Fast Parallel Construction of High-Quality Bounding Volume Hierarchies

Tero Karras Timo Aila

NVIDIA

Abstract

We propose a new massively parallel algorithm for constructing high-quality bounding volume hierarchies (BVHs) for ray tracing. The algorithm is based on modifying an existing BVH to improve its quality, and executes in linear time at a rate of almost 40M triangles/sec on NVIDIA GTX Titan. We also propose an improved approach for parallel splitting of triangles prior to tree construction. Averaged over 20 test scenes, the resulting trees offer over 90% of the ray tracing performance of the best offline construction method (SBVH), while previous fast GPU algorithms offer only about 50%. Compared to state-of-the-art, our method offers a significant improvement in the majority of practical workloads that need to construct the BVH for each frame. On the average, it gives the best overall performance when tracing between 7 million and 60 billion rays per frame. This covers most interactive applications, product and architectural design, and even movie rendering.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray tracing, bounding volume hierarchies

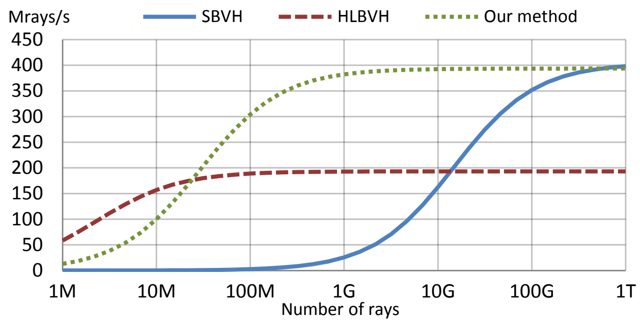


Figure 1: *Performance of constructing a BVH and then casting a number of diffuse rays with NVIDIA GTX Titan in* SODA *(2.2M triangles). SBVH [Stich et al. 2009] yields excellent ray tracing performance, but suffers from long construction times. HLBVH [Garanzha et al. 2011a] is very fast to construct, but reaches only about 50% of the performance of SBVH. Our method is able to reach 97% while still being fast enough to use in interactive applications. In this particular scene, it offers the best quality–speed tradeoff for workloads ranging from 30M to 500G rays per frame.*

1 Introduction

Ray tracing is the main ingredient in most of the realistic rendering algorithms, ranging from offline image synthesis to interactive visualization. While GPU computing has been successful in accelerating the tracing of rays [Aila and Laine 2009; Aila et al. 2012], the problem of constructing high-quality acceleration structures needed to reach this level of performance remains elusive when precomputation is not an option.

Bounding volume hierarchies (BVHs) are currently the most popular acceleration structures for GPU ray tracing because of their low memory footprint and flexibility in adapting to temporal changes in scene geometry. High-quality BVHs are typically constructed using a greedy top-down sweep [MacDonald and Booth 1990; Stich et al. 2009], commonly considered to be the gold standard in ray tracing performance. Recent methods [Kensler 2008; Bittner et al. 2013] can also provide comparable quality by restructuring an existing, lower quality BVH as a post-process. Still, the construction of high-quality BVHs is computationally intensive and difficult to parallelize, which makes these methods poorly suited for applications where the geometry changes between frames. This includes most interactive applications, product and architectural visualization, and movie production.

Recently, a large body of research has focused on tackling the problem of animated scenes by trading BVH quality for increased construction speed [Wald 2007; Pantaleoni and Luebke 2010; Garanzha et al. 2011a; Garanzha et al. 2011b; Karras 2012; Kopta et al. 2012]. Most of these methods are based on limiting the search space of the top-down sweep algorithm, and they can yield significant increases in construction speed by utilizing the massive parallelism offered by GPUs. However, the BVH quality achieved by these methods falls short of the gold standard, which makes them practical only when the expected number of rays per frame is small.

The practical problem facing many applications is that the gap between the two types of construction methods is too wide (Figure 1). For moderately sized workloads, the high-quality methods are too slow to be practical, whereas the fast ones do not achieve sufficient ray tracing performance. In this paper, we bridge the gap by presenting a novel GPU-based construction method that achieves performance close to the best offline methods, while at the same time executing fast enough to remain competitive with the fast GPU-based ones. Furthermore, our method offers a way to adjust the quality–speed tradeoff in a scene-independent manner to suit the needs of a given application.

Our main contribution is a massively parallel GPU algorithm for restructuring an existing BVH in order to maximize its expected ray tracing performance. The idea is to look at local neighborhoods of nodes, i.e., *treelets*, and solve an NP-hard problem for each treelet to find the optimal topology for its nodes. Even though the optimization itself is exponential with respect to the size of the treelet, the overall algorithm scales linearly with the size of the scene. We show that even very small treelets are powerful enough to transform a low-quality BVH that can be constructed in a matter of milliseconds into a high-quality one that is close to the gold standard in ray tracing performance.

Our second contribution is a novel heuristic for splitting triangles prior to the BVH construction that further improves ray tracing performance to within 10% of the best split-based construction method to date [Stich et al. 2009]. We extend the previous work [Ernst and Greiner 2007; Dammertz and Keller 2008] by providing a more accurate estimate for the expected benefit of splitting a given triangle, and by taking steps to ensure that the chosen split planes agree with each other to reduce node overlap more effectively.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPG 2013, July 19 – 21, 2013, Anaheim, California.
Copyright © ACM 978-1-4503-2135-8/13/07 $15.00

2 Related Work

Surface Area Heuristic Ray tracing performance is most commonly estimated using the surface area cost model, first introduced by Goldsmith and Salmon [1987] and later formalized by MacDonald and Booth [1990]. The *SAH cost* of a given acceleration structure is defined as the expected cost of tracing a non-terminating random ray through the scene:

$$C_i \sum_{n\in I} \frac{A(n)}{A(\text{root})} + C_l \sum_{l\in L} \frac{A(l)}{A(\text{root})} + C_t \sum_{l\in L} \frac{A(l)}{A(\text{root})} N(l), \quad (1)$$

where I and L are the sets of internal nodes and leaf nodes, respectively, and C_i and C_l are their associated traversal costs. C_t is the cost of a ray-triangle intersection test, and $N(l)$ denotes the number of triangles referenced by leaf node l. The surface area of the bounding volume in node n is indicated by $A(n)$, and the ratio $A(n)/A(\text{root})$ corresponds to the conditional probability that a random ray intersecting the root is also going to intersect n. In this paper, we use $C_i = 1.2$, $C_l = 0$, and $C_t = 1$, which we have verified experimentally to give the highest correlation with the measured performance.

The classic approach for constructing BVHs is based on greedy top-down partitioning of triangles that aims to minimize the SAH cost at every step [MacDonald and Booth 1990]. At each node, the triangles are classified to either side of an axis-aligned split plane according to the centroids of their axis-aligned bounding boxes (AABBs). The split plane is chosen by evaluating the SAH cost of the resulting child nodes for each potential plane, and selecting the one that results in the lowest cost. Leaf nodes are created when the SAH cost can no longer be improved through partitioning, i.e., the benefit of creating a new internal node is outweighed by its cost.

Another well-known approach is to start from the leaves and proceed in a bottom-up fashion by merging the nodes iteratively [Walter et al. 2008]. Even though this approach is often able to produce trees with a very low SAH cost, it tends to lose to the top-down algorithm in practical ray tracing performance.

Approximate Methods The most widely adopted simplification of the full top-down partitioning is the *binned SAH* [Wald 2007; Wald 2012], which limits the split planes considered at each node to a fixed number. The planes are placed uniformly along each axis to cover the spatial extent of the node, which makes it possible to *bin* the triangles into intervals between the planes according to their centroids. A further simplification is to first perform the binning globally using a fixed-size grid, and then reuse the same results for all nodes that overlap a given cell [Garanzha et al. 2011b].

Another approach is to use a *linear BVH* (LBVH) [Lauterbach et al. 2009], which can be constructed very quickly on the GPU. The idea is to first sort the triangles along a space-filling curve, and then partition them recursively so that each node ends up representing a linear range of triangles [Pantaleoni and Luebke 2010; Garanzha et al. 2011a]. Karras [2012] showed that every stage of the construction can be parallelized completely over the entire tree, which makes the rest of the stages practically free compared to the sorting.

Although linear BVHs are fast to construct, their ray tracing performance tends to be unacceptably low — usually around 50% of the gold standard. This necessitates using hybrid methods that construct important parts of the tree using a high-quality algorithm while using a fast algorithm for the expensive parts. HLBVH [Garanzha et al. 2011a], for instance, uses binned SAH for the topmost nodes while relying on linear BVH for the remaining ones.

BVH Optimization Closely related to our work, there has been some amount of research on optimizing *existing* BVHs as a post-process. Kensler [2008] proposed using local tree rotations to improve high-quality BVHs beyond the gold standard on the CPU, and Kopta et al. [2012] applied the same idea to refine BVHs during animation to combat their inherent degradation in quality. A similar approach was used in NVIDIA OptiX [2012] to improve the quality of HLBVH on the GPU. The main weakness of tree rotations is that they are prone to getting stuck in a local optimum in many scenes. To overcome this effect, one has to resort to stochastic methods that converge too slowly to be practical [Kensler 2008].

Recently, Bittner et al. [2013] presented an alternative algorithm based on iteratively removing nodes from the tree and inserting them back at optimal locations. Since there are a large number of options for modifying the tree at each step, the algorithm is able to improve the quality significantly before getting stuck. However, since the method is fundamentally serial, it is unclear whether it can be implemented efficiently on the GPU.

Triangle Splitting Several authors have noted that splitting large triangles can improve the quality of BVHs significantly in scenes that contain large variation in triangle sizes. Ernst and Greiner [2007] propose to split triangles along their longest axis if their surface area exceeds a pre-defined threshold, and Dammertz and Keller [2008] propose similar scheme to split triangle edges based on the volume of their axis-aligned bounding boxes. However, neither of these methods has been proven to work reliably in practice — they can actually end up decreasing the performance in certain scenes.

A better approach, proposed independently by Stich et al. [2009] (SBVH) and Popov et al. [2009], is to incorporate triangle splitting directly into the top-down construction algorithm. SBVH, which yields the highest ray tracing performance to date, works by considering *spatial splits* in addition to conventional partitioning of triangles. Spatial splits correspond to duplicating triangles that intersect a given split plane, so that each resulting *triangle reference* lies strictly on either side of the plane. The choice between spatial splits and triangle partitioning is made on a per-node basis according to which alternative is the most effective in reducing the SAH cost.

3 Overview

Our goal is to construct high-quality BVHs from scratch as quickly as possible. For maximum performance, we target NVIDIA Kepler GPUs using CUDA. Our approach is motivated by the insight offered by Bittner et al. [2013] that it is possible to take an existing low-quality BVH and modify it to match the quality of the best top-down methods. While Bittner et al. hypothesize that the individual tree modifications have to be global in nature for this to be possible, our intuition is that the *number* of possible modifications plays a more important role. For example, tree rotations [Kensler 2008] offer only 6 ways of modifying the tree per node. Once all of these modifications are exhausted, i.e., none of them is able to reduce the SAH cost any further, the optimization gets stuck.

Instead of looking at individual nodes, we extend the concept of tree rotations to larger neighborhoods of nodes. We define a *treelet* as the collection of immediate descendants of a given treelet root, consisting of n treelet leaves and $n-1$ treelet internal nodes (Figure 2). We require the treelet to constitute a valid binary tree on its own, but it does not necessarily have to extend all the way down to the leaves of the BVH. In other words, the children of every internal node of the treelet must be contained in the treelet as well, but its leaves can act as representatives of arbitrarily large subtrees.

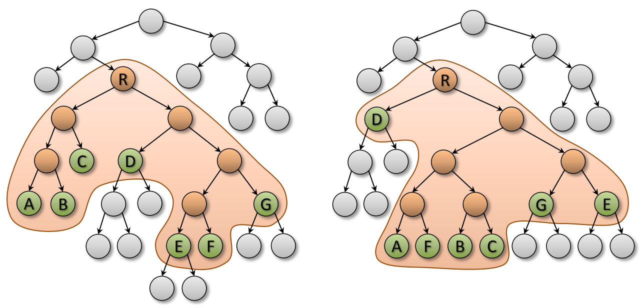
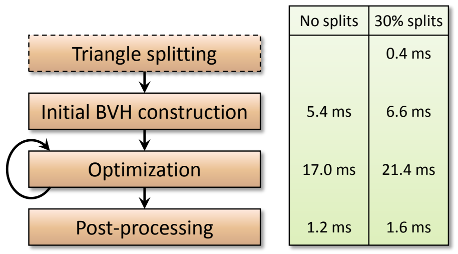


Figure 2: Left: *Treelet consisting of 7 leaves (A–G) and 6 internal nodes, including the root (R). The leaves can either be actual leaf nodes of the BVH (A, B, C, F), or they can represent arbitrary subtrees (D, E, G).* Right: *Reorganized treelet topology to minimize the overall SAH cost. Descendants of the treelet leaves are kept intact, but their location in the tree is allowed to change.*



	No splits	30% splits
Triangle splitting		0.4 ms
Initial BVH construction	5.4 ms	6.6 ms
Optimization	17.0 ms	21.4 ms
Post-processing	1.2 ms	1.6 ms

Figure 3: *Our method starts with an optional triangle splitting stage, followed the construction of an initial BVH. It then performs multiple rounds of treelet optimization and applies final post-processing to obtain a high-quality BVH suitable for ray tracing. The timings are for* DRAGON *(870K triangles) without triangle splitting, and with 30% additional triangle references allowed.*

Our basic idea is to repeatedly form treelets and restructure their nodes to minimize the overall SAH cost. We keep the treelet leaves and their associated subtrees intact during the restructuring, which means that the contents of the subtrees are not relevant as far as the optimization is concerned — we only need to consider properties of the treelet leaves themselves, such as their AABBs. Thus, the processing of each treelet is a perfectly localized operation.

Restructuring a given treelet can be viewed as discarding its existing internal nodes and then constructing a new binary tree for the same set of treelet leaves. As the number of leaves remains unchanged, there will also be the same number of internal nodes in the new treelet. The only thing that really changes, in addition to the connectivity of the nodes, is the set of bounding volumes stored by the internal nodes. In other words, restructuring offers a way to reduce the surface area of the internal nodes, which in turn translates directly to reducing the overall SAH cost (Equation 1).

Finding the optimal node topology for a given treelet is believed to be an NP-hard problem, and the best known algorithms are exponential with respect to n. However, we observe that the treelets do not need to be very large in practice. For example, $n = 7$ already provides $(2n-3)!! = 10395$ unique ways[1] for restructuring each treelet, and there are many ways for forming the treelets to begin with. We will show in Section 6 that this provides enough freedom to prevent the optimization from getting stuck prematurely.

3.1 Processing Stages

On a high level, our method works by constructing an initial BVH, optimizing its topology, and then applying final post-processing (Figure 3). For the initial BVH, we employ the method presented by Karras [2012] using 60-bit Morton codes to ensure accurate spatial partitioning even for large scenes. The initial BVH stores a single triangle reference in each leaf node, and we maintain this property throughout the optimization. However, since the size of the leaves is known to have a significant impact on ray tracing performance, we opt to collapse individual subtrees into leaf nodes during post-processing. The goal of the optimization is thus to minimize the SAH cost of the final tree that we will eventually get.

To account for the collapsing, we calculate the SAH cost of a given subtree as the minimum over the two possible outcomes:

$$C(n) = \min \begin{cases} C_i A(n) + C(n_l) + C(n_r) & (n \in I) \\ C_t A(n) N(n) & (n \in L) \end{cases} \qquad (2)$$

[1] $k!!$ denotes the *double factorial*, defined for odd k as $k \cdot (k-2) \cdots 3 \cdot 1$.

where n is the root of the subtree, n_l and n_r are its left and right child nodes, and $N(n)$ indicates the total number of triangles contained in the subtree. The first case corresponds to making n an internal node, whereas the second one corresponds to collapsing the entire subtree into a single leaf. We choose whichever alternative yields the lowest SAH cost, so $C(\text{root})/A(\text{root})$ gives the same result as Equation 1 for the final tree. In practice, we initialize $N(n)$ and $C(n)$ during the AABB fitting step of the initial BVH construction and keep them up to date throughout the optimization.

The main benefit of our cost model is that it unifies the processing of leaves and internal nodes so that we can use the same algorithm for optimizing both — moving nodes in the intermediate tree effectively allows us to refine the leaves of the final BVH.

Optimization In the optimization stage, our idea is to restructure the topology of multiple treelets in parallel, forming one treelet for each node of the BVH. To enumerate the nodes, we leverage the parallel bottom-up traversal algorithm presented by Karras [2012]. The algorithm works by traversing paths from the leaf nodes to the root in parallel, using atomic counters to terminate the first thread to enter any given node while allowing the second one to proceed. The algorithm guarantees that the nodes are visited in a strict bottom-up order: when we visit a particular node, we know that all of its descendants have already been visited. This means that we are free to restructure the descendants without the danger of other threads trying to access them at the same time. The bottom-up traversal also provides a natural way to propagate $C(n)$ up the tree.

The traversal algorithm, however, tends to have very low SIMD utilization because most of the threads terminate quickly while only a few survive until the end. This is problematic since the optimization is computationally expensive and we would ideally like to run it at full utilization. Instead of performing the restructuring independently by each thread, we choose to use a group of 32 threads to collaboratively process one treelet at a time. This requires intricate algorithm design (Section 4), but it offers a number of benefits. Since every treelet occupies 32 threads instead of one, it is enough to have only a modest number of treelets in flight to employ the entire GPU. This, in turn, means that there is more on-chip memory available for processing each treelet, and it also improves the scalability of the algorithm.

Post-processing In the end, we want to obtain a BVH that is readily usable with the fast ray tracing kernels by Aila et al. [2012]. The final post-processing stage identifies the subtrees to be collapsed into leaves, collects their triangles into linear lists, and outputs them in a format suitable for Woop's intersection test [Woop

2004]. We identify the subtrees by looking at the value of $C(n)$ for each node. If we determine that the value corresponds to the second case in Equation 2 but the same is not true for the ancestors of n, we proceed to collapse the node. This, in turn, is accomplished by traversing the subtree to identify the individual triangles, and then using an atomic counter to place them in the output array.

Triangle Splitting To match the quality of the existing split-based methods, we include an optional triangle splitting stage (Section 5) before the initial BVH construction. We control the amount of splitting by allocating space for a fixed percentage of newly created triangle references and then performing as many splits as we can without exceeding this amount. We remove duplicate references falling into the same leaf during the post-processing, but for the purposes of Equation 2, we treat each reference as a separate triangle.

4 Treelet Restructuring

Our approach for restructuring a BVH to minimize its SAH cost consists of three main ingredients. We perform *bottom-up traversal* over the nodes to determine a processing order that avoids simultaneous access to overlapping subtrees. For each node encountered during the traversal, we *form a treelet* by using the node itself as a treelet root and designating a fixed number of its descendants as treelet internal nodes and treelet leaves. We then *optimize* the treelet by constructing a new binary tree for the same set of treelet leaves that minimizes the overall SAH cost. We start by discussing treelet formation and by presenting a naive optimization algorithm, which we then refine to finally arrive at an efficient GPU implementation. Throughout this section, we use a fixed treelet size of $n = 7$ leaves to illustrate various algorithmic details in concrete terms.

4.1 Treelet Formation

Our intuition is that the surface area of a treelet's internal nodes is a good indicator of the potential for reducing its SAH cost. In order to maximize this potential, we aim to form treelets that extend over the nodes with the largest surface area. To form a treelet, we start with the designated treelet root and its two children. We then grow the treelet iteratively by choosing the treelet leaf with the largest surface area and turning it into an internal node. This is accomplished by removing the chosen node from the set of leaves and using its children as new leaves instead. Repeating this, we need 5 iterations in total to reach $n = 7$.

4.2 Naive Optimization

After forming a treelet, we wish to construct the optimal binary tree for its leaves. A straightforward way to accomplish this is to consider each possible binary tree in turn and choose the best one. This can be expressed conveniently using a recursive function, illustrated in Algorithm 1. The function takes a set of leaves S as a parameter and returns the optimal tree T_{opt} along with its SAH cost c_{opt}.

If S consists of a single leaf, the function looks up the associated SAH cost and returns (lines 3–6). Otherwise, it tries each potential way of partitioning the leaves into two subsets (line 9). A partitioning is represented by set P that indicates which leaves should go to the left subtree of the root; the rest will go the right subtree. For P to be valid, neither subtree can be empty (line 10).

For each partitioning, the algorithm proceeds to construct the subtrees in an optimal way by calling itself recursively (lines 12–13). It then calculates the SAH cost of the full tree obtained by merging the subtrees (lines 15–16). This corresponds to the first case of Equation 2, where the AABB of the root is calculated as the union of the

```
 1: function ConstructOptimalTree($S$)
 2:    // Single leaf?
 3:    if $|S| = 1$ then
 4:       $l \leftarrow S_0$
 5:       return $(l, C(l))$
 6:    end if
 7:    // Try each way of partitioning the leaves
 8:    $(T_{\text{opt}}, c_{\text{opt}}) \leftarrow (\emptyset, \infty)$
 9:    for each $P \subseteq S$ do
10:       if $P \neq \emptyset$ and $P \neq S$ then
11:          // Optimize each resulting subtree recursively
12:          $(T_l, c_l) \leftarrow$ ConstructOptimalTree($P$)
13:          $(T_r, c_r) \leftarrow$ ConstructOptimalTree($S \setminus P$)
14:          // Calculate SAH cost (first case of Equation 2)
15:          $a \leftarrow$ Area(UnionOfAABBs($S$))
16:          $c \leftarrow C_i \cdot a + c_l + c_r$
17:          // Best so far?
18:          if $c < c_{\text{opt}}$ then
19:             $T_{\text{opt}} \leftarrow$ CreateInternalNode($T_l, T_r$)
20:             $c_{\text{opt}} \leftarrow c$
21:          end if
22:       end if
23:    end for
24:    // Collapse subtree? (second case of Equation 2)
25:    $a \leftarrow$ Area(UnionOfAABBs($S$))
26:    $t \leftarrow$ TotalNumTriangles($S$)
27:    $c_{\text{opt}} \leftarrow \min\big(c_{\text{opt}}, (C_t \cdot a \cdot t)\big)$
28:    return $(T_{\text{opt}}, c_{\text{opt}})$
29: end function
```

Algorithm 1: *Naive algorithm for constructing the optimal binary tree (T_{opt}) that minimizes the SAH cost (c_{opt}) for a given set of leaves (S). The idea is to try each way of partitioning the leaves so that some of them (P) are assigned to the left subtree of the root, while the rest ($S \setminus P$) are assigned to the right subtree. The subtrees are, in turn, constructed by repeating the same process recursively.*

AABBs in S. The algorithm maintains the best solution found so far in T_{opt} and c_{opt} (line 8), and replaces it with the current one if it results in an improved SAH cost (lines 18–21).

In the end, c_{opt} corresponds to the lowest SAH cost that can be obtained by creating at least one internal node, but it does not account for the possibility of collapsing the entire subtree into a single leaf. As per our policy of maintaining one triangle per leaf throughout the optimization, we do not actually perform such collapsing until the final post-processing stage. However, we account for the possibility by evaluating the second case of Equation 2 at the end, and returning whichever of the two costs is lower (lines 25-28).

4.3 Dynamic Programming

While the naive algorithm is straightforward, it is also woefully inefficient. For instance, $n = 7$ results in a total of 1.15 million recursive function calls and an even larger number of temporary solutions that are immediately discarded afterwards. To transform the algorithm into a more efficient form that produces an identical result, we make three important modifications to it:

- Remove the recursion and perform the computation in a predetermined order instead.
- Represent S and P as bitmasks, where each bit indicates whether the corresponding leaf is included in the set.
- Memoize the optimal solution for each subset, using the bitmasks as array indices.

```
1: // Calculate surface area for each subset
2: for $\bar{s} = 1$ to $2^n - 1$ do
3:     $a[\bar{s}] \leftarrow$ AREA(UNIONOFAABBS($L, \bar{s}$))
4: end for
5: // Initialize costs of individual leaves
6: for $i = 0$ to $n - 1$ do
7:     $c_{\text{opt}}[2^i] \leftarrow C(L_i)$
8: end for
9: // Optimize every subset of leaves
10: for $k = 2$ to $n$ do
11:     for each $\bar{s} \in [1, 2^n - 1]$ with $k$ set bits do
12:         // Try each way of partitioning the leaves
13:         $(c_{\bar{s}}, \bar{p}_{\bar{s}}) \leftarrow (\infty, 0)$
14:         for each $\bar{p} \in$ {partitionings of $\bar{s}$} do
15:             $c \leftarrow c_{\text{opt}}[\bar{p}] + c_{\text{opt}}[\bar{s}$ XOR $\bar{p}]$   // $S \setminus P$
16:             if $c < c_{\bar{s}}$ then $(c_{\bar{s}}, \bar{p}_{\bar{s}}) \leftarrow (c, \bar{p})$
17:         end for
18:         // Calculate final SAH cost (Equation 2)
19:         $t \leftarrow$ TOTALNUMTRIANGLES($L, \bar{s}$)
20:         $c_{\text{opt}}[\bar{s}] \leftarrow \min\left((C_i \cdot a[\bar{s}] + c_{\bar{s}}), (C_t \cdot a[\bar{s}] \cdot t)\right)$
21:         $\bar{p}_{\text{opt}}[\bar{s}] \leftarrow \bar{p}_{\bar{s}}$
22:     end for
23: end for
```

Algorithm 2: *Finding the optimal tree using dynamic programming. L is an ordered sequence of the n treelet leaves, and $\bar{s}$ and $\bar{p}$ are bitmasks representing subsets of these leaves. The algorithm processes the subsets according to their size, starting from the smallest one. The optimal SAH cost for each subset is stored in c_{opt}, and the corresponding partitioning is stored in $\bar{p}_{opt}$. In the end, the SAH cost of the full tree is indicated by $c_{opt}[2^n - 1]$.*

These modifications lead to a bottom-up dynamic programming approach: Since we know that we need solutions to all subproblems in order to solve the full problem, we proceed to solve small subproblems first and build on their results to solve the larger ones. Given that the solution for subset S depends on the solutions for all $P \subset S$, a natural way to organize the computation is to loop over $k = 2 \ldots n$ and consider subsets of size k in each iteration. This way, every iteration depends on the results of the previous ones, but there are no dependencies within the iterations themselves.

In Algorithm 2, we represent the full set of leaves as an ordered sequence L, and use bitmasks $\bar{s}$ and $\bar{p}$ to indicate which elements of L would be included in the corresponding sets S and P in the naive variant. The algorithm starts by calculating the surface area of each potential internal node and storing the results in array a (lines 2–4). Calculating the AABBs has different computational characteristics compared to the other parts of the algorithm, so doing it in a separate loop is a good idea considering the parallel implementation.

The algorithm handles subsets corresponding to individual leaves as a special case (lines 6–8). It then proceeds to optimize the remaining subsets in increasing order of size (lines 10–11). The optimal SAH cost of each subset is stored in array c_{opt}, and the corresponding partitioning is stored in $\bar{p}_{\text{opt}}$. Keeping track of the partitionings avoids the need to construct temporary trees — once all subsets have been processed, reconstructing the optimal tree is a matter of backtracking the choices recursively starting from $\bar{p}_{\text{opt}}[2^n - 1]$.

Processing a given subset is very similar to the naive algorithm. We first try each possible way of partitioning the leaves (lines 14–17), maintaining the best solution found so far in temporary variables $c_{\bar{s}}$ and $\bar{p}_{\bar{s}}$ (line 13). We then calculate the final SAH cost and record the results in c_{opt} and $\bar{p}_{\text{opt}}$ (lines 19–21). As an optimization, we observe that the first term of the SAH cost, $C_i \cdot a[\bar{s}]$, does not actually depend on which partitioning we choose. We thus omit it from the

```
1: $\bar{\delta} \leftarrow (\bar{s} - 1)$ AND $\bar{s}$
2: $\bar{p} \leftarrow (-\bar{\delta})$ AND $\bar{s}$
3: repeat
4:     $c \leftarrow c_{\text{opt}}[\bar{p}] + c_{\text{opt}}[\bar{s}$ XOR $\bar{p}]$
5:     if $c < c_{\bar{s}}$ then $(c_{\bar{s}}, \bar{p}_{\bar{s}}) \leftarrow (c, \bar{p})$
6:     $\bar{p} \leftarrow (\bar{p} - \bar{\delta})$ AND $\bar{s}$
7: until $\bar{p} = 0$
```

Algorithm 3: *Implementing the inner loop (lines 14–17) of Algorithm 2 efficiently by utilizing the borrowing rules in two's complement arithmetic. The loop executes $2^{k-1} - 1$ iterations in total, where k is the number of set bits in $\bar{s}$.*

computation in the inner loop (line 15), and include it in the final cost instead (line 20).

Most of the computation happens in the inner loop (lines 14–17). Each iteration of the loop is very simple: we look up two values from c_{opt} and update the temporary variables $c_{\bar{s}}$ and $\bar{p}_{\bar{s}}$. The complement of $\bar{p}$, corresponding to $S \setminus P$, is obtained conveniently through logical XOR, since we know that $\bar{p}$ can only contain bits that are also set in $\bar{s}$ (line 15). Looping over the partitionings comes down to enumerating all integers that have this property (line 14). However, in addition to excluding 0 and $\bar{s}$, we would also like to exclude partitionings whose complements we have already tried. These partitionings result in mirror images of the same trees, and are thus irrelevant for the purposes of minimizing the SAH cost.

A practical way of enumerating the partitionings efficiently is shown in Algorithm 3. The idea is to clear the lowest bit of $\bar{s}$ and then step through the bit combinations of the resulting value $\bar{\delta}$. Clearing the lowest bit (line 1) means that we will always assign the first leaf represented by $\bar{s}$ to the right subtree of the root, which is enough to avoid enumerating complements of the same partitionings. We determine the successor of a given value by utilizing the borrowing rules of integer subtraction in two's complement arithmetic (line 6). The initial value of $\bar{p}$ can be thought of as being the successor of zero (line 2). For a subset of size k, the loop executes $2^{k-1} - 1$ iterations in total, after which $\bar{p}$ wraps back to zero.

4.4 Memory Space Considerations

With $n = 7$, Algorithm 2 executes $(3^n + 1)/2 - 2^n = 966$ inner loop iterations and stores $2^n - 1 = 127$ scalars in each of the arrays a, c_{opt}, and $\bar{p}_{\text{opt}}$. Compared to the naive variant, it represents roughly a thousand-fold improvement in terms of execution speed.

To turn the algorithm into an efficient GPU implementation, we must first consider the storage of temporary data. NVIDIA GPUs are built around an array of streaming multiprocessors (SMs). In the Kepler architecture, each SM can accommodate 64 warps, i.e., groups of 32 threads, and has a 256KB register file and 48KB of fast shared memory. We aim for one treelet per warp at full occupancy, which means that we can afford 32 scalar registers per thread and 768 bytes of shared memory per treelet.

Placing a, c_{opt}, and $\bar{p}_{\text{opt}}$ in shared memory using 4 bytes per element would exceed our budget by a factor of 2. To improve the situation, we make two observations. First, $a[\bar{s}]$ is only needed for calculating $c_{\text{opt}}[\bar{s}]$. We can overlay the two into the same array whose elements initially represent a until line 7 or 20 of the pseudocode turns them into c_{opt}. Second, the elements of $\bar{p}_{\text{opt}}$ are 7-bit integers, so we can save memory by storing them as bytes. This way, we are able to squeeze the arrays down to 636 bytes of shared memory.

In addition to the arrays, we also need to keep track of the bounding volumes, SAH costs, triangle counts, children, and identities

Size (k)	Subsets ($\bar{s}$)	Partitionings ($\bar{p}$)	Total work	%
2	21	1	21	2
3	35	3	105	11
4	35	7	245	25
5	21	15	315	33
6	7	31	217	22
7	1	63	63	7

Table 1: *Statistics for each subset size in Algorithm 2 with $n = 7$. The first three columns correspond to the loops on lines 10, 11, and 14 of Algorithm 2, respectively.* Total work *indicates the number of inner loop iterations executed for the given k in total, and the last column shows the overall distribution of the workload.*

Round	Subset sizes processed by 32 threads	Active
1	2 2 2 2 2 2 2 2 2 2 –	10
2	3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 – – – – – – – – – – – –	20
3	4 4 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 – – –	29
4	4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4	32
5	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 – – – – – – – – – – –	21

Table 2: *Pre-generated schedule for subsets of up to 5 leaves with $n = 7$. The processing consists of 5 rounds, and the schedule indicates which subset each thread should process in each round. We only show sizes of the subsets for clarity, not the actual values of $\bar{s}$.*

of the nodes. We store this data in the register file so that each thread stores the values corresponding to one node. To access the per-node data efficiently on NVIDIA Kepler GPUs, we utilize the *shuffle* compiler intrinsic that allows reading the registers of other threads in the same warp at a very low cost. The same functionality can be achieved on older GPUs by allocating an additional staging area in shared memory to temporarily hold one value per node. In practice, we can think of the nodes as being stored in a single array that we can read like any other array. Modifying a given node, however, is possible only from its designated thread.

4.5 Parallel Implementation

The most computationally intensive part of processing a treelet is finding the optimal partitioning for each subset of its leaves, corresponding to lines 10–23 in Algorithm 2. Since there are no dependencies between subsets of the same size, an easy way to parallelize this would be to repeatedly pick one subset for each thread until all subsets of the given size have been processed.

Table 1 shows the statistics for each subset size with $n = 7$. We see that most of the work is concentrated on sizes 4–6, whereas size 2 is practically free. We also see that the number of subsets tends to be very uneven, which means that parallelizing the computation over subsets of the same size alone will necessarily lead to low SIMD utilization. In particular, sizes 6 and 7 have the highest amount of work per subset, but offer only a few subsets to process in parallel.

Even though it is necessary for all subsets of size $k - 1$ to be ready before we can process size k to completion, it is still possible to process *some* subsets of size k before this. We can thus improve the SIMD utilization by allowing the processing of multiple subset sizes to overlap. Our approach is to process sizes $2 \dots n - 2$ in a unified fashion, and treat sizes $n - 1$ and n as special cases.

For sizes $2 \dots n - 2$, we use a pre-generated *schedule*, shown in Table 2 for $n = 7$. The schedule consists of a fixed number of processing rounds, and tells which subset each thread should process in each round, if any. The schedule can be generated for any treelet size and SIMD width using a simple algorithm that considers the rounds in reverse order and greedily includes as many subsets in the current round as it can without violating the dependency rules.

Since there are only a few subsets of size $n - 1$ and n, we choose to parallelize each subset over multiple threads. For $n - 1$, we use 4 threads per subset, and for n, we use all 32 threads to process the single subset. This is a good idea because in these cases, the number of partitionings is high enough for the inner loop to completely dominate the processing cost. Our approach is to consider only a fraction of the partitionings by each thread, and then use parallel reduction to merge the results at the end. Since $\bar{s}$ has a very specific bit pattern with $k \geq n - 1$, enumerating the partitionings considered by each thread is trivial compared to the general case.

AABB Calculation To determine the AABB for each value of $\bar{s}$ on lines 2–4 of Algorithm 2, we need to compute the minimum or maximum for the 6 scalar components of up to n individual AABBs. We parallelize the computation by assigning a group of 2^{n-5} consecutive subsets to each thread. These subsets share the same 5 highest bits of $\bar{s}$, so we first calculate an intermediate AABB considering only the leaves that correspond to these bits. To obtain the final AABBs, we then augment the result with each combination of the remaining leaves.

Treelet Formation We form the treelet by expanding it one node at a time in sequential fashion, maintaining a one-to-one mapping between nodes and threads. Even though we employ only the first $2n - 1$ threads, the overall process is still relatively cheap. At each step, we select the node with the largest surface area using parallel reduction, and then assign its children into two vacant threads. To avoid fetching full AABBs from memory for this, we maintain the values of $A(n)$ in a separate array throughout the optimization.

Reconstruction Reconstruction of the optimal treelet from $\bar{p}_{\text{opt}}$ works essentially the same way as treelet formation, except that we reuse the identities of the original internal nodes for the newly created ones. After the reconstruction, we calculate new AABBs for the internal nodes based on their children, repeating the process in parallel until the results have propagated to the treelet root. Finally, we store the nodes back to memory, bypassing the L1 cache in order to ensure that the results are visible to all SMs. As a minor optimization, we skip the output part of the algorithm in case we were not able to improve the SAH cost, i.e., $c_{\text{opt}}[2^n - 1] \geq C(\text{root})$.

4.6 Quality vs. Speed

The main loop of our BVH optimization kernel is organized according to the parallel bottom-up traversal algorithm by Karras [2012]. Each thread starts from a given leaf node and then walks up the tree, terminating as soon as it encounters a node that has not been visited by any other thread. Our idea is to form a treelet for each node encountered during the traversal, if its corresponding subtree is large enough to support our particular choice of n. We switch from per-thread processing (traversal) to per-warp processing (optimization) at the end of each traversal step, using the *ballot* compiler intrinsic to broadcast the set of valid treelet roots to the entire warp.

To determine whether a given subtree is large enough to support a treelet with n leaves, we utilize the fact that our intermediate BVH always stores one triangle per leaf. Since we already need to track the number of triangles for the purposes of Equation 2, we can use the same information to decide whether to accept a given node as a treelet root. However, the choice does not necessarily have to be made based on n — we are free to choose any $\gamma \geq n$, and only accept nodes whose respective subtrees contain at least γ triangles.

A full binary tree with m leaves can contain at most $2m/\gamma - 1$ subtrees with γ or more leaves, and we have found practical BVHs to also exhibit similar behavior. Given that our optimization kernel is virtually always dominated by treelet processing, we can describe its execution time as $\mathcal{O}(m/\gamma)$ to a sufficient degree of accuracy. This means that γ provides a very effective way to trade BVH quality for reduced construction time by concentrating less effort on the bottom-most nodes whose contribution to the SAH cost is low.

In practice, we need to execute multiple rounds of bottom-up traversal and treelet optimization in order for the SAH cost to converge. However, we have observed that the bottom part of the BVH generally tends to converge faster that the top part. This is not surprising considering that modifying the topmost nodes can potentially have a large impact on the entire tree, whereas modifying the bottom-most ones usually only affects small localized parts of the scene.

Based on this observation, it makes sense to vary the value of γ between rounds. We have found doubling the value after each round to be very effective in reducing the construction time while having only a minimal impact on BVH quality. Using $\gamma = n = 7$ as the initial value and executing 3 rounds in total has proven to be a good practical choice in all of our test scenes.

5 Triangle Splitting

Splitting large triangles is important for achieving high ray tracing performance in scenes where the distribution of triangle sizes is non-uniform. We choose to follow the idea of Ernst and Greiner [2007], where the AABB of each triangle is subdivided recursively according to a given heuristic, and the resulting set of AABBs is then used as the input for the actual BVH construction. Each AABB is associated with a pointer to the triangle that it originated from, which makes it possible to reference the same triangles from multiple parts of the BVH. The triangles themselves are not modified in the process.

Splitting triangles in a separate pass is compelling because it can be used in conjunction with any BVH construction method. However, it suffers from difficulty of being able to predict how a given split decision will affect the BVH in the end. As a consequence, existing methods [Ernst and Greiner 2007; Dammertz and Keller 2008] require considerable manual effort to select the right parameters for each particular scene, and they can even be harmful if the number of splits is too high. We aim to overcome these limitations by using carefully crafted heuristics to only perform splits that are likely to be beneficial for ray tracing performance.

5.1 Algorithm

Our approach is to first determine the number of times we wish to split each triangle, and then perform the desired number of splits in a recursive fashion. We control the overall amount of splitting with parameter β, allowing at most $s_{\text{max}} = \lfloor \beta \cdot m \rfloor$ splits for a scene consisting of m triangles. Limiting the number of splits this way results in predictable memory consumption, and it also avoids the need for dynamic memory allocation with animated scenes.

To determine how to distribute our fixed split budget among the triangles, we first calculate a heuristic priority p_t (Section 5.2) to indicate the relative importance of splitting triangle t compared to the other triangles. We then multiply p_t with an appropriately chosen scale factor D and round the result to determine the number of splits, i.e., $s_t = \lfloor D \cdot p_t \rfloor$. To fully utilize our split budget, we choose D to be as large as possible while still satisfying $\sum s_t \leq s_{\text{max}}$.

We find the value of D by first determining conservative bounds for it, and then using the bisection method to refine these bounds iteratively. We calculate the initial lower bound D_{min} based on the sum of triangle priorities, and the upper bound D_{max} by adjusting D_{min} based on how many splits it would generate:

$$D_{\text{min}} = s_{\text{max}} / \sum p_t \tag{3}$$

$$D_{\text{max}} = D_{\text{min}} \cdot s_{\text{max}} / \sum \lfloor D_{\text{min}} \cdot p_t \rfloor \tag{4}$$

We then bisect the resulting interval by iteratively replacing either D_{min} or D_{max} with their average, D_{avg}, depending on whether the average would exceed our split budget, i.e., $\sum \lfloor D_{\text{avg}} \cdot p_t \rfloor > s_{\text{max}}$. In practice, we have found 6 rounds of bisection to be enough to guarantee that the number of splits is within 1% of s_{max}. Even though this means that we execute a total of 8 parallel reductions over the triangles, finding D constitutes $\sim$1% of our total execution time.

We perform the actual splits by maintaining a set of *split tasks*, each specifying how many times we wish to split a particular AABB. To process a task, we split its AABB according to a heuristically chosen split plane. We then distribute the remaining splits among the two resulting AABBs proportional to the lengths of their longest axes. Denoting these lengths with w_a and w_b and the original split count with s, we assign $s_a = \lfloor (s-1)w_a/(w_a + w_b) + \frac{1}{2} \rfloor$ splits for the first AABB and $s_b = s - 1 - s_a$ for the second one. In case either of these counts is non-zero, we create a new task for the corresponding AABB and process it in the same fashion.

In practice, we implement the recursion in a SIMD-friendly fashion by maintaining a per-warp pool of outstanding split tasks in shared memory. We repeatedly pick 32 tasks from the pool and process them in parallel, inserting the newly created tasks back into the pool. Whenever the size of the pool drops below 32, we fetch more tasks from the global array of input triangles.

5.2 Heuristics

Intuitively, splitting triangles is beneficial for two reasons. First, it provides a way to reduce the overlap between neighboring BVH nodes by limiting their spatial extent. Second, it provides a way to represent the shape of individual triangles more accurately, reducing the number of ray-triangle intersection tests during ray traversal. In order to make good decisions about which triangles to split and which split planes to use, both effects have to be taken into account.

Split Plane Selection The most effective way to reduce node overlap is to split a large group of triangles using the same split plane. By virtue of constructing our initial BVH using spatial median partitioning, we already have a good idea which planes are going to be the most relevant. For example, we know that the root node of the initial BVH partitions the scene into two equal halves according to the spatial median of its x-axis. If a given triangle crosses this boundary, it will necessarily cause the two children of the root to overlap. Since the nodes near the root are likely to have a significant impact on the ray tracing performance, it is important to ensure that this does not happen.

Based on the above reasoning, our idea is to always split a given AABB according to the most important spatial median plane that it intersects. We define importance of a given plane in terms of how early it is considered during the construction of initial BVH. For a scene extending over the range $[0, 1]$ along each axis, the planes are given by $x = j \cdot 2^i$ where $i, j \in \mathbb{Z}$, and similarly for y and z. The value of i is negative, and values closer to zero correspond to more important planes. In practice, an efficient way to choose the split plane is to form two Morton codes based on the minimum and maximum coordinates of the AABB, and then find the highest bit that differs between them [Karras 2012].

Ideal Area To estimate how much the representation of a particular triangle can be improved, we calculate the lowest total surface area A_{ideal} of the representative AABBs that we can hope to reach through splitting. Assuming that we take the number of splits to the limit, we are essentially representing the triangle as a collection of infinitesimal boxes. For any axis-aligned projection ($\pm x, \pm y, \pm z$), the surface area of the projected triangle equals the sum of surface areas of the corresponding face of the boxes. We can thus express the total surface area of the boxes as

$$A_{\text{ideal}} = ||\mathbf{d}_1 \times \mathbf{d}_2||_1, \tag{5}$$

where $\mathbf{d}_1$ and $\mathbf{d}_2$ are edge direction vectors of the triangle and $||\cdot||_1$ indicates the L^1 norm. Each component of the cross product is equal to twice the area of a particular 2D projection. As a consequence, the result essentially represents the sum of 6 such areas, one for each face of the boxes.

Triangle Priority Splitting a given triangle is a good idea either if it crosses an important spatial median plane or if there is a lot of potential for reducing the surface area of its AABB. We thus combine these two factors to obtain an estimate for priority p_t. The fact that pre-process splitting is fundamentally an underdetermined problem makes it difficult to define the priority in a principled way. However, we have experimentally found the following formula to work robustly in all of our test scenes.

$$p_t = \left(X^i \cdot (A_{\text{aabb}} - A_{\text{ideal}})\right)^Y, \tag{6}$$

where i is the importance of the dominant spatial median plane intersected by the triangle, A_{aabb} is the surface area of the original AABB, and X and Y are free parameters. We have found $X = 2$ and $Y = \frac{1}{3}$ to give the best results in practice. However, we note that our method is not overly sensitive to the actual choice of the parameters — the important thing is to concentrate most of the splits on triangles for which both i and $A_{\text{aabb}} - A_{\text{ideal}}$ are high. It is generally a good idea to use a fairly low value for Y to avoid spending the entire split budget on a few large triangles in pathological cases.

6 Results

Our test platform is a quad-core Intel Core i7 930 and NVIDIA GTX Titan. The ray tracing performance measurements are carried out using the publicly available kernels of Aila et al. [2009; 2012]. We concentrate on diffuse inter-reflection rays because they have much lower viewpoint-dependence than primary, shadow, or specular reflection rays. The performance measurements are averages over a scene-dependent number of viewpoints. For individual objects we use two or three viewpoints, for simple architecture four or five, and for bigger scenes up to ten. The viewpoints try to capture all interesting aspects of the scene.

We use 20 test scenes to properly capture scene-to-scene variation, but due to space constraints detailed data is shown only for 4 scenes. The full result matrix is provided as auxiliary material. All statements about average, minimum, maximum refer to the full set of scenes. We focus on commonly used ray tracing test scenes, with additional city models from the Mitsuba renderer distribution. CONFERENCE, SIBENIK, CRYTEK-SPONZA, BAR, and SODA are widely used architectural models. ARMADILLO, BUDDHA, DRAGON, BLADE, MOTOR, and VEYRON are finely tessellated objects. FAIRY and BUBS have widely varying triangle sizes. HAIRBALL and VEGETATION have difficult structure. The four cities, CITY, BABYLONIAN, ARABIC, and ITALIAN show large spatial extents. SANMIGUEL combines architecture with fine geometric detail and vegetation. Thumbnails of the scenes are shown in top-right corner of Table 3.

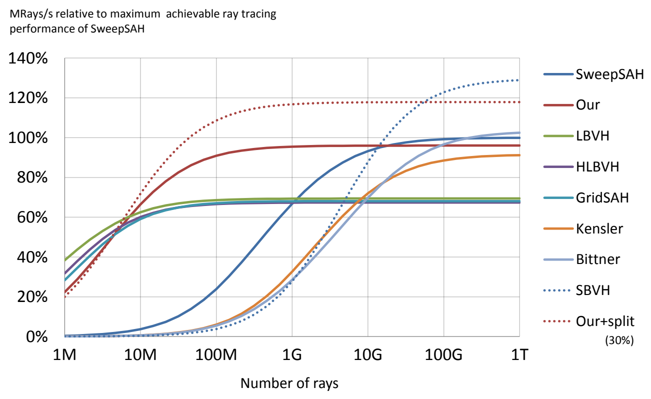


Figure 4: *Effective ray tracing performance relative to the maximum achievable ray tracing performance of SweepSAH, averaged over our 20 test scenes. We assume that the BVH is built at the beginning of the frame and then a certain number of rays (horizontal axis) are traced. Solid lines represent builders that do not use splitting, and dashes indicate splitting. Of the non-splitting variety, LBVH (GPU) offers the best tradeoff until about 7M rays per frame, our method (GPU) is best between 7M and 25G rays, at which point SweepSAH (CPU) takes over, and finally Bittner (CPU) dominates after 150G rays. The other builders are not Pareto optimal. When splitting is enabled, our method is best between 7M and 60G rays per frame, after which SBVH (CPU) becomes the best choice.*

Without Triangle Splitting We first analyze the performance without triangle splitting by comparing against six comparison methods: greedy top-down sweep SAH (SweepSAH) [MacDonald and Booth 1990], LBVH [Lauterbach et al. 2009; Karras 2012], HLBVH [Garanzha et al. 2011a], GridSAH [Garanzha et al. 2011b], tree rotations with hill climbing (Kensler) [Kensler 2008], and iterative reinsertion (Bittner) [Bittner et al. 2013], where the last two are initialized using LBVH. We use the authors' original implementations for HLBVH and GridSAH, and our own implementations for the rest of the methods. We choose SweepSAH as the baseline because it continues to be the de facto standard in BVH construction. For LBVH, we optimize the leaf nodes by collapsing subtrees into leaves to minimize SAH cost (instead of using single-triangle leaves). This is a well-known method, takes a negligible amount of time, and considerably improves the LBVH results. Of the comparison methods LBVH, HLBVH and GridSAH execute on the GPU, SweepSAH and Kensler are parallelized over four CPU cores, and Bittner runs on a single CPU core. Our method uses a fixed set of parameters for all test scenes: $n = 7$, $\gamma = 7$, we perform 3 rounds of optimization, and γ is doubled after every round.

Table 3 gives numbers for ray tracing performance, SAH cost, and build time for four scenes in addition to the average numbers over the 20 scenes. On average, our builder produces trees that offer 96% of the *maximum achievable ray tracing performance*[2] of SweepSAH (min 86% max 113%), while being significantly faster to construct. LBVH offers 3–4 times faster build than our method, but provides only 69% of the maximum ray tracing performance on the average, with significant scene-dependent variation: the performance is only 40–50% for all city scenes but over 80% for highly tessellated objects. Interestingly HLBVH and GridSAH result in lower ray tracing performance than LBVH. The primary cause of this oddity is that the original implementations of these algorithms reduce build time by using 30-bit Morton codes (instead of 60 bits), which is harmful in scenes that have larger extents, such as SAN-

[2]I.e., the ray tracing performance that can be achieved by excluding the build time, or equivalently, by tracing an infinite number of rays per frame.

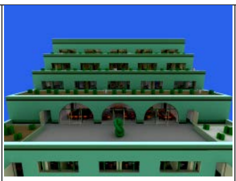
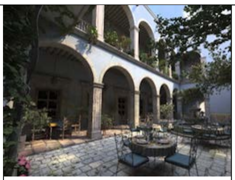
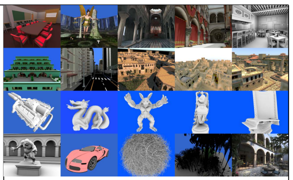
															
	CONFERENCE 282K tris			SODA 2.2M tris			DRAGON 870K tris			SANMIGUEL 10.5M tris			AVERAGE OF 20 SCENES relative to SweepSAH		
Builder	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** %	**SAH** %	**Build** %
SweepSAH	258.4	46.50	848 ms	330.2	78.26	8 s	229.7	56.74	3 s	86.7	20.13	50 s	100.0	100.0	100.00
Our	275.6	38.48	9 ms	301.8	71.04	56 ms	213.1	60.41	24 ms	83.5	17.38	274 ms	96.0	94.4	1.03
LBVH	179.4	64.59	2 ms	203.0	106.06	16 ms	188.5	70.00	7 ms	55.3	26.71	74 ms	69.4	131.5	0.28
HLBVH	185.0	60.91	6 ms	193.5	105.13	12 ms	189.1	68.54	8 ms	39.9	29.08	32 ms	67.4	129.3	0.57
GridSAH	187.8	61.96	7 ms	200.1	104.58	15 ms	193.8	66.12	11 ms	39.8	29.26	35 ms	68.1	129.1	0.70
Kensler	250.5	42.69	4 s	258.8	76.08	44 s	211.3	62.04	10 s	79.1	18.50	197 s	91.5	99.6	552.26
Bittner	278.7	36.51	1 s	356.5	59.39	60 s	215.6	57.26	50 s	98.2	15.69	334 s	103.3	87.7	1083.24

Table 3: *Measurements for seven different BVH builders in absence of triangle splitting. Performance is given in millions of diffuse rays per second, and the build time includes everything from receiving a list of triangles to being ready to cast the rays. The last three columns give average results for the full set of 20 test scenes relative to SweepSAH.*

	CONFERENCE (SBVH: 28% splits)			SODA (SBVH: 13% splits)			DRAGON (SBVH: 5% splits)			SANMIGUEL (SBVH: 13% splits)			AVERAGE OF 20 SCENES relative to SBVH		
Builder	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** Mrays/s	**SAH** **cost**	**Build** **time**	**Perf** %	**SAH** %	**Build** %
SBVH	378.4	38.71	7 s	405.2	67.33	37 s	237.9	55.59	20 s	118.5	18.57	136 s	100.0	100.0	100.00
Our (no splits)	275.8	38.48	9 ms	301.9	71.04	56 ms	213.1	60.41	24 ms	83.5	17.38	274 ms	76.5	111.5	0.14
Our (10% splits)	321.6	38.91	10 ms	386.0	68.81	63 ms	219.8	59.34	26 ms	109.2	19.95	307 ms	87.2	108.8	0.15
Our (30% splits)	358.1	39.32	12 ms	394.4	69.53	75 ms	220.9	59.22	30 ms	112.8	19.99	361 ms	91.0	108.0	0.17
Our (50% splits)	379.7	39.58	13 ms	398.9	70.43	86 ms	221.3	59.39	34 ms	112.8	20.05	413 ms	92.5	108.3	0.20
Our (match SBVH)	354.5	39.35	12 ms	386.7	68.69	65 ms	213.2	60.38	24 ms	110.1	19.94	314 ms	90.7	107.4	0.16
ESC (match SBVH)	253.5	59.71	–	276.1	85.50	–	213.0	60.52	–	92.3	29.83	–	79.1	128.3	–
EVH (match SBVH)	244.2	45.33	–	245.0	73.47	–	210.9	60.93	–	70.5	21.22	–	68.9	129.3	–

Table 4: *Measurements with triangle splitting. We test our method by limiting the splits to 0%, 10%, 30%, 50%, and to the number of splits that SBVH produced. On the last two rows we replaced our splitter with early split clipping (ESC) and edge volume heuristic (EVH).*

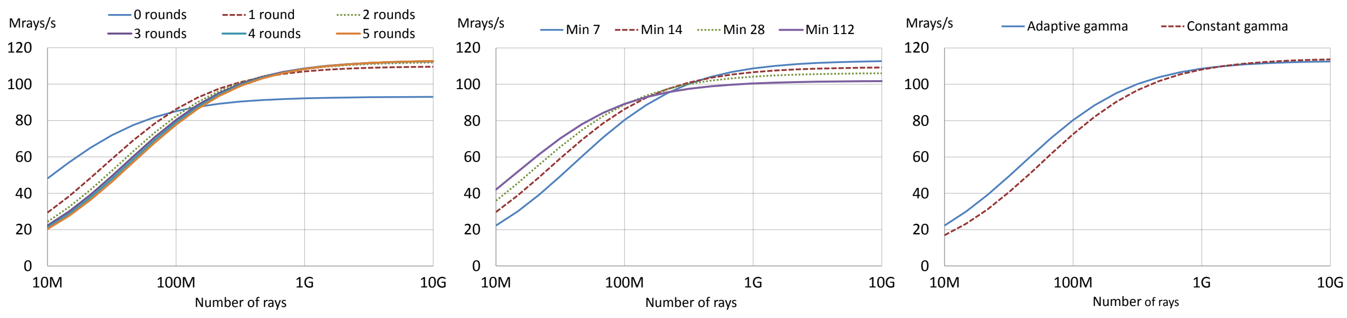
MIGUEL. Also, our LBVH implements a proper, SAH-dictated cutting to leaf nodes, whereas HLBVH and GridSAH use cruder approximations (anything ≤ 4 is considered a leaf). Bittner's method produces trees whose maximum achievable ray tracing performance is about 3% higher than for SweepSAH, but the build time is somewhat longer mainly due to lack of parallelization. Compared to Kensler's tree rotations, our method offers 5% higher ray tracing performance when averaged over all scenes. However, the difference is consistently greater for architectural models (e.g. 17% in SODA), where it is more likely for tree rotations to get stuck in a local optimum.

In order to draw principled conclusions about the relative order of different builders, the rest of this section focuses on a situation where one needs to build a BVH before tracing a certain number of rays using it. The interesting number of rays per build (or frame) depends heavily on the application: for example a game AI might need only 10K queries per frame, interactive rendering from 10M or 100M, and high-quality visualization 100G or more. At the extreme, for static parts of the scene the BVH could be pre-computed and at runtime an unbounded number of rays cast in it. Figure 4 visualizes the effective ray tracing performance of different builders, taking into account the fact that a BVH needs to be built before rays can be traced. We see that for very low ray counts LBVH is the ideal option, and for very high ray counts SweepSAH and ultimately Bittner become the best options. Our method is best for a significant part of the domain, when between 7 million and 25 billion rays need to be traced per frame.

It is worth mentioning that our method consistently produces a lower SAH cost than SweepSAH, but still leads to slightly slower ray tracing performance on the average. This suggests that the SAH cost fails to capture some important property of the BVH.

With Triangle Splitting Triangle splitting further improves the performance, as shown in Figure 4. Considering maximum achievable ray tracing speed, SBVH [Stich et al. 2009] is ∼30% faster than SweepSAH in our test scenes — although the average hides large scene-dependent variation from no benefit in highly tessellated objects to $\sim 2\times$ in VEGETATION. But since SBVH is a slow builder, the benefit starts to materialize only after 20G rays. We set SBVH's α to 10^{-5} in all scenes, except that 10^{-4} was needed in HAIRBALL and VEGETATION to avoid running out of memory and 10^{-6} in SANMIGUEL to see any splitting at all. Since our splitter is very fast, enabling it starts to pay off already around 7M rays. Interestingly, only LBVH, our method with splitting, and SBVH are on the Pareto optimal frontier in Figure 4. Our method with splitting is the fastest between 7 million and 60 billion rays per frame.

Table 4 gives numerical results for splitting. In addition to SBVH, we show our builder with different split budgets, and variants of our builder using early split clipping (ESC) [Ernst and Greiner 2007] and edge volume heuristic (EVH) [Dammertz and Keller 2008]. The maximum achievable ray tracing performance of our method increases steadily when the number of allowable splits grows, but starts to plateau around 30%, which is the budget we use for all other measurements in this paper. With this budget we achieve on



a) Number of rounds b) Minimum subtree size (γ) c) Adaptive γ

Figure 5: *Effect of our parameters in* SANMIGUEL. *a) The number of rounds affects the maximum achievable ray tracing performance in a significant way, until the second or third round. b) The minimum subtree size affects the tradeoff between build time and ray tracing performance. c) Adaptive γ reduces build time without noticeably reducing the maximum achievable ray tracing performance.*

the average 91% of the ray tracing performance of SBVH. The core strength of SBVH's splitting strategy is that it can place the splits only where they are deemed beneficial in a global sense, thanks to its top-down nature, but the downside is unpredictable memory consumption. We on the other hand cannot easily know how many splits should be done, and thus specify a limit for the memory consumption instead. Somewhat surprisingly, both ESC and EVH often fail to provide performance boost when used together with our builder. It is possible that scene-specific parameter tuning could improve their results somewhat.

Parameters All of the results in this paper were computed using $n = 7$ leaf nodes per treelet. On the average, the maximum achievable ray tracing performance is very similar also for $n = 6$ and $n = 8$. Although $n = 6$ is approximately 30–40% faster to build, it fails to find some important modifications for example in SODA, leading to lower quality trees. Going to $n = 8$ improves the trees only marginally at the cost of almost tripling the build time.

Figure 5 illustrates the sensitivity of our method for other parameters in SANMIGUEL. Increasing the number of optimization rounds makes the build slower, but significantly improves the maximum ray tracing performance for the first two or three iterations. Particularly in SODA we see a clear improvement still on the third round, and thus we use three rounds for all results in this paper. Increasing the initial value of minimum subtree size (γ) makes the build faster, but it also penalizes the maximum ray tracing performance, and we chose not to use the optimization in the measurements. Finally, it is clearly beneficial to double γ after each round.

Our BVH consumes 64B per internal node and 48B per triangle. As we have one internal node per triangle in the intermediate BVH, we need 112B per triangle in total. We do not need any additional memory besides this, because we can reuse the output triangle array to store all the temporary data needed during the optimization.

7 Future Work

Our method makes it possible to use high-quality BVHs in a large class of applications where offline construction is currently not feasible, ranging from interactive visualization and editing to rendering of feature films. We have demonstrated the general idea of treelet optimization to be a robust and effective alternative for the previous top-down approaches, which opens up a number of interesting avenues for future work. So far we have only looked at finding the optimal topology for small treelets, but it would also be possible to employ approximate methods to restructure significantly larger treelets, enabling more extensive modifications to the BVH. Since any proposed modification can be discarded if it does not improve the SAH cost, one could even combine multiple such methods to overcome the weaknesses of any single one.

The optimization could be accelerated by only restructuring treelets that actually matter, assuming there was a quick way to estimate how far a given treelet is from its optimum. We believe that our method can also be combined with other BVH quality metrics besides the SAH cost, offering a possibility to improve the ray tracing performance even further.

We encourage researchers interested in comparing against our implementation to contact us.

Acknowledgements

Samuli Laine, Jaakko Lehtinen, Sami Liedes, David McAllister, and anonymous reviewers for comments. Anat Grynberg and Greg Ward for CONFERENCE, University of Utah for FAIRY, Marko Dabrovic for SIBENIK, Ryan Vance for BUBS, Samuli Laine for HAIRBALL and VEGETATION, Guillermo Leal Laguno for SANMIGUEL, Jonathan Good for ARABIC, BABYLONIAN and ITALIAN, Stanford Computer Graphics Laboratory for ARMADILLO, BUDDHA and DRAGON, Cornell University for BAR, Georgia Institute of Technology for BLADE.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics 2009*, 145–149.

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. Tech. Rep. NVR-2012-02, NVIDIA.

BITTNER, J., HAPALA, M., AND F., H. 2013. Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1, 85–100.

DAMMERTZ, H., AND KELLER, A. 2008. The edge volume heuristic – robust triangle subdivision for improved BVH performance. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 155–158.

ERNST, M., AND GREINER, G. 2007. Early split clipping for bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 73–78.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. K. 2011. Simpler and faster HLBVH with work queues. In *Proc. High Performance Graphics*, 59–64.

GARANZHA, K., PREMOZE, S., BELY, A., AND GALAKTIONOV, V. 2011. Grid-based SAH BVH construction on a GPU. *Visual Computer 27*, 6–8, 697–706.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5, 14–20.

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. High-Performance Graphics 2012*, 33–37.

KENSLER, A. 2008. Tree rotations for improving bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 73–76.

KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. 2012. Fast, effective BVH updates for animated scenes. In *Proc. Interactive 3D Graphics and Games*, 197–204.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 3, 153–166.

NVIDIA. 2012. *NVIDIA OptiX Programming Guide 3.0*, November.

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High Performance Graphics*, 87–95.

POPOV, S., GEORGIEV, I., DIMOV, R., AND SLUSALLEK, P. 2009. Object partitioning considered harmful: space subdivision for BVHs. In *Proc. High Performance Graphics 2009*, 15–22.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics 2009*, 7–13.

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 33–40.

WALD, I. 2012. Fast construction of SAH BVHs on the Intel many integrated core (MIC) architecture. *IEEE Trans. Vis. Comput. Graph. 18*, 1, 47–57.

WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. 2008. Fast agglomerative clustering for rendering. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 81–86.

WOOP, S. 2004. A ray tracing hardware architecture for dynamic scenes. Tech. rep., Saarland University.