 3 million curve segments, in 406 ms.*

is quite remarkable that a software-only approach can be this close performance-wise to a pipeline accelerated by fixed-function tessellation & rasterization units, and by color & depth compression units (which our algorithm cannot use, nor can CudaRaster). If the same algorithms were implemented on a CPU, we argue that the performance would be more comparable to the comparison between our algorithm and CudaRaster. Still, the fact that our algorithm runs extremely well on a GPU shows that it lends itself well to massive parallelization.

We investigated how our algorithm performs with respect to image resolution. For the fur ball to the right in Figure 9, we obtained the following rendering times:

512×512	768×768	1024×1024
328 ms	422 ms	531 ms

As can be seen, the time per pixel decreases with higher resolutions. In fact, the time per pixel for 1024×1024 is less than half of that for 512×512, for example. This is to be expected since there will be fewer intervals per pixel the higher resolutions we use, and also better parallel utilization of the GPU. The rendering time of a frame is typically divided into about 2.5% for coarse rasterization and 97% for fine rasterization, while the sum of the other stages (setup, binning, sample blend) is negligible. Hence, it is clear that it is the fine rasterization stage that should be optimized for improved performance. We leave this for future work.

Below we show the memory usage of our pipeline when rendering the fur ball in the middle in Figure 9:

BC records	Bin lists	Tile lists	Interval lists	Total
25.9 MB	9.7 MB	150.7 MB	38.4 MB	225MB

Here, *BC records* are the records created in the setup stage, *bin lists* represents the memory used for binning, *tile lists* represents the memory used for coarse rasterization, and *interval lists* represents the memory used for storing intervals during fine rasterization.

Another test scene rendered with our algorithm is shown in Figure 11.

7 Conclusions and Future Work

We have presented a novel visibility engine for thin curve rendering. Contrary to previous work, our method uses a single, continuous geometric representation from all view directions and at all scales. Our new resolve procedure for intervals contributes significantly to high performance, while using our intersection algorithm between thin curves and line samples contributes to high-quality visibility. There are many avenues for future work. First, it would be interesting and challenging to extend our approach to handle motion blur, perhaps with a method similar to how motion blur was dealt with for line sampled triangles [Gribel et al. 2011]. Using stochastic simplification [Cook et al. 2007] of the thin curves, so that fewer, but fatter curves can be rendered from far away also seems worthwhile to explore, since performance would increase. It is clear that hair, especially blond hair, is transparent, and we believe that the method by Gribel [2010] for resolving for visibility could be used off the shelf for this. However, we would also like to research faster methods for transparent resolve. Finally, we will also continue working on extended shadow mapping techniques. If the intervals are compressed, it should be possible to use a line sampled shadow map.

Acknowledgements Thanks to Luca Fascione and Sebastian Sylwan from Weta Digital for the hair model. We acknowledge the Swedish Research Council, and Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

References

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *High Performance Graphics*, 145–149.

BERTAILS, F., KIM, T.-Y., CANI, M.-P., AND NEUMANN, U. 2003. Adaptive Wisp Tree: a Multiresolution Control Structure for Simulating Dynamic Clustering in Hair Motion. In *Symposium on Computer Animation*, 207–213.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

COBB, E. S. 1984. *Design of Sculptured Surfaces using the B-Spline Representation*. PhD thesis, University of Utah.

COOK, R. L., HALSTEAD, J., PLANCK, M., AND RYU, D. 2007. Stochastic Simplification of Aggregate Detail. *ACM Transactions on Graphics, 26*, 3 (July), 79:1–79:8.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2009. *Introduction to Algorithms*, third ed. MIT Press.

ELBER, G., KWON LEE, I., AND SOO KIM, M. 1997. Comparing Offset Curve Approximation Methods. *IEEE Computer Graphics and Applications, 17*, 62–71.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2011. Stochastic Transparency. *IEEE Transactions on Visualization and Computer Graphics, 17*, 8 (August), 1034–1047.

FARIN, G. 2002. *Curves and Surfaces for CAGD—A Practical Guide*, 5th ed. Morgan-Kaufmann.

GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics*, 163–172.

GRIBEL, C. J., BARRINGER, R., AND AKENINE-MÖLLER, T. 2011. High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility. *ACM Transactions on Graphics, 30*, 4 (August), 54:1–54:11.

HADAP, S., CANI, M.-P., LIN, M., KIM, T.-Y., BERTAILS, F., MARSCHNER, S., WARD, K., AND KAČIĆ-ALESIĆ, Z. 2007. Strands and Hair: Modeling, Animation, and Rendering. In *ACM SIGGRAPH 2007 courses*.

HAIN, T. F., AHMAD, A. L., RACHERLA, S. V. R., AND LANGAN, D. D. 2005. Fast, Precise Flattening of Cubic Bézier Path and Offset Curves. *Computers & Graphics, 29*, 5, 656–666.

HERY, C., AND RAMAMOORTHI, R. 2012. Importance Sampling of Reflection from Hair Fibers. *Journal of Computer Graphics Techniques, 1*, 1, 1–17.

JONES, T. R., AND PERRY, R. N. 2000. Antialiasing with Line Samples. In *Eurographics Workshop on Rendering*, 197–205.

KAJIYA, J. T., AND KAY, T. L. 1989. Rendering Fur with Three Dimensional Textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 271–280.

KIM, T.-Y., AND NEUMANN, U. 2001. Opacity Shadow Maps. In *Eurographics Workshop on Rendering Techniques*, 177–182.

LAINE, S., AND KARRAS, T. 2011. High-Performance Software Rasterization on GPUs. In *High-Performance Graphics 2011*, 79–88.

LEBLANC, A. M., TURNER, R., AND THALMANN, D. 1991. Rendering Hair using Pixel Blending and Shadow Buffers. *Journal of Visualization and Computer Animation, 2*, 3, 92–97.

LOKOVIC, T., AND VEACH, E. 2000. Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000*, 385–392.

MARSCHNER, S. R., JENSEN, H. W., CAMMARANO, M., WORLEY, S., AND HANRAHAN, P. 2003. Light Scattering from Human Hair Fibers. *ACM Transactions on Graphics, 22*, 3 (July), 780–791.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications, 14*, 4 (July), 23–32.

MOON, J. T., AND MARSCHNER, S. R. 2006. Simulating Multiple Scattering in Hair using a Photon Mapping Approach. *ACM Transactions on Graphics, 25*, 3 (July), 1067–1074.

MOON, J. T., WALTER, B., AND MARSCHNER, S. 2008. Efficient Multiple Scattering in Hair using Spherical Harmonics. *ACM Transactions on Graphics, 27*, 3 (August), 31:1–31:7.

NGUYEN, H., AND DONNELLY, W. 2005. Hair Animation and Rendering in the Nalu Demo. In *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison Wesley, ch. 23, 361–380.

RUF, E. 2011. An Inexpensive Bounding Representation for Offsets of Quadratic Curves. In *High Performance Graphics*, 143–150.

SINTORN, E., AND ASSARSSON, U. 2008. Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering. In *Symposium on Interactive 3D Graphics and Games*, 157–162.

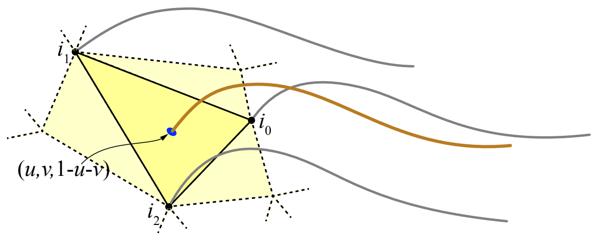


Figure 12: *Part of a triangular control mesh is shown where the grey curves are thin control curves defined at the vertices of a darker yellow triangle. Given the thin control curves, a new thin curve (brown) can be represented by thin control curve indices, (i_0, i_1, i_2), and barycentric coordinates, $(u, v, 1-u-v)$ within that triangle. This is an extremely compact representation which is valuable when rendering large numbers of thin curves.*

TILLER, W., AND HANSON, E. 1984. Offsets of Two-Dimensional Profiles. *IEEE Computer Graphics and Applications, 4*, 36–46.

TZENG, S., PATNEY, A., DAVIDSON, A., EBEIDA, M. S., MITCHELL, S. A., AND OWENS, J. D. 2012. High-Quality Parallel Depth-of-Field Using Line Samples. In *High Performance Graphics*, 23–31.

VAN WIJK, J. J. 1985. Ray Tracing Objects Defined by Sweeping a Sphere. *Computers & Graphics, 9*, 3, 283–290.

ZINKE, A., AND WEBER, A. 2007. Light scattering from filaments. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 342–356.

ZINKE, A., YUKSEL, C., WEBER, A., AND KEYSER, J. 2008. Dual Scattering Approximation for Fast Multiple Scattering in Hair. *ACM Transactions on Graphics, 27*, 3, 1–10.

ZINKE, A. 2008. *Photo-Realistic Rendering of Fiber Assemblies*. Dissertation, Universität Bonn.

A Compact Representation

Here, we present a method that makes thin curve representation more memory efficient. To do this, we define thin curves by interpolating *thin control curves* that emanate from the vertices of a triangular control mesh. This means that there will be a thin control curve defined at each vertex of the triangle, and thin curves generated "over" the triangle using interpolation of the three thin control curves. An interpolated thin curve is represented by three thin control curve indices, (i_0, i_1, i_2), and barycentric coordinates, which are used to compute the starting point inside the triangle and to blend the thin control curves. See Figure 12. This is similar to the approach taken by Nguyen and Donnelly [2005], with the exception that we keep this efficient representation throughout the rendering pipeline as well, rather than only using it for simulation. Conceptually, the triangular control mesh can be thought of as the scalp when performing hair rendering, and the landscape mesh when performing grass rendering, for example. With this approach, geometry is amplified by interpolating the thin control curves, which causes nearby curves to clump and align with each other. This behavior can be observed in, e.g., hair [Bertails et al. 2003]. In practice, interpolated thin curves are generated by distributing point samples over the triangular control mesh. It should be noted that this compact representation is not a requirement for our visibility algorithm. Instead each, e.g., hair strand, can be defined as a complete thin curve without interpolation. In Section 6, both methods are used.