Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types

I WALD[1] N MORRICAL[1,3] S ZELLMANN[4] L MA[2] W USHER[3] T HUANG[2] V PASCUCCI[3], [1]NVIDIA [2]Peking University [3]University of Utah [4]University of Cologne

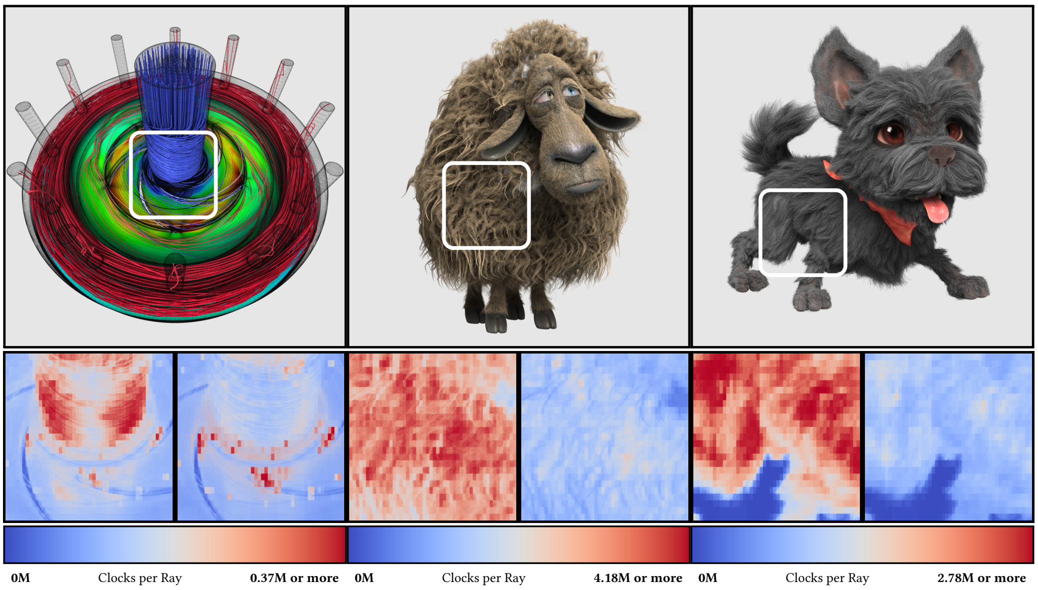


Fig. 1. Three of the models we used for evaluating our method: Left) `SciVis2011` contest data set (334.96K rounded cylinders via Quilez-style "capsules", plus 315.88K triangles). Middle,Right) hair/fur on the Blender Foundation `franck` and `autumn` models (2.4M and 3.4M "phantom" curve segments, plus 249.62K and 904.40K triangles, respectively). For these three models, our method leverages hardware ray transforms to realize a hardware-accelerated OBB culling test, achieving speedup of 1.3×, 2.0×, and 2.1×, respectively, over a traditional (but also hardware-accelerated) AABB-based BVH (both methods use exactly the same primitive intersection codes). Bottom: heat map of number of intersection program evaluations for the two methods, respectively.

Abstract. With the recent addition of hardware ray tracing capabilities, GPUs have become incredibly efficient at ray tracing both triangular geometry, and instances thereof. However, the bounding volume hierarchies that current ray tracing hardware relies on are known to struggle with long, thin primitives like cylinders and curves, because the axis-aligned bounding boxes that these hierarchies rely on cannot tightly bound such primitives. In this paper, we evaluate the use of RTX ray tracing capabilities to accelerate these primitives by tricking the GPU's instancing units into executing a hardware-accelerated oriented bounding box (OBB) rejection test before calling the user's intersection program. We show that this can be done with minimal changes to the intersection programs and demonstrate speedups of up to 5.9× on a variety of data sets.

1 INTRODUCTION

Modern ray tracers rely on bounding volume hierarchies (BVHs) for fast ray traversal performance. Generally speaking, BVHs have been proven to be very efficient solutions, and on RTX-enabled GPUs are even available with hardware acceleration support; however, the axis-aligned bounding boxes (AABBs) used by traditional BVHs are known to struggle when dealing with long, thin, and in particular, diagonally oriented primitives such as cylinders, curves, certain glyphs, or hair. This is because a primitive's likelihood of being intersected by a random ray is proportional to the surface area of its bounding volume, and for BVHs limited to AABBs, these bounding volumes become large—leading to a large number of ray-primitive intersections per ray that a tighter bounding volume could have avoided.

A common solution to this problem is to use more general BVHs with *oriented* bounding boxes (OBBs), but these are harder to build and traverse, and consequently not supported by today's ray tracing frameworks such as OptiX, DXR, or Vulkan RT.

In this paper, we propose and investigate a technique that "tricks" the AABB-based BVH traversal hardware on Turing into first doing a hardware-accelerated oriented-bounding box (OBB) rejection test before switching to software execution and calling the expensive user intersection program. We do this by "masquerading" the user's world-space intersection program(s) in a way that the hardware first executes an instance transform and an object space bounding box test (which together effectively executes an oriented bounding box (OBB) test), while then un-doing the instance transform and calling the original intersection program. This allows for performing a hardware-accelerated OBB test without the user's program even having to be aware of it, and therefore works for arbitrary primitive types.

We demonstrate that for a variety of different real-world models this technique results in speedups of up to 5.9× over the (also hardware-accelerated) reference BVH, with otherwise identical intersection programs.

2 RELATED WORK

Ray Tracing and RTX. While different acceleration structures and traversal methods for ray tracing are still an active area of research, most ray tracing-based software today uses some sort of ray tracing middleware API, such as Embree for CPUs [20], or Vulkan [8], DXR [10], and OptiX [11, 14] for GPUs.

Of particular interest to this paper is the Turing architecture, whose RTX capabilities allow for hardware-accelerated ray tracing, and in particular hardware-accelerated BVH traversal as well as instance transformations. For more details on the Turing architecture, we refer to recent descriptions from NVIDIA [7, 13]. For our implementation we use the Turing hardware RTX capabilities through OptiX [14] (more specifically, through OWL [19] and OptiX 7 [11]), but observe that the exact same ideas would also work in Vulkan Ray Tracing (VKR) [8] and DirectX Ray Tracing (DXR) [10], or any other API that exposed hardware instancing and user-code intersection programs.

Cylinders, Curves, and Hair. This paper is about accelerating the ray tracing of predominantly long and thin primitives such as cylinders, rounded cone stumps, curves, ribbons, etc. In scientific visualization such shapes often appear in the form of glyphs or stream lines. The latter can be realized effectively by linearly connecting a set of control points through rounded and/or varying-radius cylinder segments (see, e.g., Han et al. [5]).

In photo-realistic rendering, the most common occurrence for long, thin, primitives are hair and fur, which are typically rendered as either linear, quadratic, or cubic curve segments. For linear (but still varying-radius) segments ray-primitive intersectors have been proposed, for example, by Han et al. [5] and Quilez [15], who called those primitives *links* and *capsules*, respectively. Various ray-curve intersectors have been proposed, for example, by Nakamaru et al. [12], Woop et al. [20, 21], Binder et al. [1], and Reshetov et al. [16, 17]. We explicitly emphasize that our method is completely orthogonal to these ray-primitive intersection techniques: it is *not* another ray-curve/hair intersector, but a general acceleration method applicable to *any* such primitive type.

Oriented Bounding Volumes. Irrespective of whether we are dealing with cylinders, rounded cone stumps, or cubic segments, these primitive types all suffer from the same issue; namely, that the axis-aligned bounding boxes that modern ray tracers rely on often fail to tightly bound these shapes, leading to a large number of costly ray-primitive intersections that mostly result in misses. For collision detection, Gottschalk et al.[4] proposed the use of oriented bounding boxes (OBBs) as early as 1996. In ray tracing, Woop et al. [21] proposed a technique that uses a top-down BVH builder that aims at finding groups of primitives with similar orientation, and when found, bound those more tightly with special oriented bounding box nodes that they added for that purpose. Woop et al. reported speedups of up to 2.5× over the best non-OBB BVH. However, their technique requires modifications to the underlying ray traversal code and BVH layout, and relies on a non-trivial OBB builder, while its final speedup is limited due to the expensive OBB tests themselves. In contrast, we avoid the hierarchy construction problem by executing these OBB tests only at the primitive level, and exploit the hardware's ray transform units to essentially do the OBB test for free.

3 REALIZING OBB TESTS USING INSTANCING HARDWARE

This paper is based on five fundamental observations:

Observation 1: The problem with thin, diagonal geometry stems primarily from the large surface area of the enclosing AABBs, leading to a large number of ray-primitive intersection tests. These tests will most likely fail early, but are still costly.

Observation 2: While Observation 1 is true in general, this issue is even more problematic in the context of hardware ray tracing, where every call to a user-code intersection program requires the hardware to interrupt traversal and do a context switch to invoke this intersection program in the general compute cores. These context switches are expensive, in particular compared to the cost of hardware-accelerated operations such as BVH traversal or instance transforms. Thus, regardless of how cheap the intersection program itself may be, a large number of (largely missing) intersection program invocations will carry a huge cost.

Observation 3: More tightly fitting bounding objects would significantly reduce such failed tests. At least for long and thin primitives, either oriented bounding boxes (OBBs), bounding cylinders, etc., could significantly reduce the number of final ray-primitive intersections.

Observation 4: Observation 3 *is* true, but in practice matters only if this culling test is hardware-accelerated as well. Though OBB (or cylinder) "early-out" tests can indeed reduce the number of final primitive tests, without hardware support the only way to implement them is to do such a test in an intersection program—but that merely trades one intersection program for another one; if the real primitive test is very expensive (it usually is not!) such a software culling test may still save *something*, but not much. Specifically, the number of times the GPU has to switch from hardware traversal to software intersection remains unchanged, and the root cause of the problem (Observations 1 and 2) remains unsolved.

Observation 5: The ray tracing hardware units on Turing can already do *most* of what a hardware OBB test would require: The canonical way of performing an OBB test is to inversely transform the ray to a space where that OBB maps to the unit box, then intersect that transformed ray with that box, and only consider the OBB's content if the transformed ray hit that box. Turing does not support OBBs; but it does support instancing. Each time a ray traverses an instance, it is transformed using that instance's affine transform, and traverses the child BVH if the transformed ray hits the child BVH's root node. Arguably, the fundamental insight of this paper is that though not exactly the same, these operations are functionally similar enough to allow us—with some additional techniques we describe below—to *reduce* the problem of a hardware-accelerated OBB test to a combination of instance transforms and a suitably designed intersection program.

With these observations, the core idea of this paper is to use the hardware units that are already there—in particular, the instancing hardware and the flexibility to call arbitrary shader programs—to realize what is essentially a hardware-accelerated OBB test for arbitrary primitive intersectors.

3.1 Instance Transforms to Reduce Surface Area: Proof of Concept

Before looking into the core problem addressed in this paper—namely, how to trick instancing hardware to do OBB tests for *arbitrary* primitives—let us first look at an intentionally simplified proof of concept. For now, let us consider the simple case where all N primitives are regular cylinders with a fixed radius, and without rounded caps. In this case, each cylinder can be represented in one of two ways: either through its intrinsic parameters (e.g., center, orientation, length, and radius), or alternatively through an affine transform of a common unit cylinder. In practice, very few primitives can be represented with this affine transform approach (in particular, neither Han's *links*, nor Quilez' *capsules*, nor any curve representations can) but for now, this approach is easier to think about.

Now, assuming an input model is made up of N such cylinder primitives, there are similarly two different ways of representing those primitives in a hardware-accelerated BVH. First, in the most general approach, one can represent all primitives through their respective AABBs. These AABBs would be placed into a user-geometry "bottom-level acceleration structure" (BLAS), and would use an intersection program that performs a world-space ray-primitive intersection for any primitive whose bounding box the ray encounters.

In addition, *if* all primitives can be represented as affine copies of a base primitive, then an alternative way of representing the same scene would be to instead create N instances of a single unit cylinder object. We would first create a single BLAS with a single unit cylinder, then for each cylinder in the scene, we would create an instance with the given affine transform. We would then build a second-level "top-level acceleration structure" (TLAS) over those instances. The intersection program and shading code would necessarily have to be aware of the instance transforms, but otherwise these scenes would be identical.

Impact of GPU Hardware acceleration for the two methods. For this hypothetical scenario, let us now look at how a GPU with hardware ray tracing would actually behave in each of those two cases (also see Figure 2): For the canonical solution, each time a ray traversing a BLAS hits a cylinder's AABB, the ray has to interrupt hardware traversal and call the software intersection program. For long and thin primitives, this will result in many costly context switches.

For the instance-based method, there would be exactly one instance per primitive, and since the top-level acceleration structure (TLAS) also uses AABBs, we would, in fact, end up with almost exactly the same AABBs in the TLAS that the reference method had in the BLAS[1]. Seen this way, we have not *reduced* the number of primitives that get traversed. Rather, we have *shifted* the traversal of these AABBs from the BLAS to the TLAS. However, there is one critical difference in what the hardware does when an instanced primitive is hit. In the instance BVH, even if a ray does hit one of the large AABBs in the TLAS, traversal remains in hardware. The ray then first undergoes an instance transform, starts traversal of the single-primitive BLAS, then intersects the root node of that single-primitive BLAS, and *only* switches to the software intersection if the ray actually intersects this BLAS's root bounding box. Specifically, that root bounding box of the BLAS is specified in *local* space, and will typically be much tighter than the primitive's AABB in world space—effectively acting as an oriented bounding box (OBB) in world space.

[1]in fact, they are slightly worse, since the AABB of a transformed AABB is usually slightly bigger than the best AABB for the transformed primitive

In summary, it is important to realize that this instance-based variant would in some sense actually do *more* work than the canonical method. However, all of the work up to and including the ray-intersection test with the (local-space) root bounding box would be done entirely in hardware, while the costly user intersection program would be called only if the ray did hit what is exactly the equivalent of a OBB test in the given instance transform space. A graphical sketch of this (intentionally simplified) case is also given in Figure 2.

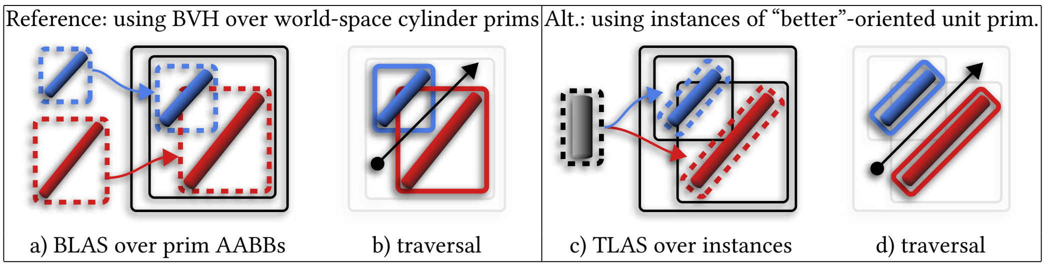

Reference: using BVH over world-space cylinder prims
Alt.: using instances of "better"-oriented unit prim.
a) BLAS over prim AABBs
b) traversal
c) TLAS over instances
d) traversal

Fig. 2. Conceptual illustration of using instance transforms to reduce number of primitive intersections, in this simplification using cylinders that can be handled via affine copies of a unit cylinder (we will relax this later on). a+b) When building a bottom-level acceleration structure (BLAS) over the world-space primitives (a), the AABBs of long, thin, and diagonally oriented cylinders become large. Rays traversing this (b) are likely to hit these AABBs, and trigger lots of intersection program calls, many of which miss, but all of which will interrupt hardware traversal. c+d) Replacing the primitives with instances of a "better oriented" unit primitive results in a top-level acceleration structure (TLAS) whose leaf nodes (black) look almost the same as in a), but the root nodes of the BLAS's they are instantiating (dotted red and blue rectangles in c) are now the equivalent of OBBs. d) Rays traversing c will still hit the instances in the TLAS, but will remain in hardware traversal through the instance transform successive BLAS traversal, and will leave hardware traversal only if the transformed ray did hit the much tighter BLAS root node (ie, if the world ray did hit the OBB). c+d do more total traversal steps, but all in hardware; while calling significantly fewer intersection programs.

Quantifying the potential speedup. The caveat of the proof-of-concept we just made is that in the way it is described, our method will only work for N affine copies of a single base primitive, which as argued above has limited applications. However, before describing how we can generalize this to non-affine shapes, we can already use this simplified case to quantify just how big the performance difference between these two modes could end up being.

To do so, we created artificial scenes containing bundles of N parallel cylinders of length 1 and radius r, using a bundle radius of $R = 0.5$; these bundles are then rotated in the X-Y plane to form an angle α to the Y axis. By controlling the parameters r, N, and α we can now simulate models with varying degrees of how thin (r), dense (N), and non-axis aligned (α) the primitives in that model are to be. The result of various configurations of this experiment (using the framework described later on) is given in Figure 3 and Figure 4. As expected, we observe that, the thinner and more diagonal the primitives, the higher the speedup, with speedups reaching up to 3.8×.

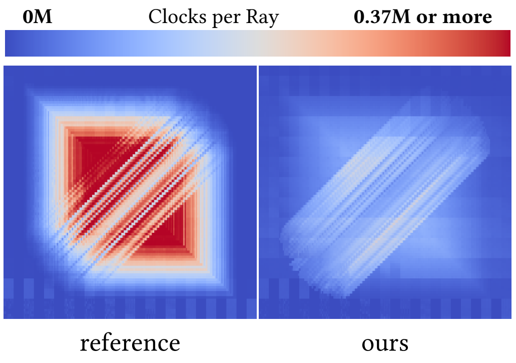
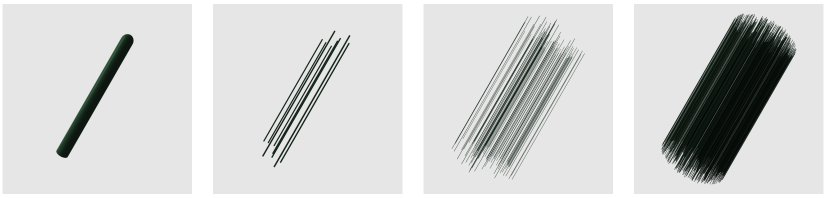


Fig. 3. Heat map showing number of clocks required per ray, for the $N = 1000, r = 0.002$, tilt $= 45^\circ$ models from Figure 4.



	N=1,r=0.1	N=10,r=0.01	N=100,r=0.002	N=1000,r=0.002
tilt=0°	**1.05×**	**1.03×**	**1.04×**	**1.04×**
tilt=15°	**1.16×**	**1.16×**	**2.01×**	**3.83×**
tilt=30°	**1.10×**	**1.21×**	**1.91×**	**3.32×**
tilt=45°	**1.12×**	**1.22×**	**1.90×**	**3.20×**

Fig. 4. Relative performance of our method over the reference method (using the framework described in Section 4), for different configurations of our artificial test scene set-up.

3.2 Extension to Arbitrary Primitive Types

Though the speedups observed for our hypothetical set-up are encouraging, as described so far our method would necessarily be limited to affine copies of a base primitive, with obvious limitations. To make this method useful in practice we therefore need to extend it to handle arbitrary primitive types such as rounded cylinders, varying-radius cylinders, curves, etc.

To do this, we can draw on two core insights that can be gleaned from the previous section's hypothetical experiment: First, we observe that the majority of the speedups achieved by the instance-based variants is *not* because of savings in the intersection program itself, but instead, is due to the fact that the hardware traversal of the transformed ray in the BLAS led to this ray being tested with a tighter bounding box that avoided calling the intersection program in the first place. It is the effect of this specific test—the intersection test between the (transformed) ray and the (tight) root bounding box of the BLAS—that together forms what is effectively an OBB test.

Second, we observe that for the previous observation to hold true, it does not matter at all what exactly the intersection program computes within the space of that BLAS's single primitive bounding box. In particular, it does not matter if this program were to perform *different* computations for one instance of this primitive than it were to perform for another. This clearly is not how instancing usually works, but since the intersection program is completely within the user's control we can, after the ray has passed the test with the transformed box, make it do whatever we want. In particular, there is no rule that requires different calls to the intersection program to always compute the same code, nor even that this program has to use the object-space ray to do this.

Using these two insights means we can write an intersection program that *masquerades* primitives as instances of a "black box" that doesn't do anything other than calling back to the original, world-space intersection test from the reference method, testing against a different primitive based on which instance is currently being traversed. This masquerading program would compute the same result as the reference program, but unlike the reference program, would only get called if and when a ray has traversed the transformed unit box. An illustration of our method is given in Figure 5.

3.3 Implementation

As input to our method, we assume that there are N different input primitives, for example, Quilez's "capsules", Han's varying radius "links", Reshetov's "phantom" curves, etc. We emphasize that we do *not* make any assumptions about what shapes the primitives have; instead, all we require is that the user provides exactly two inputs: First, for each of those primitives, an oriented bounding box (specified through an affine transform of a unit bounding box) that is guaranteed to cover the primitive in world space; and second, a CUDA intersection program that, given an integer primitive ID, computes a (world-space) intersection with a ray and the specified primitive.

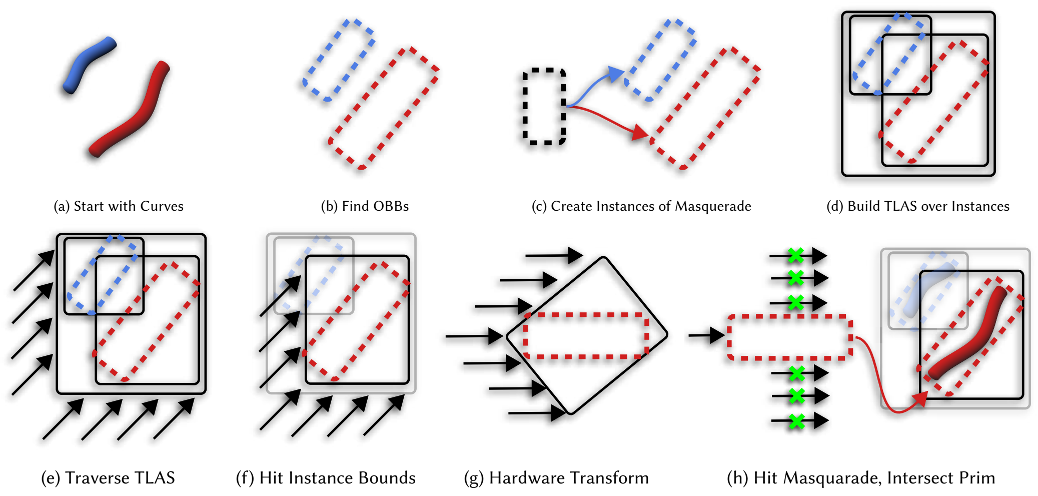

(a) Start with Curves
(b) Find OBBs
(c) Create Instances of Masquerade
(d) Build TLAS over Instances
(e) Traverse TLAS
(f) Hit Instance Bounds
(g) Hardware Transform
(h) Hit Masquarade, Intersect Prim

Fig. 5. Illustration of our complete method, including the masquerading to allow instance transforms to do OBB test for general, non-affine primitives that couldn't be handled by mere instantiation. Given a scene with arbitrary primitive types (a), we first compute oriented bounding boxes (OBBs) that contain these primitives (b). We then create a BLAS with a single, unit-box "masquerading" primitive (dotted black), and create one instance of that for each OBB, such that that masquerading program maps to each OBB (c). The resulting TLAS (d) has large boxes for the instance's AABBs (black), but the instanced masquerading primitive's bounds only cover the tighter OBBs (red and blue). Rays traced into this scene will traverse the TLAS with hardware support (e), and when they hit the (big) instance AABBs (f) will first get transformed and traversed through the BLAS (g), and will only call the masquerading program if they do hit the OBB. The masquerading program undoes the transform (h), and calls back to the original primitive behind the masquerade, with a possibly different program for each masquerade, and without this primitive even knowing that it was masqueraded.

We then create a single bottom-level acceleration structure over a single primitive with a unit bounding box to create our masquerading primitive. The intersection program for this masquerading primitive first queries the object's *instance* ID, then calls the user-provided intersection routine, passing it the instance ID as input for the primitive ID, along with the world-space ray to intersect with (see Figure 6).

We observe that asking the user to provide exactly one OBB per primitive is similar in spirit to how APIs such as Embree or OptiX 6 have in the past handled *bounding box programs* for user geometry, with the sole difference that we expect an OBB, while those APIs expect AABBs. This suggests a simple API where the user provides both intersection program and a bounding program that returns an OBB.

The method as described uses instances and intersection programs in a way that is clearly different from how it is intended, but nevertheless is necessarily correct: If the original OBB bounded the primitive, then the world-space transform of our masquerading BLAS's root box will also bound the world-space primitive (i.e., any ray that would intersect the world-space OBB that the user provided is guaranteed to also intersect our masquerading primitive); and since we use exactly the same intersection program, the outcome is necessarily the same. This observation also leads to another way of visualizing *how* our method works: essentially, we are not instantiating the primitives, but instantiating an OBB rejection test.

Similarly, though the instance transform is necessarily restricted to affine transforms, we do not actually use that transformed ray in the intersection program (the masquerading program

```
// actual intersection code
__device__ void intersectPrim
           (const Prim &prim, const Ray &ray)
{
  if (...)  ... optixReportIntersection(t_hit);
}

// reference method
__device__ void intersect_canonical()
{
  // Use *object* space ray and *primitive* index
  Ray     ray     = Ray(optixGetObjectRayOrigin(),...);
  int     primIdx = optixGetPrimitiveIndex();
  Model *model    = (Model *)optixGetSBTData();
  intersectPrim(model->prim[primIdx], ray);
}

//  masquerading method
__device__ void intersect_masquerading()
{
  // Use *world* space ray and *instance* index
  Ray     ray     = Ray(optixGetWorldRayOrigin(),...);
  int     primIdx = optixGetInstanceIndex();
  Model *model    = (Model *)optixGetSBTData();
  intersectPrim(model->prim[primIdx], ray);
}
```

Fig. 6. Intersection programs for canonical and masquerading methods. Our masquerading method can use exactly the same data and code, and just uses world-space ray and *instance* index instead of the object-space ray and primitive index.

un-did this transform before it was called), so different instances computing different shapes is perfectly valid. In fact, we could even use the same scheme to masquerade totally different *kinds* of primitives (e.g., one a cubic curve and another one a linear capsule) within the same TLAS. All our method requires for correctness is that every intersection that the user would expect from his/her intersection program is accurately bounded by the OBB he/she provided.

The key to this working is that the intersection program behind this single proxy primitive is fully user-programmable code, allowing us to undo what would usually happen in an instance, and exchange object space ray and primitive ID with the "real" world-space ray and primitive ID before calling the user's code. On the downside, this also means that this method only works for user intersection programs, not for diagonally oriented primitives that are hardware-accelerated.

4 PERFORMANCE EVALUATION

To evaluate our technique on real world models, we implemented our method on top of OptiX 7.0 [11]. For this evaluation we use models that consist of one set of curves (specified through control points and radii), and an optional triangle mesh. For the reference method all user primitives are put into a single BLAS, with a tiny TLAS combining the curve and hardware-accelerated triangle BLAS for the mesh geometry, where applicable. For our method, the TLAS contains both the masquerading instances and an optional instance for the triangle BLAS.

Irrespective of the method we use, all curve data is stored in a single linear buffer, a pointer of which is written into the respective curve geometry's Shader Binding Table (SBT) record; i.e., even though our method's BLAS only has a single entry it has access to the entire buffer containing all primitives. Both methods use exactly the same ray generation program and shading code, using a path tracer with Disney BRDF [6, 9], the implementation of which we obtained from Usher's ChameleonRT [18].

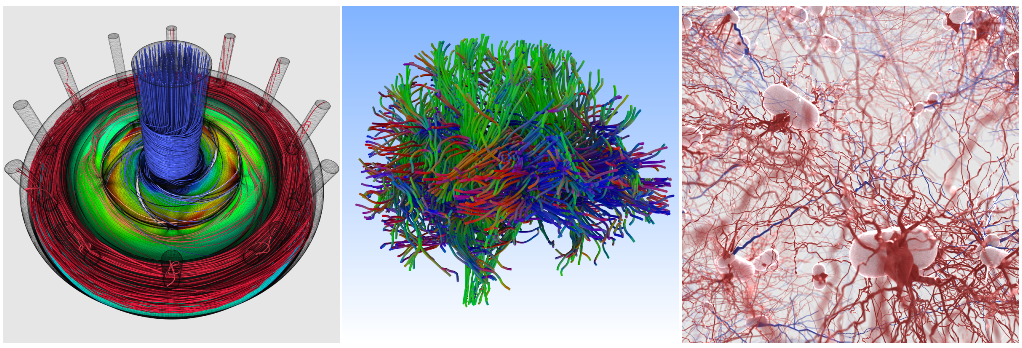
Evaluated Primitive Types. To evaluate different primitive types we implemented both Quilez's linear *capsule* intersector [15] and Reshetov's cubic *phantom* intersector [17]. For the *capsule* intersector we added a numerical accuracy fix that temporarily moves the ray origin closer to the model before intersection, but otherwise use a literal copy of Quilez' code from ShaderToy. For *phantom* we followed Reshetov's suggestion of using an early-out cylinder test (although our method should not need this, and should run slightly faster without), and added his suggested optimization of pre-splitting each segment into N sub-segments.

For the *capsule*s we compute OBBs that simply cover the two rounded caps, for the *phantom* curves we compute our OBB based on the way that Reshetov computes the bounding cylinder: We compute a central axis that connects first and last control point, and then widen this until the entire curve is covered. We observe that this OBB is correct, but for many real-world curves is by no means optimal—and that a tighter OBB would naturally result in even better results.

Data Sets. To capture a wide range of data sets we chose models from both scientific visualization and hair rendering in our evaluation. For the former, we use the `SciVis2011`, `DTI`, and `Neurons` data sets given in Figure 7; for hair data, we included two typical hair rendering research models (`single curl` and `curly hair`), along with two complete models from Blender Foundation movies: `autumn` from the movie "Spring" [2]; and `franck` from "Cosmos Laundromat" [3].

For the sci-vis data sets we always use first order primitives (Quilez' *capsule*s, Han's *link*s); for the hair models, we compare both the capsule intersector (connecting each pair of control points) and cubic *phantom* curves that each cover four control points. Representative images and the number of curves and triangles for the models are given in Figures 7 and 8.

Hardware Set-up. To evaluate this set-up we use a workstation with an Intel Xeon CPU (8 cores, 2.2 GHz), 128 GB of RAM, and two NVIDIA Quadro RTX 8000 GPUs (4608 cuda cores, 72 RT cores,



SciVis2011	DTI	Neurons
324K links + 158K tris	5.65K links	4.44M links
Quilez: 260M rays/second (1.25×)	Quilez: 226M rays/second (1.02×)	Quilez: 193M rays/second (1.02×)
Han: 167M rays/second (1.19×)	Han: 134M rays/second (0.89×)	Han: 185M rays/second (1.09×)

Fig. 7. Performance evaluation for three SCI-Vis data sets, each rendered at 800x800px and with 4SPP, using both the varying radius linear intersector from Han et al. [5] as well as a cheaper capsule intersector by Quilez [15]. While our method works for all three, only the `SciVis2011` data set sees any measurable speedups: for both `DTI` and `neurons` the capsules are so short relative to the chosen thickness that AABBs are just as good as OBBs, precluding any larger speedups.

and 48 GBs of GDDR6 VRAM each). As software environment we use Ubuntu Linux 18.04.3, OptiX 7.0, CUDA 10.1, and NVidia driver 440.44.

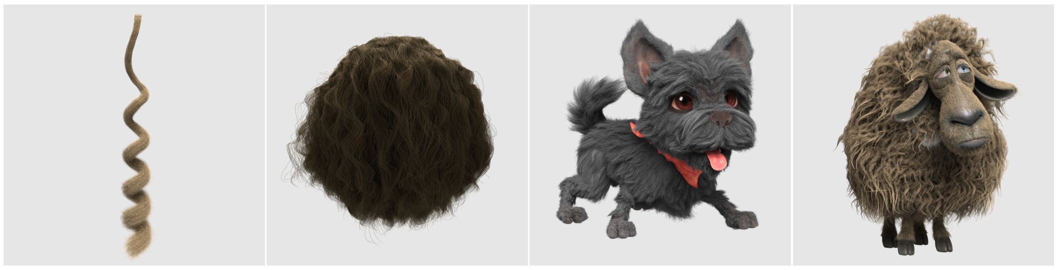


Fig. 8. The four hair/fur models we use for our evaluation, each rendered at 800x800px and with 4SPP. Since our method is completely orthogonal to the actual ray-primitive intersector we can render each of these models with either Quilez' *capsule* linear segments, or Reshetov's *phantom* curves.

	single curl			curly-hair			blender "autumn"			blender "franck"		
	ref	ours		ref	ours		ref	ours		ref	ours	
Quilez "Capsule" intersector (linear)												
	4.1M	15.8M	3.82×	15.5M	49.2M	3.17×	106.2M	209.6M	1.97×	46.2M	128.8M	2.79×
Han "Sphere-Truncated-Cone-Stump" intersector (linear)												
	4.2M	16.8M	4.03×	15.0M	48.9M	3.27×	87.1M	129.3M	1.48×	45.6M	111.9M	2.46×
Reshetov "Phantom" curve intersector (w/ various pre-subdivisions)												
1x	81.3K	141.2K	1.74×	1.0M	2.2M	2.06×	8.8M	52.3M	5.92×	2.4M	2.3M	0.99×
2x	343.0K	813.8K	2.37×	3.7M	10.0M	2.73×	26.6M	123.5M	4.65×	6.4M	10.1M	1.59×
4x	1.3M	4.4M	3.44×	12.2M	35.3M	2.88×	68.4M	144.3M	2.11×	18.7M	37.1M	1.98×
8x	5.0M	18.2M	3.65×	n/a	n/a		n/a	n/a		n/a	n/a	

Table 1. Performance (in rays per second, at four paths per pixel and at a resolution of 800^2) and relative speedup (between the reference method and ours) for the four hair/fur models shown in Figure 8. Irrespective of ray-primitive intersection method, and with otherwise un-modified rendering and intersection code, our method consistently outperforms the reference method (which also uses hardware BVH traversal, just without our OBB extension) by up to 3.6×. Note that `n/a` means that this configuration exceeded the maximum number of primitives respectively instances that OptiX allows in a single BVH.

4.1 Render Performance

Final render performance on the different models is given in Figure 7 for the sci-vis data sets, and Table 1 for the hair models. For the sci-vis models, we test against two intersector types: Quilez' capsule and Han's "truncated cone stump". These intersector types are more useful for our sci-vis models than a higher order intersector would be, as the underlying segments' vertex positions come from real world data. For these models, it is important that the segment's graphical representation reaches the underlying vertices to create an accurate visualization.

For these datasets, we largely see mixed results: though our method is never much worse, it does not achieve significant speedups, either. In retrospect, the segments for these datasets can be quite short, as a shorter distance between integration steps or between samples leads to a more accurate visualization. As a result, this significanty reduces the difference in surface area between the OBBs used for our masquerading technique and for the AABBs that the reference method uses

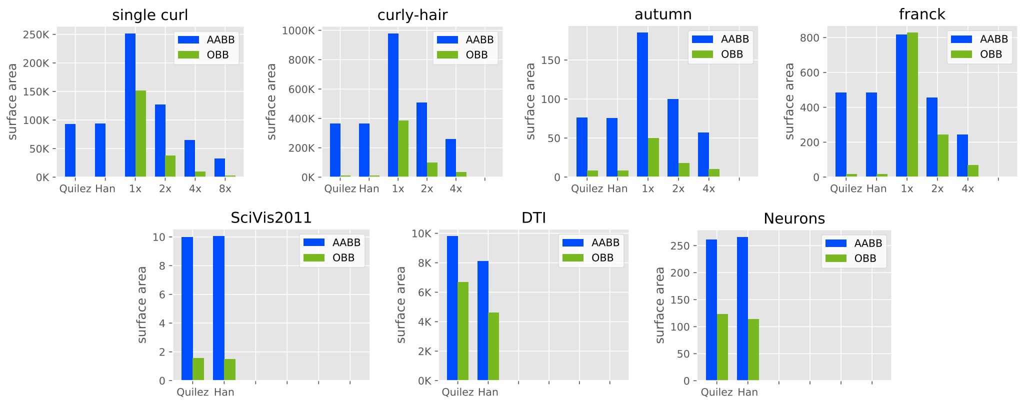


Fig. 9. Surface area of the bounding boxes that an AABB BVH computes vs. the surface area of the OBBs computed using our masquerading technique, for both Quilez capsules as well as Phantom curves with various splitting factors. We can see a significant reduction in surface area for the hair models, while the reduction for the SCI-Vis data sets is less severe. The difference in surface area roughly mirrors the speedup we observe (cf. Figure 7 and Table 8). As we don't use optimal OBBs for the Phantom curves, the reduction in surface area is more pronounced for the Quilez capsules than for the former.

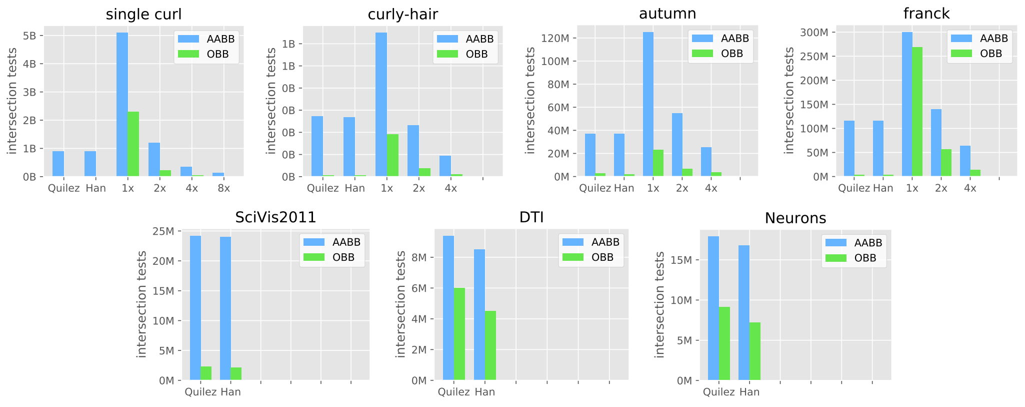


Fig. 10. Number of intersection tests performed during OptiX BVH traversal, both with the AABB-based reference implementation and with our OBB-based masquerading technique. We present those numbers for the Quilez capsule intersection test and—where applicable—for the Phantom curve intersection test. We can see that the number of intersection tests correlates with the observed speedup (cf. Figure 7 and Table 8) as well as the surface area of the bounding boxes (Figure 9).

(as shown in Figure 9). This lack in speedup is also partly due to the Quilez and Han intersectors being so cheap that only very large reductions in calls to these intersectors would make a significant difference. In fact, switching between the Quilez and Han intersectors, we do see some minor differences in performance. However, we mostly attribute these performance differences to slight variations in implementation and data layout.

Ultimately though, we found that for all three sci-vis models, since the connected linear segments are so short relative to their thickness, even without our method almost every intersection results in a hit. As a result, our method can only exclude a few failed ray-primitive intersections, especially for our DTI and Neurons datasets as shown in Figure 10.

For the hair models, in contrast, except for one outlier our method consistently outperforms the reference method, by 1.59 – 5.92×. One interesting observation is that when using the *phantom* intersector the speedup *increases* with the number of pre-subdivisions, in particular for very curly models such as `franck`. Our analysis regarding the surface area of the AABBs generated with the reference method and the OBBs with our masquerading technique (see Figure 9) indicates that this is due to the subdivision leading to more cylinder-like segments. Those can be more tightly enclosed by an OBB than a highly curved segment could (in particular for the chosen way of computing the bounding cylinder, see Section 4).

Nevertheless, speedups are significant across the board, in particular considering that both variants use exactly the same rendering code with the same number and distribution of rays traced, and with a non-trivial constant cost for shading in both methods.

4.2 Memory Usage

The speedups achieved by our method were higher than initially expected, but come at the price of higher memory consumption. For the reference method, each primitive costs only a few bytes for that primitive's corresponding BLAS memory, while in our method we need to store an entire instance for each primitive, which currently seems to require on the order of 250 bytes.

Total memory consumption for both methods is given in Table 2. Given how much memory OptiX uses per instance, our method's current memory overhead is quite significant, at roughly 3×. This relative overhead does of course depend on what other data the renderer might use for other geometries, texture, etc., but in absolute terms it is still significant. We do believe that there should be room to reduce this per-instance memory cost on the OptiX side, but this would require changes to the GPU driver, and as such is not within our control.

model	Common Mem		Accel Data (links)		total		
	links	mesh	naive	ours	naive	ours	(+x%)
SciVis	7.44 MB	9.04 MB	11.8 MB	71.4 MB	28.2 MB	87.8 MB	+211%
neurons	107 MB	0 MB	168 MB	0.99 GB	275 MB	1.1 GB	+311%
DTI	1.1 MB	0 MB	1.76 MB	10.7 MB	2.87 MB	11.8 MB	+310%
autumn	234 MB	12.4 MB	369 MB	2.2 GB	615 MB	2.43 GB	+304%
franck	167 MB	13.8 MB	264 MB	1.6 GB	445 MB	1.74 GB	+302%

Table 2. Memory usage for our method relative to the reference method. Since OptiX currently spends significantly more memory per instance than per primitive our method's current memory usage is quite significant, at up to 300%.

4.3 Comparison to Woop et al.'s OBB BVH

Given the importance of hair-like primitives, and the speedups that OBBs can apparently achieve for them, the number of techniques to compare our system to is almost shockingly small. In fact, we found only one obvious comparison—the "local similarity" method by Woop et al. [21]. Even for Woop et al., a direct comparison is hard because we could not get access to the models they used.

Regarding model size, the largest model demonstrated by Woop is the Yeti at 153 million hair segments; we in contrast are limited not only by GPU memory, but in particular by the maximum number of instances that OptiX allows in the TLAS, which is currently 16 million. The other three models, however, are well within the same range as `franck` and `autumn`.

Regarding performance, Woop et al. report speedups of roughly 2 – 2.5×, which is roughly comparable to the average 2.7× speedup we see for `franck` and `autumn`. This is particularly surprising given that Woop et al. could modify the entire BVH—and thus, also save traversal steps—while our method is currently limited to the primitive level. In terms of memory, Woop et al. report roughly 1.5× overhead for the OBB, while ours is almost twice as high; however, we again observe that Woop et al. could perform additional compression of their BVH nodes, which we can not; conversely, if their reference BVH used the same compression as their OBB variant (which at the time it didn't) their memory overhead would be larger, too.

In summary, our method already achieves roughly similar results as Woop et al., at significantly lower code complexity, and even without changing the underlying BVH at all. Conversely, a potential extension of our method where the ray transforms would operate on entire subtrees rather than individual primitives (possibly on subtrees computed using the techniques proposed by Woop et al.) sounds quite promising, but has not been investigated, yet.

5 SUMMARY AND DISCUSSION

In this paper, we have presented a method that leverages existing ray tracing hardware's instance transform units to realize what is essentially a hardware-accelerated OBB culling test for arbitrary user programs. Though our method does bear some similarities to traditional instantiation, it only "abuses" hardware instancing to perform a general OBB test, but does in fact operate differently: In particular, whereas instancing is traditionally limited to creating multiple affine copies of a base primitive, our method isn't. In particular, neither the Quilez capsules nor the hair segments could have been realized with instantiation, while our method can even use the same intersection code.

Using our method allows for rendering non-trivial hair and fur models at up to 5.9× the performance of a reference method. Despite this speedup, in its current incarnation our method comes with several major limitations. The biggest such limitation clearly is the massive memory overhead of up to 3× compared to the reference version. In some applications the speedups achieved by our method may well be worth that memory cost, but for others this will be prohibitive. Reducing this overhead would require for OptiX to spend less memory per instance, or to extend our method to work on subtrees of multiple primitives, thus amortizing the memory overhead per primitive.

In addition to memory overhead, the maximum reasonable model size is also limited by how many instances OptiX allows in an instance acceleration structure (TLAS), which at the time of this writing is 16 M. Though we expect future architectures to relax this limitation, at this time it poses a hard limit on what our method can do.

Since we already use OptiX' instancing capability to create our masquerading OBBs we also currently cannot create "real" instances of those geometries that use this technique, at least not without changes to the intersection program we currently employ.

Finally, there are some obvious cases where our method (or any other OBB method, for that matter) would simply not be applicable: the core assumption of any OBB method is that the surface area of the OBBs is significantly smaller than the surface area of the corresponding AABBs. If this is not the case—e.g., for short and squat primitives, or those mostly aligned to coordinate axes, or that are so curled as to form circles or half-circles, etc.—then OBBs will rarely achieve any speedup. Similarly, our method is only applicable to user geometries, as triangles are already done in hardware, and do not have any intersection programs.

On the upside, our experiments show that, where applicable, our method can achieve significant speedups of up to 5.9×, and on average we see a speedup of about 2.5×. These speedups were significantly higher than we had initially expected, and also compare quite well to those reported by Woop et al. [21], albeit for different models. The latter is particularly surprising given that Woop et al. could modify the entire hierarchy (i.e., they could also create OBB nodes for entire BVH subtrees), while our method can currently only do OBB tests for individual primitives.

6 CONCLUSION

We have presented a method that “tricks” recent ray tracing enabled GPUs’ hardware instancing units into performing what is essentially a hardware-accelerated OBB rejection test for arbitrary primitive types. Our method is generally applicable to different primitive intersection programs and achieves significant speedups even for non-trivially sized hair and fur models. Even beyond our specific masquerading technique, maybe the core insight of this paper is how useful hardware ray transforms can be in “rotating” geometry (either individual primitives, or possibly entire BVH subtrees) to minimize BVH node surface area in the underlying BLAS.

As future work, we believe it would be promising to look at combining our method with Woop et al.’s OBB BVH. In particular, it would be interesting to build a hybrid system where Woop et al.’s similarity metric was used to find similarly oriented subtrees, while our method would then provide a hardware-accelerated way of traversing this.

REFERENCES

[1] Nikolaus Binder and Alexander Keller. 2018. Fast, high precision ray/fiber intersection using tight, disjoint bounding volumes. arXiv preprint arXiv:1811.03374.

[2] Blender Foundation 2019. Spring - A poetic fantasy film written and directed by Andy Goralczyk. https://cloud.blender.org/p/spring/

[3] Blender Foundation 2020. Cosmos Laundromat (Project Gooseberry). https://gooseberry.blender.org/about/

[4] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. 1996. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proceedings of ACM SIGGRAPH*.

[5] Mengjiao Han, Ingo Wald, Will Usher, Qi Wu, Feng Wang, Valerio Pascucci, Charles D Hansen, and Chris R Johnson. 2019. Ray Tracing Generalized Tube Primitives: Method and Applications. 38, 3 (2019).

[6] Naty Hoffman, Wenzel Jakob, Michal Iwanicki, Pesce Angelo, Danny Chan, David Neubelt, Matt Pettineo, Brent Burley, and Luca Fascione. 2015. Physically Based Shading in Theory and Practice. In *ACM SIGGRAPH 2015 Courses* (Los Angeles, CA) *(SIGGRAPH ’15)*. Association for Computing Machinery, New York, NY, USA, Article Article 6, 19 pages.

[7] Emmett Kilgariff, Henry Moreton, Nick Stam, and Brandon Bell. 2018. NVIDIA Turing Architecture In-Depth. https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/ NVidia Developer Blog.

[8] Daniel Koch, Tobias Hector, Joshua Barczak, and Eric Werness. 2020. "Ray Tracing in Vulkan". https://www.khronos.org/blog/ray-tracing-in-vulkan

[9] Stephen McAuley, Stephen Hill, Naty Hoffman, Yoshiharu Gotanda, Brian Smits, Brent Burley, and Adam Martinez. 2012. Practical Physically-Based Shading in Film and Game Production. In *ACM SIGGRAPH 2012 Courses* (Los Angeles, CA) *(SIGGRAPH ’12)*. Association for Computing Machinery, New York, NY, USA, Article Article 10, 7 pages.

[10] Microsoft 2020. DirectX Ray Tracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html version 1.11.

[11] Keith Morley. 2019. How to Get Started with OptiX 7. NVIDIA Developer Blog. https://devblogs.nvidia.com/how-to-get-started-with-optix-7/

[12] Koji Nakamaru and Yoshio Ohno. 2002. Ray Tracing for Curves Primitive. In *Proceedings of Winter School of Computer Graphics (WSCG)*.

[13] NVIDIA Corp. [n.d.]. NVidia Turing GPU Archictecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf WP-09183-001_v01.

[14] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, and Austin Robison. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* 29, 4 (2010).

[15] Inigo Quilez. 2016. Capsule - intersection. Available on ShaderToy. https://www.shadertoy.com/view/Xt3SzX

[16] Alexander Reshetov. 2017. Exploiting Budan-Fourier and Vincent’s Theorems for Ray Tracing 3D Bezier Curves. In *Proceedings of High Performance Graphics (HPG)*.

[17] Alexander Reshetov and David Luebke. 2018. Phantom Ray-Hair Intersector. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018).

[18] Will Usher. 2019. ChameleonRT. https://github.com/Twinklebear/ChameleonRT.

[19] Ingo Wald. 2020. OWL - A Node Graph Library Abstraction for OptiX 7. http://owl-project.github.io

[20] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 33, 4 (2014).

[21] 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*.