Phantom Ray-Hair Intersector

Alexander Reshetov, David Luebke
NVIDIA

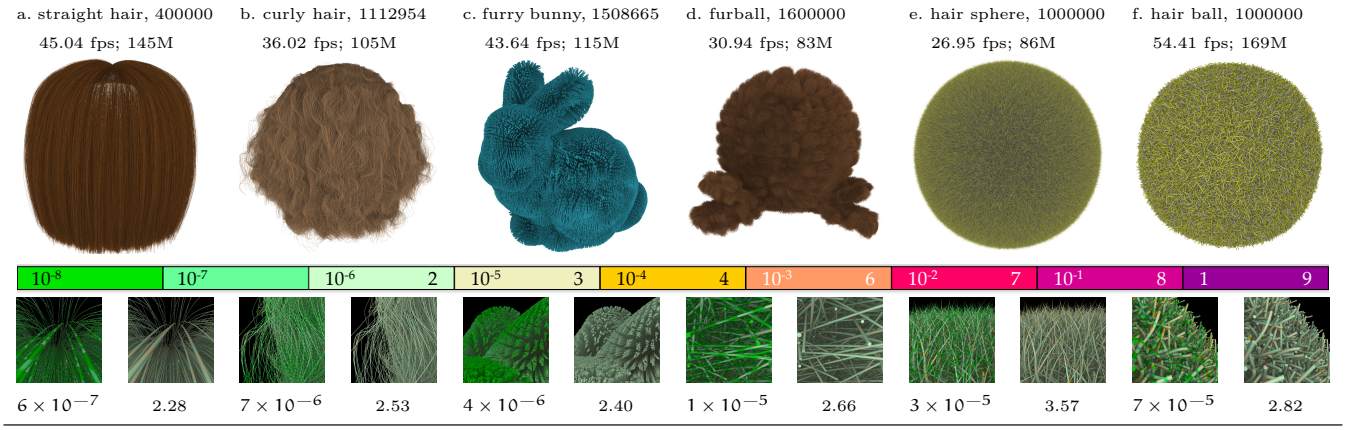

Figure 1. *Different models rendered with our technique and number of curves in the model. Second line: the achieved frame rate on a Titan Xp and the total number of rays traced per second, including one primary ray for each pixel at 1000×1000 screen resolution and four ambient occlusion rays for each hit point. For each model, the bottom row gives the average error in curve parameter* t *(left) and the number of iterations (right). The corresponding inserts show these values for each primary ray according to the heatmap strip in the middle.*

ABSTRACT

We present a new approach to ray tracing swept volumes along trajectories defined by cubic Bézier curves. It performs at two-thirds of the speed of ray-triangle intersection, allowing essentially even treatment of such primitives in ray tracing applications that require hair, fur, or yarn rendering.

At each iteration, we approximate a radially symmetric swept volume with a tangential cone. A distance from the ray-cone intersection to the cone's base is then used to compute the next curve parameter t. When this distance is zero, the ray intersects the swept volume and the cone at the same point and we stop the iterations. To enforce continuity of the iterative root finding, we introduce "phantom" intersection, padding the cone until it touches the ray if the ray-cone intersection does not exist.

HPG'18, August, 2018, Vancouver, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/3233307

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**; **Parametric curve and surface models**;

KEYWORDS

Ray tracing, Bézier curves, swept volumes, generalized cylinders, hair and fur rendering

ACM Reference Format:
Alexander Reshetov, David Luebke. 2018. Phantom Ray-Hair Intersector. In *Proceedings of HPG'18*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3233307

1 INTRODUCTION AND PRIOR ART

Rasterization renderers send a list of 3D primitives into a graphics pipeline in which each primitive is projected onto a screen and sampled at the screen resolution cadence. Ray tracing applications reverse this process: primary rays are defined by screen pixels and traced through the scene, potentially generating secondary rays. The ray tracing approach corresponds more closely to the asset production and utilization logic in film [Christensen and Jarosz 2016; Fascione et al. 2017], while rasterization has historically been used in games.

When primitives are triangles, the cost of finding the ray-triangle intersection is tantamount to solving a linear system; resolving a triangle's pixel coverage during rasterization is also a linear problem. For such primitives, performance characteristics of these two techniques stem from other issues

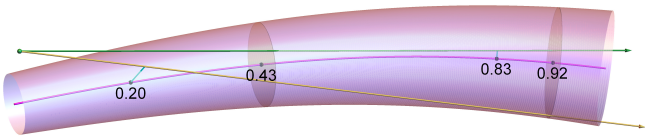

Figure 2. *The closest point of approach (CPA) failure cases. The minimum distance between the green ray and the curve is achieved at* $t = 0.83$*, whereas the ray intersects the swept volume at* $t = 0.43$*. The minimum distance between the yellow ray and the curve at* $t = 0.2$ *is* 40% *bigger than the volume's radius at this position. Therefore, it is rejected. Yet, the ray intersects the volume at* $t = 0.92$*.*

(number of pixels *vs.* number of triangles, occlusion, retained *vs.* immediate processing modes, etc).

The situation is different for higher order primitives. For rasterization, sampling in a parametric domain has a moderate and predictable cost. Finding a ray-primitive intersection though involves solving non-linear equations, which is typically carried out through iterations [Benthin et al. 2006; Kajiya 1982]. The traditional way of sidestepping these problems—by using off-line tessellation—necessitates maintaining assets in the two different formats. It also goes against the general trend of relying on run-time computations instead of using pre-processed data.

In hair and fur rendering [Ward et al. 2006], off-line tessellation might be prohibitively expensive. For this reason, the prevalent approach is to tessellate at run-time by dynamically splitting a curve's parametric domain until a curve fragment can be safely approximated with a straight line [Barringer et al. 2012; Chiang et al. 2015; Nakamaru and Ohno 2002; Qin et al. 2014; Woop et al. 2014]. Recently, Reshetov [2017] proposed to eliminate the linearization step altogether by exploiting the advanced algebraic properties of the underlying polynomials.

In all these methods, the point on the curve closest to the ray is found first. The distance d from this point to the ray is then compared with the curve's maximum half-width at this point $r(t)$. If $d < r(t)$, the 'intersection' is asserted and the shading normal is constructed. Such methods—called "the closest point of approach" or CPA—were first developed in nautical science [Morrel 1961] to reduce ship collisions.

The CPA 'intersection' might be quite different from the actual one. For variable $r(t)$, it is also possible to miss one as well, as explained in Figure 2. The CPA method is acceptable when $r(t)$ is small and is not changing quickly. Even in this case, it introduces artificial high-order frequencies in the rendered image that are manifested when a ray and a curve directions are close. Since such frequencies are caused by the systematic errors, they cannot be removed with either pixel oversampling or additional curve splitting.

CPA is commonly used in practice since it is computationally simpler than computing ray-surface intersections. For cubic curves, finding the closest distance is algebraically equivalent to finding roots of a $5th$ degree polynomial. For bona fide ray-swept sphere intersection, van Wijk [1985] shows that the degree of the polynomial is 10, provided the degree of $r(t)$ is less or equal to 3. Note also that sometimes it might be advantageous to pad a very thin hair viewed from a distance to facilitate antialiasing. The Arnold system [2018] allows to make extremely thin hair strands thicker in screen space while at the same time making them transparent.

The credit for thoroughly exploring the mathematical apparatus in ray-swept volume intersections goes to van Wijk [1985] and Bronsvoort and Klok [1985]. They also suggested subdivision as the preferred way of finding such intersections.

We build on these contributions and propose a solution that is well-suited for GPU. We consider only radially symmetric swept volumes defined by a parametric radius $r(t)$. At the core of our algorithm is a ray-cone intersector in a ray-centric coordinate system. Ordinarily, for a thin hair, a probability of a ray-cone intersection is low. We reformulate the intersector so it always returns the intersection. If there is no geometric intersection as such, we pad the cone until the ray just touches it. Such "phantom" intersections help guide the iterative process toward the actual closest intersection (if it exists). We also update ray and curve parameters in unison and come up with an iterative scheme that has a simple geometric interpretation and requires only two or three iterations in most cases.

We describe the basic ideas behind our approach in section 2, followed by CPU and GPU implementation details in section 3. Our CPU version, while finding the exact intersections, still improves performance by about 25%, in comparison with the fastest CPA implementation [Reshetov 2017]. To take advantage of a throughput-oriented GPU architecture, we simplified and streamlined our algorithm, achieving two-thirds of the performance of the optimized ray-triangle intersector in the OptiX system [Parker et al. 2010]. This result was quite unexpected, given the non-linear nature of a swept volume primitive. In section 4, we provide the relevant statistical data in order to quantify the execution aspects of our algorithm.

We consider only issues related to a geometric ray-primitive intersection. Modeling of such a complex phenomenon as light-hair interaction is extensively discussed in work of other researchers [Andersen et al. 2016; Chiang et al. 2015; Wu and Yuksel 2017; Yan et al. 2015].

2 BASIC IDEAS

The swept volume is constructed by dragging a circle with a variable radius $r(t)$ along the trajectory defined by a given curve $\mathbf{c}(t)$.

A ray-swept volume intersection point is defined by the parameter s along the ray $\mathbf{o} + s\,\mathbf{d}$; we also need the corresponding parameter t on the curve to compute the surface normal. When the curve is a straight line and the radius $r(t)$ is linear, the swept volume is a cone. In such a case, s is the smallest root of the quadratic equation for the ray-cone intersection (i.e. one that is closest to the ray origin). Parameter t can be subsequently computed by finding a distance from the intersection point to the cone's base plane.

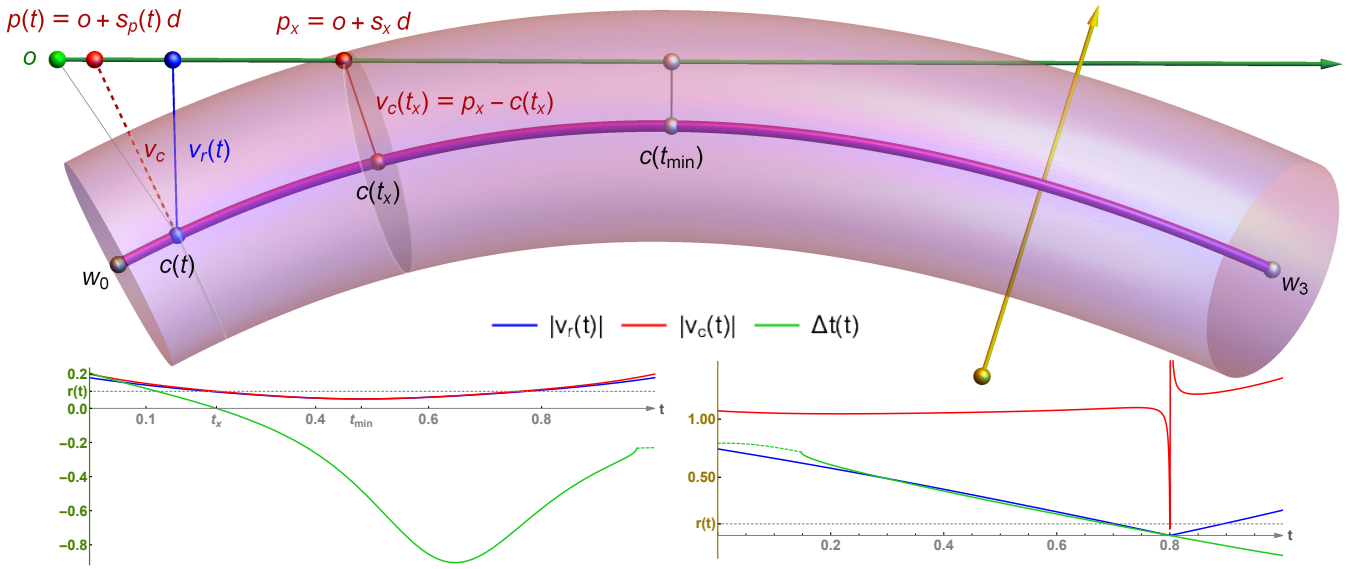

Figure 3. *In the CPA method, the length of vector* $\mathbf{v}_r(t)$ *from a point on the curve* $\mathbf{c}(t)$ *to ray* $\mathbf{o} + s\ \mathbf{d}$ *is minimized. For the green ray, this minimum is at* t_{min}*. To find ray-swept volume intersections, we consider the plane passing through* $\mathbf{c}(t)$ *that is orthogonal to the curve's tangent vector* $\mathbf{c}'(t)$*. The ray intersects this plane at* $\mathbf{p}(t) = \mathbf{o} + s_p(t)\ \mathbf{d}$*, where* $s_p(t)$ *is given by (2). Once the length of the vector* $\mathbf{v}_c(t)$ *from* $\mathbf{p}(t)$ *to* $\mathbf{c}(t)$ *is equal to the radius of the swept volume* $r(t)$*, the ray intersects the swept volume (at point* $\mathbf{p}_x$ *for the green ray). However, the equation* $|\mathbf{v}_c(t)| = r(t)$ *is difficult to solve numerically. Instead, we search for the roots of* $\Delta t(t)$ *function, which is described in Figures 4 and 5. Two charts show the functions that can be used to find the intersections (left chart for the green ray, and right—for the yellow).*

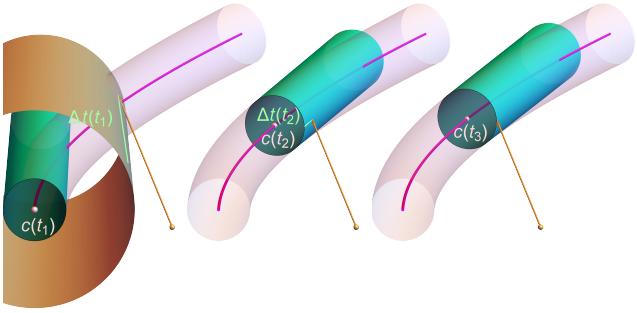

Figure 4. *Three iterations of the phantom algorithm. At initial* t_1*, the ray does not intersect the corresponding tangent cone, but it touches the padded cone. We project the intersection point to the cone's base to find the next* $t_2 = t_1 + \Delta t(t_1)$ *and repeat the process until* $\Delta t \approx 0$*.*

In a general case, we could iteratively apply the similar technique. We start at t_1 and consider the cone with the base defined by the swept volume circle at t_1. In the case shown in Figure 4, the ray will not intersect such a cone, but it will touch the padded cone. It allows us to find the next $t_2 = t_1 + \Delta t(t_1)$. The function $\Delta t(t)$ is a distance between the ray-cone intersection and the cone base, multiplied by the length of the curve's tangent vector $\mathbf{c}'(t_1)$—which is also the cone axis—as shown in Figure 5. Two additional iterations converge to the intersection point at t_3. Note that the ray will intersect the swept volume and the cone defined by t_3 at the same point and t_3 is a root of the equation $\Delta t(t) = 0$. If the subsequent Δt values have the opposite signs, they can be furnished into the secant method to find the zero-crossing of the abscissa axis for better accuracy. We will now describe the details of this algorithm.

2.1 Notation

A cubic Bézier curve can be represented as either Bernstein or a univariate polynomial by

$$\begin{aligned} \mathbf{c}(t) &= (1-t)^3\mathbf{w}_0 + 3(1-t)^2 t\mathbf{w}_1 + 3(1-t)t^2\mathbf{w}_2 + t^3\mathbf{w}_3 \\ \mathbf{c}(t) &= \mathbf{u}_0 + \mathbf{u}_1 t + \mathbf{u}_2 t^2 + \mathbf{u}_3 t^3 \end{aligned} \tag{1}$$

where $\mathbf{w}_i$ and $\mathbf{u}_i$ are 3D vectors and $t \in [0,1]$ is a scalar parameter. Endpoints of the curve (corresponding to $t = 0$ and $t = 1$) are $\mathbf{w}_0$ and $\mathbf{w}_3$ (Figure 3).

A length of the shortest vector $\mathbf{v}_r(t)$ from a point on the curve $\mathbf{c}(t)$ to ray $\mathbf{o} + s\ \mathbf{d}$ with origin $\mathbf{o}$ and unit direction $\mathbf{d}$ can be found by using properties of right triangles. It is equal to the length of the cross product vector $(\mathbf{c}(t) - \mathbf{o}) \times \mathbf{d}$ (this is shown as blue plot in Figure 3). Vector $\mathbf{v}_r(t)$ is orthogonal to the ray and it could intersect the ray at a negative or a positive value of the ray's parameter s.

Let's now consider the plane passing through $\mathbf{c}(t)$ that is orthogonal to the curve's tangent vector $\mathbf{c}'(t)$. The ray intersects this plane when s is equal to

$$s_p(t) = \frac{(\mathbf{c}(t) - \mathbf{o}) \cdot \mathbf{c}'(t)}{\mathbf{d} \cdot \mathbf{c}'(t)} \tag{2}$$

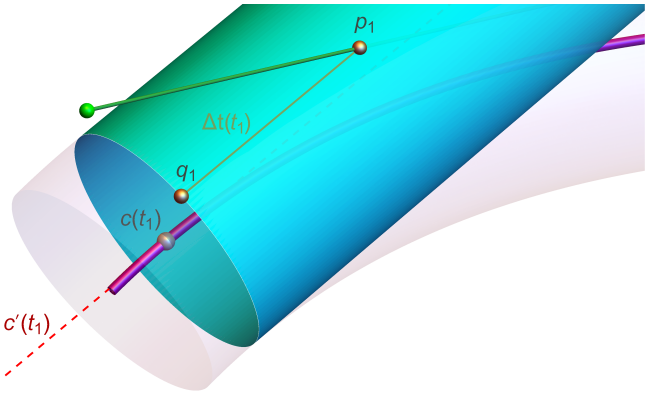

Figure 5. *For a given* t_1*, we find ray-cone intersection* $\mathbf{p}_1$ *and its projection* $\mathbf{q}_1$ *on the cone's base plane that passes through* $\mathbf{c}(t_1)$ *and has normal* $\mathbf{c}'(t_1)$*. It allows to set* $t_2 = t_1 + \Delta t(t_1) = t_1 + (\mathbf{p}_1 - \mathbf{q}_1) \cdot \mathbf{c}'(t_1)$.

Vector $\mathbf{v_c}(t)$ connects $\mathbf{c}(t)$ with the ray-plane intersection point $\mathbf{p}(t) = \mathbf{o} + s_p(t)\ \mathbf{d}$. By design, it is orthogonal to $\mathbf{c}'(t)$. We use subscript r for vectors orthogonal to the ray, and subscript c for vectors orthogonal to the curve. When $|\mathbf{v_c}(t)| = r(t)$, the ray intersects the swept volume. Such intersections come in pairs: one for the ray entering the volume and another for exiting (these two points coincide in a tangential hit). Among all such t-values in all possible pairs, we are interested in one that yields the intersection closest to the ray origin.

It is easy to compute $|\mathbf{v_c}(t)|$ as a length of the vector from the point on the curve to the ray-plane intersection. Nevertheless, solving $|\mathbf{v_c}(t)| = r(t)$ is not very accommodating if we are interested in ray-swept volume intersections. First, we have to differentiate between entry and exit positions. Additional computations are required to find the ray's parameter s for the found roots (generally, we need the smallest positive s). There could be numerical issues when the denominator in (2) is close to 0. The biggest problem though is that in such situations ($\mathbf{d} \cdot \mathbf{c}'(t) \approx 0$), i.e. when a ray is almost orthogonal to a curve, function $|\mathbf{v_c}(t)|$ is very steep and finding its intersections with $r(t)$ can be challenging (the yellow ray in Figure 3).

2.2 Phantom Intersector

The basic step of our algorithm is a ray-cone intersection (Figures 4 and 5). Starting at $t = t_1$, we set $t_2 = t_1 + \Delta t(t_1)$ using the intersection of the ray with the cone that has axis $\mathbf{c}'(t_1)$ and base centered at $\mathbf{c}(t_1)$. The base radius is $r(t_1)$ and the cone's slant is $r'(t_1)$, both of which are defined by the hair model.

To compute function $\Delta t(t_1)$, we consider a point $\mathbf{a} = \mathbf{c}(t_1) + \mathbf{c}'(t_1)\Delta t$ on the cone axis. Let $\mathbf{p}_1 = \mathbf{o} + s_1\mathbf{d}$ be the intersection of the ray with the plane that passes through $\mathbf{a}$ and has normal $\mathbf{c}'(t_1)$, When $|\mathbf{p}_1 - \mathbf{a}| = r(t_1) + r'(t_1)\Delta t$, the ray intersects the cone. Using the $(\mathbf{p}_1 - \mathbf{a}) \cdot \mathbf{c}'(t_1) = 0$ identity, we can eliminate Δt and get a quadratic equation for the ray's parameter s. We only need the smallest root, i.e. one that yields an intersection that is closer to the ray origin. Once the root s_1 is found, we set $\Delta t = (\mathbf{p}_1 - \mathbf{q}_1) \cdot \mathbf{c}'(t_1)$, where $\mathbf{q}_1$ is the projection of $\mathbf{p}_1$ to the cone's base plane.

If the determinant of the equation for s is negative, we set the determinant to 0. Such a "phantom" solution (shown in Figure 4, left) still allows us to move closer to the actual ray-swept volume intersection, and using it produces branch-free code for one iteration of our algorithm.

At this point, we could either continue this process to get t_3, or—which is numerically more accurate—find the zero crossing of the line passing though two 2D points $[(t_1, \Delta t(t_1)]$ and $[t_2, \Delta t(t_2)]$. We use this *regula falsi* ("false position") method, credited to ancient Babylonian mathematicians, if $\Delta t(t_1)\ \Delta t(t_2) < 0$, otherwise we set $t_3 = t_2 + \Delta t(t_2)$.

This process yields a sequence of the alternating t and s values $\{t_1, s_1,\ t_2, s_2,\ t_3, s_3,\ \ldots\}$. Weaving computations of these two parameters results in a rapid rate of convergence, reducing the error by over 100× at each iteration (to obtain this estimation, we compute the ratio of `log10(error_reduction)` to `number_of_iterations` using data in Figure 1). Even though in a general case *regula falsi* converges linearly, we take advantage of the fact that in a typical hair model, curve's curvature is not changing rapidly. As an added benefit, once accuracy requirements are satisfied (section 3) and we have a real intersection—not a phantom one—we get both s and t at once. This enables us to immediately compute the hit point and the surface normal. All such computations are especially simple in a ray-centric coordinate system (RCC) in which one of the axes is ray direction $\mathbf{d}$. We provide our C++/Cuda implementation in Appendix A as a reference and also to facilitate the discussion of the performance bottlenecks in section 4.

2.3 Roots of Δt Function

Δt function is designed to "predict" its roots. When a curve is a straight line, it is a linear function. In a general case, we search for its roots, eliminate phantom ones (for which the ray only touches the padded cone), and choose the intersection closest to the ray origin. To design a cost-efficient strategy, let's consider a few examples.

The ray enters the swept volume of the top curve in Figure 6 at two points. Yet, Δt function has three roots. The middle root corresponds to the phantom value. If we start iterations at $t = 0$, we will converge to root t_1, if we start at 1—to t_3. We generally need root t_1 since it is closer to the ray origin. This is a typical situation when a curve is arc-shaped and a ray enters its swept volume at two points.

The left part of the middle curve is similarly shaped and Δt function has three roots on $[0, 0.5]$ interval. Yet the whole curve is zigzag-shaped and there are two additional (phantom) roots. Starting at $t = 0$, we will converge to t_1, at 1—to t_5. Yet, the correct solution is t_3. If this curve were split, we would not have any problems.

Characteristically, starting at the bottom curve's endpoints, we find the correct intersection since it is not occluded

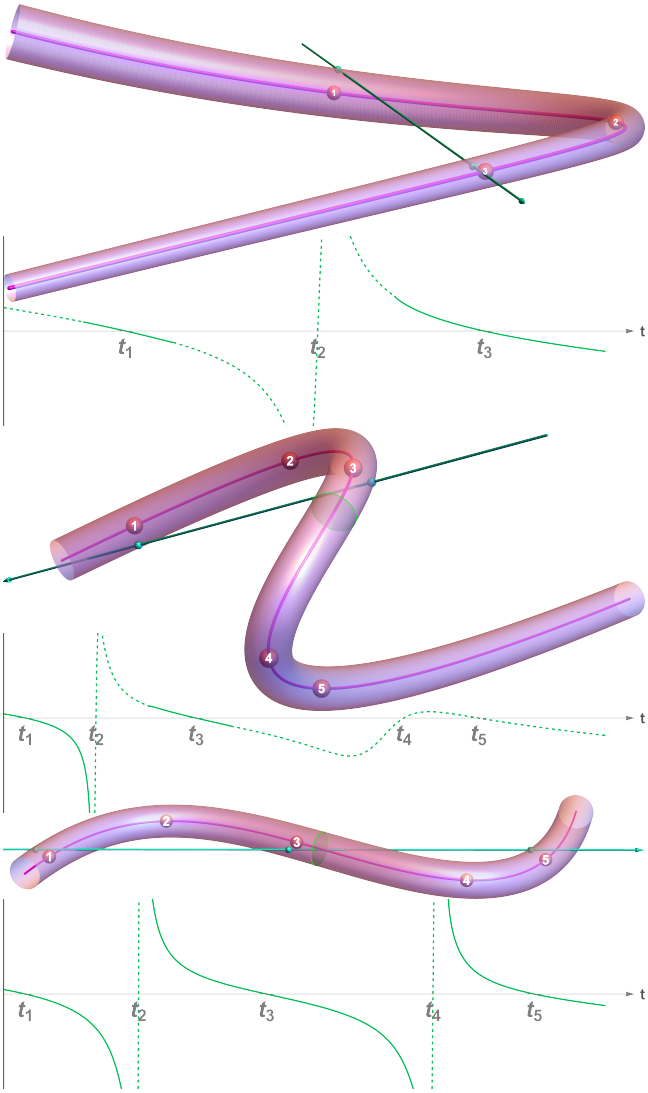

Figure 6. *Three examples of* Δt *function. Starting at curve's endpoints, we find the correct ray-swept volume intersection for the top curve (*t_1*), but not for the middle one (*t_3*). Splitting the middle curve allows to find the correct root. The ray enters the bottom curve at three separate places. We find the right root (*t_1*) since it is closest to* $t = 0$.

by the phantom roots. For other rays though we might miss an intersection. Splitting such zigzag curves will take care of this problem. It makes sense to split such curves at minimums of the curvature $\kappa = |\mathbf{c}' \times \mathbf{c}''|/|\mathbf{c}'|^3$ (indicated by green circles in Figure 6). Such positions—called curve's *vertices*—represent the salient points on the curve [Fuster and Sedykh 1995; Yan et al. 2017b]. We could also find the roots of the cubic polynomial $\mathbf{c}' \times \mathbf{c}''' \cdot \mathbf{c}' \times \mathbf{c}''$, which is obtained by differentiating the numerator of κ^2. It is computationally simpler and yields positions at which $\kappa' \approx 0$. We numerically validated this conjecture for arbitrary curves (it requires no more than 2 splits since a function maximum separates its minimums). For the real models, it is even simpler to perform the additional uniform splits (we use 8 segments for hair models in this paper). It yields a good acceleration structure and allows finding all intersections with a very simple strategy: starting iterations only at segment's endpoints.

There is another possible optimization for curves for which $\mathbf{c}'(t) \cdot (\mathbf{w}_3 - \mathbf{w}_0) > 0$ for any $t \in [0, 1]$, i.e. when the curve's tangent goes in the same direction as the base of the curve. This is true for any curve in all real models we have experimented with, let alone the split segments. We then could start iterations at the curve's endpoint closest to the ray origin. For the top curve in Figure 6, it will be $t = 0$. If we find the real intersection, we then return it immediately, without processing the other endpoint ($t = 1$). It improves performance by less than 1% but has no detrimental impact on the quality. Strictly speaking, this tweak is not correct for the self-overlapping volumes with $r(t)$ commensurable with the curvature radius as in Figure 15. We assume that for real models this will not happen.

Note that the intersection defined by t_1 for the top curve is a superficial one and Δt function has phantom values near it. Yet, it is continuous and finding t_1 is easy. The situation with root t_2 is different. Δt function changes from $-\infty$ to $+\infty$ at t_2. Geometrically, this corresponds to the ray parallel to the cone's line segment. Ordinarily, this would not result in the swept volume intersection and can be safely ignored. The only situation when this is not true is when the ray just touches the swept volume at a single point and it is parallel to the curve's tangent. During iterations, we cap the absolute Δt values by 0.5 and our iterative scheme (section 3.3) degenerates to bisection in such a case. This increases the total number of iterations, especially when there are no intersections but the ray comes near the curve (Figure 11).

2.4 Non-linear $r(t)$

All available hair and fur models, which we have tested (Figure 1a–d), have linearly varying $r(t)$. For a mostly linear function, we could predict its behavior just by looking at its values on a small interval. This is why an initial guess in the Δt root finding algorithm is not very important.

To understand the possible limitations of our technique in a more general case, we modulated curve's width as $r(t) = 0.05\,(0.25 + \sin^6 \pi t)$. Figure 7 shows a model with 1000 of such randomized curves. It turns out that our technique is still applicable with one minor modification: even though $r'(t)$ can be analytically computed, we must use the discrete derivatives $\big(r(t_2) - r(t_1)\big)/(t_2 - t_1)$, etc. The cones slanted by these values better capture the overall shape of the swept volume around the curve. Without such modification, it is possible to miss an intersection when starting at the curve's position at which $r(t)$ is still mostly constant and the ray is collinear with the curve direction. Models with wildly varying $r(t)$ would require more splits of the original curve, but such splits are dictated by the logic of the optimal

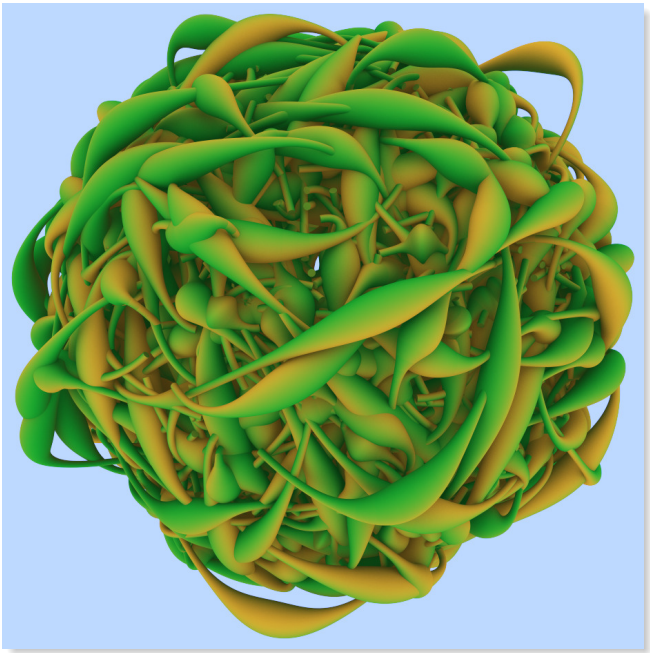

Figure 7. *1000 primitives with a non-linear* $r(t)$.

acceleration structure anyway. It also would be more appropriate to enforce the monotonic behavior of $r(t)$ on the split intervals, though we did not do so and still see artifact-free rendering.

This model runs at 96 fps on an NVIDIA Titan Xp (4 ns per ray). If we render one million similarly-shaped primitives, the performance drops only by 50% since the majority of the primitives become occluded, while the number of the traversal steps increases only logarithmically.

When finding a ray-cone intersection, it is possible to drop the terms proportional to $r'(t)^2$. Such modification is implemented in Appendix A by undefining the KEEP_DR2 preprocessor macro. Both versions (with and without $r'(t)^2$ terms) work for all the tested models. For slowly changing r', version without these terms is slightly faster. For the complex model in Figure 7, the opposite is true: using all the terms improves the performance by 0.2% by slightly reducing the total number of iterations.

2.5 Buttend Hits

There is a non-zero probability of a ray hitting the swept volume around the curve at its buttend (the base of the corresponding cone), as happens for the yellow ray in Figure 8. In such a case, Δt will not be 0 and we have to handle it separately from a general situation.

Δt is a non-linear function that could also have infinite values (when a ray is parallel to cone's surface). Fortunately, in the practical cases when a ray does intersect the surface at some angle, the function is almost linear. If we had used $\mathbf{v_c}(t)$—ray-plane intersection—we would have got an opposite behavior: easy cases with collinear directions and difficult

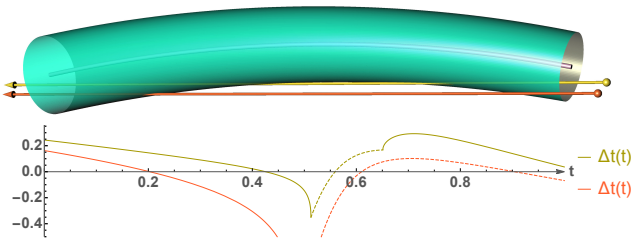

Figure 8. *Yellow ray hits the curve's volume at* $t = 1$ *buttend and root* $t = 0.43$ *has to be ignored. Red ray intersects the volume near* $t = 0.2$.

minimization problems in the practical situations. As a side effect of computing Δt values, we also find $|\mathbf{v_c}(t)|$. This allows us to use both these functions jointly to our advantage.

In the inner loop of our technique, we iteratively look for Δt roots. If the algorithm tells us to go beyond $[0, 1]$ interval and $|\mathbf{v_c}| < r$, we hit a buttend (there is Δt root for the yellow ray in Figure 8 at $t = 1.04$). A ray enters a buttend from the outside if $\mathbf{d} \cdot \mathbf{c}'(0) > 0$ or $\mathbf{d} \cdot \mathbf{c}'(1) < 0$ for the two possible buttends. In RCC, it is equivalent to using only z-component of $\mathbf{c}'$ and can be expressed succinctly as `t == cd.z < 0`. If this happens, we use the ray's parameter for the ray-plane intersection (variable `sp` in Appendix A).

In all examples in this paper we use a flat shading defined by $\mathbf{c}'$ normal for buttend hits. In principle, this behavior could be application-specific.

For connected curves, buttends will be occluded by the adjacent swept volumes. We do not have to handle such cases differently, as the ray-tracing logic will take care of it.

3 IMPLEMENTATION DETAILS

Generally, a ray-primitive intersector has to test for the existence of the intersection and then find the hit point and the surface normal. We carry it out through the following steps:

3.1. Check the ray against the curve's enclosing cylinder. Exit if no such intersection exists.

3.2. Transform the curve into the ray-centric coordinate system.

3.3. Iterate on $t \in [0, 1]$ interval as (3)
- If $\Delta t(0) < 0$ and $\Delta t(1) > 0$, ignore the interval.
- Start iterations at $t = 0$ if $(\mathbf{w}_3 - \mathbf{w}_0) \cdot \mathbf{d} > 0$, or at $t = 1$ otherwise (start at the "closer end").
- Test for convergence. If the intersection is found, report it, otherwise start at the other endpoint.

We implemented this algorithm on CPU in PBRT system [Pharr et al. 2016] and on GPU (OptiX [Parker et al. 2010]) using axis-aligned bounding volume hierarchy (BVH) acceleration structure. Now we will describe algorithm (3) in more detail.

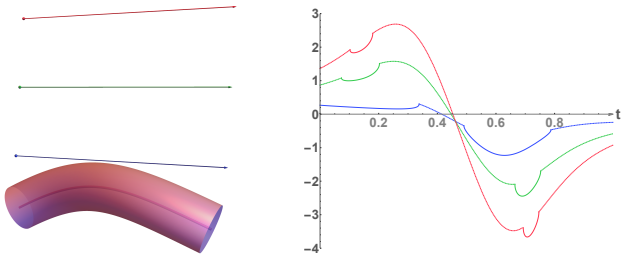

Figure 9. Δt *function for three rays roughly collinear with the curve. We have to find its roots before eliminating them. We could avoid testing the red ray since it does not intersect the curve's enclosing cylinder.*

3.1 Early Exit Check

It is possible to precompute the enclosing cylinder for the swept volume. It is easier though to find such a cylinder for the curve itself and then pad it by $r_{max} = \max_{t\in[0,1]} r(t)$ expressly as:

4.1. Set the enclosing cylinder's axis $\mathbf{d}_e$ to $\mathbf{w}_3 - \mathbf{w}_0$, i.e. the difference $\mathbf{c}(1) - \mathbf{c}(0)$ in (1). Compute a point on the axis $\mathbf{o}_e = \big((\mathbf{w}_3 + \mathbf{w}_0)/2 + \mathbf{c}(0.5)\big)/2$.
4.2. Find a conservative maximum distance d_e from the points on the curve to this axis. (4)

This is a heuristic algorithm that works well in practice. We choose the cylinder to go in the same direction as the curve's base $\mathbf{w}_3 - \mathbf{w}_0$. We then average the middle point of the base and the point on the curve at $t = 0.5$ and stipulate that the cylinder's axis goes through this point. At the last step, we split the curve a predefined number of times and compute the maximum distance from the resulting control points to the axis.

We rule out the intersection if the distance between the ray and the enclosing cylinder axis is greater than $r_{max} + d_e$. The square of such distance can be computed as

$$\mathbf{n} = \mathbf{d} \times \mathbf{d}_e$$
$$d_{re}^2 = \frac{((\mathbf{o} - \mathbf{o}_e)\cdot\mathbf{n})^2}{\mathbf{n}\cdot\mathbf{n}} \quad (5)$$

Note that we perform these computations before transforming the curve to RCC and that d_{re} is the distance between lines (ignoring the finite cylinder height). Consequently, this check might not resolve the cases when a ray is at a significant distance from the enclosure but comes close to its axis well outside the curve's bounding box. For example, this happens for the green ray in Figure 9, but not the red ray.

More elaborate approaches, which we tested, do resolve such cases but decrease the overall performance.

Figure 10 shows a few curves and enclosures computed with algorithm (4). Its impact on performance is stated in Table 1.

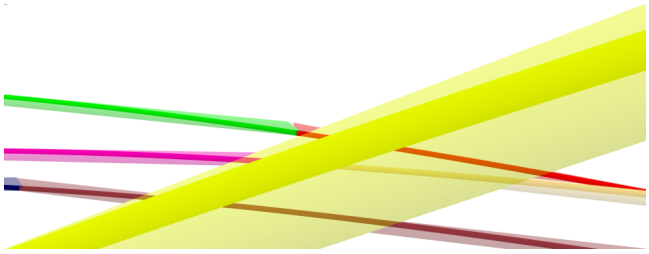

Figure 10. *Enclosing cylinders help with early eliminations.*

3.2 Ray-Centric Coordinate System

For rays that intersect both a leaf's bounding box and its enclosing cylinder, further computations are simplified by transforming the curve's coefficients $\mathbf{u}_i$ (1) to a ray-centric coordinate system in which $\mathbf{o} = \{0,0,0\}$ and $\mathbf{d} = \{0,0,1\}$. There is still one degree of freedom left in finding such a transformation—rotation angle around axis $\overrightarrow{\mathbf{od}}$.

We tested three such transformations:

1. One suggested by Reshetov [2017] to eliminate cubic term in one of the transformed curve coordinates.
2. `class optix::Onb` in OptiX [2010] system.
3. The branchless version of Duff et al [2017].

The first transformation depends on curve's coefficients and it could additionally eliminate some tests by analyzing the properties of the resulting polynomials. It helps with the performance of the CPU version (section 3.4), which does not use ray packets. For GPU, it has net negative effect. Instead, we use Duff's code that is about 10% faster by itself than the customary version 2 and, yet, has similar accuracy.

3.3 Root Finding Technique

We designed $\Delta t(t)$ to directly predict the next $t_2 = t_1 + \Delta t(t_1)$. Mostly, it is a well-behaved function, especially when there is an intersection or a near-miss. In other cases, when a ray is far away from a curve and collinear with the prevalent curve's direction, the absolute value of $\Delta t(t)$ could significantly exceed 1. Yet, the function might still have a zero-crossing on the $t \in [0,1]$ interval (Figure 9) and we have to find it before discarding it.

The derivative-free regula falsi method calculates t values iteratively as $t_{n+1} = (\Delta t_n t_{n-1} - \Delta t_{n-1} t_n)/(\Delta t_n - \Delta t_{n-1})$ and adjusts $[t_n, t_{n+1}]$ bounds to keep the root bracketed. To handle all cases uniformly, we clamp the magnitude of Δt by 0.5. For the green ray in Figure 9, the regula falsi with the clamped values yields $t_{n+1} = (t_{n-1} + t_n)/2$, i.e. the bisection value. Once we come closer to the root and the clamping is voided, the convergence becomes super-linear.

The regula falsi method generally converges to the root faster than the bisection, though sometimes it gets stuck on the one side of the root. There are many ways to handle this [Wikipedia 2016], we use the simplest possible approach by switching to the bisection every $4th$ iteration: `if ((iteration & 3) == 0) tnext = 0.5f * (tpos + tneg)`.

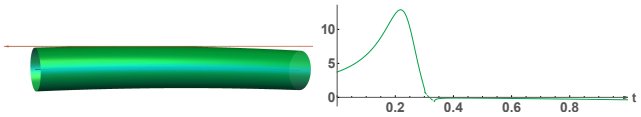

Figure 11. *The maximum number of iterations (36) in the curly hair model (the ray does not intersect the volume).*

The minimum number of iterations in our algorithm is 2. It is achieved when we start at the endpoint closest to the ray origin, compute $t_2 = t_1 + \Delta t(t_1)$ and then $\Delta t(t_2)$. If $\Delta t(t_2)$ is sufficiently small and the corresponding determinant is positive, we declare the intersection. There is no need to check the other endpoint due to the considerations discussed at the end of section 2.2. In all examples in this paper we stop iterations when $\Delta t < 5 \times 10^{-5}$. Once it happens and the found root is a real one, we still could improve t with the regula falsi, without calculating the next Δt. This yields another order of magnitude reduction in average error for all real models in Figure 1. This setting requires less than 3 iterations on average.

Average error is a more stable benchmark than maximum error while comparing hair intersection methods, since the outcome is threshold-based (hit or no hit). For this reason, the maximum error is very sensitive to the viewpoint, especially for the tangential hits.

Typically, misses need more iterations than hits. For the curly hair model, the maximum number is 36. We start at $t = 1$ and progress slowly due to the small negative values of Δt (Figure 11). Surprisingly, such outliers do not hinder the performance, even on GPU, so we did not try improving the convergence in such cases.

3.4 CPU version

We tested the phantom intersector in PBRT system [Pharr et al. 2016], which is designed to trace single rays. Accordingly, it benefits even from a low percentage test elimination if it can be resolved quickly.

While converting a curve to a ray-centric coordinate system, we set one of the RCC axes to normalized $\mathbf{u}_3 \times \mathbf{d}$. In such a transformation, the corresponding cubic term vanishes [Reshetov 2017], allowing to use the properties of quadratic polynomials to eliminate some tests.

Indeed, the distance $|\mathbf{v}_c(t)|$ from the ray-swept volume intersection to $\mathbf{c}(t)$ is greater than the distance $|\mathbf{v}_r(t)|$. It follows from the analysis of right triangles (in which a hypotenuse is the longest side), as shown in Figure 3. Furthermore, since the ray direction is $\{0, 0, 1\}$ in RCC, $|\mathbf{v}_r(t)|$ is greater than the quadratic coordinate of the transformed curve $c_2(t)$. We can find extrema of $|c_2(t)|$ for $t \in [0, 1]$ and if $\min|c_2(t)| > \max r(t)$ then $|\mathbf{v}_c(t)| > \max r(t)$, i.e. there is no intersection possible.

This helps eliminating about 5% tests for the curly hair in Figure 1 and 3%—for the straight hair model. It results in the corresponding performance improvement on the CPU, taking advantage of its branch prediction logic.

3.5 GPU version

GPU needs thread coherency to approach its potential peak performance. It is less efficient for the code with multiple branches. For this reason, we attempt quickly discarding the intersection only once, while checking the ray separation from the curve's enclosing cylinder (section 3.1).

We use the OptiX system [Parker et al. 2010] with BVH8 acceleration structure: `context->createAcceleration("Bvh8")`. It outperforms other types of bounding volume hierarchies by 30% or more. OptiX does not support customized ray payloads at this time, so we have to recompute RCC axes in all ray-curve tests, despite the fact that the chosen transformation depends only on ray data. It causes a small performance impediment in our implementation.

The phantom ray-cone intersector (Appendix A) is the main part of the inner loop of our algorithm. Even though it is more complex than a typical ray-triangle intersector, it runs significantly faster. The examination of the compiled kernel reveals that

- the compiler was able to convert a significant amount of this code to use fused-multiply-add operations and
- some computations were deferred till the end of the iterations.

In particular, inside the loop we only need the ray's parameter s relative to the z-component of the transformed $c(t)$ vector. Accordingly, the compiler computes $c_z(t)$ only when we actually have an intersection.

4 PERFORMANCE RESULTS

To broaden the scope of the investigation, we created two artificial parameterized models by placing randomized hair strands near the unit sphere. The hair ball model, in which the seeded hair direction is parallel to the sphere surface (before randomization), significantly outperforms the hair sphere model with the mostly radial direction. This is despite the fact that the average hair width is the same in these models (linearly changing from 0.01 to 0 for the hair sphere and constant 0.005 for the hair ball). This is, of course, due to the efficient occlusion handling in ray tracing applications. We set the ambient occlusion distance to ∞ and let such rays terminate at "any hit" for all the models in the paper.

We compared performance of the phantom intersector with the fastest CPA implementation [Reshetov 2017] on Intel i7-3930K at 3.2GHz. For the first 4 models in Figure 1, the performance improvement is 25, 17, 23, and 53%.

The phantom intersector was specifically designed to take advantage of the throughput-oriented GPU architecture. To put things in a familiar context, we compare its performance with OptiX ray-triangle intersector. We do so by measuring the overall execution time and counting the total number of tests (summing up per-pixel counts on CPU to avoid GPU thread synchronization). OptiX 5.0 can report the percent of the execution time inside the intersector ($\approx 30\%$ on average), allowing to calculate the kernel timing. We gather this data for the two polygonal models with the vastly different number of triangles (Stanford Bunny model and Happy Buddha) and

a. 58537 curves 412M rays/s | b. 139506 curves 400M rays/s | c. 729628 curves 322M rays/s | d. 271902 curves 515M rays/s | e. 756453 curves 531M rays/s

Figure 12. *A Titan Xp performance for the different yarn models [Wu and Yuksel 2017; Yuksel et al. 2012] for one primary and four ambient occlusion rays at 1000×1000 screen resolution (the total number of the traced rays per second).*

compare it with the first four models in Figure 1. The found ratio is 0.65, i.e. the phantom intersector runs at about two-thirds the speed of the ray-triangle intersector.

To reduce the overlap of the leaf nodes in the constructed BVH, we pre-split each curve in hair models into 8 segments. Such splits for different curves are not coordinated with each other, resulting in a less efficient BVH. For yarn models (Figure 12), splits are unnecessary since the original curves are already rather small to represent the intricate knitting patterns (and there are no zigzag-shaped curves as well). Due to the spatial regularity of such patterns, the achieved performance is significantly higher—around half billion rays per second. We assume that the performance of hair models may also be improved by tuning the building algorithm.

4.1 Memory Utilization

To avoid chained de-indexing, we copy the split curve data into the segment structure. This is not a memory-efficient approach. To study alternative memory layouts, we created three additional versions of our algorithm by storing the following data inside the segment primitive (which is a part of the curve between two t values):

1. Segment's t values and curve's index.
2. Segment's coefficients, t values, and curve's index.
3. The enclosing cylinder, t values, and curve's index.

Table 1 lists the achieved performance for such layouts and the main version of our algorithm. The enclosing cylinder data include its axis, point on the axis, and the radius. The first two entries can be derived from the curve data at run-time as well, as described in section 3.1.

For the models in this paper, storing parent's curve index, rather than segment's coefficients, results in 10-20% performance penalty (the third configuration). For the bigger models though, it might be the option of choice as it avoids bloating the memory footprint.

4.2 Accuracy

The phantom intersector accuracy can be increased by executing more iterations. This process is illustrated in Figure 13.

Table 1. *Performance (in nanoseconds per ray on a Titan Xp) for different memory layouts. One primary and four ambient occlusion rays (for hit points) are traced for each pixel at 1000×1000 screen resolution.*

model: / memory layout:	a. straight	b. curly	c. bunny	d. furball
1. t,i	10.84	13.78	12.37	18.88
2. t,i,u_{0123}	9.31	11.79	9.63	14.13
3. t,i,enclosure	8.38	10.6	9.6	13.57
4. all	6.9	9.52	8.7	12.05

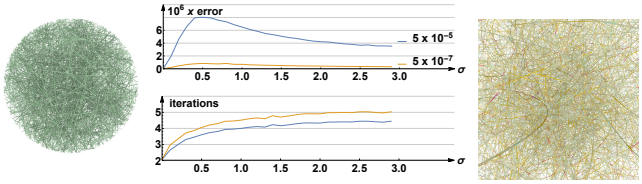

Figure 13. *We create different models by choosing equidistant control points $\mathbf{w}_i$ on straight lines with $|\mathbf{w}_3 - \mathbf{w}_0| = 1$. Each control point is then randomly modulated with a scale defined by a given σ. Left: a model consisting of all straight curves. Right: a model modulated by $\sigma = 3$. Pixel colors show number of iterations as defined by the heatmap in Figure 1. Average error (top) and number of iterations (bottom) are computed for the two different sought accuracy values: 5×10^{-5} (blue) and 5×10^{-7} (brown).*

We create a plurality of different models by randomly perturbing control points with values chosen from a Gaussian distribution with a given variance.

We use two accuracy settings that define the Δt value when iterations are concluded. As expected, for the completely straight curves, the number of iterations is 2 (we find the hit point at the first iteration and then have to validate it at the second iteration). For such models, the achieved accuracy corresponds to the single float precision. More wavering models require more iterations and the average error increases. Interestingly, for $\sigma > 0.5$, the error starts decreasing, driven by the increased number of iterations.

These measurements show that the phantom intersector can handle curves with a high curvature. Still, it does have some restrictions, which are described in the next section.

5 LIMITATIONS

The accuracy of our approach depends on the root-finding technique used. For performance reasons, we implemented a very simple version of the *regula falsi* method, starting iterations at two segments' endpoints. This approach might cause incorrect results when the radius of the swept volume is large (in comparison with the curve's curvature) and changing quickly. It is especially troublesome near sharp features ('γ' tip). One such case is shown in Figure 14.

We cancel the processing of the second endpoint if the first one leads to the intersection (see discussion at the end of section 2.2). Figure 15 shows the situation when this cancellation fails. We start iterations at the cyan end of the curve as it is closer to the ray origin. Consequently, we find the intersection and report it immediately. Yet, there is another intersection that is closer to the ray origin (near the blue curve's endpoint).

Another limitation—which can be easily fixed—deals with how roots for curve's parameter s are chosen. Such roots are a solution of a quadratic equation. We take one that corresponds to the situation when ray's origin is outside the curve's volume (see line 27 in Appendix A). If, au contraire, the ray origin is inside the volume, this would give the negative value of s. Consequently, it will be rejected. The phantom intersector could be modified to handle such situations as well, which might be necessary to model sub-surface scattering. Characteristically, the chosen approach (rejecting origins inside a hair) permits automatic handling of transparency. In such a case, the hit segment will not be intersected again, allowing to find the intersections further away along the ray.

Our lazy root-finding approach fails if the real root is bracketed by the phantom ones, as shown in Figure 16 (left). The situations depicted in Figures 14–16 can be avoided if

1. $\mathbf{c}'(t_1) \cdot \mathbf{c}'(t_2) > 0$, i.e. curve's tangent lines go in the same direction,
2. there are no curvature extrema inside the interval, and
3. the radius of curve's curvature is greater than $\max r(t)$.

We surmise that under such conditions the phantom intersector always finds the correct root, which is corroborated by the numerical experiments using linearly varying $r(t)$. The conditions 1–2 can be enforced by splitting the original curve. If, instead of searching for extrema of the true curvature

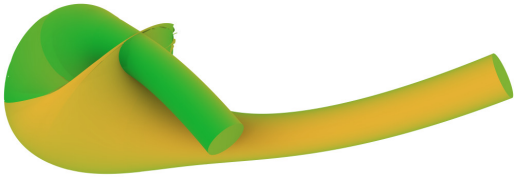

Figure 14. *Artifacts caused by rapidly varying* $r(t)$ *in the vicinity of high curve's curvature.*

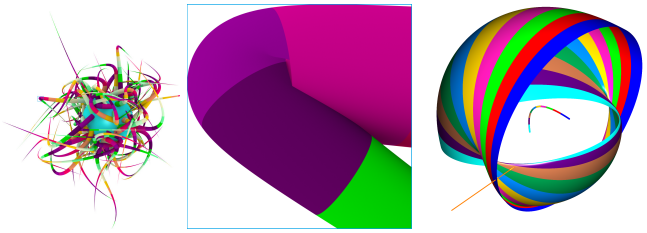

Figure 15. *Self-overlapping curves might cause problems: there are 2 ray-volume intersections near the segment's endpoints. If we abort the processing after the first one is found (near the cyan end), we will miss the second one, which is closer to the ray origin.*

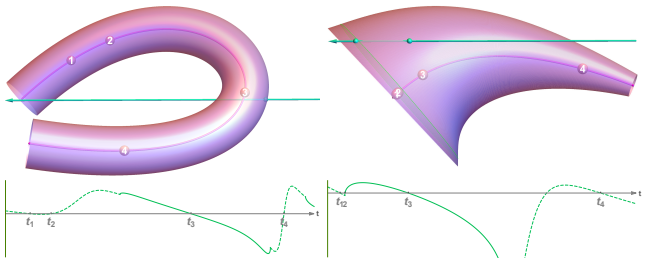

Figure 16. *Left: the real root* t_3 *is between the phantom roots and cannot be found. Right: there is a curvature minimum inside the interval (at green circle), causing the closest real root* t_3 *to be occluded by the phantom ones.*

$\kappa = |\mathbf{c}' \times \mathbf{c}''|/|\mathbf{c}'|^3$ we solve $\mathbf{c}' \times \mathbf{c}''' \cdot \mathbf{c}' \times \mathbf{c}'' = 0$ (see discussion in section 2.3), the real root may be occluded by the phantom ones, as shown in Figure 16 (right). Such contrived cases are unlikely to happen in the real models.

6 COMPARISON WITH ALTERNATIVE IMPLEMENTATIONS

We propose the first-of-a-kind adaptive solution for GPU ray tracing swept volumes around hair strands.

Adaptivity is a well-established method for ray tracing hair on CPU using the closest point of approach. On GPU platforms, less-accurate 2D and volumetric texture approximations have typically been used in practice [Andersen et al. 2016; Hadap et al. 2007; Kajiya and Kay 1989; Lengyel et al. 2001; Petrovic et al. 2005; Ren et al. 2010; Sintorn and Assarsson 2009].

Vendor-specific solutions, such as AMD TressFX [2014] or NVidia HairWorks [2017], use billboards. These systems aim

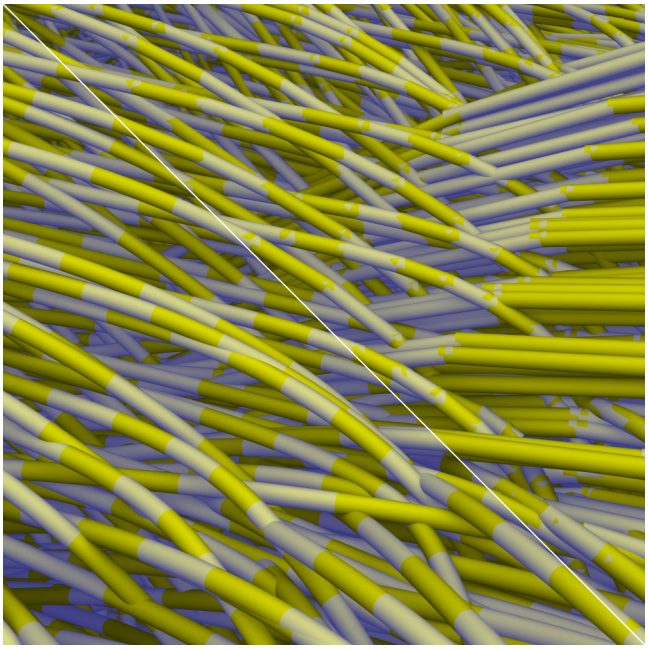

Figure 17. *Comparing phantom (left-bottom) and cylinder primitives (top-right) in the curly hair model. Alternating colors are used to show different segments.*

at the efficient hair/fur authoring, simulation, and rendering in games, using rasterization.

GPU ray-tracing systems, in which hair strands split into the predefined number of cylinders, were proposed by Sedaghat [2010] and Martins [2016].

To compare such techniques with the phantom intersector and eliminate platform and model-specific factors, we had implemented two additional methods in our system, treating each curve's segment as either a billboard or a cylinder with the maximum swept volume's radius. In all our implementations, the original hair strands are split into the same number of segments and we use the same BVH defined by the bounding boxes of the swept shapes.

We did not try optimizing billboard primitives, treating each one as 2 triangles. Ray-cylinder intersections though were implemented with just 5 dot products, similar to the ray-cone intersection (Appendix A), but in the world coordinates. Curiously, there is no much performance difference between these two implementations and the phantom one: billboards are about 3% slower and cylinders are 4% faster. Rendering distant viewpoints with the cylindrical primitives looks similar to one with the phantom primitives, but at close-ups (Figure 17) the discontinuous nature of the fixed cylindrical approximation becomes apparent.

Intersections of rays with (rational) polynomial surfaces are widely used in CAD/CAM/CAE rendering. Polynomial solvers were used in the classic papers of van Wijk [1985] and Bronsvoort and Klok [1985]. Some of their ideas are very promising for the future research. We assume that the generic solvers are slower than our geometric approach, which typically converges in 2 or 3 iterations. The phantom method may be used for the arbitrary curves and radius $\mathtt{r}(\mathtt{t})$. We use cubic curves because it is a format of the available models and the cubic curves allow the exact bounding box computations.

7 CONCLUSION AND FUTURE WORK

At the core of our approach is the ability to find the intersections of a ray and a quadratic surface. It is formulated as a solution of the corresponding quadratic equation and classified into phantom or real roots depending on the sign of the equation determinant.

We used only symmetric swept volumes since all available models have this property, allowing a rather efficient implementation in a ray-centric coordinate system (Appendix A). It might be possible to extend this approach to elliptic cones, though its performance characteristics cannot be easily ascertained for all the possible swept volume profiles.

The situation with grass-like models is less clear. Conceivably, each grass blade might be approximated by a part of a quadratic ruled surface, but this is an open problem.

Another interesting research direction is the study of efficient memory layouts, perhaps using quantization relative to the curve's leaf node. To maintain the compatibility between CPU and GPU versions of our codebase, we use a univariate polynomial representation of Bézier curves (1). For GPU, this is not necessary. We surmise that only the Bernstein forms can be reliably quantified.

Our phantom ray-hair intersector achieves two-thirds of the speed of a ray-triangle intersector. We hope that it will remove the performance considerations in choosing primitive types in ray tracing applications, allowing developers to concentrate instead on top-level system design issues [Lee et al. 2017; Pérard-Gayot et al. 2017] or more evolved reflectance models [Yan et al. 2017a]. Modern content creation tools can directly design higher-order surfaces and curves. Embracing the "as-is" principle in rendering applications should simplify asset management and production development.

ACKNOWLEDGMENTS

We are deeply grateful to Cem Yuksel and Benedikt Bitterli for providing a hair and yarn models which are used under a creative commons attribution license.

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

Tobias Grønbeck Andersen, Viggo Falster, Jeppe Revall Frisvad, and Niels Jørgen Christensen. 2016. Hybrid Fur Rendering: Combining Volumetric Fur with Explicit Hair Strands. *Vis. Comput.* 32, 6-8 (June 2016), 739–749.

Rasmus Barringer, Carl Johan Gribel, and Tomas Akenine-Möller. 2012. High-quality Curve Rendering using Line Sampled Visibility. *ACM Trans. Graph.* 31, 6, Article 162 (Nov. 2012), 10 pages.

Carsten Benthin, Ingo Wald, and Philipp Slusallek. 2006. Techniques for Interactive Ray Tracing of Bézier Surfaces. *Journal of Graphics Tools* 11, 2 (2006), 1–16.

Willem F. Bronsvoort and Fopke Klok. 1985. Ray Tracing Generalized Cylinders. *ACM Trans. Graph.* 4, 4 (Oct. 1985), 291–303.

Matt Jen-Yuan Chiang, Benedikt Bitterli, Chuck Tappan, and Brent Burley. 2015. A Practical and Controllable Hair and Fur Model for Production Path Tracing. In *SIGGRAPH Talks*. ACM.

Per H. Christensen and Wojciech Jarosz. 2016. The Path to Path-Traced Movies. *Foundations and Trends® in Computer Graphics and Vision* 10, 2 (2016).

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin. 2017. Building an Orthonormal Basis, Revisited. *Journal of Computer Graphics Techniques (JCGT)* 6, 1 (27 March 2017), 1–8.

Luca Fascione, Johannes Hanika, Marcos Fajardo, Per Christensen, Brent Burley, and Brian Green. 2017. Path Tracing in Production - Part 1: Production Renderers. In *ACM SIGGRAPH 2017 Courses (SIGGRAPH'17)*. Article 13, 39 pages.

M.C.R. Fuster and V.D Sedykh. 1995. On the Number of Singularities, Zero Curvature Points and Vertices of a Simple Convex Space Curve. *J Geom* 52, 168 (1995).

Lee Griggs. 2018. *Arnold for Maya User Guide 5*.

Sunil Hadap, Marie-Paule Cani, Ming Lin, Tae-Yong Kim, Florence Bertails, Steve Marschner, Kelly Ward, and Zoran Kačić-Alesić. 2007. Strands and Hair: Modeling, Animation, and Rendering. In *ACM SIGGRAPH 2007 Courses (SIGGRAPH'07)*. 1–150.

James T. Kajiya. 1982. Ray Tracing Parametric Patches. *SIGGRAPH Comput. Graph.* 16, 3 (July 1982), 245–254.

J. T. Kajiya and T. L. Kay. 1989. Rendering Fur with Three Dimensional Textures. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 271–280.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics (HPG'17)*. Article 10, 11 pages.

Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. 2001. Real-time Fur over Arbitrary Surfaces. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics (I3D'01)*. 227–232.

Timothy Martin, Wolfgang Engel, Nicolas Thibieroz, Jason C. Yang, and Jason Lacroix. 2014. TressFX: Advanced Real-Time Hair Rendering. In *GPU Pro 5*, Wolfgang Engel (Ed.). CRC Press, 193–209.

Jorge R. Martins. 2016. *Cone Tracing of Human Hair Fibers*. Master's thesis. Técnico Lisboa, Instituto Superior Técnico, University of Lisbon, Lisboa, Portugal.

J. S. Morrel. 1961. The Physics of Collision at Sea. *Journal of Navigation* 14, 2 (1961), 163–184.

Koji Nakamaru and Yoshio Ohno. 2002. Ray Tracing for Curves Primitive. In *WSCG*. 311–316.

Nvidia. 2017. NVIDIA HairWorks. (2017). https://developer.nvidia.com/hairworks

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH'10)*. Article 66, 13 pages.

Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leissa, and Sebastian Hack. 2017. RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 157–168.

Lena Petrovic, Mark Henne, and John Anderson. 2005. Volumetric methods for simulation and rendering of hair. *Pixar Animation Studios* 2, 4 (2005).

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hao Qin, Menglei Chai, Qiming Hou, Zhong Ren, and Kun Zhou. 2014. Cone Tracing for Furry Object Rendering. *IEEE Trans. Vis. Comput. Graph.* 20 (2014), 1178–1188.

Zhong Ren, Kun Zhou, Tengfei Li, Wei Hua, and Baining Guo. 2010. Interactive Hair Rendering Under Environment Lighting. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH'10)*. Article 55, 8 pages.

Alexander Reshetov. 2017. Exploiting Budan-Fourier and Vincent's Theorems for Ray Tracing 3D Bézier Curves. In *Proceedings of High Performance Graphics (HPG'17)*. Article 5, 11 pages.

Nasim Sedaghat. 2010. *Real-Time Hair Modeling and Rendering Using Ray Tracing on GPU: Introducing "Continual Cylinders" to Represent Hairs*. Lambert Academic Publishing.

Erik Sintorn and Ulf Assarsson. 2009. Hair Self Shadowing and Transparency Depth Ordering Using Occupancy Maps. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D'09)*. 67–74.

Jarke J. van Wijk. 1985. Ray tracing objects defined by sweeping a sphere. *Computers & Graphics* 9, 3 (1985), 283–290.

Kelly Ward, Florence Bertails, Tae-Yong Kim, Stephen R. Marschner, Marie-Paule Cani, and Ming C. Lin. 2006. A survey on hair modeling: styling, simulation, and rendering. In *IEEE Transaction on Visualization and Computer Graphics*. 213–234.

Wikipedia. 2016. Root-finding Algorithms — Wikipedia, The Free Encyclopedia. (2016). https://en.wikipedia.org/wiki/Category:Root-finding_algorithms

Sven Woop, Carsten Benthin, Ingo Wald, Gregory S Johnson, and Eric Tabellion. 2014. Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur. In *High Performance Graphics*. 41–49.

Kui Wu and Cem Yuksel. 2017. Real-time Fiber-level Cloth Rendering. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D'17)*. Article 5, 8 pages.

Ling-Qi Yan, Henrik Wann Jensen, and Ravi Ramamoorthi. 2017a. An Efficient and Practical Near and Far Field Fur Reflectance Model. *ACM Trans. Graph.* 36, 4, Article 67 (July 2017), 13 pages.

Ling-Qi Yan, Chi-Wei Tseng, Henrik Wann Jensen, and Ravi Ramamoorthi. 2015. Physically-accurate Fur Reflectance: Modeling, Measurement and Rendering. *ACM Trans. Graph.* 34, 6, Article 185 (Oct. 2015), 13 pages.

Zhipei Yan, Stephen Schiller, Gregg Wilensky, Nathan Carr, and Scott Schaefer. 2017b. K-curves: Interpolation at Local Maximum Curvature. *ACM Trans. Graph.* 36, 4, Article 129 (July 2017), 7 pages.

Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2012. Stitch Meshes for Modeling Knitted Clothing with Yarn-level Detail. *ACM Trans. Graph.* 31, 4, Article 37 (July 2012), 12 pages.

A PHANTOM RAY-CONE INTERSECTION

```
struct RayConeIntersection {  // ray.o = {0,0,0}; ray.d = {0,0,1};
  inline bool intersect(float r, float dr) {
    // cone is defined by base center c0, radius r,
    // axis cd, and slant dr
    float r2  = r * r;  // dr could be either positive
    float drr = r * dr; // or negative (0 for cylinder)

    float ddd = cd.x*cd.x + cd.y*cd.y; // all possible
    dp        = c0.x*c0.x + c0.y*c0.y; // combinations
    float cdd = c0.x*cd.x + c0.y*cd.y; // of x*y terms
    float cxd = c0.x*cd.y - c0.y*cd.x; // (c0 × cd)z

    float c = ddd;                    // compute a,b,c in
    float b = cd.z * (drr - cdd); //   a - 2 b s + c s^2
    float cdz2 = cd.z*cd.z;           //   (s for ray ∩ cone)
    ddd += cdz2;                      // now it is cd·cd
    float a = 2*drr*cdd + cxd*cxd - ddd*r2 + dp*cdz2;
    #if defined(KEEP_DR2)             // dr^2 adjustments
    float qs = (dr*dr)/ddd;           // (it does not help
    a -= qs * cdd*cdd;                //  much with neither
    b -= qs * cd.z*cdd;               //  performance nor
    c -= qs * cdz2;                   //  accuracy)
    #endif

    // We will add c0.z to s and sp latter if needed
    float det = b*b - a*c;     // for a - 2 b s + c s^2
    s   = (b - (det > 0? sqrt(det) : 0))/c; // c > 0
    dt  = (s*cd.z - cdd)/ddd; // wrt t
    dc  = s*s + dp;           // |(ray ∩ cone)  - c0|^2
    sp  = cdd/cd.z;           // will add c0.z latter
    dp += sp*sp;              // |(ray ∩ plane) - c0|^2

    return det > 0;           // true (real) or false (phantom)
  }

  float3   c0; // curve(t)   in RCC (base center)
  float3   cd; // tangent(t) in RCC (cone's axis)
  float     s; // ray.s - c0.z for ray ∩ cone(t)
  float    dt; // dt to the (ray ∩ cone) from t
  float    dp; // |(ray ∩ plane(t)) - curve(t)|^2
  float    dc; // |(ray ∩ cone(t))  - curve(t)|^2
  float    sp; // ray.s - c0.z for ray ∩ plane(t)
};
```