

Ray Specialized Contraction on Bounding Volume Hierarchies

Yan Gu Yong He Guy E. Blelloch

Carnegie Mellon University

Abstract

In this paper we propose a simple but effective method to modify a BVH based on ray distribution for improved ray tracing performance. Our method starts with an initial BVH generated by any state-of-the-art offline algorithm. Then by traversing a small set of sample rays we collect statistics at each node of the BVH. Finally, a simple but ultra-fast BVH contraction algorithm modifies the initial binary BVH to a multi-way BVH. The overall acceleration for ray-primitive testing is about 25% for incoherent diffuse rays and 30% for shadow rays, which is significant as a data structure optimization. Similar results are also presented for packet ray tracing, and for Quad-BVHs the improvement is 10% to 15%. The approach has the advantages of being simple, and compatible with almost any existing BVH and ray tracing techniques, and it requires very little extra work to generate the modified tree.

Keywords: ray tracing, bounding volume hierarchy, ray distribution, data-driven optimization

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

1. Introduction

The pursuit of high performance in ray tracing systems has led to a rapid evolution in acceleration structures, such as KD-trees and bounding volume hierarchies (BVHs), and in architectural optimizations for these structures on many-core CPUs and modern GPUs. BVH techniques are widely used in a number of ray tracing engines [PBD*10, WWB*14] because of its simplicity, flexibility and construction parallelism [KKW*13].

State-of-the-art ray tracers usually build BVHs offline—after the model is loaded, the BVH is fully constructed using surface-area heuristics that approximates the probability of a ray intersecting a volume. Building an optimal BVH based on these heuristics is believed to be NP-hard [Hav00, PGDS09], so in practice a BVH is constructed using a variety of greedily approaches (top-down, bottom-up, post-optimization, spatial splits, or a combination of them). These approaches trade off among BVH quality, construction speed and parallelism. However, all of these approaches assume that rays are distributed **uniformly**, approaching from all directions at equal frequency. Therefore, the constructed BVH depends only on the mesh geometry, without consideration of the actual ray distribution in a scene.

BVH quality can be greatly improved by considering a specific ray distribution. However, as the actual ray distribution depends heavily on run-time settings such as the positioning of camera and lights, the BVH shall be updated whenever these parameters change. This requires any ray-distribution-aware system to collect ray distribution and reconstruct a new BVH in very short amount of time. Previously, [BH09, FLF12] have proposed techniques to incorporate ray distribution into BVH construction, but all of these algorithms reconstruct the entire BVH to adapt for new ray distribution every frame, offsetting the benefit of a more optimized tree. Meanwhile, their BVH qualities are also no better than the newest offline algorithms [GHFB13, KA13].

In this paper, we propose an algorithm to leverage ray distribution with very little overhead. Our approach (refer to RDTA, in Section 4.2.2) starts with an arbitrary binary BVH tree that can be built very quickly using any of existing methods, and restructures the initial tree into a multi-way BVH optimized for the given ray distribution. The ray distribution is collected through a fast and light-weight process that involves tracing a sample set of input rays to estimate how often each node is visited. We show that our new BVHs have achieved 25-35% performance improvement in terms of both the number of ray-box tests and actual rendering time (in-

cluding both tree-restructuring and ray-tracing process) over the standard SAH-based binary BVHs, and a 10% to 15% speedup (refer to CBTC-RD in Section 4.4) on Quad-BVHs over CBTC-SA [WBB08] in ten typical test scenes.

Our method is compatible with almost all of the existing ray tracing techniques: packet traversal [WSBW01], treelet reorganization [AK10], reordering for expensive computation materials [LKA13], multi-branch BVH traversal [WB-B08, DHK08], etc. The algorithm can be easily integrated into any ray tracing systems with very little change to existing system architecture.

2. Related Work

Top-down BVH construction using the Surface Area Heuristic (SAH) [GS87] is widely considered as the “gold standard”, because it results relatively high BVH quality (in reducing ray-box testing in traversal). Further research focused on accelerating construction time by approximation and better parallelism. We refer the readers to [Wal07] and [GP-BG11] for excellent summaries for modern approaches.

While some GPU-based BVH construction algorithms [LGS*09, PL10, GPM11, Kar12] abandon SAH to trade off BVH quality for faster BVH construction of deformable objects, other algorithms that attempt to improve BVH quality use SAH to guide tree building. Agglomerative clustering [WBKP08] builds BVH bottom-up, and an approximate version [GHFB13] greatly reduces the construction time. Post-optimization BVH constructions [Ken08, BHH13] start with an initial BVH, and iteratively modifies it by applying rotation, deletion and re-insertion operations. Alternatively, BVH constructions with spatial splits [SFD09, PGDS09] build BVHs top-down, and consider object and spatial splits together. Karras et al. [KA13] efficiently combine post-optimization and spatial splits. In summary, significant research has done on improving the BVH quality or construction speed. Our algorithm relies on these state-of-the-art techniques in constructing an initial BVH, and further improves its quality using a BVH contraction algorithm.

A variety of papers attempted to combine ray distribution into BVH construction. Early works [Hav00, HM08] (not only with BVHs) have sought to analytically modify the heuristic for common non-uniform ray distribution. Later works [BH09, FLF12] directly compute new heuristics by testing a few sample rays, generated by a first-pass of ray tracing. However, these solutions suffer from several unsolved problems that make it less effective. Firstly, an whole BVH is constructed when camera position or view direction is changed, which is inefficient since their constructions are slow and not compatible with other offline algorithms. Secondly, insufficient improvement (almost none for non-shadow rays) is acquired comparing to SAH [GS87]. Nabata et al.’s idea [NIDN13] is similar to Feltman et al.’s [FLF12] and accelerate divide-and-conquer ray trac-

ing (DACRT [Mor11]), but it is hard to integrate to standard ray tracer since DACRT uses a completely different pipeline, and also its performance is not competitive with the standard approaches. In conclusion, the extra work spent in these algorithms is hard to trade off for the improvement in traversal, comparing with the best offline BVH construction algorithms (e.g. [GHFB13, KA13]). We will show how our new algorithm overcomes these problems in the this paper.

It is also worth to be pointed out that accurately analyzing and predicting the performance of a BVH is an interesting but open problem [AKL13]. Although we have not given an exact answer in this paper, some algorithmic analysis could provide useful insights to this problem.

The idea of skipping ray-box testing when traversing a BVH has appeared in previous papers. CPU-based ray tracing techniques [WBB08, DHK08] tried to utilize SIMD instructions. Dammertz et al.’s technique [DHK08] contracts every other level to generate a 4-way BVH. Wald et al.’s method [WBB08] contracts BVH by surface area, which is a special case in our algorithm. Nabata et al.’s approach on DACRT [NIDN13] decides whether a packet of rays skips ray-box testing or not by testing a sample set. Unfortunately, such an approach is incompatible with a standard ray tracer. They also use a similar criterion as Eqn. 2, but we use it in a different way: they implicitly generate a BVH every time for a set of rays and use this criterion to skip testing for these rays, while we combine it with other heuristics (e.g. surface area and ray distribution) to generate general high-quality BVHs that can be used to trace any rays.

3. Binary BVH Traversal

For completeness, we review the standard BVH ray-traversal algorithms. Algorithm 1 shows the pseudocode for first-hit ray traversal, which returns the first ray-scene intersection (used for primary, specular-reflection, ambient-occlusion and diffuse inter-reflection rays). Algorithm 2 provides the pseudocode for any-hit ray traversal, and its goal is to decide if a ray occluded by any scene objects, usually called by shadow ray queries. Notice that instead of running ray-box test immediately in line 4 in Algorithm 1, the test is postponed to the next level in any-hit traversal (line 1 in Algorithm 2). The reason is that, when an intersection is found in any-hit traversal, the function will return “yes”, and the rest ray-box tests can be skipped and saved. Efficient implementations of these algorithms can be found in [WBS07] and [AL09].

4. The BVH Contraction Algorithm

4.1. The Definition of BVH Contraction

In this section, we propose a novel BVH contraction algorithm that reduces unnecessary ray-box tests. To begin with, we first cover some basic concepts.

Algorithm 1: FirstHitTraverse(R, N) // for primary and diffuse rays

Input: Ray R , BVHNode N

Output: First intersection (can be empty)

```

1 if  $N.isLeaf$  then
2   | Check all triangles;
3 else
4   | Ray-box tests for all children;
5   | foreach intersected child  $C$  in front-to-end order do
6     |   | if  $Distance(R, C) < R.FirstInter$  then
7       |   |   | FirstHitTraverse( $R, C$ );
8   | return first intersection;
    
```

Algorithm 2: AnyHitTraverse(R, N) // for shadow rays

Input: Ray R , BVHNode N

Output: Boolean hit

```

1 if not Ray-box tests( $R, N.bbox$ ) then
2   | return false;
3 if  $N.isLeaf$  then
4   | Check all triangles;
5 else
6   | foreach child  $C \in N.children$  do
7     |   | if AnyHitTraverse( $R, C$ ) then
8       |   |   | return true;
9   | return false;
    
```

We define the set of nodes whose bounding box is actually tested to the input ray (or packet) in Algorithms 1 and 2 to be a “traversed subset-tree” of that ray (or packet). Ray-box tests of nodes in a “traversed subset-tree” can be classified into two categories:

- “Pass-test”: at least one child or primitive (i.e. triangle) of this node is explored, which indicates that line 7 in FirstHitTraverse is called, or test on line 1 in AnyHitTraverse returns “hit”. This node is either an interior node of the traversed subset-tree, or a leaf node that directly contains primitives.
- “Prune-test”: none of its children or primitives are further traversed. This node is a leaf node in traversed subset-tree.

Notice that the categorization has no relationship with the result of a ray-box test (either a hit or a miss).

BVH traversal is usually considered to cost logarithmic time for a first-hit ray query. This is because when a Prune-test occurs, the whole subtree is skipped. Hence, the total number of ray-box tests are much less than the number of nodes in a BVH. However, a Pass-test cannot provide any extra information more than “there might be an intersection in this subtree”. Thus, the key observation for faster BVH traversal is to reduce the number of Pass-tests. For a shadow ray query, traversal can also terminate when an occlusion is found, and corresponding optimization can be found in Section 4.2.2.

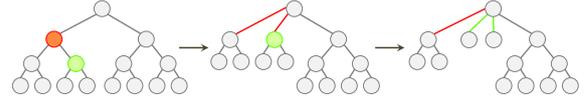


Figure 1: Two cascade node-“contraction” operations: first the red node, then the green node are removed from the tree.

Algorithm 3: BVHContract(N)

Input: BVHNode N

```

1 if  $N.isLeaf$  or StopCriterion( $N$ ) then
2   | return;
3 else
4   |  $S \leftarrow \{N.left, N.right\}$ ;
5   | while ContractionCriterion( $S$ ) do
6     |   |  $s \leftarrow Select(S)$ ;
7     |   |  $S \leftarrow S - \{s\} + \{s.left + s.right\}$ ;
8   |  $N.child \leftarrow S$ ;
9   | foreach  $s \in S$  do
10    |   | BVHContract( $s$ );
    
```

We now analyze the traversed subset-tree of a standard binary BVH in FirstHitTraverse. The traversed subset-tree is a binary tree (two children are tested together in line 4), which means that the number of interior nodes of this tree always equals to the number of leaves minus one. Note that the tests in all interior nodes are Pass-tests, which are almost half of the overall tests. Ideally, if we can skip all of these interior nodes and directly check their children (recurse if they are still interior nodes), half of the ray-box tests can be saved.

Since the traversed subset-trees varies for different rays (or ray packets), no oracle exists to determine a test to be Pass-test or Prune-test ahead of time. We propose a novel statistical method to produce an approximate prediction of the category of a ray-box test, in order to reduce the number of Pass-tests. This is done by the node “contraction” operation:

Definition 1. A “contraction” operation for an interior BVH node is to hoist all its children to its parent, and remove this node from the tree.

An example of contraction is shown in Figure 1. In this case, if a ray hits both colored nodes, the two Pass-tests are eliminated when traversing the contracted BVH.

To decide whether a BVH node should be contracted or not, we define a cost function $\delta(N)$ to return cost of using a contracting node N versus keeping the original BVH unchanged. If a Pass-test happens, $(1 + n_{N.child})C_B$ is paid where C_B is the cost of a ray-box testing and $n_{N.child}$ is the number of children for node N ; otherwise, C_B is spent and a Prune-test happens. If node N is contracted, $n_{N.child} \cdot C_B$ is the cost to test all children. Hence, if the probability for Pass-test in node N is α_N ,

$$\begin{aligned} \delta(N) &= n_{N.child} \cdot C_B - (\alpha_N (1 + n_{N.child}) + (1 - \alpha_N)) C_B \\ &= ((1 - \alpha_N) \cdot n_{N.child} - 1) \cdot C_B \end{aligned} \quad (1)$$

A contraction operation holds beneficial if $\delta(N) < 0$, which is equivalent to:

$$\alpha_N > 1 - \frac{1}{n_{N.child}} \quad (2)$$

We propose a cascade BVH contraction algorithm (Algorithm 3) that greedily optimizes a BVH by applying node contraction guided by the cost function. The basic idea is that for each node, check its children to see whether there exist a child skipping whose ray-box test is beneficial (as defined by the cost function), and contract that node if so. Then the algorithm iteratively check again until no such child is found. We keep a set S to be the candidate children of node N ; the function *ContractionCriterion* (line 5) guides the process of cascade contraction and decide whether there exist nodes to be further contracted; in each iteration, one child node s is selected from S by function *Select*; finally, function *StopCriterion* determines when to terminate the recursion so we only modify the top levels of the BVH (which are a small fraction of the entire BVH but has large impact on ray tracing performance) for faster contraction process (see an example in Section 4.2.2).

Although contractions can be operated in any order, we implement our algorithm in a top-down approach, which has two advantages: first, when the iteration finished after line 7, all children are kept in a continuous memory, which optimizes data locality when traversing; second, function *StopCriterion* is able to control the number of nodes to be executed, which guarantees a fast modification. Note that the pseudocode in Algorithm 3 is a high-level abstraction that parameterizes a set of functions and simplifies the description in Section 4.2. In practice, a careful implementation of BVHContract takes linear time regarding to the number of visited nodes, since each node N is only checked once: if $\delta(N) < 0$ then it will be contracted immediately; otherwise, they node will stay and never be checked again.

A combination of *ContractionCriterion*, *Select* and *StopCriterion* is called the parameters of BVH contraction.

4.2. Parameters of BVH Contraction

Now we introduce the way to select appropriate parameters for better ray-scene test performance.

Basic data structure intuition indicates that an ideal binary tree data structure will split the range of space in half each time, and thus for each check, the probability to traverse each subtree is approximately the same and close to half, which maximizes the information gain (entropy) for this check. Nevertheless, this property does not always hold for BVHs built by many commonly-used algorithms for the following two reasons:

- **The bias in the data structure (structural imbalance).** Objects do not distribute in the space evenly, and a common BVH construction heuristic, like SAH, usually bi-

sects the space so that the smaller subspace contains more primitives and vice versa. This can lead to a large difference in the probability of visiting each child.

- **The bias in the queries (ray-distribution imbalance).** Even for incoherent rays, like diffuse inter-reflection rays with multiply bounces from a given camera position, parts of the scene are much harder to reach than the rest. Moreover, this imbalance is very hard to measure analytically.

4.2.1. Surface-Area Guided Contraction

A common way (i.e. [WBB08]) to measure the probability to traverse a child is by the ratio of the surface area of the child to that of the parent. This is based on the assumption that rays are infinitely long and their distribution is completely random. Therefore, the corresponding parameters (surface-area guided tree contraction, SATC) for Algorithm 3 are:

$$\alpha_N = \frac{SA(N)}{SA(N.parent)}$$

$$ContractionCriterion(S) = \begin{cases} True, & \exists s \in S, \delta(s) < 0 \\ False, & \forall s \in S, \delta(s) \geq 0 \end{cases}$$

$$Select(S) = any\ s \in S\ s.t.\ \delta(s) < 0$$

$$StopCriterion(N) = False$$

where $SA(N)$ is the surface area of the bounding box for node N . SATC only captures the structural imbalance of BVH itself caused by the heuristic.

Since BVHContract is executed top-down and $n_{N.child}$ in Eqn. 2 is unknown at the moment, we empirically set $1 - 1/n_{N.child}$ to be 0.6. (A bottom-up approach with the exact threshold for α_N provides similar BVH quality.) Therefore, the checking for $\delta(s) < 0$ is equivalent to $\alpha_N > 0.6$, so that a BVH node with surface area larger than 0.6 times its parent's surface area will be contracted.

SATC does not involve any information of ray distribution, but it is a good example to understand the BVH contraction algorithm, and useful in further analysis in the experiment section. Experimental results using SATC can be found in Table 1 and discussion in Section 5.2.

4.2.2. Ray-Distribution Guided Contraction

To explore the bias in queries, we introduce our ray-distribution guided BVH contraction. Feltman et al. [FLF12] found that a small set of sample rays are able to sufficiently represent a ray distribution. However, their implementation stores the sample rays and checks them with all possible bounding boxes in BVH construction, which makes it very inefficient. Instead, our algorithm focuses on the probability of traversing both children directly. We keep a counter named "visitCount" in each BVH node, indicating the number of times this node is traversed by sample rays. Every time a Pass-test occurs on a node, its counter will increase by one.

The new structure to store a BVH node is shown below,

which has a total size of 32 bytes, so a standard CPU cache line fits two BVH nodes. Four elements including number of children (4 bits, since the new generated BVH nodes are forced to have at most 16 children), leaf flag (1 bit), contracted flag (1 bit, for implementation purpose) and pointer to the first child (26 bits, supporting at most 64M triangles), are stored together using 32 bits.

```

struct Node {
    BoundingBox  bbox;
    Int          numChild   :4;
    Boolean      isLeaf     :1;
    Boolean      ContrFlag  :1;
    Pointer      ptr        :26;
    Int         visitCount;
}

```

Using the statistical data provided by the counter in each BVH node, we now estimate the probability that a ray-box test of a node is a Pass-test to be the ratio of the numbers in its parent's counter and its own counter. For example, if the counter for node A is 400 and the counter for A 's parent is 500, then the probability that a Pass-test for node A is 80% for this ray distribution.

It is worth to point out that this method provides a much more accurate estimation for tree imbalances. An extra usage of the counters in the tree nodes is to constrain the contraction process on the "important" part on the BVH. This is because the importance of a node is proportional to the number in the counter, and a good parameter of contraction is able to neglect "less" important parts and only reconstruct a small fraction of tree nodes (usually a few thousands of tree nodes). Therefore, the time spent for this whole procedure can be negligible (no more than 1ms).

The new parameters (aka. ray-distribution guided tree contraction, RDTC) for Algorithm 3 are (the rest is the same as previous):

$$\alpha_N = \frac{\text{visitCount}(N)}{\text{visitCount}(N.\text{parent})}$$

$$\text{StopCriterion}(N) = \text{visitCount}(N) < t$$

where t is a constant threshold. RDTC captures both structural and ray-distribution imbalances by directly acquiring the probability to different subtrees. In practice, sampling about 0.1% – 0.5% of rays or tracing a few thousand pixels are sufficient for high-quality BVH contraction.

4.2.3. Pipeline for Sampling-Based Ray Tracing

Since our new algorithm requires statistical information from sample rays, the process of ray sampling needs to be integrated, and the ray tracing pipeline needs to be slightly modified. The new procedure is:

1. Create a BVH using any BVH construction algorithm.
2. Pre-render a small sample of pixels and track the counters for each node.

3. Apply BVHContract (Algorithm 3).

4. Render the rest of pixels.

The extra work introduced in step 2 and 3 has negligible impact on run-time performance. In step 2, the algorithm traces and stores the intersection result of a sub-set of input rays, which will be traced anyway by the system. In some parallel systems, the overhead of atomic operations on counters can be high. However, an approximate estimation is sufficient for our algorithm, and we find that removing atomicity boosts performance without impacting output BVH quality. For GPU implementations that atomic operations are expensive, an alternate solution is to trace less sample rays. Based on our experiment, tracing only 0.02% of rays to be samples only affect the overall performance for less than 2% comparing to the setting tracing 0.4% sample rays. In step 3, our RDTC algorithm modifies only a few thousand BVH nodes. This extra cost is very small compared to the entire work of tracing hundreds of millions of rays. In Section 5.3, we will show that tracing the new BVH in step 4 is also efficient.

4.3. Ray-Distribution Order for Shadow Ray Traversal

Some previous papers [IH11, FLF12, NM14] tried to design specialized traversal order or BVH to accelerate shadow-ray traverse, since instead of finding the first intersection along the ray, their algorithms use some heuristics to decide the probability of having occlusions for a random ray in a subtree, and first traverse the subtree based on this priority.

In our approach, we already know an approximate probability for a ray to traverse all children. Thus the traversal order for shadow rays is to always test and traverse the subtree with larger number in node counter first, and keep this order for cascade contraction in BVHContraction.

4.4. Extension to Packet Ray Tracing and n -ary BVHs

To utilize the SIMD units, two major directions for better parallelism in ray tracing are ray parallelism (packet ray tracing) [WSBW01] and box-test parallelism [WB08, DHK08].

Extension to packet ray tracing is straightforward and no change to RDTC is required. The only difference is that the probability to traverse a certain child (α_N) will increase, because the node will be visited if any rays in the packet hit its bounding box. This will lead to better contraction results. However, in packet ray tracing, the ratio of execution time spending in ray-triangle testing increases, and our contraction will not reduce the cost in this part. Our experiment shows that the overall acceleration is almost the same compared to trace a single ray every time, and the detailed results are shown in Table 1.

n -ary ($n = 2^k, k = 2, 3$) BVHs, or Quad-BVHs when $n = 4$,

was proposed in [WBB08, DHK08], where multiple bounding boxes are tested simultaneously using vector units AVX and SSE on CPUs. To apply the new BVH contraction algorithm in this case, we only need to fix the maximum number of branches to a certain number (4 or 8).

Intuitively, nodes with a higher probability of being traversed should be contracted first [WBB08]. In our setting, this priority is indicated by higher Pass-test rate, provided by the ratio of numbers in the counters. We estimate the probability directly using the numbers in the counter if many (at least t) rays hit this node; otherwise, we use the surface area to estimate the probability. Hence, parameters for n -ary BVH (constant branches tree contraction, CBTC) is:

$$\alpha_N = \begin{cases} \frac{SA(N)}{SA(N.parent)}, & visitCount(N) < t \\ \frac{visitCount(N)}{visitCount(N.parent)}, & visitCount(N) \geq t \end{cases}$$

$$ContractionCriterion(S) = |S| < 2^k \text{ and } NI(S) \neq \emptyset$$

$$Select(S) = \arg \max_{s \in NI(S)} \alpha_s$$

$$StopCriterion(N) = False$$

whereas $NI(S)$ maps non-leaf nodes in S into a new set, and t is still a constant threshold as previously mentioned. It is interesting to point out that Wald et al.'s method [WBB08] generate the same BVH as CBTC if t is set to be $+\infty$. We name this special case as CBTC-SA because it only relies on surface area, and the rest setting as CBTC-RD, because ray distribution is integrated in the contraction process.

5. Evaluation

Due to the page limit, we provide a brief version of our experiment in this section. We refer the reader to the supplemental material for a comprehensive algorithm analysis.

5.1. Experiment Setup

The evaluation in our paper focuses on both the number of ray-box tests, and the actual execution time on a many-core CPU. The number of ray-box tests is a good indicator for BVH quality, because this number does not depend on details of the hardware or implementation, and can be easily reproduced. Since we cannot implement all of the state-of-the-art ray tracers on different platforms, we report our actual running time on a 40-core machine consists of four 10-core Intel E7-8870 Xeon processors (1066 MHz bus). Parallel implementations were compiled with CilkPlus, which is included in G++. Our ray tracing code borrows the ideas from some of the recent works [BWB08, Tsa09], but we implement it separately. The performance of our ray tracer is competitive since the test scenes we used are relatively complex, and the tracing speed for each scene is provided in the supplemental material. More implementation details and analysis can be found in Section 5.3. Notice that tree structure needs to be rearranged for tracing n -ary BVH to utilize the SIMD units.

We use 15 test scenes in our experiments, which contain significant scene-to-scene variations. Our method tends to reduce ray-box tests due to BVH imbalances in complex geometry models, so we mainly focus on 10 real-world scenes, which include: 3 widely used architectural models CONFERENCE, CRYTEK-SPONZA and SAN-MIGUEL; a complex building SODA-HALL to be rendered separately inside and outside; 2 city models ARABIC and BABYLONIAN from the Mitsuba distribution [Jak10] showing large spatial extends; and 3 game scenes TRAIN-STATION, EPISODE2 and WAREHOUSE from HalfLife2, with complex geometry. Experimental results for the other 5 scenes are given in the supplemental material.

We show the benefits of our method by studying the performance improvement based on starting with the BVH constructed by three different algorithms: a top-down full-sweep SAH build (short for SAH) [GS87], a bottom-up build using approximate agglomerative clustering with HQ parameters (short for AAC) [GHFB13], and a top-down build using spatial splits with default parameters (short for SBVH) [SFD09]. These algorithms generate high-quality BVHs using different approaches, so the evaluation results are representative. Renderings use 32 sample rays per pixel and one to several area light sources depending on scene complexity. More than 5 camera positions for outdoor scenes and 3-5 for indoor scenes are used, with the results averaged. We pre-render 1 pixel per 16×16 block in screen space, and use these sample rays to generate statistics on the BVH. Our experiment shows that the threshold t in *StopCriterion* in RDTC is insensitive, and in the experiments we use the maximum number of rays for a single sample pixel. More details are provided in the supplemental material.

In our experiment, we extensively use the “relative ratio” or “relative performance” to show the acceleration of our approach, and here it is defined as the total amount of work (number of ray-box tests or wall clock time) on the contracted BVH divided by that on the original binary BVH.

5.2. Scene-by-Scene Acceleration

To start with, we first analyze the improvement of performance by BVH contraction on different scenes. Table 1 compares the relative performance based on different parameters, with both number of ray-box tests (for SATC and RDTC) and wall clock time (for RDTC). The table also provides the tree imbalance, number of contracted nodes for RDTC, relative node depth, and average number of branches for new generated node. All data are generated by single ray tracing, but the running time for packet ray tracing is also provided, which shows a similar speedup.

The SATC heuristic (introduced in Section 4.2.1), which tries to avoid unnecessary ray-box tests caused by structural imbalance, can reduce the tests by up to 25% on diffuse rays and 45% on shadow rays (column SATC in “Rel. # ray-

Scene	Initial BVH type	# ray-box tests non-opt BVH diff / shad	Rel. # ray-box tests		Relative runtime			BVH imbalance		Contracted nodes		Rel. node depth	Ave. num. of branch / pass
			SATC diff / shad	RDTC diff / shad	Single diff / shad	Packet diff / shad	SATC	RDTC	num.	pct.			
	SAH	42.9 / 31.3	0.75 / 0.72	0.71 / 0.67	0.74 / 0.69	0.76 / 0.70	0.51	0.58	1.9K	1.1%	0.31	5.2 / 1.3	
	AAC	35.0 / 24.9	0.76 / 0.72	0.73 / 0.71	0.76 / 0.75	0.79 / 0.77	0.24	0.39	1.8K	1.5%	0.36	4.6 / 1.2	
	SBVH	37.4 / 28.9	0.79 / 0.73	0.75 / 0.66	0.78 / 0.69	0.80 / 0.69	0.49	0.61	2.3K	1.1%	0.30	5.3 / 1.5	
	SAH	105.4 / 86.1	0.80 / 0.56	0.72 / 0.50	0.75 / 0.56	0.77 / 0.54	0.37	0.49	5.6K	3.6%	0.30	6.1 / 1.5	
	AAC	87.5 / 54.8	0.82 / 0.66	0.74 / 0.55	0.71 / 0.61	0.70 / 0.59	0.21	0.44	5.2K	4.8%	0.25	5.2 / 1.3	
	SBVH	71.6 / 51.7	0.88 / 0.85	0.81 / 0.57	0.83 / 0.65	0.85 / 0.61	0.30	0.45	5.9K	2.8%	0.34	4.6 / 1.2	
	SAH	68.5 / 40.3	0.90 / 0.70	0.69 / 0.46	0.70 / 0.54	0.73 / 0.53	0.32	0.65	1.2K	0.1%	0.15	6.6 / 1.2	
	AAC	139.9 / 65.3	0.76 / 0.77	0.46 / 0.48	0.43 / 0.55	0.42 / 0.57	0.20	0.45	4.7K	0.6%	0.15	7.0 / 1.2	
	SBVH	67.0 / 50.1	0.95 / 0.78	0.70 / 0.58	0.66 / 0.65	0.73 / 0.65	0.26	0.61	1.4K	0.1%	0.23	5.3 / 1.1	
	SAH	44.9 / 30.7	0.87 / 0.76	0.76 / 0.68	0.83 / 0.69	0.76 / 0.70	0.37	0.56	3.2K	0.2%	0.30	4.9 / 1.1	
	AAC	54.0 / 36.7	0.83 / 0.82	0.75 / 0.73	0.73 / 0.68	0.73 / 0.74	0.20	0.42	3.7K	0.4%	0.33	4.2 / 1.2	
	SBVH	34.8 / 27.6	0.97 / 0.86	0.80 / 0.68	0.90 / 0.68	0.83 / 0.71	0.21	0.50	3.6K	0.2%	0.43	3.8 / 1.0	
	SAH	88.5 / 53.7	0.97 / 0.98	0.76 / 0.69	0.83 / 0.77	0.80 / 0.72	0.36	0.49	4.7K	1.7%	0.33	4.5 / 1.3	
	AAC	75.0 / 44.6	0.85 / 0.85	0.76 / 0.73	0.82 / 0.81	0.84 / 0.81	0.20	0.45	3.8K	2.1%	0.35	4.4 / 1.3	
	SBVH	50.9 / 38.3	0.92 / 0.87	0.79 / 0.75	0.84 / 0.82	0.86 / 0.84	0.24	0.48	5.8K	1.5%	0.37	4.0 / 1.1	
	SAH	61.0 / 39.2	0.88 / 0.84	0.77 / 0.68	0.79 / 0.73	0.85 / 0.76	0.34	0.51	2.2K	0.7%	0.34	4.2 / 1.1	
	AAC	65.4 / 43.9	0.83 / 0.72	0.77 / 0.76	0.73 / 0.77	0.80 / 0.74	0.20	0.46	2.3K	1.0%	0.24	5.8 / 1.4	
	SBVH	45.1 / 30.0	0.92 / 0.93	0.80 / 0.71	0.78 / 0.78	0.78 / 0.80	0.26	0.57	2.5K	0.5%	0.34	4.1 / 1.0	
	SAH	64.1 / 39.2	0.89 / 0.86	0.79 / 0.71	0.75 / 0.77	0.80 / 0.70	0.29	0.49	2.0K	1.7%	0.34	4.0 / 1.2	
	AAC	65.1 / 37.3	0.93 / 0.96	0.78 / 0.68	0.77 / 0.75	0.82 / 0.81	0.17	0.41	1.8K	2.3%	0.31	4.1 / 1.2	
	SBVH	57.1 / 37.2	0.91 / 0.76	0.80 / 0.69	0.84 / 0.72	0.79 / 0.72	0.25	0.48	2.4K	1.7%	0.39	3.9 / 1.1	
	SAH	72.9 / 47.8	0.91 / 0.91	0.74 / 0.64	0.67 / 0.59	0.72 / 0.62	0.27	0.56	2.9K	0.4%	0.21	5.3 / 1.3	
	AAC	74.3 / 48.2	0.93 / 0.93	0.74 / 0.72	0.75 / 0.70	0.76 / 0.68	0.17	0.56	2.6K	0.6%	0.17	5.5 / 1.3	
	SBVH	68.4 / 42.6	0.95 / 0.95	0.74 / 0.66	0.78 / 0.71	0.82 / 0.68	0.25	0.62	3.1K	0.4%	0.20	5.1 / 1.2	
	SAH	73.5 / 57.5	0.93 / 1.08	0.68 / 0.65	0.67 / 0.58	0.65 / 0.64	0.30	0.59	3.0K	2.2%	0.29	4.9 / 1.3	
	AAC	72.1 / 55.8	0.84 / 0.84	0.67 / 0.63	0.68 / 0.66	0.68 / 0.68	0.20	0.57	2.4K	3.1%	0.23	5.0 / 1.3	
	SBVH	58.5 / 54.7	0.94 / 0.92	0.75 / 0.68	0.82 / 0.70	0.80 / 0.69	0.28	0.58	3.6K	2.1%	0.35	4.0 / 1.2	
	SAH	142.7 / 67.0	0.89 / 0.72	0.80 / 0.75	0.87 / 0.74	0.90 / 0.70	0.28	0.37	14.4K	0.3%	0.46	3.9 / 1.2	
	AAC	143.0 / 60.8	0.85 / 0.88	0.79 / 0.81	0.82 / 0.87	0.88 / 0.78	0.18	0.32	14.9K	0.4%	0.43	3.9 / 1.2	
	SBVH	106.2 / 55.9	0.94 / 0.75	0.83 / 0.67	0.91 / 0.71	0.90 / 0.69	0.25	0.37	13.1K	0.2%	0.51	3.7 / 1.2	
Average 10	SAH		0.88 / 0.81	0.75 / 0.64	0.76 / 0.66	0.77 / 0.66					0.30	4.9 / 1.2	
	AAC		0.84 / 0.82	0.72 / 0.68	0.71 / 0.71	0.74 / 0.72					0.28	5.0 / 1.3	
	SBVH		0.92 / 0.85	0.78 / 0.66	0.80 / 0.70	0.81 / 0.70					0.35	4.4 / 1.2	

Table 1: Detail experiment results for different scenes with various initial BVHs. Results for numbers of ray-box tests for non-optimized BVHs, relative ratios of ray-box test for both SATC and RDTC comparing to non-optimized BVHs, relative ratios on actual wall-clock time for ray-primitive testing for RDTC on both single and packet ray tracing (actual running time in supplemental material), BVH imbalance descriptors for both SATC and RDTC, reconstructed BVH nodes for RDTC, relative node depth to reach triangles for ray-box testing between RDTC and initial BVH, and average numbers of branches and Pass-tests for new contracted node are provided. The data in the last column are averaged on the weight of the VisitCount in each node. “diff / shad” means diffuse rays (and other rays querying for first intersection) / shadow rays.

box tests”). However, this number varies significantly across the scenes and BVH construction methods, and can even be negative. The RDTC heuristic (introduced in Section 4.2.2), however, captures inefficiencies due to both structural and ray-distribution imbalance, gets a consistent improvement of 20-30% (average 25%) for diffuse rays, and 25-55% (average 35%) for shadow rays (column RDTC in “Rel. # ray-box tests”). Similar improvements in runtime are also observed. Moreover, these improvements are less related to BVH construction approaches, but are more scene dependent. Such reductions in the number of ray-box tests for ray tracing are significant since the ray-primitive testing has logarithmic time complexity.

We also use the “imbalance descriptor” (Imb) from 0 to 1 to measure the imbalance of a BVH:

$$Imb(S) = \frac{\sum_{s \in S} (visitCount(s) |\alpha_{s.left} - \alpha_{s.right}|)}{\sum_{s \in S} visitCount(s)} \quad (3)$$

where α is measured by different parameters of the BVH contraction (introduced in Section 4.2). The argument S can be the set of all the nodes in a BVH in Table 1, or the nodes in a specific level in Figure 2. We claim that this function predicts the improvement by our method very well, and the linear regression between them are also shown in the supplemental material.

The number of contracted nodes for RDTC is provided, and

usually only a few thousand tree nodes (1.2K to 5.9K, except for San-Miguel which contains 8M triangles) are reconstructed in our methods. The consumed time for BVH contraction is very short (usually less than 1ms), and about the computations to trace a few hundred rays. Therefore, it is affordable to run the BVH contraction algorithm on **every** frame. We tested the hybrid parameters for SATC and RDTC to a full BVH contraction (i.e. to use α_N in CBTC in Section 4.4), and the difference between the hybrid parameters and RDTC in relative ratio in ray-box tests is less than 1% in all scenes with any initial BVH. Hence, we believe that only this small fraction of the tree (the contracted part, 0.1% to 3% of overall tree nodes) covers most of the structure and ray distribution imbalance. Meanwhile, Table 1 also shows that an average of 8-15% improvement on diffuse rays and 13-18% on shadow rays is caused by structural imbalance and caught by SATC, and an extra 12-14% and 13-20% improvement is caused by extra ray-distribution imbalance and caught by RDTC.

We investigate the benefits of our method by further looking at four representative scenes in Figure 2, which are architectural model CONFERENCE with mainly structural imbalance, building SODA-HALL with imbalance in ray distribution, game scene TRAIN-STATION with imbalance on both, and finely tessellated objects HAIRBALL that has a balanced initial BVH. All their traversal details are computed with an initial BVH generated by top-down SAH.

In column (a), bars indicate the number of Pass-tests in each level. Pass-tests should be avoided in ray-primitive testing because they cannot provide useful pruning during traversal and creates extra data accesses. As we anticipated, this number reduce by a factor of 60% to 80% on the first 3 scenes, and about 40% for balanced initial BVH.

Figure 2, Column (b) shows structural and ray distribution imbalances by levels, which are computed by the imbalance descriptor with α separately from SATC and RDTC. These figures show that most significant imbalance happens at the top (around 10) levels in the BVH, which is where our algorithm focuses on. This is further shown by column (c), which indicates that few contractions happen beyond the top levels since the average branches drop down to 2 quickly. Moreover, even if we have a multi-branch (up to 16 branches) BVH at the first several levels, the actual number of Pass-tests is relatively low (≤ 3.3 at root node, ≤ 1.6 in 2 to 10 levels, average 1.2 to 1.3 for contracted node as shown in Table 1), which means our BVH contraction will not require sorting many boxes to order them from front to back.

5.3. Implementation Details on Traverse

The code to traversing binary BVH is usually highly optimized, including hand-tune operations, dedicate register allocation, etc. For contracted BVH, only the top levels are reconstructed, and the contracted flag of these nodes

are marked as `True`. For non-contracted node, the highly-optimized code for binary BVH traversing is still able to use, because the whole subtree is not changed. For contracted node, an extra loop variable and a more complex sorting process are needed. However, we claim that the extra steps will not affect the running speed.

For non-contracted nodes, multiple box-tests occur together (average 4.4-5.0 shown in Table 1). Nevertheless, since the contraction reduces Pass-tests, only average 1.2 to 1.3 Pass-tests occur on each node. Since only a small fraction (about 25%) of Pass-tests are on each node, sorting the children from-to-end is cheap, since we are usually sorting no more than 2 elements except for the root node (Column (c) in Figure 2). Moreover, the nodes that need sorting are much less and reduced by about 70% on average (Column “relative node depth” in Table 1). Overall, the time spend in sorting process is actually faster than that without BVH contraction. Furthermore, The path to reach triangles are much shortened (average length of 2.6 to reach triangles), comparing to a path with usually 6-15 levels to reach the corresponding BVH nodes. The reduction on average depth can largely speedup the traversing process on both stack and stackless implementation, and overcome the extra cost to use the loop variable. The testing result in Table 1 shows that the acceleration on runtime and the number of box-tests are similar.

The experiment results in Table 1 and Figure 2 shows that traversing the new contracted multi-branch BVHs will not cause inefficiency comparing to binary BVHs, since the relative ratio of actual running times is similar to the ratio of decreased ray-box tests.

5.4. Evaluation on Details

From the previous section we show that the improvement on overall performance on contracted BVHs. We also carefully evaluated the performance of this algorithm with the impacts on different rendering settings, including sample size, lighting environment, camera position, and the order of reflection rays. Due to the length limit of the paper, the evaluation details are provided in the supplemental material, and here we summarized the useful conclusions.

Sample size. Taking a sample pixel over a pixel block with size between 8-by-8 to 32-to-32 usually provides the best overall improvement. In practice, one thousand sample pixels are sufficient for our approach.

Lighting setting. Higher ratios of occluded shadow rays can speedup ray-traversing on contracted BVHs since intersections are found faster after contraction. The performance for diffuse rays should not be affected by light environments. In our experiments shown in the supplemental material, the performance to traverse shadow rays on contracted BVHs get better improvement when a higher ratio of occluded rays appear. Moreover, better coherence of shadow rays generally improves the performance of our approach.

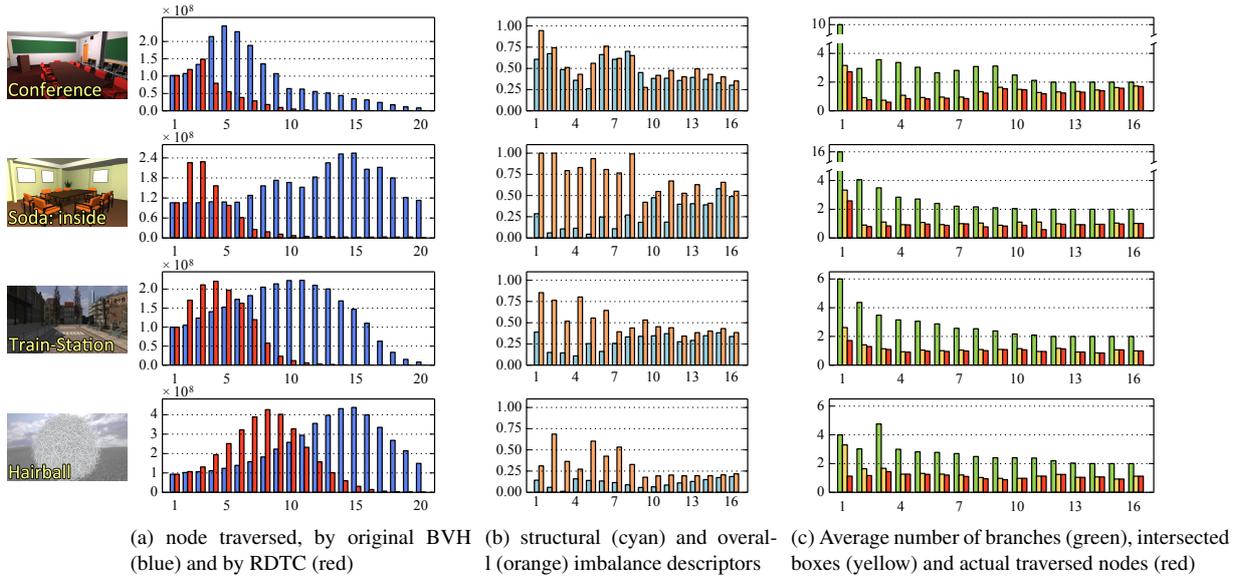


Figure 2: Detail results by levels in the original and contracted BVH. Column (a) shows the nodes that actually traversed (i.e. by Pass-tests) in each level. Column (b) provides structural and overall imbalance descriptors defined in Section 5.2. Column (c) gives the average number of branches, intersected boxes and actual traversed nodes on contracted BVH, and each node is weighted by the number in its counter. Initial BVHs are top-down SAH BVHs.

Camera positions. In our experiments, we show that different camera positions do not impact the performance of our approach much. This is because high quality ray tracing requires numerous diffuse rays to generate global illumination effects, and hence such a large number of incoherent rays distribute fairly randomly no matter where the camera positions are (usually less than 5% differences between the extreme cases in one scene).

Higher-order diffuse-bouncing rays. In our experiments, we also observed that consistent performance improvements are achieved with higher-order diffuse-bouncing rays, and the differences in relative ratios between 1st, 2nd and 3rd order diffuse-bouncing rays are between 1% to 3% on all 3 scenes.

5.5. *N*-ary BVHs

Now we briefly summarize some experimental results when our approach is applied to *n*-ary BVHs (Quad-BVHs or Oct-BVHs). In this case, multiple bounding boxes can simultaneously test together using the SIMD units on a CPU. We compare BVHs constructed by our method (using the parameter of ray distribution and surface area, refer to the CBTC-RD column in Table 8 in the supplemental material) with two existing *n*-ary BVH construction methods: directly collapsing [DHK08] (direct collapse column), and only by surface area [WBB08] (CBTC-SA column).

We provided experiment results on 10 outdoor scenes combining with 3 initial BVH construction algorithms in the supplemental material. The conclusion is that CBTC-RD pro-

vides a 15% / 21% (diffuse / shadow ray) improvement for Quad-BVH and 25% / 30% for Oct-BVH compared with direct collapsing, and 10% / 12% for Quad-BVH and 14% / 15% for Oct-BVH compared with CBTC-SA. Notice that this improvement is measured by number of box testing. This is because the same code can be used as ray-primitive tests on all different parameters since the tree structures are the same. Thus, the reduction of ray-box tests is irrelevant to the code implementation, and any code optimization will accelerate all BVHs. Again, due to the page limit, the full experiment data and result analysis are provided in the supplemental material.

6. Conclusion

In this paper we demonstrated a novel method to efficiently contract BVHs that accelerates ray-primitive testing significantly. This contraction is based on the statistics generated from a sample of the rays. Our algorithm can start with any initial binary BVH created by a state-of-the-art algorithm, and is compatible with other ray-tracing techniques for accelerating ray-primitive testing. Unlike previous method to rebuild an extra BVH, we directly keep statistics in the BVH, so a post-optimization can be easily applied.

Notice that our method significantly decreases Pass-tests at the cost of increasing Prune-tests (overall tests are decreased). An interesting direction for future work is the similar idea but in opposite direction: a method to re-group the nodes to avoid Prune-tests. A heuristic need be developed to acquire the statistics for newly generated nodes.

Acknowledgments

This research was supported in part by NSF grants CCF-1218188, CCF-1314590, and CCF-1533858, and by the Intel Science and Technology Center for Cloud Computing.

References

- [AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics* (2010). 2
- [AKL13] AILA T., KARRAS T., LAINE S.: On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics* (2013). 2
- [AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics* (2009). 2
- [BH09] BITTNER J., HAVRAN V.: RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *25th Spring Conference on Computer Graphics (SCCG)* (2009), Hauser H., (Ed.). 1, 2
- [BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum* 32, 1 (2013), 85–100. 2
- [BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. In *Proc. Interactive Ray Tracing* (2008). 6
- [DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Proc. Eurographics Conference on Rendering* (2008). 2, 5, 6, 9
- [FLF12] FELTMAN N., LEE M., FATAHALIAN K.: SRDH: Specializing BVH construction and traversal order using representative shadow ray sets. In *Proc. High-Performance Graphics* (2012). 1, 2, 4, 5
- [GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient BVH construction via approximate agglomerative clustering. In *Proc. High-Performance Graphics* (2013). 1, 2, 6
- [GPBG11] GARANZHA K., PREMOZE S., BELY A., GALAKTIONOV V.: Grid-based SAH BVH construction on a GPU. *The Visual Computer* 27, 6-8 (2011), 697–706. 2
- [GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics* (2011). 2
- [GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5 (May 1987), 14–20. 2, 6
- [Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 1, 2
- [HM08] HUNT W., MARK W. R.: Ray-specialized acceleration structures for ray tracing. In *Proc. Interactive Ray Tracing* (2008). 2
- [IH11] IZE T., HANSEN C. D.: RTSAH traversal order for occlusion rays. *Comput. Graph. Forum* 30, 2 (2011), 297–305. 5
- [Jak10] JAKOB W.: <http://mitsuba-renderer.org>. 6
- [KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. High-Performance Graphics* (2013). 1, 2
- [Kar12] KARRAS T.: Maximizing parallelism in the construction of BVHs, octrees, and K-D trees. In *Proc. High-Performance Graphics* (2012). 2
- [Ken08] KENSLER A.: Tree rotations for improving bounding volume hierarchies. In *Proc. Interactive Ray Tracing* (2008). 2
- [KKW*13] KELLER A., KARRAS T., WALD I., AILA T., LAINE S., BIKKER J., GRIBBLE C., LEE W.-J., MCCOMBE J.: Ray tracing is the future and ever will be. In *ACM SIGGRAPH Courses* (2013). 1
- [LGS*09] LAUTERBACH C., GARL M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. In *Proc. Eurographics* (2009). 2
- [LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proc. High-Performance Graphics* (2013). 2
- [Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph.* 30, 5 (Oct. 2011), 117:1–117:12. 2
- [NIDN13] NABATA K., IWASAKI K., DOBASHI Y., NISHITA T.: Efficient divide-and-conquer ray tracing using ray sampling. In *Proc. High-Performance Graphics* (2013). 2
- [NM14] NAH J.-H., MANOCHA D.: Sato: Surface area traversal order for shadow ray tracing. *Computer Graphics Forum* (2014). 5
- [PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., ET AL.: Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 66. 1
- [PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object partitioning considered harmful: Space subdivision for BVHs. In *Proc. High Performance Graphics* (2009). 1, 2
- [PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical L-BVH construction for real-time ray tracing of dynamic geometry. In *Proc. High-Performance Graphics* (2010). 2
- [SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *High-Performance Graphics* (2009). 2, 6
- [Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-bvh ray stream tracing. In *Proc. High Performance Graphics* (2009). 6
- [Wal07] WALD I.: On fast construction of SAH-based bounding volume hierarchies. In *Proc. Interactive Ray Tracing* (2007). 2
- [WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. Interactive Ray Tracing* (2008). 2, 4, 5, 6, 9
- [WBKP08] WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast agglomerative clustering for rendering. In *Proc. Interactive Ray Tracing* (2008). 2
- [WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. on Graphics* 26, 1 (2007). 2
- [WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. In *Computer graphics forum* (2001), vol. 20, pp. 153–165. 2, 5
- [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 (TOG)* 33, 4 (2014), 143. 1