High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur

Sven Woop[†], Carsten Benthin[†], Ingo Wald[†], Gregory S. Johnson[†], and Eric Tabellion[‡]

[†] Intel Corporation [‡] DreamWorks Animation

Figure 1: *A zoomed-in region of the* Yeti *model (153M hair segments) from the DreamWorks Animation movie* "Rise of the Guardians". *Our technique exploits similarity in the orientation of neighboring hairs to enable efficient ray traversal of complex hair structure such as in this model.*

Abstract

Hair and fur typically consist of a large number of thin, curved, and densely packed strands which are difficult to ray trace efficiently. A tight fitting spatial data structure, such as a bounding volume hierarchy (BVH), is needed to quickly determine which hair a ray hits. However, the large number of hairs can yield a BVH with a large memory footprint (particularly when hairs are pre-tessellated), and curved or diagonal hairs cannot be tightly bounded within axis aligned bounding boxes. In this paper, we describe an approach to ray tracing hair and fur with improved efficiency, by combining parametrically defined hairs with a BVH that uses both axis-aligned and oriented bounding boxes. This BVH exploits similarity in the orientation of neighboring hairs to increase ray culling efficiency compared to purely axis-aligned BVHs. Our approach achieves about 2× the performance of ray tracing pre-tessellated hair models, while requiring significantly less memory.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Hair and fur are fundamental to the realistic appearance of rendered characters. However, rendering high quality hair in which strands are modeled individually, is computationally and memory intensive due to the number of hairs in a given model and the intrinsic geometric properties of a strand. For example, a typical character can contain 150K hairs in the case of a human, or millions in the case of a furry animal (Figure 1). Each hair may be tessellated into many triangles, with finer tessellation required to accurately represent strands with greater curvature. Further, hairs are often thinner than a pixel and shiny, resulting in high frequency changes in depth, surface normals, and shading across neighboring pixels in space and time. As a result, high sampling rates are needed to avoid aliasing and noise.

In a ray tracer, high sampling rates require a large number of rays to be traced per pixel. Worse, the computational cost of tracing a given ray through hair is itself high. The large number of primitives produced by tessellating hair models leads to correspondingly large spatial index structures, which are both memory intensive and costly to traverse. As an alternative, rays can be intersected directly with a parametric representation (e.g. cubic Bézier splines) of the individual hairs [SN90, NSK90, NO02]. Fewer primitives are needed to accurately capture hair curvature resulting in smaller spatial index structures. However, ray-spline intersection is more

expensive than ray-triangle intersection. Moreover, it is difficult to tightly bound long hair segments within axis-aligned spatial index structures (Figure 2a), resulting in many intersection tests for relatively few hits. Intersection efficiency is further reduced in regions with nearby hairs, where axis-aligned bounding boxes may overlap (Figure 2b).

The focus of this paper is on reducing the per-ray cost of intersection with a hair model in a general-purpose ray tracer or path tracer, rather than on sampling techniques to reduce the number of rays traced. Our approach is informed by two observations. First, it is possible to achieve tight bounds for parametric hair segments using one (or a small number) of bounding boxes locally oriented to the primary axis of the segment (Figure 2d and 2e), enabling individual hairs and their bounds to be represented compactly. Second, neighboring hairs typically exhibit a natural similarity in orientation. As a result, the oriented bounding boxes (OBBs) of nearby hairs overlap minimally (Figure 2f), enabling an efficient spatial index structure to be built over the full hair model. We show that this approach achieves higher performance with less memory than methods using pre-tessellated hair geometry, in production quality scenes with millions of separate hairs.

For brevity, we largely restrict our discussion to nearest-hit ray traversal of static hair models. However, our approach can easily be extended to handle shadow rays or cone tracing [QCH*13]. The data structures and kernels described here are built on top of the open source Embree ray tracing kernel framework, for which we direct the reader to [WWB*14] for more detail.

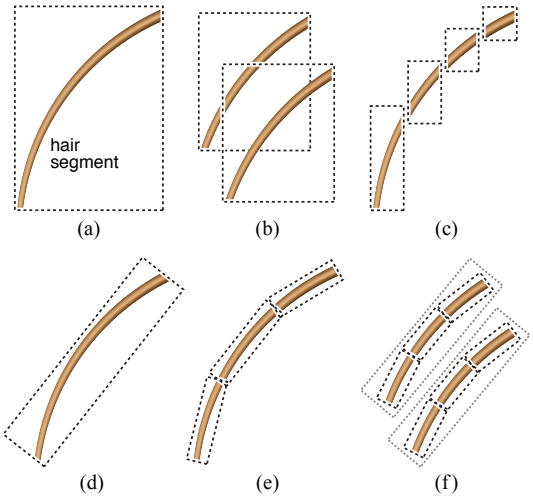


Figure 2: *An axis-aligned bounding box (AABB) does not efficiently bound a curved hair* (a)*, resulting in many ray-box intersection tests with few ray-hair hits. Worse, where hairs are densely packed* (b)*, rays must be tested against many overlapping boxes. Somewhat tighter bounds can be achieved by using multiple AABBs for a single hair* (c)*. A bounding box oriented to the primary axis of the hair* (d) *yields tighter bounds than a single AABB in the general case, and very tight bounds* (e) *can be achieved using multiple oriented bounding boxes (OBBs). In cases where nearby hairs are similarly oriented* (f) *the OBBs (and OBBs of OBBs) overlap minimally.*

2. Hair Segment Intersection

Our approach combines a highly optimized ray-hair intersection test (described here), with a specialized spatial index structure that yields tight bounds for hair-like geometry and which exploits similarity in the orientation of neighboring hairs to increase ray culling efficiency (Section 3). This technique is intended to enable efficient ray traversal in production quality scenes composed of millions of individual hairs (Figure 1, 4). We assume that non-hair geometry is handled separately using conventional spatial index structures and intersection methods.

2.1. Hair Representation and Intersection Test

In our system, hairs are modeled individually using a small number of connected cubic Bézier segments, which we refer to as *hair segments* in the remainder of the paper. Each Bézier control point has a 3D position and an associated radius. Thus the thickness of a hair can vary along its length. Specifically, the thickness of a hair at any given point is interpolated from the radii at the control points of the respective segment.

The intersection of a ray and a hair segment is performed using the test described by Nakamaru and Ohno [NO02], extended to support hair segments with variable thickness. In this test, a Bézier curve is approximated with a number of line segments obtained by recursively subdividing the curve. For each line segment, the test computes the point P along the segment that is nearest the ray, and determines if this distance is less than the thickness of the segment. To efficiently calculate P, the Nakamaru method projects the Bézier curve into a coordinate space in which the ray starts at the origin and runs parallel to the Z axis. Distance and culling tests are performed relative to the origin in this space, yielding simpler arithmetic and fewer operations.

This intersection test is approximate. Rather than finding the ray entry and exit points on the hair, the test yields only the nearest distance between a ray and a line segment. This distance is typically less than the interpolated radius, and consequently most ray "hit" points lie inside the hair. As a result, the origin of secondary rays produced during shading must be pushed outward from the line segment to the hair surface.

2.2. Vectorization of the Intersection Kernel

In principle, it is possible to vectorize the intersection kernel by testing multiple rays in parallel with a given hair. However in practice, rays lack the spatial coherence needed to achieve high vector utilization when rendering fine hair geometry. For this reason, we vectorize the intersection of a single ray with a single Bézier segment. This process is described in pseudocode in Figure 3 in terms of 8-wide AVX instructions found in modern CPUs.

```
void intersect(Ray ray, BezierCurve3D curve_in) {

  /* transform Bezier curve into ray space */
  BezierCurve3D curve = transform(ray.space, curve_in);

  /* directly evaluate 8 line segments (P0, P1) */
  avx3f P0, P1;  curve.eval(P0, P1);

  /* project ray origin (0, 0) onto 8 line segments */
  avx3f A = -P0;
  avx3f B = P1 - P0;
  avxf d0 = A.x * B.x + A.y * B.y;
  avxf d1 = B.x * B.x + B.y * B.y;

  /* calculate closest points P on line segments */
  avxf  u = clamp(d0 * rcp(d1), 1.0f, 0.0f);
  avx3f P = P0 + u * B;

  /* the z-component holds hit distance */
  avxf t = P.z;

  /* the w-component interpolates the curve radius */
  avxf r = P.w;

  /* if distance to nearest point P <= curve radius ... */
  avxf r2 = r * r;
  avxf d2 = P.x * P.x + P.y * P.y;
  avxb mask = d2 <= r2 & ray.tnear < t & t < ray.tfar;

  /* find closest hit along ray by horizontal reduction */
  if (any(mask)) {

     size_t i = select_horizontal_min(mask, t);
     ...

  }

}
```

Figure 3: *Pseudocode for our ray-hair segment intersection test using C++ wrapper classes for the 8-wide AVX instructions. The hair segment is approximated using 8 cones which are tested for intersection against the ray in parallel.*

The four control points of the Bézier curve are first projected into ray space. This projection can be formulated as four SIMD matrix-vector multiplications. Once in ray space, the Bézier curve is divided into 8 line segments (extension to 16 or more segments is straightforward), rather than recursively subdividing the curve. The start and end points (and radii) of each line segment are evaluated in parallel from precomputed Bézier coefficients. The ray is then tested for intersection against the 8 line segments in parallel. In the case of multiple hit points, the nearest is selected using a horizontal reduction operation. This kernel achieves 1.3× to 1.6× the total rendering performance of a naïve vectorized reference implementation of the Nakamaru and Ohno kernel, for the *Tighten* and *Sophie* models (Figure 4).

For our models we have found 8 line segments to provide sufficient geometric accuracy. However, for very curved hair segments a higher level of subdivision is needed. As these cases are rare, we recommend pre-subdividing problematic hairs using a flatness criteria. This approach provides high geometric accuracy and allows the use of our fast ray-hair intersection kernel, at the cost of a small increase in memory.

3. A Spatial Index Structure for Hair Geometry

One advantage of rendering tessellated hair primitives rather than directly rendering Bézier segments, is that the tessellation process breaks long, thin hairs into smaller (and less thin) triangles. These triangles can be more tightly bounded in an axis-aligned spatial index structure compared to the longer Bézier segments (a similar observation has motivated the development of spatial split techniques for general-purpose geometric primitives [EG07, SFD09]). However, accurately capturing the curvature of a hair may require 50 to 100 triangles per Bézier segment. For scenes containing millions of hairs, the memory occupied by these primitives and the associated data structure may be prohibitive.

What is needed is a spatial index structure which tightly bounds long and thin Bézier segments, enabling efficient ray traversal and culling with reduced memory consumption. A BVH consisting only of axis-aligned bounding boxes is insufficient, since the boxes of neighboring hairs frequently overlap (Figure 2b). As a result, these hairs are often not separated during BVH construction when using the surface-area heuristic (SAH). Even when the build kernel is coerced into separating hairs, the neighboring regions still overlap and must be tested during ray traversal.

3.1. Local Orientation Similarity

Our extends previous work on acceleration structures using oriented bounding volumes [AK89, GLM96, LAM01]. We note that a BVH of oriented bounding boxes can efficiently bound and separate hair segments. A single hair segment that is almost straight can be bounded very effectively using an OBB (Figure 2d). An OBB is also more effective at bounding a curly hair than an AABB (keep in mind that we build a BVH over hair *segments*, not entire hair *curves*). Further, the orientation of individual hairs is typically not random but is *locally* similar, minimizing overlap in the OBBs of neighboring hairs (Figure 2f). However, OBBs are generally more expensive to store and traverse than AABBs.

For this reason, we utilize a bounding volume hierarchy that combines both axis-aligned and oriented bounding boxes, and spatial and object splits. This mixed BVH is able to adapt to the local orientation of individual hairs, and exploit similarity in the orientation of neighboring hairs to minimize overlap between bounding boxes, while reducing the memory and compute overhead associated with a BVH built solely from OBBs.

The choice of where to use an AABB or OBB is decided at run time in the BVH construction kernel (Section 3.3) based on local geometric properties of the scene. Generally speaking, AABBs are better suited for bounding relatively short hair segments, and for the upper nodes of the BVH where they are faster to traverse than OBBs. In contrast, OBBs are used in nodes lower in the BVH where they bound tufts or strands of hair in which local orientation similarity can be maximally exploited.

It may happen that no single well-defined major hair direction exists, but two or more (e.g. two hair strands cross at some location). This happens less frequently for real-world models, but can happen for synthetic models. We address this issue by binning hair segments into sets of similar orientation during the BVH build process.

Oriented bounding boxes do not efficiently bound groups of hairs that are not only long and densely packed, but also highly curved. We address these cases by performing spatial splits during BVH construction [SFD09, PGDS09]. Spatial splits shorten very curved hair segments and reduce the curvature per sub-segment, improving the bounding efficiency of OBBs.

A case we cannot handle efficiently is densely grouped hairs where each hair fiber points in a random direction. In principle, the use of spatial splits in our BVH build kernel enables such random hair fibers to be split into smaller segments which can be more tightly bounded. However, in practice this approach increases memory consumption, reducing the comparative benefit of a hybrid AABB / OBB BVH relative to a BVH consisting only of AABBs.

3.2. SIMD-Friendly Data Layout

The goal of our data structure is to achieve significant speedups by reducing the number of traversal steps and primitive intersection tests. On the other hand, these savings could easily be lost if traversal was considerably more costly than for a regular BVH: if our data structure was limited to scalar traversal codes, the savings would have to be immense to even compete with the SIMD-optimized traversal kernels available for regular BVHs.

BVH Data Layout. To eventually allow for fast single-ray SIMD traversal we follow the data layouts of the original Embree kernels as closely as possible. In particular, we use a BVH with a branching factor of four that will allow for always intersecting four child-bounds in parallel (Section 3.4). This data-parallel intersection in particular requires that every group of four sibling nodes have to be of the same type: If only one prefers OBBs, all four nodes have to be OBB nodes. However, as argued above the distribution of AABB vs. OBB nodes is not random, but closely correlated to their depth in the hierarchy, so this in practice is not a major limitation.

Node References. A node stores 64-bit *node references* to point to its children. These node references are decorated pointers, where we use the lower 4 bits to encode the type of inner node we reference (AABB node or OBB node) or the number of hair segments pointed to by a leaf node. During traversal we can use simple bit operations to separate the node type information from the aligned pointer.

AABB nodes. For nodes with axis aligned bounds, we store four bounding boxes in a SIMD friendly structure-of-array layout (SOA). In addition to the single-precision floating point coordinates for the four AABBs this node also stores the four 64-bit node references, making a total of 128 bytes (exactly two 64-byte cache lines).

OBB nodes. For nodes with non-axis aligned bounds we store, for each node, the affine transformation matrix that transforms the respective OBB to the unit AABB $((0,0,0),(1,1,1))$. This transformation allows for a very efficient ray-OBB intersection test, by first transforming the ray with this matrix, and then intersecting the transformed ray with the unit AABB (in which case many terms become simpler). The four OBB nodes' transformations are stored in a SIMD friendly SOA layout, requiring a total of 192 bytes. Together with the 4 node references this makes a total of 224 bytes for an OBB node, or roughly twice as much as for an AABB node.

Leaf Nodes specify a list of references to Bézier segments contained in this leaf. For each Bézier segment reference we store a 64-bit pointer to the Bézier-segment's first control point, which allows us to share the first and last control point of neighboring hair segments. Additionally we store two 32-bit integers to encode some group ID and hair ID, which allow a shader to look up per-hair shading data (as done in the case of the Yeti in Figure 1) and can be used for mailboxing in case spatial splits are enabled.

Bézier Segments are stored in one or more groups as specified by the application, where each Bézier segment is specified as a list of 4 control points. The arrays of Bézier segments are stored with the application, and no separate copy of this data is required, as our leaf representation can directly point into these arrays. We also experimented with a version where the Bézier segment references are replaced with copies of the actual four control points. This saves a pointer indirection and allows more efficient prefetching, which translated to roughly 5% higher performance; however, due to the higher memory consumption this mode is currently not being used.

3.3. Spatial Index Construction

Though our combination of AABBs and OBBs *can*, in theory, better enclose geometry, how well it will do so in practice will depend on how exactly the data structure is built; in

(a) Tighten
2.2M hair segments, 420K triangles

(b) Tiger
6.5M hair segments, 83K triangles

(c) Sophie
13.3M hair segments, 75K triangles

Figure 4: *Three reference models from the DreamWorks Animation movies* "Megamind" (a), "The Croods" (b), *and* "Rise of the Guardians" (c) *featuring greatly varying complexity and style of the hair geometry.*

particular, which orientations will be chosen for the OBBs, and how the builder will decide which node types to choose.

3.3.1. Determining Hair Space

Non-axis aligned partitioning strategies are performed in a special coordinate frame which we call the *hair space*, that is well aligned to a set of hair segments. To calculate this space we randomly pick a small set of candidate hair segments. For each such candidate we compute a dominant axis spanning from the first to the last vertex and a randomized orientation around this dominant axis, to obtain an orthogonal *candidate space*.

For each such candidate space, we compute how well it is aligned with the given hair segments by transforming each hair segment into the candidate space, and computing the surface area of the transformed segment's bounding boxes. We then select the candidate space with the smallest sum of surface areas.

If most of the hair segments are oriented into a similar direction, picking a single candidate segment will very likely already yield a suitable candidate space. To reduce the likelihood of selecting a misaligned candidate hair segment we pick 4 random candidates.

3.3.2. Partitioning Strategies

To construct the tree we use a top-down surface area heuristic (SAH) based approach that automatically selects the proper node type based on the lowest cost. In particular, the cost function we use has different cost factors for both AABB and OBB splits to properly reflect the higher cost of traversing OBB nodes.

At any time during the construction process, we first compute a suitable hair space for the given hair segments as outlined previously, and then compute the expected SAH cost for the following five techniques:

i Traditional AABB object partitioning in world space

ii *Spatial splits* in world space

iii Object partitioning in hair space (producing OBBs)

iv Spatial splits in hair space (producing OBBs)

v A *similar orientation clustering* step that partitions a set of hair segments into two distinct clusters with different dominant orientation.

Object Partitioning. For the partitioning strategies ('i' and 'iii") we use a binning approach using 16 "bins" as described in [Wal07] to compute the SAH cost using the proper cost factors for AABBs vs. OBBs. Binning is performed in world space and in hair space and bounding boxes are always calculated in the space the partitioning happens.

Spatial Splits. For the spatial splits ('ii' and 'iv') we use an approach similar to Stich et al. [SFD09], except that we did not implement their "un-splitting" optimization. We again use 16 "bins" (meaning 15 candidate split planes), and again perform spatial split testing in both world and the same hair space as used for object partitioning.

Spatial splits improve performance, but also produce a large number of additional nodes and primitive references, potentially leading to excessive memory consumption. To control memory consumption we use a predetermined threshold value that tells the builder how many additional references it

is allowed to produce; once this threshold is reached spatial splits become automatically disabled.

To prevent a single "bad" subtree from consuming all the limited number of spatial splits (and thus, having none left for other regions of the scene) we use a breadth-first builder higher up in the tree. However, since depth-first construction generally results in better locality of the generated nodes and primitive references we eventually switch to depth-first construction as soon as a subtree's primitive count falls below a given value (currently 128).

Similar orientation clustering is an additional technique (strategy 'v') we developed to better handle cases where two or more differently-oriented strands intersect with each other—which happens in particular for curly hair. In this case, both hair strands by themselves have local orientation similarity and could be well handled by OBBs, but neither spatial splits nor binned object partitioning would be able to partition these strands.

To handle such cases we first pick a random hair segment, and then determine the segment that is *most misaligned* to this initial hair (i.e., whose orientation spans the widest angle). We then iterate over all other hair segments, and cluster them into two distinct clusters depending on which of those two hairs they are most aligned with.

As the orientation of the separated hair strands can differ much, we use the OBBs calculated in separate hair spaces of each of the two strands to calculate the SAH cost. This way the bounds used to calculate the SAH are well aligned with each of the two strands. This is different to the object partitioning and spatial splitting approach, that always calculate bounds in the space the binning operations were performed.

3.3.3. Tree Construction

Using these 5 strategies we could easily build a binary tree in a top-down manner. To actually build a 4-wide BVH we proceed as follows: We start by putting all hair segments into a single set, and partition that set using the strategy (i)-(v) with lowest SAH cost into two sets. We then iteratively pick the set with largest surface area and split it until a total of four such sets are obtained.

If any of these four sets was created with an OBB split we create a quad-node of OBBs (even for those nodes that did not require any) and store the OBB calculated in the hair spaces of each of these sets. Else we create a quad-node that stores world space AABBs. We then proceed with all non-leaf nodes in an obvious manner until the entire tree is constructed.

3.4. SIMD-Enabled Traversal

Given the combination of very high geometric detail and likely incoherence of the ray distributions we decided not to pursue any packetization, and instead opted for a pure *single-ray SIMD* approach.

Our single ray traversal algorithm is heavily inspired by the corresponding single-ray BVH4 kernel found in Embree [WWB*14], except that the leaves contain Bézier segments, and in that some of our nodes are OBB nodes.

AABB and OBB tests. Given that the majority of traversal steps will operate on AABB nodes (as they are located at the top of the tree) we first test if a node is indeed an AABB node (checking some bits in its node reference), and if so, use exactly Embree's code for this case. If not, we test if the node is a leaf or an OBB node; in the latter case we use the respective node's four affine transformation matrices—already stored in SIMD-friendly structure-of-array layout—and use SIMD instructions to transform the ray's origin and direction into the respective four spaces (in parallel).

Since the transformation changes the ray's direction we can no longer use precomputed reciprocal values for the ray direction; however, computing the four inverse directions again maps to straightforward and very efficient vector code. Once the ray is in the proper space, intersection with the unit box is trivial, and is actually somewhat cheaper than intersecting with a non-unit box. Altogether, performing an OBB test in our implementation is roughly 50% more expensive than an AABB node test.

Mailboxing. When using spatial splits, the resulting replication of references can lead to the same primitive being encountered multiple times during traversal. For highly optimized triangle tests this is often ok, and the cost and complexity of potentially adding mailboxing often outweighs its savings. However, even with our vectorized intersection test from Section 2, a hair segment intersection is significantly more expensive than a ray-triangle test.

To avoid these costly intersection tests we use a simple mailboxing technique that tracks the IDs of the last 8 primitives encountered during traversal; we obviously use an 8-wide vector intrinsic to test if this list contains a primitive in question, and simply skip it if this is the case.

3.5. OBB Node Quantization

Production-rendering models are generally large, and while our method has significantly lower memory consumption compared to tessellating the model the amount of memory required can still be high, in particular when using spatial splits. To reduce memory consumption we extended our technique to (optionally) store the OBB nodes in a compressed form using *quantization* of OBB coordinates.

In difference to normal OBB nodes, compressed OBB nodes require a common orientation for all 4 OBB bounds to enable efficient compression. This common orientation is stored as an orthonormal linear transformation that gets

scaled with 127 and quantized using *signed chars*. In this common quantized space we store 4 AABB for the 4 children, each quantized using *unsigned chars* relative to the merged bounding box of all children. For decompression we store the lower bounds and extend of the merged bounding box as floating point values.

Each quad-OBB node requires only 96 bytes, which is less than half that of an uncompressed OBB node, and even 25% less than a regular AABB node. The common orientation requires some changes to the construction algorithm, in particular to use the same hair space for all splits of a single node. We only use quantization for the OBB nodes: for those the savings are greater, and since they are traversed less often the decoding overhead is low. The traversal code itself hardly changes at all—except for the node de-quantization, which in SIMD is straightforward, and cheap (Section 4.3).

4. Results

With all components of our method together we can now evaluate its performance. In particular, we are interested in how our method compares to the two alternative approaches an existing production renderer would have had access to without our method: tessellation into triangles on one side, and a traditional axis aligned BVH over hair segments on the other.

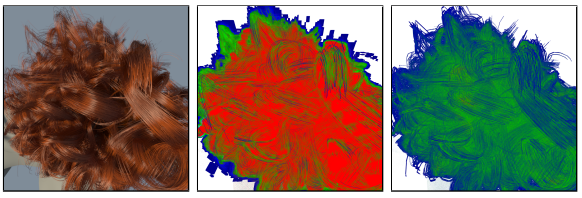
4.1. Comparison Methodology

For all our experiments we use actual movie-content characters graciously provided for testing by DreamWorks Animation: *Tighten* from *"Megamind"*, *Tiger* from *"The Croods"*, and *Sophie* and *Yeti* from *"Rise of the Guardians"* (Figures 1 and 4). To render these images and for all evaluations, we wrote a path tracer that uses the hair scattering model and importance sampling described by Ou et al. [OXKP12]. All of the non-hair base geometry uses triangle meshes with a simple lambertian diffuse material. The renderer also supports texturing, which is used to control the surface diffuse and hair internal color in Figures 1 and 4b).

All tests were run on a workstation with 64 GBs of memory and two Intel® E5-2697 CPUs (12 cores at 2.7 GHz each), using 1 sample path (up to 10 bounces) per pixel per frame, at a resolution of 1024 × 1024. All performance results are given in frames per second and measure total render time including shading, texturing and the surface and hair scattering model (R, TRT and TT lobes [OXKP12]). Our data structures (and all reference codes) have been integrated into a modified version of Embree. All experiments are run on Linux, using Intel® Compiler version 14.0.2.

As a reference solution for an axis-aligned BVH built over Bézier curves we employ Embree's high-quality BVH4 and use our Bézier curve intersector for primitive intersection. For spatial splits, we use a threshold that allows at most 2× as many primitive references to be generated than the number of hair segments in the scene.

For triangular geometry we use Embree's default kernels. With both our technique and with the reference curve implementation the triangles are handled in a separate acceleration structure; for the tessellation reference all triangles—both tessellated hair and base mesh—end up in a single BVH.



(a) Original View (b) Intersection Tests AABB BVH (c) Intersection Tests AABB / OBB BVH

Figure 5: *The number of ray-hair segment intersection tests per pixel is shown for a zoomed-in region of the Tighten model* (a)*, for a traditional axis-aligned BVH4* (b)*, and our hybrid AABB / OBB BVH* (c)*. Blue indicates 0 to 10 tests, green 10 to 40, yellow 40 to 70, and red greater than 70.*

4.2. Culling Efficiency

The key rationale of our technique is its ability to better adapt to long, thin, and diagonal geometry to reduce the number of costly ray-hair segment intersections. To quantify the degree to which this is the case we have annotated our code and measured both traversal tests and hair segment intersections for a closeup of the Tighten model. For this configuration, Figure 5 visualizes the number of ray-hair segment intersections performed per pixel. For some areas of the model (where hair is diagonally oriented) the reference axis-aligned BVH4 has to perform a large number of hair segment intersections, while our approach performs well for all parts of the model. Our approach traverses on average only 110 nodes and intersects 5.7 hair segments per ray, while the BVH4 traverses on average 174 nodes (1.5× more) and intersects 47.2 hair segments (8.2× more). Thus our approach achieves more than an 8× reduction in hair segment intersection tests by performing even less traversal steps.

4.3. Memory Consumption vs. Performance

Though overall fully automatic, our method has two knobs that allow for trading off performance vs. memory consumption: Spatial splits generally lead to better culling, but require more memory; and OBB quantization saves memory, but comes with some run-time overhead. To quantify these effects, we have run both combinations, and measured their respective performance and (total) memory consumption.

According to Table 1, spatial splits are always beneficial to rendering performance, and can yield speedups of over 30% (for the Tiger) at the expense of an 2.5x slower build. Even with our memory usage threshold in some cases spatial splits

almost double total memory consumption. OBB compression can somewhat counter that, and reduces memory consumption by about 30%. In the case of the *Yeti* model, using compressed OBB nodes is actually *fastest*, likely due to improved cache behavior.

spatial splits:		no	no	yes	yes
compression:		no	yes	no	yes
Tighten	perf	6.6fps	6.2fps	7.5fps	7.3fps
	mem	387MB	267MB	633MB	404MB
	build	6.4s	5.3s	15s	13.7s
Tiger	perf	2.1fps	1.8fps	2.7fps	2.5fps
	mem	1.1GB	761MB	1.8GB	1.1GB
	build	19.7s	16.3s	41.0s	38.6s
Sophie	perf	7.1fps	7fps	7.3fps	7.1fps
	mem	2.1GB	1.4GB	3.3GB	2.7GB
	build	42.4s	33.9s	64.8s	59.3s
Yeti	perf	2.6fps	2.6fps	3.1fps	3.2fps
	mem	21.7GB	17.7GB	34.4GB	24.9GB
	build	393.7s	373.7s	880.6s	834.0s

Table 1: *The impact of spatial splits and node compression on memory consumption and performance (in frames per second) for our hybrid AABB / OBB BVH.*

Tuning these parameters allows an animator some control of memory consumption vs. render time, and it is likely that different applications would pick different configurations. However, to simplify the evaluations done in the remainder of this paper we will from now on assume a "standard" configuration that we believe makes a good trade-off between performance and memory consumption, and assume that spatial splits and OBB compression are both enabled.

4.4. Comparison to Tessellation Reference

To compare against tessellation we tessellate each hair segment into 8 triangles. Even with this extremely low tessellation rate, tessellation requires $2-3\times$ more memory than our technique (Table 2), which causes the *Yeti* model to exceeded available memory when tessellated. In terms of performance, all models perform about $2\times$ faster with our technique than with tessellation. Even though the BVH of the tesselated version is build over 8x as many primitives, its build times are about 5x faster than ours. This is mainly due to the default Embree builder being very optimized and our approach tests multiple heuristics, of which unaligned partitionings and spatial splits are quite expensive.

4.5. Comparison to Curve Reference

Table 2 also allows for comparing our method to the reference implementation using an axis-aligned BVH4 over hair segments (see Section 4.1), which we consider to be the state

		reference BVH4 triangles		ours	reference BVH4 hair segments	
Tighten	perf	3.5fps	0.48x	**7.3fps**	3.7fps	0.51x
	mem	1.1GB	2.72x	404MB	289MB	0.72x
	build	2.6s	0.19x	13.7s	0.28s	0.02x
Tiger	perf	1.44fps	0.58x	**2.5fps**	1.0fps	0.40x
	mem	3.5GB	3.18x	1.1GB	816MB	0.74x
	build	8.4s	0.21x	38.6s	1.1s	0.028x
Sophie	perf	4.2fps	0.59x	**7.1fps**	3.5fps	0.49x
	mem	6.8GB	2.51x	2.7GB	1.6GB	0.59x
	build	16.9s	0.28x	59.3s	2.0s	0.033x
Yeti	perf	—	—	**3.2fps**	1.8fps	0.56x
	mem	—	—	24.9GB	18.6GB	0.75x
	build	—	—	833s	75.7s	0.09x

Table 2: *Performance and memory consumption for our method (with spatial splits and compression enabled) versus both a reference BVH4 (using object binning) over tesselated hair segments, and a reference BVH4 over hair segments using our SIMD-optimized ray-hair segment intersection test.*

of the art for somebody building a production renderer for hair.

Compared to this reference our method requires $1.3-1.6\times$ more memory. This is not due to OBB nodes being bigger than AABB nodes—this is amply counteracted by the OBB compression—but as mentioned previously, by the fact that our method can better separate different hair segments and consequently produces more nodes—plus the additional node and reference count due to spatial splits. In particular, when used in its memory conservative mode—with spatial splits disabled and OBB compression enabled—our method actually needs slightly *less* memory than the reference implementation (Table 1). In terms of performance, our method handily outperforms the reference implementation for all models (even without spatial splits), and in the case of the Tiger is actually $2.5\times$ as fast.

Our build performance is about 10x - 50x slower compared to the curve reference, which uses the fast Embree BVH builders. This performance discrepancy comes from two sources: First, we test additional heuristics (such as unaligned partitionings and spatial splits), and second we did not yet fully optimize our implementation. Some additional optimizations, like fully parallelizing our build procedure, and enabling expensive splitting strategies, such as non-axis aligned partitionings and spatial splits, only if cheap axis aligned object partitioning was not successfull will improve performance of our build procedure.

For all models we spend about 50% of the total rendering time tracing rays through hair geometry: Sophie 47.9%, Tiger 62.5%, Tighten 50.3% and Yeti 48.7%. The remaining time is spent in ray generation, ray tracing triangular geometry, and shading. Consequently, the speedup of our method

when measuring the pure ray tracing time for hair geometry is even higher.

4.6. Comparison to Multilayer Shadow Maps

We have also attempted to compare tracing shadow rays using our approach with epipolar ray-tracing of multilayer shadow maps, as described in Xie et al. [XTP07]. Memory considerations put aside and although the comparison is not straightforward, the general sense is that our approach compares very favorably in terms of ray traversal performance.

5. Conclusion

Our approach is designed to enable efficient ray traversal of production quality hair models with potentially millions of individual strands. To do so, we use a new spatial data structure which incorporates axis-aligned and oriented bounding boxes. The use of oriented bounding boxes enables long, parametrically defined hair segments to be tightly bounded, and minimizes overlap in the bounding boxes of neighboring hairs which improves ray culling efficiency. We pair this data structure with a vectorized ray-hair intersection kernel to achieve higher performance than ray tracing pre-tessellated models while using significantly less memory.

Though our method is intended for use in production rendering, more work is needed, notably support for motion blur. In addition, it is unclear how best to scale the 8-wide single-ray SIMD intersection kernel described in Section 2 to 16-wide SIMD as is available in the Xeon Phi architecture. Doubling the line segments per hair segment to 16, may not substantially improve image quality in all cases. Finally, we anticipate that total rendering performance could be improved by modifying our data structure to allow both triangle and hair geometry to be handled in a single hierarchy.

Elements of this work are potentially useful beyond hair rendering. For example, the *hybrid* AABB / OBB data structure we described may be applicable to other rendering workloads in which the geometric primitives are not otherwise well bounded by axis-aligned spatial partitions, and which exhibit local orientation similarity (e.g. stream lines in scientific visualization applications).

Acknowledgments

We would like to thank DreamWorks Animation for providing the models in Figure 4 and 1 and for supporting this work. We also want to thank Harrison McKenzie Chapter for early discussions, and Sebastian Fernandez for help in exporting the models.

References

[AK89] ARVO J., KIRK D.: A Survey of Ray Tracing Acceleration Techniques. In *An Introduction to Ray Tracing*, Glassner A. S., (Ed.). Academic Press, San Diego, CA, 1989.

[EG07] ERNST M., GREINER G.: Early Split Clipping for Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (2007), IEEE Computer Society, pp. 73–78.

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proceedings of Computer Graphics and Interactive Techniques* (1996), SIGGRAPH '96, ACM, pp. 171–180.

[LAM01] LEXT J., AKENINE-MÖLLER T.: Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics Short Presentations* (2001), pp. 311–318.

[NO02] NAKAMARU K., OHNO Y.: Ray Tracing for Curves Primitive. In *Proceedings of Winter School of Computer Graphics (WSCG)* (2002), pp. 311–316.

[NSK90] NISHITA T., SEDERBERG T. W., KAKIMOTO M.: Ray Tracing Trimmed Rational Surface Patches. *Computer Graphics (Proceedings of SIGGRAPH '90)* (1990), 337–345.

[OXKP12] OU J., XIE F., KRISHNAMACHARI P., PELLACINI F.: ISHair: Importance Sampling for Hair Scattering. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 1537–1545.

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 15–22.

[QCH*13] QIN H., CHAI M., HOU Q., REN Z., ZHOU K.: Cone Tracing for Furry Object Rendering. *IEEE Transactions on Visualization and Computer Graphics* (2013).

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 7–13.

[SN90] SEDERBERG T. W., NISHITA T.: Curve Intersection using Bezier Clipping. *Computer-Aided Design 22*, 9 (1990), 538–549.

[Wal07] WALD I.: On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 33–40.

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree–A Ray Tracing Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2014). (to appear).

[XTP07] XIE F., TABELLION E., PEARCE A.: Soft Shadows by Ray Tracing Multilayer Transparent Shadow Maps. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGSR'07, Eurographics Association, pp. 265–276.