Eurographics Symposium on Parallel Graphics and Visualization (2018)
H. Childs, F. Cucchietti (Editors)

Robust Iterative Find-Next-Hit Ray Traversal

Ingo Wald Jefferson Amstutz Carsten Benthin

Intel Corporation

Abstract

We present two different methods for improving the performance and robustness of ray tracing algorithms that require iterating through multiple successive intersections along the same ray. Our methods do so without ever missing intersections, even in cases of co-planar or numerically close surfaces, and without having to rely on a dedicated all-hit traversal kernel. Furthermore, our evaluation shows that, in many cases, our methods consistently outperform existing all-hit kernels.

1. Introduction

Ray tracing applications in any domain use to a set of kernels for finding intersections between rays and geometric primitives. Whereas most ray tracing-based rendering primarily relies on so-called "first-hit" and "any-hit" kernels, physics-based simulations often require another category of kernel for finding either all, or the closest N, hits along a ray (otherwise known as "all-hit" and "multi-hit" kernels).

Traditional first-hit kernels have issues with scenes in which multiple surfaces are co-planar or numerically close to each other. In this case, traditional first-hit kernels require ray epsilon offsets to avoid ad infinitum self-intersections—but this in turn means multiple same-distance surfaces will only report one of the multiple valid intersections. This is often tolerable in rendering—either because only one such intersection is sufficient, or because the resulting artifacts do not matter much—but is generally not tolerable in simulation applications that have to accurately account for multiple successive surfaces (see Figure 1). Multi-hit kernels, however, are often not implementable or only partially supported by modern high-performance ray tracing frameworks. Furthermore, even when such kernels are supported, scenes with high depth complexity in applications that require but a few intersections along each ray may actually come with significant overhead.

In this paper, we propose two techniques that aim at obviating the need for multi-hit kernels by adopting the paradigm of a robust *next-hit* kernel that, in successive calls, returns the respectively *next* hit along a ray in a safe, robust, and well-defined fashion, without ever skipping or multiply reporting any intersections. In particular, we describe two implementations of this paradigm: First, we propose a simple modification to closest-hit kernels that is easy to integrate into existing frameworks such as Embree [WWB*14] and OptiX [PBD*10], and which allows those kernels to be safely used in a next-hit fashion—but requires to a fresh ray traversal for every call. Second, we introduce a novel *iterator* based kernel that tracks traversal state across successive calls, and thus avoids re-traversing nodes. We evaluate the cost characteristics of the two methods for multiple ray depths required by the renderer.

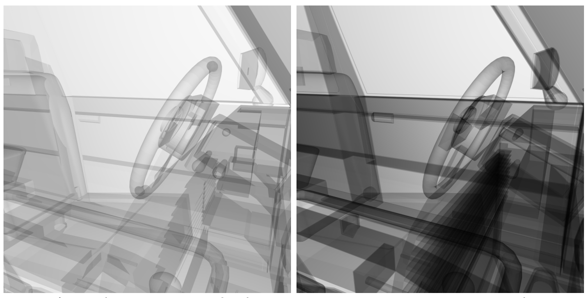


Figure 1: *The impact of "losing" intersections at co-planar surfaces with traditional closest-hit kernel and epsilon-offsetting, illustrated using a rendering of the* truck *model with each surface made partially transparent. Left: rendered with closest-hit and epsilon-offsetting (using a hand-tuned epsilon that just barely avoids self-intersections); right: with a correct traversal kernel that properly finds all surfaces without skipping any. Though the image on the left may look "plausible" in a rendering application it is in fact completely wrong; for a physics-based simulation code this outcome would be quite problematic.*

2. Background and Related Work

At its most general, ray tracing refers to a set of algorithms that allows for performing arbitrary 3D visibility queries. In particular, there are three kernels for a given ray $R(t) = R_{org} + tR_{dir}$ ($t \in [t_0, t_1]$): finding the *closest* intersection (*first-hit*), finding whether there is *any* intersection with this ray (*any-hit*), and, though less common, finding *all* (or at least, the first M) intersections along this ray (so-called *all-hit* or *multi-hit* kernels).

Ray tracing is best known for its use in rendering, where it is used to simulate light transport between camera, lights, and surfaces. It is, however, just as important for non-graphical domains, such as ballistic vulnerability analysis [BS07, AG14], radio/wireless signal propagation [Des72, ANM00, GA15], sound propagation [TCAM09], xray simulation [SdRCJC11], etc. Given this importance, it comes as little surprise that over the years a plethora of different kernels, data structures, and implementations have been proposed (see, e.g., [Gla89, Hav01] for an overview).

Virtually all this work falls in either of the three categories described above: closest-hit, any-hit, and multi-hit. Among those three, most efforts focusses on the first two, with relatively little attention spent on multi-hit. We argue that this is because the lion's share of attention (and consequently, research) has been captured by rendering, which traditionally relies more of the first two kernels. In rendering, light tends to get reflected or refracted at surfaces, thus a ray hitting a surface will typically leave that surface in another direction—in this case, finding more than one intersection along the original ray is generally less useful.

2.1. Multi-Hit Kernels

In simulation applications, however, the situation is different: the particles or waves that the rays represent often *penetrate* through multiple layers of surfaces—this makes closest-hit kernels actually problematic, as originally described by Gribble et al. [GNK14]. First, tracing multiple individual rays to step "through" multiple layers of surfaces is generally more expensive than tracing a single ray that finds all those surfaces. Second, and even more importantly, traditional closest-hit kernels have issues with "self-intersections", where a ray cast right off a surface returns that same surface as "closest" hit point ad infinitum.

The generally accepted way of avoiding this problem is through utilizing ray epsilon offsets, where the secondary ray's valid ray interval is set to start an epsilon-distance *behind* or in front of the previous hit point. This can indeed avoid self-intersections, but in turn is guaranteed to never find more than one intersection in an epsilon-interval around any hit point, even if more than one surface is present at that location. The resulting loss of some intersections may be tolerable for many graphical applications, but for simulation codes it is not. When objects with different material properties abut, such codes *must* be able to find both the surface where the ray/particle/wave leaves the one object *and* the (same- or similar-distance!) surface where it enters the next one—or the simulation result may be completely useless (also see Figure 1). Unfortunately, scenes where more than one surface are co-located—and where epsilon-offsetting thus results in many skipped intersections—are not the exception, but the norm.

In summary, there are two *different* problems with relying on only closest-hit kernels for penetration applications: *efficiency*—the cost of re-casting a ray for every penetrated surface—and *correctness*—losing intersections for co-planar surfaces. Multi-hit kernels can, in theory, solve both of these problems.

They are, however, not a panacea either for two reasons: First, since it is generally not possible to predict how many intersections a given ray will have, all-hit kernels require dynamic memory management, which in particular on modern, high-throughput architectures can be costly. Second, in most real-world applications the rays will typically *not* always penetrate through all surfaces. Thus always computing all surface intersections for any ray is potentially very wasteful—in particular if the number of actually required intersections is small relative to the scene's average depth complexity.

Both memory allocation and—to some degree—wastefulness can be alleviated by switching from all-hits to specialized N-hits kernels that find, for a fixed N, the first N hits along this ray. However, picking the right *N* can be challenging, too: when too large it is still wasteful, yet when too small it still requires dealing with how to handle intersections not yet found in this query.

Eventually, all the issues we discussed thus far—the issues caused by epsilon-offsetting for closest-hit, the potential wastefulness of all-hits, and the issues with finding the proper *N* for N-hits—would largely disappear if only there was an efficient way of iteratively stepping through multiple hits along the same ray—always returning but one "next" hit—in a robust and correct way, without ever losing any intersections, and, ideally, without having to re-start traversal for every next step.

2.2. Related Work

An overview of ray tracing in graphics can be found, for example, in [SM03]. Havran's thesis [Hav01] gives a comprehensive overview over different data structures and implementations up until 2001. Since then, research has mostly focussed optimizing ray tracing for modern CPUs and GPUs [WWB*14, AL09].

Historically, most applications of ray tracing considered the actual ray tracing kernel implementation as an integral part of that application, meaning that every such application came with their own ray tracing kernel implementations. At least in graphics, what we are seeing today is a shift towards ray tracing framework libraries (ie, OptiX [PBD*10] for GPUs, and Embree [WWB*14] for CPUs) that get re-used across many different renderers.

While adoption of Embree and OptiX continues to grow in graphics, non-rendering codes (such as, for example, the DoD's BRLCAD package [Dyk13]) still maintain their own ray tracing kernel implementations largely because they require efficient next-hit/multi-hit kernels that rendering packages tend to give less care than their highly optimized first-hit and any-hit kernels. The downside of this de-coupling of ray tracing for rendering and ray tracing for non-rendering applications is that these non-rendering uses of ray tracing have not benefitted from the—often significant—performance advances on the rendering front. As a consequence, performance of such kernels are often several multiples off that for the fastest known ray tracing implementations in rendering.

To bridge this gap, several researchers have recently started to bring rendering-side advances in ray tracing technology to non-rendering applications of ray tracing. Butler and Stephens [BS07] were the first to demonstrate the potential of doing so by adding a multi-hit capability to one of then fastest ray tracing systems (Manta [BSP06]). More recently, Gribble et al. proposed a new high-performance ray tracing framework for simulation applications (called RayForce [GN13]) that aimed at leveraging GPUs and spatial indexing structures such as KD-trees.

Rather than competing with Embree and OptiX, Amstutz et al. [AGGW15] showed that multi-hit can also be realized using Embree's "intersection filter" call-backs and OptiX's "any-hit program" features. Most recently, Gribble [Gri16] proposed a specialized node-culling traversal technique for Embree, and showed that this, in some cases, requires only a fraction of the ray-triangle intersections of an all-hit kernel. Furthermore, Gribble et al. [GWA16] showed how this technique can be implemented in Embree.

3. Robust First-Hit Kernel

As argued above there are two different—and largely orthogonal—reasons for using multi-hit kernels: *correctness* and *efficiency*. Of those two, efficiency depends on how many hits are required compared to how many hits exist along the ray. However, the correctness argument is independently valid. In particular, we argue that for applications that require to find only few hits are primarily motivated by correctness. In other words, an application may well prefer a single-hit kernel if only it could be guaranteed that successive calls to this kernel would correctly iterate through ray intersections. Therefore, our first method looks at making first-hit ray traversals able to correctly traverse all hits along the ray exactly once.

Identifying the problem Hits with the same (or numerically similar) distance are missed by first-hit kernels because of epsilon-offset issues, which in turn is required because hits with same distance otherwise leads to self-intersections. The root of this problem is that first-hit kernels are typically defined as *closest*-hit kernels, where comparing different hits only by distance is ambiguous, allowing the traversal algorithm a way to determine which hits are *closer* than others, but not allowing it to put a clear, unambiguous order on hits that have the same distance. As long as no deterministic ordering exists, a respective "first" or "next" hit along a ray does not make sense.

Fixing the problem All that is required to enforce a global order on hit points is to define a comparison operator $<_{hit}$ that implements a global ordering relationship on the space of all possible hits. There are no requirements to use a particular ordering, just that there exists an ordering for hit points. For example, assuming each hit point H could be uniquely identified through a primitive ID $H.p$ and a distance $H.t$, then all that remains is to use the primitive ID to disambiguate hits with same distance:

$$H_1 < H_2 = \begin{array}{rl} true & : for\, H_1.t < H_2.t \\ true & : for\, H_1.t = H_2.t\ and\ H_1.p < H_2.p \\ false & : otherwise \end{array}$$

In using such an ordering, the order of hit points along a given ray becomes well defined, making it possible to query exactly the *next*—hit, the ray that is $<_{hit}$ any other hit points, but that comes "after" another hit h_0. Similarly, the concept of a "valid t interval" $t \in [t_0, t_1]$ can be replaced by a "valid hit interval" $h \in [H_0, H_1]$ that only allows for hits H that fulfill $H_0 < H < H_1$. For applications that only want to specify a minimum *distance*, t_0 even if there is no intersection at that distance, it is still possible to specify a minimum hit $H_0 = (t_0, -1)$ (assuming that -1 is a invalid primitive ID).

Once a complying comparison test is integrated into a given find-first-hit traversal kernel, an application that needs to step through multiple hits along a ray can easily do so knowing that each hit will be reported exactly once— given an initial ray interval $[t_0, t_1]$, the application would start by querying the first hit in the $[(t_0, -1), (t_1, -1)]$ interval and get hit point $\hat{H}_0$; in the next iteration it would query the first hit in $[\hat{H}_0, (t_1, -1)]$ and get $\hat{H}_2$, etc.

Variations and Extensions While we have considered only a single primitive ID, extension to a multi-level addressing scheme (such as Embree's $(instID, meshID, primID)$) is straightforward. Similarly, any other comparison method is fine as long as any pair of hit points has a consistent and well defined order. In particular, since the only requirement is to apply this method to the distance test between two hit points, the method is completely compatible with any acceleration structure, primitive type, traversal order, SIMD optimization, or other traversal variant that is based on tracking the "closest" hit inside of a given interval.

We can, in fact, engineer the ordering relation to also achieve other goals that may be important or desirable for the application. For example, for applications that need to track which objects a ray enters or leaves, we could make the find-next-hit kernel always return "leave" events before "enter" events. Similarly in rendering, decals objects can made to be returned before a base geometry, etc.

4. Next-Hit via Iterative Front-to-Back Traversal

While the previous section's method does alleviate the *most* important reason for a multi-hit kernel (correctness), it still requires a complete re-start of the data structure traversal every time the next hit point is queried—a cost which can be expensive for applications needing more than a few hit points.

To avoid this a separate kernel that can resume traversal where a previous call has left off is needed. To do this, we first need a means of passing information from a previous traversal to the respective "next" hit in the *same* traversal sequence. For this, we first have to break with the paradigm that all traversals are independent and introduce a new, two-step paradigm that is similar to an "iterator"—after requesting a new "next hit" iterator for a given ray, subsequent calls to that iterator will successively return the respective next correct hit point, where the iterator tracks the information about what has already been traversed.

With this new ray-iterator concept, implementing an efficient next-hit kernel is straightforward. First, we keep track of nodes that have not yet been traversed, and only traverse nodes until we are sure that the next hit has been found. We use some sort of stack or priority queue of untraversed nodes, and traverse until the current closest hit is guaranteed to be closer then any as yet untraversed node. Second, we keep track of *every* hit encountered during traversal—even a hit that is not currently closest, as it may be the closest hit in a future iteration. Therefore, the iterator also must keep track of all hits already found (using some form of heap or sorted list).

Though the core ideas just described are also applicable to other data structures, we apply them to Bounding Volume Hierarchies (BVHs) because BVHs are by now the most prevalent acceleration structures we see in practice. Thus we will from now on only consider BVHs of arbitrary branching factors. Also, though the same algorithm can also be implemented in recursive depth-first traversal schemes it is most easily be explained in a front-to-back BVH traversal, which we will from now on assume.

4.1. Front-to-back Traversal for BVHs

Unlike spatial hierarchies such as grids or kd-trees, BVHs are object hierarchies in which different subtrees can overlap. This means that recursive BVH traversal cannot guarantee that nodes (or leaves) will be traversed in a front to back order, even if each traversal step properly sorts its children.

Though front-to-back traversal is sometimes believed to not be possible with BVHs, it is in fact easy to achieve: all that is required is to replace the logical traversal *stack* of still-to-be-traversed nodes with a *priority queue* (e.g, a heap, sorted list, etc) of such nodes, with each node's priority in this queue being its distance to the ray origin, which is known from the ray-bounding box intersection test. Once such a priority queue exists, all that is required is that each traversal step always picks the closest node, tests its children against the ray, and then inserts all intersected children with their respective distance.

Though inherently simple, this algorithm is not widely used nor well described in the literature: though algorithmically superior to recursive traversal the cost of maintaining and updating the priority queue may well negate any savings in fewer traversal steps. This is particularly true if the BVH is well built, and if the recursive traversal order has "fast path"s for cases for the common cases of intersecting 0, 1, or 2 hit points (see, e.g., [BWW*12]).

```
struct NextHitIterator {
  priority_queue<float,Hit>  hitQueue;
  priority_queue<float,Node> nodeQueue;
};

/* initialize a new next-hit traversal sequence */
NextHitIterator *initNextHit(Ray ray)  {
  NextHitIterator *it = new NextHitIterator;
  it->ray = ray;
  /* add BVH root node as (only) yet-to-traverse node */
  it->nodeQueue = { scene.rootNode() };
  /* no hits found yet: */
  it->hitQueue = { empty() };
  return it;
}

/*! find next hit along the ray (if one exists) */
Hit findNextHit(NextHitIterator *it) {
  while (it->nodeQueue not empty) {
    /* pick closest un-traversed node/subtree */
    Node closestNode = it->nodeQueue.front();
    /* check if any hits are closer than closest node */
    if (it->hitQueue not empty AND
        hitQueue.front().t < closestNode.t)
      /* this hit is guaranteed to be the next hit:
         it's the closest known hit,
         and closer than any other subtree */
      return hitQueue.pop_front();
    /* no closer hits: remove node from queue... */
    nodeQueue.pop_front();
    /* ... and traverse it: */
    if (node is leaf) {
      foreach pritmitive in node /* in SIMD */ {
        if ((hit = intersect(primitive)) != NO_HIT)
          it->hitQueue.insert(hit.t,hit);
      }
    } else {
      /* ... test every child of this node ... */
      foreach child of node /* possibly in SIMD */ {
        /* enqueue every child that intersects the ray,
           with its respective distance */
        if (child.box intersects it->ray)
          it->nodeQueue.insert(distanceTo(child),child);
      }
    }
  }
  /* no more expandable nodes: return closest hit */
  if (it->hitQueue is empty)
    /* no more hits - we're done */
    return NO_HIT;
  /* return closest hit in hitlist */
  return it->hitQueue.pop_front();
}

/*! terminate next-hit traversal: just release state */
void endNextHit(NextHitIterator *state)
{ delete state; }
```

Algorithm 1: Pseudo-Code for our iterative *findNextHit* kernel, which consists of three parts: Initializing a new iterative traversal (`initNextHit`), stepping from one the last found hit to the respective next hit along that ray, and finally cleaning up state (`endNextHit`).

4.2. Find-next-hit via Front-to-Back Traversal

Once we have such a front-to-back traversal, the ground work for a find-next-hit scheme is done—in addition to a priority queue for the as yet untraversed nodes we also maintain a priority queue of already found hits. Then, as soon as the closest hit in the queue is closer than the closest not yet traversed node we know that this is the guaranteed to be the closest hit, where it can be returned to the application. Otherwise, we pop the closest yet-untraversed node from the priority queue, and traverse it: if it is a leaf node, we intersect all primitives and add all found hits to the hit list, otherwise if an inner node we test all children and add them to the node list. Finally, the next-hit sequence gets initialized with a node queue that contains the BVH root node. A pseudo-code sketch of this algorithm is given in Algorithm 1.

5. Implementation and Evaluation

We evaluate our kernels by implementing them inside of Embree v2.11.0 and benchmark them through the OSPRay ray tracing framework [WJA*16]. Embree supports a large number of different CPU instruction sets, BVH types, primitive types, and other variants—the full support of which is beyond the scope of this paper. Thus, our current implementation only supports what we considered the most representative such configuration: the AVX2 instruction set, triangles, and a single-ray interface. In this case, Embree uses a `BVH8` data structure, a `Triangle4` primitive, and a `Triangle4Intersector1MoellerTrumbore` intersector. To expose the new traversal kernel to applications, we added three new API calls: `rtcNhBegin`, `rtcNhPlusPlus`, and `rtcNhEnd`.

We extended OSPRay with a custom "X-Ray"-style renderer that treats each surface as partially transparent (using a transparency of $alpha = 0.5 + 0.5|dot(ray.dir, hit.normal)|$). Since OSPRay itself uses ISPC [PM12], we also added the ISPC-equivalent of `nhBegin`, `nhPlusPlus`, and `nhEnd` API calls to step through the transparent surfaces in a front-to-back manner.

Each of these three ISPC-side API calls is internally implemented in one of three different, compile-time selectable ways:

restart does nothing in *ospNhBegin/End* and uses successive calls to Embree's `rtcIntersect` in *ospNhPlusPlus* using the modified distance test described in Section 3.

filter realizes the next-hit iterator using a multi-hit kernel as described in [AGGW15]: *ospNhBegin* uses Embree's intersection filter to find *all* hits along the ray (in sorted order), and stores those in the state. `ospHnPlusPlus` then merely returns the respectively next hit from this precomputed list.

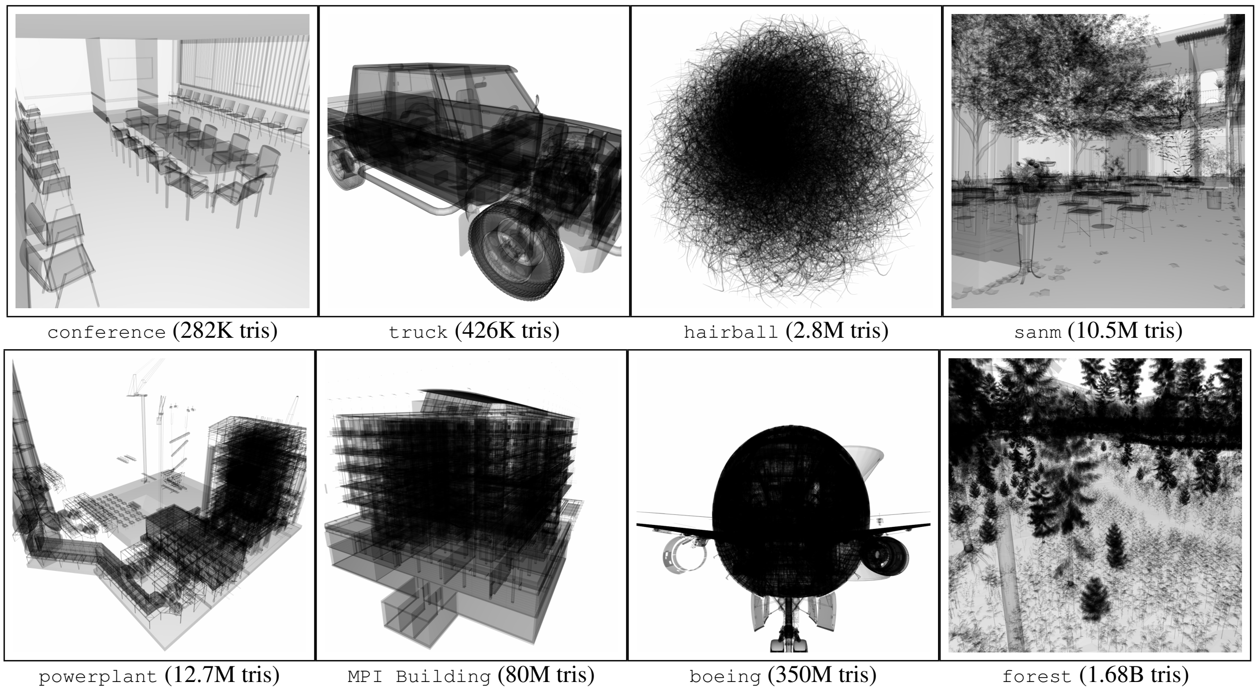


Figure 2: *The scenes we used for evaluating our method, rendered with a "XRay" shader in which each surface is partly transparent.*

iterative uses our queue-based iterative next-hit traversal as previously described.

All three kernels internally operate on individual rays, where we use ISPC's `foreach_active` capability to iterate over ray packets with scalar code.

5.1. Evaluation Framework

The scenes we use for evaluation are shown in Figure 2, where they intentionally cover different application scenarios (engineering, architectural, and outdoors)—in particular, they cover a wide range of geometric complexity and depth complexity (from a few hundred thousand to over 1.6 billion primitives, and from a few dozen to up to hundreds of hits per ray).

To simulate workloads that require fewer than all hits, we specify a maximum number of surfaces that this renderer will step through, and ran all experiments with maximum number of 1, 5, and 15 hits, as well as one ("all") that always ran through all hit points until no more could be found. Note that for the hardcoded maximum number of hits *only* the renderer that is aware of this number—the kernels themselves are not aware of this information and thus do not optimize for this maximum hit depth in any way.

For each of these configurations—kernel type, maximum number of hits, and scene—we ran our set-up on a workstation with dual Intel-Xeon 2699 v4 "Broadwell" CPUs (totalling 44 cores at 2.2GHz), and measured both (maximum) frame rate and ray depth complexity information, using a screen resolution of 1024×1024 pixels. The results of these measurements are given in Figure 3.

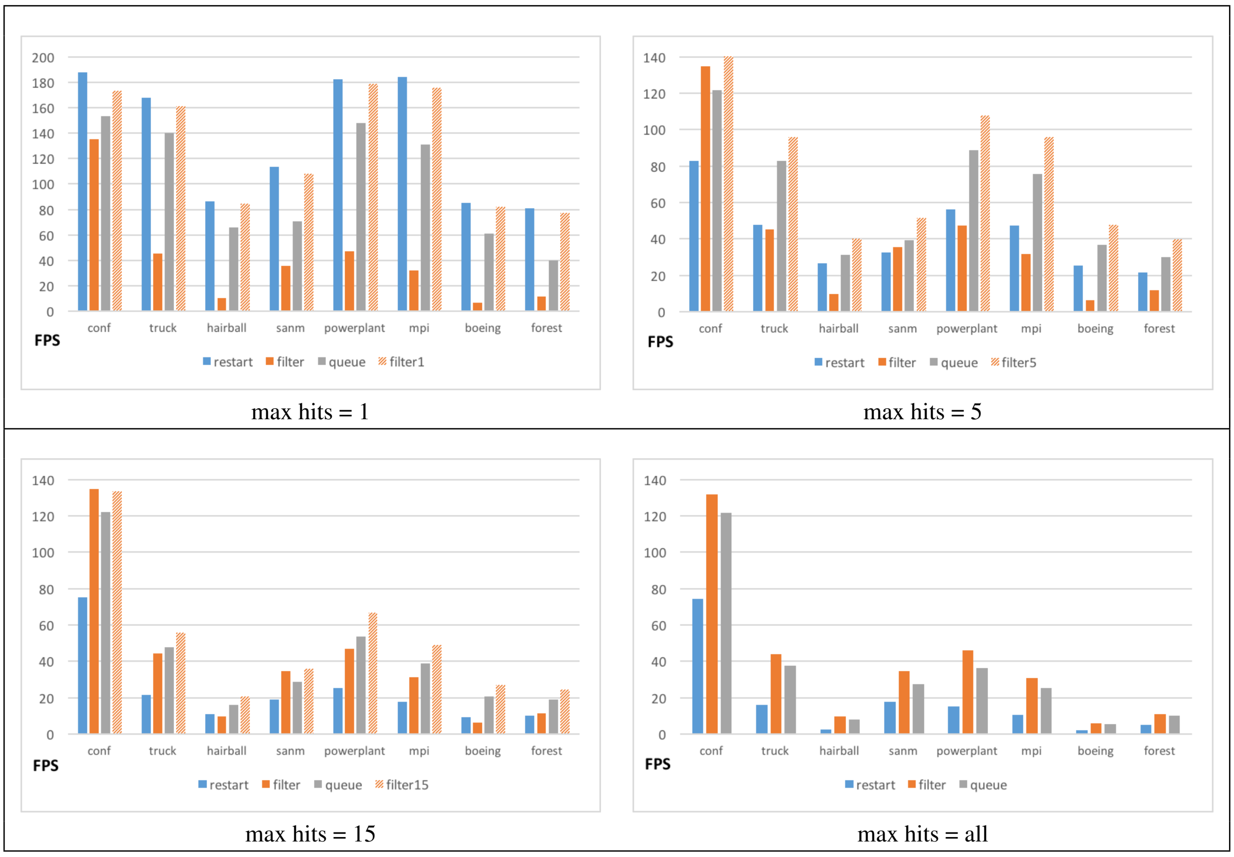
5.2. Performance Results

Expectations The results obtained by our measurements largely confirmed our expectations, but also contained some surprises. Broadly speaking, what we *expect* to happen is for the `filter` method to exhibit a cost that is nearly independent of how many hits get queried, and nearly linear in the average depth complexity. Consequently, the filter method should always win for the "all hits" case, but should be very expensive for the cases where only few hits get queried.

On the opposite end of the spectrum, the `restart` method is but a tiny modification of Embree's fastest first-hit kernel, and therefore ought to be the fastest for the case where only 1 or 2 surfaces are queried. However, its cost should increase linearly with the number N of hits queried, making it far less effective for large N.

Finally, the `queue` method should have a significant overhead relative to restart (for managing the priority queue(s)), but should be able to amortize cost over successive queries, and thus eventually overtake `restart`. It should also be faster than `filter` for few queried hits, but eventually be overcome in larger N hit cases.

Measured Results Looking at measured performance in Table 3, in virtually all cases (with the sole exception of the smallest scene, `conference`) the performance characteristics of the three kernels turned out to be exactly as predicted. When comparing `filter` and `restart`, `restart` is always faster when only one hit is required, and significantly so for high-depth complexity models. Also as expected, in the all-hits case these roles are reversed, with margins increasing with the scene's depth complexity. What came a surprise, though, is how well the `queue` method performed: we had expected this method to win only for a low to medium number of queried hits, but to lose significantly against `filter` for all hits, and against `restart` for 1 or 2 hits. As it turns out, `restart` and `filter` are indeed the fastest methods, but by a much smaller margin than expected; with the queue method winning in both the 5 and 15 hits cases, and losing but marginally even in the extreme cases.



	ray complexity		1 hit			5 hits			15 hits			all		
scene	avg	max	restart	filter	queue	restart	filter	queue	restart	filter	queue	restart	filter	queue
conf	2	20	**188**	135	153	83	**135**	122	75	**135**	122	74	**132**	122
truck	15	68	**168**	46	140	48	45	**83**	21	44	**48**	16	**44**	38
hairball	22	194	**86**	10	66	27	10	**31**	11	10	**16**	3	**10**	8
sanm	6	74	**114**	36	71	32	35	**39**	19	**35**	29	18	**35**	28
pplant	13	134	**182**	47	148	56	47	**89**	25	47	**54**	15	**46**	36
mpi	19	98	**184**	32	131	47	32	**76**	18	31	**39**	10	**31**	25
boeing	66	345	**85**	7	61	25	6	**37**	9	6	**21**	2	**6**	6
forest	14	190	**81**	12	40	22	12	**21**	10	12	**13**	5	**11**	10

Figure 3: *Performance (in frames per second) for the three find-next-hit methods discussed in the text: finding all hits up front using an intersection filter (filter), iterating through all surfaces using the robust find-first-hit method from Section 3 (restart), and using our new iterative method described in Section 4 (queue). For each method, we show measurements for different numbers of intersections queried by the renderer. For comparison we also include performance for the hypothetical node-culling filter1, filter5, and filter15 methods.*

Our results show that in all cases our method is either the fastest, or close to the fastest, independent of scene and number of queried hits. This is particularly important for applications that cannot predict how many hits will be required: though we cannot guarantee that it will always be the best, it will never be bad, and always be close to the best possible results.

5.3. Comparison to FilterN

Throughout the previous section's evaluations we have assumed what we argue to be the most realistic scenario: that the application does not know a priori how many hits it will actually require. While our tests *do* use an artificial ray depth for benchmarking purposes, we assert that almost all applications will not know this information up front as the decision to need more intersections is largely determined by what surfaces the ray encounters *during* traversal. As such, we have so far not compared our methods to the node-culling technique described by Gribble et al. [Gri16, GWA16].

For completeness, we also implemented the node-culling technique for the 1, 5, and 15 hits, using the same algorithm as described in [GWA16]; this will always find exactly the closest 1, 5, and 15 hits, respectively, and avoid traversing those regions of the BVH that are guaranteed to only have hits further than those N already found, thus saving potentially many traversal steps in particular for high depth complexity models.

model	1 hit		5 hits		15 hits	
	filter-1*	queue	filter-5*	queue	filter-15*	queue
conf	173*	153	140*	122	133*	122
truck	161*	140	96*	83	56*	48
hairball	85*	66	40*	31	21*	16
sanm	108*	71	51*	39	36*	28
pplant	179*	148	108*	89	67*	54
mpi	176*	131	96*	76	49*	39
boeing	82*	61	48*	37	27*	21
forest	77*	40	40*	30	24*	19

Table 1: *Comparison to Gribble et al.'s specialized node-culling traversal as described in [GWA16], using this technique's best-case scenario where the application has some form of a priori knowledge about exactly how many hit points it will eventually need (filter-1, filter-5, and filter-15). In this (mostly hypothetical) scenario, node culling is indeed the fastest technique across all experiments, while our (more general)* `queue` *technique remains competitive.*
(* Requires advance knowledge of number of queried hits).

The results of this comparison can be seen in Table 1. As expected, in this (for this technique best-case) scenario the node culling technique with hard-coded number of hits is indeed always faster than our queue method. However, even without requiring such a priori knowledge our `queue` method is close, and if this technique is run with only slightly more hits than the application actually requires the `queue` method once again win.

5.4. Correctness

For all three kernels we have verified that the same results get computed. Since neither `filter` nor `queue` enforce any ordering on same-distance hit points we can not guarantee that all three kernels return same-distance hits in the same order, but we did verify that neither of the three methods skips or misses intersections. For the sake of completeness, we have also run some of our examples with the modified distance test disabled. In this case, just as expected, the `filter` and `queue` methods do not change, but `restart` encounters infinite loops from self-intersections, while adding an epsilon-offset leads it to missed intersections as shown in Figure 1.

6. Summary and Conclusion

In this paper, we argued that the main reason for the existence of multi-hit and all-hit kernels is correctness—ie, the need to find successive hits without missing any—and have proposed two alternative ways of solving that same problem without having to rely on the multi-hit paradigm.

The first of these two is a simple modification of existing first hit kernels that only requires modification of the distance-test used to compare two hit points. Using this, we can guarantee that any traversal for any acceleration structure will properly report the respectively "first" hit in the specified search interval, allowing correct iteration through all hits by simply advancing the valid interval to the respectively last found hit. We believe this to be an obvious win for any ray tracer (irrespective of data structure and/or hardware), at a near negligible implementation—and runtime—cost.

Our second kernel requires a new "find next hit" paradigm that requires changes to the API, but enables iteration through successive hits faster by tracking traversal state between hits. Somewhat surprisingly, this kernel performed even better than expected, beating both the robust first-hit kernel as well as Amstutz's intersection filter based all-hits implementation [AGGW15] in almost all cases—sometimes significantly so.

Both methods are guaranteed to be robust in that they never missing or multiply report any hits. We believe this to be an important step towards bridging the rendering and non-rendering communities which use ray tracing, hopefully making it easier to adopt fast ray tracing solutions such as Embree or OptiX to non-rendering use cases. Though primarily designed for non-rendering use cases such as simulations, allowing renderers to get rid of epsilon-offsetting along a ray also has important use cases in rendering, in particular when dealing with transparent surfaces, decals, etc.

7. Discussion and Future Work

We were quite surprised to find how well the `queue` method performed. If our `queue` method can get this close to embree's fastest first-hit kernel for the 1-hit case—despite having a guaranteed overhead in terms of maintaining the hits list, and despite it not yet being fully tuned—then a priority-queue based specialization for first-hit may lead to further performance improvements for first-hit kernels as well.

Though we are pleased with the observed results, some issues remain. For the queue method, we currently use hard-coded maximum list sizes. Though we could easily over-provision for any of the models we used, there are always models that will require even larger lists. Handling these cases with this will require some sort of dynamic allocation, at least when reaching a certain queue size.

Even without dynamic allocation, the non-trivial memory overhead required for the queue data structures can be problematic in particular when memory is scarce (e.g., on a GPU), or when dealing with lots of active rays (e.g., operating on ray streams).

Aside from performance considerations, one limitation of the queue method is that it cannot be added into existing ray tracing frameworks without changing their APIs. The `restart` kernel from Section 3, in contrast, allows this, which keeps this kernel rather interesting despite the commonly lower performance.

In terms of non-rendering simulation codes, our method solves what we believe to be one of the most challenging problems—robustness and correctness—but others will undoubtedly remain. In particular, many such codes require double-precision (which Embree currently does not support), and many rely on CSG operations that we do not even address.

Acknowledgements

We thank Lee Butler and Christiaan Gribble for their input in discussions regarding multi-hit ray tracing. We also thank the Embree team for their support with the implementation used to evaluate the methods discussed in this paper.

References

[AG14] AMSTUTZ J., GRIBBLE C.: Leveraging GPUs for Ballistic Simulation. *DSIAC Journal 2*, 1 (2014). 1

[AGGW15] AMSTUTZ J., GRIBBLE C., GÜNTHER J., WALD I.: An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels. *Journal of Computer Graphics Techniques 4*, 4 (2015). 2, 4, 7

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics* (2009). 2

[ANM00] ATHANASIADOU G. E., NIX A. R., MCGEEHAN J. P.: A microcellular ray-tracing propagation model and evaluation of its narrow-band and wide-band predictions. *IEEE Journal on Selected Areas in Communications 18*, 3 (2000). 1

[BS07] BUTLER L., STEPHENS A.: Bullet Ray Vision. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007). 1, 2

[BSP06] BIGLER J., STEPHENS A., PARKER S. G.: Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 187–196. 2

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W. R.: Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (2012). 4

[Des72] DESCHAMPS G. A.: Ray techniques in electromagnetics. *Proceedings of the IEEE 60*, 9 (1972). 1

[Dyk13] DYKSTRA P. C.: *The BRL-CAD package: An overview*. Tech. rep., DTIC Document, 2013. 2

[GA15] GRIBBLE C., AMSTUTZ J.: StingRay: High-Performance RF Energy Propagation Modeling in Complex Environments. *DSIAC (Defense Systems Information Analysis Center) Journal 2*, 2 (2015). 1

[Gla89] GLASSNER A. S. (Ed.): *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK, 1989. 1

[GN13] GRIBBLE C., NAVEROS A.: GPU ray tracing with Rayforce. In *ACM SIGGRAPH 2013 Posters* (2013). 2

[GNK14] GRIBBLE C., NAVEROS A., KERZNER E.: Multi-Hit Ray Traversal. *Journal of Computer Graphics Techniques 3*, 1 (2014). 2

[Gri16] GRIBBLE C.: Node Culling Multi-Hit BVH Traversal. In *Eurographics Symposium on Rendering (EI&I track)* (2016). 2, 6

[GWA16] GRIBBLE C., WALD I., AMSTUTZ J.: Implementing Node Culling Multi-Hit BVH Traversal[in Embree. *Journal of Computer Graphics Techniques 5*, 4 (2016). 2, 6, 7

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech TU in Prague, 2001. 1, 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings ACM SIGGRAPH) 29*, 4 (2010). 1, 2

[PM12] PHARR M., MARK B.: ISPC: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings of Innovative Parallel Computing (inPar)* (2012), pp. 184–196. 4

[SdRCJC11] SANCHEZ DEL RIO M., CANESTRARI N., JIANG F., CERRINA F.: SHADOW3: a new version of the synchrotron x-ray optics modelling package. *Journal of Synchrotron Radiation 18*, 5 (2011). 1

[SM03] SHIRLEY P., MORLEY R. K.: *Realistic Ray Tracing*, second ed. A K Peters, 2003. ISBN 1-56881-198-5. 2

[TCAM09] TAYLOR M. T., CHANDAK A., ANTANI L., MANOCHA D.: RESound: interactive sound rendering for dynamic virtual environments. In *Proceedings of the 17th ACM international conference on Multimedia* (2009). 1

[WJA*16] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J. L., GÜNTHER J., NAVRATIL P.: OSPRay—A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics (Proceedings IEEE VisWeek)* (2016). 4

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 33* (2014). 1, 2