Eurographics Symposium on Rendering - Experimental Ideas & Implementations (2016)
E. Eisemann and E. Fiume (Editors)

Node Culling Multi-Hit BVH Traversal

Christiaan Gribble†

Applied Technology Operation
SURVICE Engineering

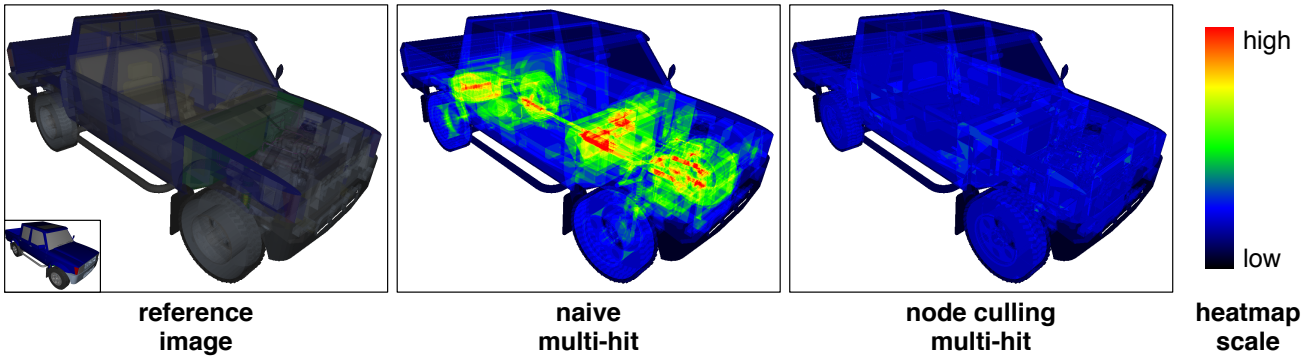

Figure 1: Faster multi-hit ray tracing in a BVH with node culling. *We introduce node culling multi-hit BVH traversal, which significantly improves multi-hit performance in a BVH when users request fewer-than-all hits. Here, heatmap visualizations depicting the number of valid ray/primitive intersections identified for each ray before correctly satisfying the multi-hit query reveal that far fewer intersections must be identified when using our node culling algorithm compared to using naive multi-hit traversal in a BVH.*

Abstract

We introduce node culling multi-hit BVH traversal to enable faster multi-hit ray tracing in a bounding volume hierarchy (BVH). Existing, widely used ray tracing engines expose API features that enable implementation of multi-hit traversal without modifying their underlying—and highly optimized—BVH construction and traversal routines; however, this approach requires naive multi-hit traversal to guarantee correctness. We evaluate two low-overhead, minimally invasive, and flexible API mechanisms that enable node culling implementation entirely with user-level code, thereby leveraging existing BVH construction and traversal routines. Results show that node culling offers potentially significant improvement in multi-hit performance in a BVH for cases in which users request fewer-than-all hits.

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

1. Introduction

Multi-hit ray traversal is a class of ray traversal algorithms that finds one or more, and possibly all, primitives intersected by a ray ordered by point of intersection. Gribble et al. [GNK14] introduce two algorithms for multi-hit traversal in acceleration structures based on spatial subdivision: a naive algorithm, which essentially implements all-hit traversal but returns (at most) the N closest hit points, and a buffered algorithm with early-exit, which exploits ordered traversal to terminate after (at least) the N closest hit points have been found.

With acceleration structures based on spatial subdivision, leaf nodes do not overlap, so ordered traversal is straightforward. However, ordered traversal in a structure based on object partitioning, such as a bounding volume hierarchy (BVH), is not achieved so easily: though at any level primitives belong to only one node, the nodes themselves may overlap, so strict front-to-back traversal can-

† christiaan.gribble@survice.com

DOI: 10.2312/sre.20161213

not be resolved easily. This characteristic is at odds with the ordered traversal guarantee exploited by buffered multi-hit ray traversal.

Recently, Amstutz et al. [AGGW15] evaluate the use of intersection callbacks (ICBs) to implement multi-hit traversal in Embree [WWB*14] and OptiX [PBD*10], while leveraging these engines' existing, highly optimized BVH construction and traversal algorithms. Their work demonstrates that with careful attention to performance concerns and characteristics of the underlying hardware architecture, ICBs enable correct implementation of multi-hit ray tracing in a BVH, despite the apparent conflict between ordered hit points and overlapping nodes. Moreover, this ICB-based approach integrates multi-hit traversal into these engines using only user-level code; this characteristic is important for maintainability, particularly in production environments.

Unfortunately, use of ICBs in Embree and OptiX requires naive multi-hit traversal to guarantee correctness: lack of strict front-to-back traversal in a BVH requires traversal of all nodes with which a ray interacts before the multi-hit query can be answered, as depicted in the middle panel of Figure 1. Though simple and effective, naive multi-hit traversal is potentially very slow, particularly for cases in which users request fewer-than-all hits.

We introduce node culling multi-hit BVH traversal to enable faster multi-hit ray tracing in a BVH, and we evaluate two low-overhead, minimally invasive, and flexible API mechanisms that enable node culling implementation entirely with user-level code, thereby leveraging existing BVH construction and traversal routines. Results demonstrate that node culling offers potentially significant improvement in multi-hit performance in a BVH for cases in which users request fewer-than-all hits, as shown in the right panel of Figure 1.

2. Node Culling Multi-Hit BVH Traversal

As noted above and discussed in detail by Amstutz et al. [AGGW15], the problem of multi-hit ray traversal in a BVH is compounded by overlapping nodes: correctness requires either naive multi-hit traversal, which is potentially slow, or modification of BVH construction or traversal routines, which imposes potentially significant development and maintenance burden in production environments.

If we dismiss naive traversal as too slow, we must seek an approach that performs well but that also minimizes the engine-level code specific to multi-hit traversal. To improve multi-hit performance, we focus on an algorithm that exploits knowledge of the currently valid ray interval throughout traversal to cull subtrees corresponding to interior nodes or to avoid ray/primitive intersection tests arising in leaf nodes. To minimize development and maintenance burden, we highlight two callback-based approaches that implement the node culling algorithm without engine-level traversal routines specific to multi-hit ray tracing.

2.1. Algorithm

First-hit BVH traversal algorithms typically consider the currently valid interval along a ray, $[\varepsilon, t_{near}]$, to cull nodes based on t_{near}, the distance to the nearest valid intersection found so far. If during traversal a ray enters a node at $t_{enter} > t_{near}$, the node is skipped: traversing the node cannot possibly produce a valid intersection closer to the ray origin than the one already identified.

We would like to apply a similar optimization during multi-hit ray traversal in a BVH, but such an approach imposes additional traversal constraints:

- We can cull nodes only after we have collected at least $N \geq N_{query}$ hits; we must therefore count the number of valid ray/primitive intersections encountered so far.
- We can cull nodes using only the distance to the farthest valid hit point among the $N \geq N_{query}$ collected so far; we must therefore track this value, say t_{far}, throughout traversal.
- We can cull nodes for which $t_{enter} > t_{far}$ only after $N \geq N_{query}$ hit points have been gathered; we must therefore update the ray interval only after finding $N \geq N_{query}$ hits.

Algorithm 1 provides pseudocode for multi-hit BVH traversal that includes the node culling optimization.

```
 1: function TRAVERSE(root, ray)
 2:     INITIALIZE(travStack, hitList)
 3:     PUSH(travStack, root)
 4:     while !EMPTY(travStack) do
 5:         node ← POP(travStack)
 6:         if !INTERSECT(node, ray) then
 7:             continue
 8:         if ISLEAF(node) then
 9:             for primitive in node do
10:                 if INTERSECT(primitive, ray) then
11:                     hitData ← (t, u, v, tID, ...)
12:                     INSERT(hitList, hitData)
13:             if SIZE(hitList) ≥ N_query then
14:                 UPDATEINTERVAL(t_far)
15:             continue
16:         far ← FARCHILD(node)
17:         near ← NEARCHILD(node)
18:         PUSH(travStack, far)
19:         PUSH(travStack, near)
20:     return hitList
```

Algorithm 1: Node culling multi-hit BVH traversal. *This algorithm, which is very similar to standard stack-based BVH traversal, accepts/rejects node intersections based on the currently valid ray interval (line 6), counts the number of valid ray/primitive intersections and tracks the distance to the farthest valid hit point, t_{far} (line 12), and updates the currently valid ray interval (lines 13–14), to begin culling nodes for which $t_{enter} > t_{far}$ once $N \geq N_{query}$ hit points have been gathered.*

Importantly, node culling multi-hit BVH traversal is very similar to standard stack-based BVH traversal, which suggests that an implementation with only user-level code is feasible: appropriate callback mechanisms need only be exposed to implement this algorithm; all other traversal-related operations remain the same. Likewise, node culling multi-hit BVH traversal exposes opportunities for early-exit despite lack of ordered traversal in a BVH. Early-exit

is a key feature of first-hit BVH traversal and of buffered multi-hit traversal in acceleration structures based on spatial subdivision; we thus expect significant performance impact for multi-hit traversal in a BVH for cases in which users request fewer-than-all hits.

2.2. Implementation

We now require a means to implement Algorithm 1. We dismiss direct implementation in a traversal routine specific to multi-hit, as this approach runs counter to our primary goal: to support high performance multi-hit ray traversal in a BVH with low development and maintenance costs. This restriction is important in production environments, where minimizing the effort required to design, implement, and, ultimately, maintain performance- and memory-efficient BVH data structures is critical: we simply cannot afford to increase the already-large maintenance burden imposed by production ray tracing engines. We thus highlight two callback-based mechanisms to implement node culling multi-hit BVH traversal: intersection and traversal callbacks (TCBs).

In addition to reducing development and maintenance burden, callback-based implementations promote flexibility: mechanisms supporting node culling multi-hit BVH traversal can also be leveraged for other rendering tasks. Widely used ray tracing engines such as Embree and OptiX already support a form of ICBs that are used for purposes other than multi-hit ray tracing, and we expect TCBs to provide a similarly flexible API feature beyond the node culling algorithm.

Intersection callbacks. ICBs are an obvious choice: Algorithm 1 suggests a combination of per-intersection operations (for counting hit points and tracking t_{far}, line 12) and per-leaf operations (for updating ray intervals, lines 13–14), but implementation with only per-intersection operations would suffice if we accept the overhead of executing ray interval update operations more often than strictly necessary in favor of a straightforward, already-supported implementation mechanism.

However, in an ICB-based multi-hit implementation [AGGW15], the interval over which intersections are considered valid is $[\varepsilon, t_{near} = t_{max}]$, where $t_{max} \geq t_{exit}$, the distance at which the ray exits the root-level bounding volume. This interval is required to ensure that all nodes are visited during traversal and, as a result, that all hit points are gathered.

Once initialized, this interval does not—and cannot—change during traversal due to restrictions on ICBs imposed by current ray tracing APIs. For example, the multi-hit traversal ICB for Embree simply saves data corresponding to each valid hit point in a per-ray data structure and rejects the intersection to continue traversal. In this *reject-intersection* case, the original value of t_{near}—that is, t_{max}—is restored by a post-ICB routine automatically invoked by Embree and inaccessible to the client.

As a result, Algorithm 1 cannot be implemented entirely in user-level code with currently available ICB mechanisms; either ICB restrictions must change or another mechanism must be exposed to accomplish the task at hand. Thus, in Section 3, we evaluate both an unconstrained ICB (uICB) mechanism that permits ray interval updates from within the callback itself and TCBs as possible implementation alternatives.

Traversal callbacks. As noted, Algorithm 1 suggests a combination of per-intersection and per-leaf operations. Together with currently available ICB mechanisms, the addition of TCBs represents an alternative approach to enable node culling implementation with only user-level code.

Just as ICBs enable client applications to inject arbitrary logic to be executed whenever a valid ray/primitive intersection is found, TCBs enable clients to inject arbitrary logic to be executed on a per-node basis during traversal of an acceleration structure.

Two concerns arise immediately with the introduction of this mechanism: First, what is the performance impact of introducing per-node operations when the corresponding feature is unused? That is, how much does exposing this API feature degrade performance in typical rendering scenarios relative to performance in these scenarios without support for TCBs? Second, what is the maintenance impact of introducing per-node operations? That is, how much effort is required to develop and maintain the API mechanism by which user-defined per-node operations are exposed?

Answers to both questions depend on the combination of programming language, compiler, and existing feature set of any ray tracing API in which TCBs are ultimately exposed. However, our experience with the experimental reference implementation highlighted in Section 3 shows that TCBs can be exposed with no additional performance overhead in cases where the feature is unused. Users assume responsibility for performance of operations injected during traversal via TCBs, but this situation is no different than the responsibility assumed for operations injected during primitive intersection via ICBs. From a performance perspective, then, TCBs are in no way fundamentally different than ICBs.

Likewise, introduction of TCBs impacts only BVH traversal, not construction. This point, though obvious, is important in practice: the significant effort required to develop and maintain performance- and memory-efficient BVH data structures remains intact. In this respect, too, TCBs are not fundamentally different than ICBs, and therefore offer a compelling implementation alternative for the node culling algorithm.

In either case, clients need simply define appropriate user-level callbacks and invoke the engine-level BVH traversal function to implement multi-hit BVH traversal with node culling. Importantly, these approaches implement the node culling algorithm without any code specific to multi-hit in the engine-level traversal function. Specifically, support for unconstrained per-intersection ICBs or per-node TCBs is sufficient to implement multi-hit ray traversal using a standard—and possibly architecture-specific or otherwise optimized—BVH traversal function. Intersection and traversal callbacks thus offer low-overhead, minimally invasive, and—ultimately—flexible API mechanisms to implement node culling multi-hit BVH traversal.

3. Results

To understand the potential impact of node culling multi-hit BVH traversal, we investigate performance in an experimental reference implementation using eight scenes of varying geometric and depth complexity rendered from the viewpoints depicted in Figure 2.

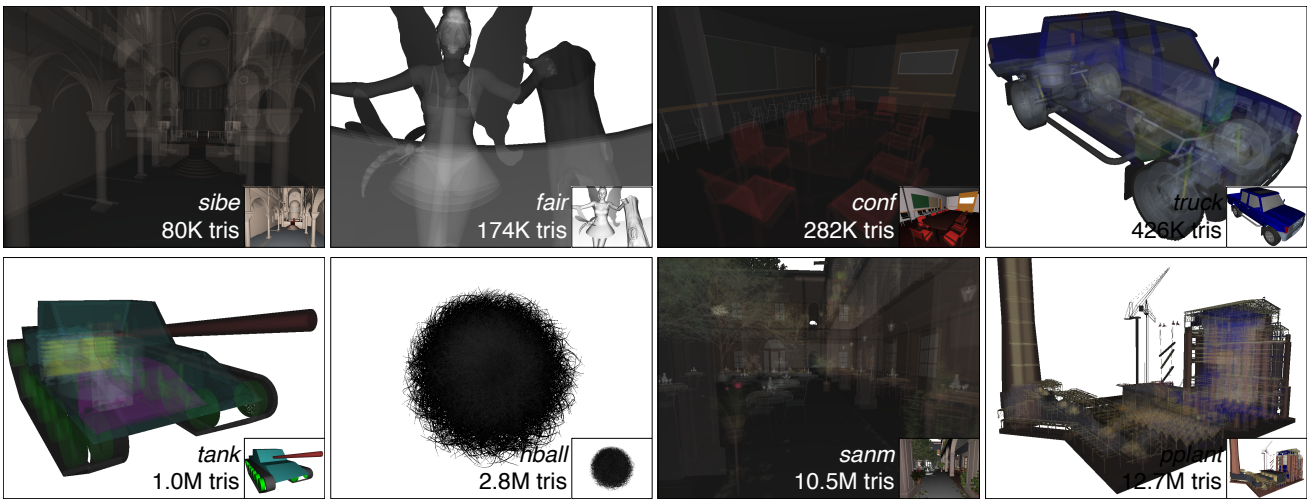

Figure 2: Scenes used for performance evaluation. *Eight scenes of varying geometric and depth complexity are used to evaluate the performance of node culling multi-hit BVH traversal. First-hit visible surfaces hide significant internal complexity in many of these scenes, making them particularly useful in tests of multi-hit traversal performance.*

3.1. Reference Implementation

We are primarily interested in multi-hit performance using node culling relative to naive multi-hit ray traversal, so we evaluate these algorithms in a reference implementation that supports both ICBs and TCBs.

Specifically, we use C++11 lambdas to implement the per-intersection operations (Algorithm 1, line 12) and the per-leaf operations (Algorithm 1, lines 13–14) required by multi-hit BVH traversal with node culling. As noted, these callback-based approaches implement the node culling multi-hit algorithm without any code specific to multi-hit in the engine-level traversal function.

We opt for clarity and simplicity in the reference implementation rather than aggressive optimization to maximize absolute performance on the test platform. As such, the renderer is both scalar and single-threaded.

Likewise, we implement the progressive insertion sort variant of multi-hit traversal [AGGW15] using `std::multiset` as the underlying data structure for collecting per-ray hit point data—here, too, we opt for clarity and simplicity rather than aggressive optimization.

Finally, node culling multi-hit traversal does not impact (and is not impacted by) BVH construction, so we use a readily available surface area heuristic construction algorithm [WBS07]; however, relative performance with and without node culling translates directly to BVH data structures resulting from other construction algorithms.

In the following experiments, ICB-based node culling uses the uICB mechanism described above. As noted, such an implementation is not possible with the current ICB mechanisms exposed by Embree and OptiX, but represents an alternative to TCB-based node culling for multi-hit traversal in engines that employ BVHs.

Similarly, we evaluate two TCB-based node culling alternatives:

- **Every-node TCBs (eTCBs)**, which are applied to every node processed during traversal; in this context, a node is *processed* if it is popped from the traversal stack.
- **Leaf-node TCBs (lTCBs)**, which are applied only to leaf nodes processed during traversal; here, a leaf node is *processed* if it is successfully intersected during traversal.

With eTCB-based node culling, only every-node TCBs implement per-leaf operations required by Algorithm 1, whereas lTCB-based node culling uses only leaf-node TCBs for these operations. In both TCB-based implementations, ICBs implement the per-intersection operations required by Algorithm 1.

3.2. Multi-Hit Ray Tracing Performance

We now consider the impact of node culling on multi-hit performance in several scenarios representative of those for which multi-hit traversal might be used in production. Specifically, we consider the values of N_{query} outlined in Table 1 across various combinations of node culling implementations and our test scenes.

For each test, we render a series of 5 warm-up frames followed by 50 benchmark frames at 1024×768 pixel resolution using visibility rays from a pinhole camera and a single sample per pixel. Results are averaged over the 50 benchmark frames.

Find-first-intersection. First, we measure the impact of node culling when specializing multi-hit ray traversal to first-hit traversal. Figure 3 compares performance in seconds per frame of finding the nearest hit using a standard first-hit traversal implementation against finding the nearest hit using both naive multi-hit traversal and multi-hit traversal with lTCB-based node culling (with $N_{query} = 1$). The advantage of node culling is clearly evident in

scene	max # hits	N_{query} 10%	30%	70%
sibe	22	2	7	15
fair	26	3	8	18
conf	21	2	6	15
truck	86	9	26	60
tank	38	4	11	27
hball	170	17	51	119
sanm	113	11	34	79
pplant	139	14	42	97

Table 1: Ray/primitive intersection characteristics for our test scenes. *Here, the table reports the maximum number of valid ray/primitive intersections encountered along any one ray, as well as the number of requested hit points that comprise 10%, 30%, and 70% of the corresponding maximum. Scenes clearly vary not only in geometric complexity, but in depth complexity as well.*

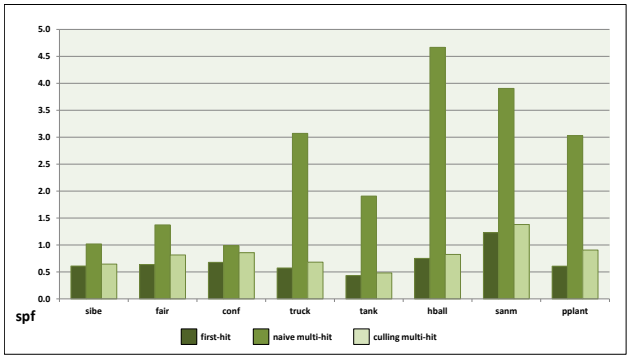

Figure 3: Performance of standard first-hit and multi-hit variants for *find-first-intersection. Here, the graph compares performance in seconds per frame (spf) among standard first-hit traversal, naive multi-hit, and lTCB-based node culling when* $N_{query} = 1$. *On average, node culling improves rendering performance to within about 20% of first-hit performance, whereas naive multi-hit requires more than a factor of* $2.5\times$ *of first-hit performance.*

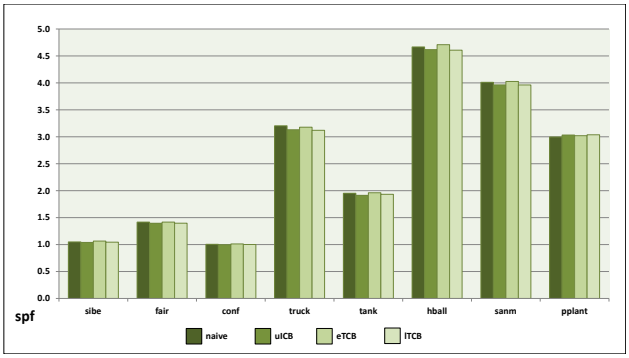

Figure 4: Performance of multi-hit variants for *find-all-intersections. Here, the graph compares multi-hit performance in seconds per frame (spf) between naive multi-hit and node culling when* $N_{query} = \infty$. *Differences in performance are within the expected variability among trials for the* find-all-intersections *case.*

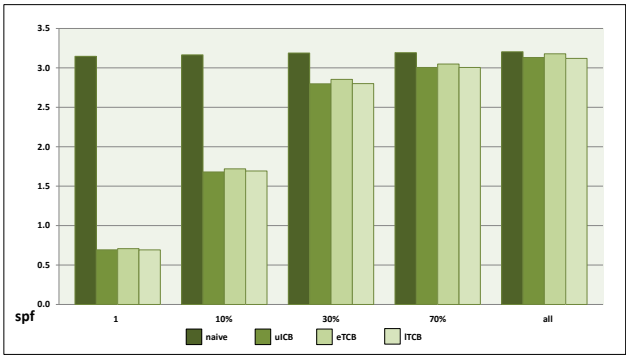

Figure 5: Multi-hit performance in the *truck* scene. *Here, the graph compares multi-hit performance in seconds per frame (spf) among multi-hit implementations for various values of* N_{query}. *For this scene, node culling improves rendering performance by just less than 40% (on average) relative to naive multi-hit.*

this case: frame time with node culling multi-hit BVH traversal approaches that of standard first-hit traversal (to within about 20%, on average), whereas frame time with naive multi-hit traversal is more than a factor of $2.5\times$ greater than that with first-hit traversal (on average) for our test scenes.

Find-all-intersections. Next, we measure the impact of node culling using multi-hit ray traversal to implement all-hit traversal ($N_{query} = \infty$). Figure 4 compares performance in seconds per frame when using each multi-hit variant to gather all hit points along a ray. Not surprisingly, node culling offers no significant advantage in this case, and differences in performance are within the expected variability among trials.

Find-some-intersections. Finally, we measure multi-hit performance for the values of N_{query} enumerated in Table 1. The *find-some-intersections* case is perhaps the most interesting, given that multi-hit traversal cannot be specialized to either first-hit or all-hit algorithms in this case. For brevity, we examine only results for the *truck* scene here; however, we present multi-hit performance (in average seconds per frame), as well as the number of ray/node intersection tests, node traversal operations, and ray/primitive intersection tests, for each test scene in the supplemental data accompanying this paper. Generally speaking, we observe trends present in results for the *truck* scene in results for the other scenes as well.

The impact of node culling varies directly with the number of requested intersections. In particular, Figure 5 shows that as $N_{query} \to \infty$, the impact of node culling on performance improvement relative to naive multi-hit decreases from nearly a factor of $3\times$ when $N_{query} = 1$ to effectively zero when $N_{query} = \infty$. The relative advantage of node culling is clearly evident when users request fewer-than-all hits: fewer requested hits lead to greater performance gains.

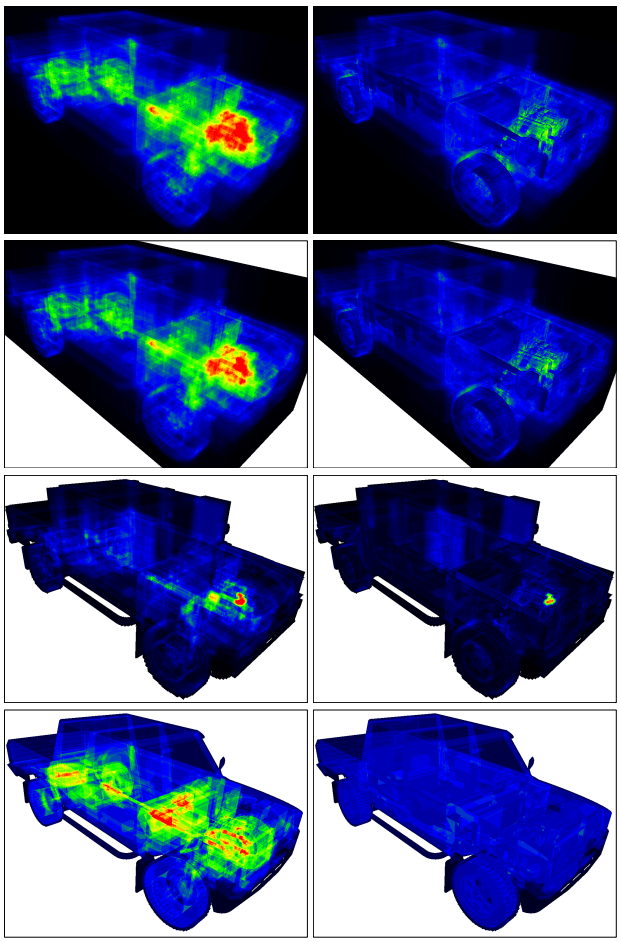

Figure 6: Efficiency visualization. *Heatmap visualizations using the color scale depicted in Figure 1 reveal that far less work must be done per-ray when using node culling (right) compared to using naive multi-hit traversal (left). From top-to-bottom: number of ray/node intersection tests, node traversal operations, ray/primitive intersection tests, and valid ray/primitive intersections identified before satisfying the multi-hit query ($N_{query} = 9$).*

3.3. Discussion

Results show that node culling multi-hit traversal significantly improves performance relative to naive multi-hit in cases when users request fewer-than-all hits. The factors leading to improved performance are crystallized when viewing the heatmap visualizations in Figure 6: node culling reduces the number of ray/node intersection tests, node traversal operations, ray/primitive intersection tests, and valid primitive intersections found before correctly satisfying multi-hit queries. Significantly less works translates directly to significantly better performance.

We also observe that, when averaged across values of N_{query}, node culling using lTCBs offers a slight but measurable advantage over the other node culling implementations. When comparing these approaches, the only source of difference is per-intersection execution of per-leaf operations (Algorithm 1, lines 13-14) with uICBs, and per-node execution of these operations with eTCBs. The operations themselves are just not costly enough to impose significant overhead, even when executed more often than strictly necessary. This result suggests a certain degree of flexibility in the choice of mechanism—uICBs, eTCBs, or lTCBs—an engine could expose to support multi-hit BVH traversal with node culling.

The current limitation inherent to our results is lack of support for vectorized traversal in the experimental reference implementation. BVHs are well-suited to vector processing via packet tracing [WBWS01, WBS07]. Although results demonstrating the significant performance differences between node culling and naive multi-hit BVH traversal are not tightly coupled to the underlying BVH data structure, complexities introduced by packet-based traversal may affect results in an unintuitive way. Thorough investigation of packet-based node culling multi-hit BVH traversal thus represents an interesting avenue of future work.

4. Conclusion

We explore faster multi-hit ray traversal in a BVH using a new node culling multi-hit BVH traversal algorithm. We evaluate the potential impact of our algorithm in an experimental reference implementation with support for intersection and traversal callbacks. These callback mechanisms permit implementation of node culling multi-hit BVH traversal entirely with user-level code, which is an important characteristic in production environments. Results show that node culling multi-hit BVH traversal offers potentially significantly improvement in performance relative to naive multi-hit in a BVH for cases in which users request fewer-than-all hits.

Acknowledgments

We thank Jefferson Amstutz (Intel Corporation) for many constructive discussions regarding multi-hit ray traversal generally, and in a BVH specifically.

References

[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), 72–90. 2, 3, 4

[GNK14] GRIBBLE C., NAVEROS A., KERZNER E.: Multi-hit ray traversal. *Journal of Computer Graphics Techniques 3*, 1 (2014), 1–17. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH K., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGRUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2010) 29*, 4 (2010). 2

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (January 2007), 6. 4, 6

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (September 2001), 153–164. 6

[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 33*, 4 (July 2014), 143:1–143:8. 2