Improved Two-Level BVHs using Partial Re-Braiding

Carsten Benthin  Sven Woop  Ingo Wald  Attila T. Áfra
Intel Corporation  Intel Corporation  Intel Corporation  Intel Corporation

Figure 1: Example scenes to which we applied our technique: Trees (12k instances, 522M instanced triangles), San Miguel (254 objects, 10.3M static and 200k dynamic triangles), Boeing (720k objects, 330M static triangles), Powerplant (56 objects, 12.3M static triangles), Rungholt (84 objects, 6.7M static triangles), and Crown (850 objects, 4.8M static triangles). While being nearly as fast to build as traditional two-level BVHs, using our partially merged two-level BVH leads to lower spatial overlap, which in the shown models results in $1.2 \times 2.1 \times$ higher rendering performance.

ABSTRACT

We propose a novel approach for improving the quality of two-level BVHs (i.e., a two-level data structure that uses a top-level BVH built over second-level object BVHs). After building an individual, high-quality BVH for each object, our new top-level BVH build approach selectively re-braids (opens and merges) object BVHs during the build process to reduce overlap and improve SAH quality. We demonstrate that compared to the two main state-of-the-art techniques—brute-force re-construction of a single, flat BVH; and building a traditional two-level BVH over objects, respectively—the proposed approach achieves build times significantly faster than the former, while simultaneously yielding traversal performance that is much higher than the latter.

CCS CONCEPTS

- Computing methodologies → Ray tracing; Visibility;

KEYWORDS

ray tracing, instancing, bounding volume hierarchy

1 INTRODUCTION

To achieve high ray traversal performance, ray tracers employ acceleration data structures such as BVHs, k-d trees, etc. While these structures significantly speed up rendering, they introduce a high up-front cost every time the data structure must be (re-)built. With parallel, well-tuned BVH builders provided by state-of-the-art ray-tracing frameworks such as OptiX [Parker et al. 2010] and Embree [Wald et al. 2014], BVHs can today be built, from scratch, at many million primitives per second. Nevertheless, for interactive rendering of animated content the roughly linear cost [Wald and Havran 2006] of building BVHs puts an upper limit on the number of primitives which can be animated per frame.

The alternative to building a single acceleration structure over all geometric primitives is to employ so-called two-level (or multi-level) BVH. For such two-level BVHs the model’s geometric primitives are grouped into separate objects, each with their own BVH, and with a top-level BVH built over these objects (this allows for updating only those objects—and the top-level BVH—that have changed in a given frame). This approach works well in particular in the common rendering scenarios where one or more animated objects are set within static background geometry. In addition, two-level BVHs
while having BVH quality closer to a single BVH. Like two-level work well with object instancing and rigid body animation using ray transformations.

The caveat of two-level BVHs is that they often incur significant traversal overhead: since the top-level BVH can only separate logical objects—not the geometric primitives they are comprised of—the object partitioning done at the top-most level is typically much worse than what a single, flat BVH would have been able to do. Consequently, rays often have to first traverse the “wrong” subtree for some time before eventually traversing the “right” one. This in turn means lower traversal performance (typically in the range of 1.5 — 2x). With that, users are left with one of two mediocre choices: high build overhead to get a faster-to-traverse single BVH over all primitives, or a fast-to-built, but slower-to-traverse two-level BVH.

In this paper we propose an alternative data structure (and its build algorithms) that is nearly as fast to build as two-level BVHs, while having BVH quality closer to a single BVH. Like two-level BVHs, we use separate objects with their own BVHs and a top-level BVH built over them. However, rather than have the top-level BVH only refer to individual, monolithic objects, we allow the top-level builder to reach into the object BVHs, and to “open up” object nodes where appropriate. This allows the top-level BVH to create new top-level nodes (brown) that address individual subtrees in the object BVHs, resulting in improved BVH quality.

2 RELATED WORK

Acceleration Data Structures. The goal of any ray tracing acceleration structure is to employ some sort of spatial and/or hierarchical indexing to minimize the number of ray-primitive intersections that must be performed. In practice, this involves a trade-off of three factors: how efficient a data structure is in reducing the number of intersections; how quickly it can be traversed on a given hardware; and what it costs to build and maintain it. Though many such data structures have been proposed (see, e.g. [Haines et al. 1989; Pharr and Humphreys 2010]) today most ray tracers use some sort of bounding volume hierarchies (BVHs). In particular, both of today’s fastest ray tracing frameworks—OptiX [Parker et al. 2010] and Embree [Wald et al. 2014]—use BVHs.

Fast BVH builds. With every more widespread use of BVHs, many researchers investigated ways of improving the build time of BVHs, typically involving aggressive parallelization and/or quality trade-offs [Fuetterling et al. 2016; Ganestam et al. 2015; Ganestam and Doggett 2016; Gu et al. 2013; Hendrich et al. 2017; Hou et al. 2010; Karras and Aila 2013; Lauterbach et al. 2009; Parker et al. 2010; Vinkler et al. 2016; Wald et al. 2014]. Our technique is completely orthogonal to such high-performance BVH builders; we use these same techniques for the lower-level object hierarchies, and have our top-level BVH point into these such-generated BVHs.

Two-Level BVHs. Even with the fastest BVH builders, rebuilding the entire data structure every frame is costly. Two-level data structures—first proposed for k-d trees [Wald et al. 2003], but since applied also to BVHs [Parker et al. 2010; Wald et al. 2014]—avoid this by not building a single BVH over all primitives, but grouping primitives into logical objects, building BVHs for those objects, and building a second—the “top-level”—BVH over those objects. This allows for selectively updating only changed objects, as well as for efficient rigid-body transformation, instancing, etc. Our method uses a similar concept, but builds the top-level BVH in a way that allows it to reach into the lower-level object BVHs, thus partially merging the top-level with the object BVHs, where appropriate.

Repairing BVHs. Our method can also be seen as a way of repairing overlap in a two-level BVH. This is similar in spirit to the trie rotations as proposed by Kensler et al. [Kensler 2008], as well as to the selective restructuring proposed by Yoon et al. [Yoon et al. 2007]. Unlike those methods we start out with just a list of high-quality object BVHs, which allows to concentrate all repair operations into a single, quick, and parallel top-level BVH construction pass.

Build-from-hierarchy. Yet another way of viewing our technique is as a variant of the build-from-hierarchy concept as proposed by Hunt et al. [Hunt et al. 2007]: Hunt proposed to accelerate k-d tree construction by using the input scene graph’s hierarchy information to reduce the number of potential split position candidates. Based on this work, multiple build-from-hierarchy variants have been proposed [Ganestam et al. 2015; Gu et al. 2013; Hendrich et al. 2017; Karras and Aila 2013], each of which first build a low-quality auxiliary hierarchy, and then use this auxiliary data structure to build the final, high-quality hierarchy at lower cost. We, too, use a two-step approach in which the second step looks “into” a pre-existing hierarchy, but with two crucial differences: First, we do not rely on a user-supplied scene graph nor do we need to create any low-quality hierarchy first. Second, rather than having the second stage build the entire hierarchy from scratch we only “repair” overlap in the upper levels of the data structure, and otherwise reuse large parts of the existing object BVHs.
3 METHOD OVERVIEW

Our method is motivated by three observations: First, that the performance degradation often seen with two-level BVHs is mainly caused by different objects’ BVHs overlapping each other in the top-level BVH; second, that each subtree of a BVH is also a BVH; and third, that even when different objects’ BVHs overlap significantly, at some deep enough level their respective subtrees overlap significantly less than the objects’ root nodes. Based on these three observations, the core idea of our approach is to

- (1) start with object BVHs in the same way a traditional two-level BVH would;
- (2) find a suitable “cut” through each object’s BVH such that the resulting set of BVH subtrees has lower overlap; and
- (3) build a top-level BVH over those resulting subtrees.

The result is similar to a two-level BVH in which the top-level and object level had been partly merged together by eliminating some of the upper levels of the object BVHs.

3.1 Prerequisites

Our approach requires a list of already built object BVHs as input. Construction of these object BVHs can be done by any suitable construction/update method, but we do assume these BVHs to be of reasonably good quality (for their respective objects) to start with. Our implementation uses a high-quality binned-SAHT builder [Wald and Havran 2006; Wald et al. 2014] for every object BVH.

In addition to those object BVHs we require—just like a regular two-level BVH—a list of instances of these objects. Each such instance refers to an object (and its BVH), and can—but does not have to—contain a transformation matrix: Our common use case is that animated objects get re-built per frame in world space, but our method is fully applicable to scenes containing possibly multiple instances of objects, too.

3.2 BRefs

Given this input, a traditional two-level BVH would build a BVH over exactly those instances, using the instances’ world-space bounding box during top-level BVH construction. Since our top-level BVH will eventually refer to subtrees additional information are required. Throughout the rest of this paper we refer to what we call BVH node build references (or BRefs), which stores the essential data needed for the top-level BVH construction. Each BRef contains a reference pointer to a BVH node inside an object BVH (initialized with the root BVH node), the corresponding world-space bounding information of this node (including transformation, if required), and the ID of the object/instante the BVH node belongs to:

```c
struct BRef {
    BVHNodeReference ref;
    AABB bounds;
    unsigned int objectID;
    unsigned int numPrims;
};
```

Since BRefs can refer to subtrees of vastly different size we also have each BRef track the (possibly estimated) number of primitives in the given subtree; this, together with the bounding box, allows the builder to estimate the SAH cost of a given subtree.

4 CONSTRUCTION ALGORITHM

The key idea of our approach—i.e., partially merging top-level and object BVHs—is valid independent of how exactly the data structure is going to be built (i.e., which object subtrees get selected for the top-level BVH, and how that top-level BVH connects them).

In its simplest form, an algorithm would operate in two distinct phases: one “opening” phase that “opens” object BVHs to produce a list of subtrees to build the top-level BVH over (resulting in a list of BRefs), followed by a second “merge” phase that merges the resulting BRefs in a top-level BVH. The first phase could, for example, start with one BRef per object/instante BVH, and could iteratively pick one such BRef (using some suitable heuristic), open it up, and replace it with the BRefs for its children, until some suitable termination criterion is reached (e.g., until a maximum number of BRefs is created).

4.1 Recursive Top-Down Build

Instead of using such two strictly separated phases, we follow a second approach in which the opening of nodes is built directly into a top-level BVH builder. This top-level BVH builder starts with a list of BRefs, and refines this list continuously by replacing and adding new BRefs on the fly, as required. This list of BRefs is internally stored in a single pre-allocated array (see Section 5.2).

In each recursive partitioning step, the builder looks at a “segment” (subset of contiguous elements) of this array and performs the following steps:

- (1) For the current segment, use an opening heuristic (Section 4.2) to determine which BRefs to open.
- (2) Open the selected candidates by replacing them with BRefs to their respective children.
- (3) Apply a SAH-based binning and partitioning step to split the current segment of BRefs into left and right sub-segments.
- (4) Recursively apply this algorithm for left and right sub-segments until some suitable termination heuristic (see Section 4.4) is reached. This termination heuristic is applied right before step (1), and effectively avoids unnecessary work.

An illustration of the impact of the opening steps is given in Figure 3: In Figure 3(a), the builder starts out with only two BRefs (one blue, one green), so a traditional top-level BVH could not do better than separate these two nodes, resulting in blue and green object BVHs with significant overlap. Using the opening step in our example the builder opens up the blue BRef and replaces it with its children (Figure 3(b)), then allowing the partitioning step to create a better partitioning with lower overlap.

4.2 Opening Criteria

Selecting a subset of BRefs to open is performed according to a spatial extent heuristic. Based on the AABB of the current BRef segment we first determine the dimension $dim$ of the segment’s maximum extent $ext$, where $ext = AABB_{max}[dim] - AABB_{min}[dim]$.

A BRef is opened if its BRef does not refer to a leaf, and if its AABB in dimension $dim$ is wider than $10\%$ of the segment extent $ext$ (the $10\%$ threshold has been determined to work well in practice across a variety of test scenes; see Section 6). The extent criterion makes sure that larger nodes are selected first, and in addition increases the probability that after a couple of opening steps the
with less spatial overlap.

AABBs of the BRefs are more equally sized. It will also keep a set of relatively small BRefs intact, avoiding unnecessary opening.

A threshold of 10% will still open a relatively large number of nodes. This is, however, not a problem, as our chosen termination criterion (Section 4.4) ensures that we only open nodes where different objects overlap.

### 4.3 Binning

Once the opening step completes we can use a traditional SAH binning step to compute a partition of these BRefs, with only small modifications: Unlike standard SAH-based binning we do not operate on individual primitives, but on BRefs that represent entire subtrees with possibly many primitives. We use this information during binning and SAH evaluation by tracking, for each BRef, the number of primitives in this subtree. If that BRef gets added to a bin, we increase that bin’s primitive counter not by one, but by the BRef’s `numPrims` entry.

### 4.4 Termination Criterion

Opening BRefs can be beneficial for removing spatial overlap but excessive opening without a significant gain in SAH quality will waste memory and construction time. Excessive BVH opening is avoided by first testing whether all BRefs refer to the same object by comparing their `objectID` entries. If that is the case we stop all opening for the current segment and for all subsequent build steps (keeping the BRef array segment unmodified from this point on in the recursion). This simple but efficient termination criteria relies on the fact that the underlying object BVH is already of high quality and the opening of BRefs all belonging to the same object won’t improve SAH quality further.

If the number of BRefs is small (≤ 4), we test whether the corresponding bounding boxes overlap using a cheap, SIMD-optimized test. If there is only a small overlap the opening process is terminated even though the respective `objectIDs` might be different.

Finally, in our particular implementation there is a third, implicit, termination criterion caused by our builder’s specific way of handling memory allocations (Section 5.2), which limits the total number of BRefs to a given multiple of the number of input objects—which in effect implies an upper limit on how many opening steps the builder can possibly perform.

### 5 IMPLEMENTATION

The previous sections’ data structure and construction algorithm are general, and could be implemented in a variety of ways. For this particular paper, we have implemented these concepts within the Embree framework [Wald et al. 2014], with an emphasis on high performance through effective threading and memory allocation.

#### 5.1 Thread Parallelism

To achieve high build performance our implementation makes heavy use of Embree’s tasking system, in which a number of threads operate on tasks that themselves can spawn new tasks. Each thread typically picks a different task (if possible), but can also join another, already running, task if no independent work is available.

Building the input objects’ BVHs can easily be done in parallel. Different worker threads build different objects if possible, but are allowed to join other threads’ build tasks if no more independent work is available.

For the top-level build, we again inherit from Embree’s existing parallel BVH builders, which follow a recursive spawning of sub-tasks for each subtree: After a node is built, we launch a task for each child, allowing those subtrees to be built in parallel. If the number of BRefs for a given node is large enough we also allow the opening, SAH binning, and partitioning stages themselves to be split into smaller sub-tasks, allowing multiple workers to work simultaneously on the same task. This is particularly important in the early stages of the build where jobs are large and costly, and only few independent subtrees are being worked on, yet.

#### 5.2 Memory Allocation

Frequent memory allocation from many threads often is a severe performance bottleneck. Traditional top-level BVH builders only need to re-order BRefs in a single, fixed-size array, but for our method the constant opening of nodes requires “allocating” new BRefs all the time, by possibly many different threads in parallel.

To avoid any actual memory allocations we follow the approach proposed by Füetterling et al. [Füetterling et al. 2016] and Ganestam et al. [Ganestam and Doggett 2016], and treat the opening process similar to how they handle spatial splits: We pre-allocate a single static BRef array of a given maximum size (larger than the initial number of BRefs), and keep track of the extra (i.e., not yet used) space during the build. In this approach, each “list” of BRefs corresponds to a segment in this array that can represented by the triple (sstart,send,send_extra), in which [sstart,send) contains the actual list of valid BRefs, and the range (send:send_extra) tracks the extra space into which the opening stage can store new BRef entries when required. As initial size (including the extra space) of the BRef array we use the simple heuristic of `nprimitives/1000` for a tree built over objects, and `4 × ninstance` for those built over instances.

After the partitioning step—which splits such an input segment into left and right sub-segments—we distribute the extra space...
across the child segments heuristically, proportionally to the number of 
BRefs meeting the opening criteria in the left and right segment, respectively. After the partition is done we have to make 
sure that both left and right segments are once again in the proper 
data layout described above, which requires moving the right side’s 
BRefs as much to the right as is required to free up the extra space 
for the left segment. This data movement for the right segment 
can be done efficiently in parallel and in place (it does not require 
maintaining the order of the right side’s BRefs)

Note that our way of proportionally distributing the extra space 
to left and right child has some interesting properties that are easily 
overlooked: First, it means that the number of BRefs a subtree can 
create is independent of the order in which these subtrees are 
being processed, ensuring that the tree that is built is completely 
deterministic despite the heavy threading.

Second, it means that no subtree can ever open more than the 
extra space it got allocated as a fraction of its parent. This ensures 
that the budget for opening nodes gets distributed evenly across 
the entire tree, and, since each child’s budget will get continuously 
smaller the further we go down the tree, also helps avoid any 
excessive opening operations, which keeps the top-level tree small.

5.3 Node Opening and Child BRef Creation
Opening a BRef consists of dereferencing the corresponding BVH 
node reference to access the N children of an N-wide BVH node, 
and creating a new BRef for each of those children. In terms of 
memory allocation, the first child BRef replaces the original parent 
BRef, while its N − 1 sibling BRefs get appended to the end of 
the segment, updating the extra space counter as required. Each 
newly generated child BRef is initialized on the fly, using the child’s 
BVH node reference, the corresponding bounding information, the 
objectID of the parent BRef, and an estimated primitive count.

For each child BRef, our builder needs to know the (approximate) 
number of primitives in each child. The easiest way to obtain 
that information would be to store the actual primitive count in 
each BVH node, but this would require extending the actual BVH 
node layout, including higher memory requirements and likely 
performance degradation. To avoid this we instead compute a 
course approximation by assuming that each subtree’s primitives 
are divided equally across its children, yielding \( \text{child.numPrims} = \frac{\text{parent.numPrims}}{\text{parent.numChildren}} \). The root node’s number of primitives is set 
to the number of primitives of the object it refers to.

5.4 Supporting Instances
For each BRef, we also need to store that BRef’s (world-space) 
bounding box. For objects that got built in world space, this is 
simply the child node’s AABB. For BRefs that refer to an instanced 
object, we compute a conservative AABB based on the child’s AABB 
transformed by the instance’s transformation matrix.

This transformation is done on the fly when generating the 
child BRefs, meaning that our approach is fully applicable to 
instances. In particular, the top-level BVH can and will—in a fully 
automatic manner—select different subtrees of an instanced object 
to be opened depending on where and how it is instantiated.

Applying our method to instanced objects will slightly increase 
the total number of BVH nodes, because an instanced BVH node 
may be opened from multiple parents. However, node openings are 
concentrated in the upper tree levels, so this effect is small.

Computing a conservative AABB for each instance means that 
boxes are often larger than they would need to be, increasing the 
chance that those node get intersected by a ray. In our method, 
however, our ability to open large nodes means that the impact 
of this is actually less than for traditional top-level BVHs. Also, 
large boxes become prime targets for opening, making our method 
particularly useful for scenes with lots of overlapping instances.

5.5 Traversal
By using as much of the existing Embree BVH builder infrastructure 
as possible, the data structure being produced is exactly the same 
as the original two-level BVH, and existing traversal kernels can 
operate on it without any special modifications. When operating on 
instances the node opening means that a ray may now need to get 
transformed to the same instance coordinate system several times. 
However, initial experiments have proven these transformations to 
be cheap enough to not be worth any efforts to avoid them, meaning 
we can use the same BVH traversal code as before. Overall, the 
(smaller) overhead of possibly transforming some rays multiple times 
is easily paid for by the higher quality BVH.

6 RESULTS
As mentioned previously we have integrated our method into Embree 2.15, replacing the existing top-level BVH builder for objects 
and instances. This allows for easily comparing our method—in both 
build time and traversal performance—to both Embree’s existing 
standard two-level BVH and single BVH built over all primitives. All measurements were performed on a dual-socket Intel® Xeon® E5- 
2699 v3 workstation (36 cores total) with 64 GB of memory.

For evaluation we selected a variety of different scenes: one with 
lots of overlapping tree instances (generated using Xfrog [Deussen et al. 1998]); one consisting of dynamic geometry within a complex 
static environment (an animated robot placed into the San Miguel 
scene); one representing a typical CAD model (Boeing); an architect-

ural scene with lots of long thin geometry (Powerplant); a complex 
"MineCraft" model with mostly regular tessellation (Rungholt); and 
the Imperial Crown of Austria model. All models consist of multiple 
individual objects, with varying degrees of overlap between them.

All of the following measurements use the default opening thresh-
old of 10%, as determined by Figure 4.

![Figure 4: Render performance for different opening thresh-
old values, normalized to the performance of our default of 
10%. Though performance can vary a lot, a 10% threshold has 
shown to work close to optimally for all tested scenes.](image-url)
Table 1: Total SAH for building a top-level BVH over all object BVHs (two-level), for our top-level BVH approach with re-braiding (ours), and for a traditional single, high-quality BVH (single) built over all primitives. Compared to two-level, our approach reduces SAH costs and therefore increases rendering performance by 1.2 – 2.1x. For the models we tested our method achieves SAH statistics that are significantly better than those for a traditional top-level BVH, and for most scenes are within 10-20% of those of a single, high-quality BVH.

6.1 Render Performance
Table 1 shows that due to spatial overlap the two-level approach has the highest SAH costs. A single high-quality BVH over all primitives achieves the lowest SAH at 0.29 – 0.85x the reference two-level costs, while our approach achieves 0.44 – 0.89x the two-level SAH costs. For the Trees scene, a single BVH over all objects (without instancing) exceeded the available amount of memory (64 GB) on the system. The improved SAH quality of our approach has a direct impact on rendering performance (measured with a diffuse pathtracer, using up to 8 bounces), yielding roughly 1.18 – 2.1x higher performance than a regular two-level BVH.

Interestingly, the Rungholt model shows the largest performance gain from our approach (2.1x over two-level) while at the same time still benefits the most from a single high-quality BVH (2.9x over two-level) over all primitives. The Crown model has the lowest spatial overlap between objects and therefore benefits the least from our method (being only about 1.18x faster than two-level).

6.2 Build Performance and Time-to-Image
Besides rendering performance, BVH build performance is often critical in terms of time to first image and for handling dynamic scenes in general. Table ?? shows that the standard two-level approach provides 1.6 – 4x faster build times compared to building a single BVH. This is due to the single BVH approach having to iterate over all primitives multiple times in the beginning to find and create the initial partitions for the top of the BVH tree. These operations are costly and in particular often exceeding the CPU cache capacity, making them typically memory bandwidth bound. The two-level approaches avoid these costly first steps (similar to [Ganestam et al. 2015; Gu et al. 2013; Hendrich et al. 2017]) as they build the smaller object BVHs first with a small top-level BVH on top, which results in the vastly higher BVH build performance (66-139 vs. 34-42 million primitives/s). The downside is the reduced SAH quality which our approach is able to significantly regain.

Looking just at the build times for the top-level, our re-braiding approach is on average ~ 2.5x more costly than a simple top-level BVH build over all initial objects due to the additional opening phase and increased number of BRefs in general. The absolute times (in ms) show that even with re-braiding the top-level build time is still only a fraction of total build time. Looking at the combined time-to-image numbers (total build + rendering times) our re-braiding approach essentially combines the fast build times of the two-level approach with the high rendering performance of the single BVH, making it the fastest approach overall.

6.3 Discussion
Due to the fixed memory footprint reserved for the top-level BVH, our re-braising two-level approach increases the total number of BVH nodes by less than 0.05%, adding a negligible overhead to the total BVH memory consumption. Due to the efficient thread parallelism scheme (see Section 5.1), it reaches a scalability in the number of CPU cores of over 90%, making it a good fit for future architectures with even more cores/threads.

The small absolute run-time cost of the re-braiding approach makes it applicable for improving SAH costs in ‘dynamic-in-static’ scenarios, where a set of dynamic objects are rebuilt per frame and then combined with a set of static objects which remain constant over all frames. Table 3 shows build and render times (primary visibility only) for the San Miguel scene with 10.3M static and 200k dynamic triangles (an animated robot character). For interactive scenarios where per-frame rebuild time is most important, rebuilding the entire scene from with using a high-quality (binned-SAH) builder is too slow (256 ms / 41.2 Mprims/s). Even though a fast Morton code-based builder [Lauterbach et al. 2009; Wald et al. 2014] provides up to 5x faster rebuild times (52 ms / 201.9 Mprims/s) for this scene, it is still not fast enough and the BVH quality is significantly inferior (high rendering times) to the binned-SAH variant. The standard two-level approach which rebuilds only the dynamic objects and in a second step the top-level BVH over all static and dynamic objects is significantly faster (1.9 ms) than any single BVH build variant. However, the BVH quality is the lowest resulting in the highest rendering time. Relative to a traditional two-level
We have presented a novel BVH build algorithm that addresses BVH single/multilevel performance, and freely available reference implementation we believe able (open-source). Given its combination of simplicity, perfor-
tation was written for CPUs, the method is just as applicable to GPUs; in fact, the fixed memory footprint makes it a very suitable
tifications to existing traversal kernels; making it easy to add it to
tings such two-level BVHs). Our method works by detecting
node overlap in two-level BVHes (which is, arguably, the main lim-
ity or cost of nodes; do the top-level build in a lazy fash-
crease the probability to catch these extreme cases, but would
storage for creating new nodes. Increasing the node budget would
object BVHs simply because eventually the subtrees will run out of
structure, but might fail to reduce it deep down in overlapping
algorithm. This limitation is not fundamental to the data structure,
leaf nodes cannot be opened, thus never get merged by our build
opening of the object BVHs is currently limited to inner nodes:
One of the few remaining restrictions of our approach is that the
re-braiding approach increases the top-level and dynamic
objects rebuild times by 2.4× (to 4.6 ms)—but vastly improves the
BVH quality, cutting the rendering time in half (from 31 ms to 15
ms), and making it the fastest approach overall with just 19.6 ms per frame.

6.4 Comparison to Related Work
Our approach—indeed, in particular, the top-down opening phase used by
our approach—shares some similarities with work by Ganestam et
al. [Ganestam and Doggett 2016] and Hendrich et al. [Hendrich et al.
2017]. Both of these approaches identify BVH nodes which exceed
certain surface area thresholds and replace these nodes by their
children to obtain a set of more equally sized subtrees. However,
they are targeted at building a single high quality BVH over a scene
from scratch, while our approach targets reducing overlap in an
already existing two-level BVH.

7 SUMMARY AND CONCLUSION
We have presented a novel BVH build algorithm that addresses BVH
node overlap in two-level BVHes (which is, arguably, the main lim-
iation of such two-level BVHs). Our method works by detecting
cases where objects overlap, and reduces the overlap by selectively
opening BVH nodes, and allowing the top-level BVH to reach di-
rectly into the object BVHs where appropriate. Furthermore, we
have described an efficient implementation within the Embree ray
tracing framework, and have shown that this approach outperforms
Embree’s existing two-level BVH, while retaining all of a two-level
BVH’s advantages (i.e., instancing, support for different primitive
types, and fast build times).

Our method is easy to implement, and does not require any mod-
ifications to existing traversal kernels; making it easy to add it to
existing ray tracing frameworks. Though our reference implementa-
tion was written for CPUs, the method is just as applicable to
GPUs; in fact, the fixed memory footprint makes it a very suitable
candidate for such architectures.

The implementation of our method will be made publicly avail-
able (open-source). Given its combination of simplicity, perform-
ance, and freely available reference implementation we believe
this will become the method of choice for improving the quality of
two-level BVHs.

<table>
<thead>
<tr>
<th></th>
<th>Trees</th>
<th>San Miguel</th>
<th>Boeing</th>
<th>Powerplant</th>
<th>Rungholt</th>
<th>Crown</th>
</tr>
</thead>
<tbody>
<tr>
<td>build time (in ms) / build performance (Mprim/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>single</td>
<td>–/–</td>
<td>256/41.2</td>
<td>10/34.9</td>
<td>298/42.6</td>
<td>160/41.6</td>
<td>113/42.9</td>
</tr>
<tr>
<td>two-level</td>
<td>130/–</td>
<td>119/88.2</td>
<td>2.5/139.7</td>
<td>186/68.2</td>
<td>95/70.0</td>
<td>52/93.5</td>
</tr>
<tr>
<td>ours</td>
<td>145/–</td>
<td>122/86.0</td>
<td>2.5/139.7</td>
<td>191/66.6</td>
<td>98/68.4</td>
<td>53/91.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>top-level</th>
<th>ours</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>build time (top-Level only), in ms</td>
<td>15</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>ours</td>
<td>30</td>
<td>2.6</td>
<td>89</td>
</tr>
<tr>
<td>ours</td>
<td>1220</td>
<td>668</td>
<td>2698</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>time to image (build + rendering), in ms</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>–</td>
<td>782</td>
<td>10.2k</td>
<td>469</td>
<td>315</td>
</tr>
<tr>
<td>two-level</td>
<td>1739</td>
<td>1109</td>
<td>2792</td>
<td>425</td>
<td>549</td>
</tr>
<tr>
<td>ours</td>
<td>1220</td>
<td>668</td>
<td>2698</td>
<td>386</td>
<td>314</td>
</tr>
</tbody>
</table>

Table 2: BVH build and time-to-image (build + path traced rendering) in ms for a single BVH over all primitives (single), for a
top-level BVH over all object BVHs (two-level), and for our approach (ours). Our method is slightly slower in build time than
two-level, and slightly slower in rendering than single, but outperforms both in total time-to-image for all examples scenes.

<table>
<thead>
<tr>
<th></th>
<th>build</th>
<th>render</th>
<th>build + render</th>
</tr>
</thead>
<tbody>
<tr>
<td>single (binned)</td>
<td>256</td>
<td>13</td>
<td>269</td>
</tr>
<tr>
<td>single (morton)</td>
<td>52</td>
<td>20</td>
<td>72</td>
</tr>
<tr>
<td>two-level + dyn</td>
<td>1.9</td>
<td>31</td>
<td>32.9 (1.0×)</td>
</tr>
<tr>
<td>ours + dyn</td>
<td>4.6</td>
<td>15</td>
<td>19.6 (0.59×)</td>
</tr>
</tbody>
</table>

Table 3: Rebuild and rendering times (primary visibility + simple shading, in ms) for San Miguel. Timings for the two-
level approaches include rebuild for the dynamic objects and
for the top-level; the single BVH times are for building all
primitives. Both two-level variants are significantly faster
than the single BVH, but our approach provides much faster
rendering performance (due to improved SAH quality), and
consequently best overall time-to-image.

Remaining Issues and Future Work
One of the few remaining restrictions of our approach is that the
opening of the object BVHs is currently limited to inner nodes;
leaf nodes cannot be opened, thus never get merged by our build
algorithm. This limitation is not fundamental to the data structure,
however, and could be addressed in a modified implementation.

Our approach reduces overlap in the upper levels of the data
structure, but might fail to reduce it deep down in overlapping
object BVHs simply because eventually the subtrees will run out of
storage for creating new nodes. Increasing the node budget would
increase the probability to catch these extreme cases, but would
come at the costs of higher memory consumption and build time.

On the upside, there are multiple promising avenues for further
improving the build algorithm. For example, one might open
multiple levels at one; might modify the opening heuristic to com-
pute the actual overlap of nodes; might to prioritize node opening
based on size or cost of nodes; do the top-level build in a lazy fash-
ion; etc. Covering all such possible extensions will require further
investigation, but might make the method even more powerful.
ACKNOWLEDGMENTS

The Imperial Crown of Austria model was provided by Martin Lu-
lich. The Rungholt and Powerplant scenes are courtesy of Morgan
McGuire. The robot model is courtesy of Epic Games, Inc. Finally,
we would like to thank Jeff Amstutz and Johannes Günther for their
valuable feedback.

REFERENCES

O. Deussen, P. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prasinkiewicz.
Parallel Spatial Splits in Bounding Volume Hierarchies. Eurographics Symposium
on Parallel Graphics and Visualization (2016).
Bounding Volume Hierarchy Generation using Mini Trees. Journal of Computer
Graphics Techniques (JCGT) 4, 3 (2015), 23–42.
Per Ganestam and Michael Doggett. 2016. SAH Guided Spatial Split Partitioning for
Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH Construc-
Jakub Hendrich, Daniel Mester, and JiÅźÃŋ Bittner. 2017. Parallel BVH Construc-
Memory-Scalable GPU Spatial Hierarchy Construction. IEEE Transactions on Visu-
alization & Computer Graphics (June 2010).
Warren Hunt, William R. Mark, and Don Fussell. 2007. Fast and Lazy Build of Acceler-
ation Structures from Scene Hierarchies. In IEEE/EG Symposium on Interactive Ray
Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-Quality Bounding
Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and
Dinesh Manocha. 2009. Fast BVH Construction on GPUs. In Proceedings of Euro-
graphics ‘09.
S.G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister,
Matt Pharr and Greg Humphreys. 2010. Physically Based Rendering: From Theory to
Implementation (2nd ed.). Morgan Kaufman.
Marek Vinkler, Vlastimil Havran, and JiÅźÃŋ Bittner. 2016. Performance Comparison
of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. Computer
Graphics Forum 35, 8 (2016), 68–79.
Ingo Wald, Carsten Benthin, and Philipp Slusallek. 2003. Distributed Interactive Ray
Tracing of Dynamic Scenes. In Proceedings of the IEEE Symposium on Parallel and
Large-Data Visualization and Graphics. 11–20.
Ingo Wald and Vlastimil Havran. 2006. On building fast kd-trees for ray tracing, and on
doing that in O(N log N). In Proceedings of the 2006 IEEE Symposium on Interactive
Ray Tracing. 61–69.
Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014.
Embree: A Kernel Framework for Efficient CPU Ray Tracing. ACM Transactions on
Graphics 33, 4, Article 143 (2014), 8 pages.
Sung-Eui Yoon, Sean Curtis, and Dinesh Manocha. 2007. Ray Tracing Dynamic Scenes
using Selective Restructuring. In Eurographics Symposium on Rendering.