# Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees

P. Ganestam[1,2], R. Barringer[2], M. Doggett[2], and T. Akenine-Möller[1,2]
[1]Intel Corporation [2]Lund University

**Figure 1.** The San-Miguel scene (7,842,744 triangles) overlaid with a visualization of its Bonsai bounding volume hierarchy (BVH). The Bonsai BVH of San-Miguel is constructed in 478 ms using a 2.6 GHz quad core laptop CPU (Intel 4950HQ) and rendering performance is 107% improved compared to the rendering performance of the same scene using a BVH built with the sweep SAH method.

## Abstract

We present an algorithm, called Bonsai, for rapidly building bounding volume hierarchies for ray tracing. Our method starts by computing midpoints of the triangle bounding boxes and then performs a rough hierarchical top-down split using the midpoints, creating triangle groups with tight bounding boxes. For each triangle group, a mini tree is built using an improved sweep SAH method. Once all mini trees have been built, we use them as leaves when building the top tree of the bounding volume hierarchy. We also introduce a novel and inexpensive optimization technique, called mini-tree pruning, that can be used to detect and improve poorly built parts of the tree. We achieve a little better than 100% in ray-tracing performance compared to a "ground truth" greedy top-down sweep SAH method, and our build times are the lowest we have seen with comparable tree quality.

 ISSN 2331-7418

## 1. Introduction

In order to ray trace [Whitted 1980] a scene with path tracing [Kajiya 1986], for example, a spatial acceleration data structure [Kay and Kajiya 1986; Pharr and Humphreys 2010] needs to be built. The task of this structure is to speed up the determination of what a ray intersects in a three-dimensional scene. One of the most popular spatial acceleration data structures is the bounding volume hierarchy (BVH). For animated scenes, the entire BVH, or parts of it, needs to be rebuilt every frame, and therefore, the BVH generation needs to be fast. However, it is also important that the generated trees are of high quality so the subsequent ray-tracing process becomes as fast as possible.

Top-down, greedy sweep surface area heuristic (SAH) methods [MacDonald and Booth 1990], simply abbreviated *sweep* SAH here, are known to generate high-quality trees. We present a highly efficient implementation of the sweep SAH method and use that as a building block in our new algorithm for generating BVHs. Our algorithm is surprisingly simple, parallelizes well, and is easy to implement. As we will show in our results, our BVHs can be built faster than binning SAH methods [Wald 2007], and our tree quality is better in that the subsequent ray tracing is faster.

In Sections 2 and 3 we review previous work and BVH generation background. In Section 4, we present our implementation of the sweep SAH method with some extra optimizations, followed by our novel BVH generation algorithm. Implementation details are described in Section 6 and results are presented in Section 7. Finally, we offer some conclusions.

## 2. Previous Work

An important component of light transport simulation performance is the time it takes a ray to find the surface intersection. Tremendous gains in ray-tracing performance have been achieved through improved traversal algorithms and improved data structures. One of the first uses of hierarchical storage was presented by Clark [1976], who used them to improve the determination of visible surfaces. Later Rubin and Whitted [1980] developed this idea using parallelepipeds for ray tracing. To ensure that the best hierarchy was constructed, Goldsmith and Salmon [1987] presented the surface area heuristic (SAH) that computes the surface area for new nodes to find the best potential split of a bounding volume. MacDonald and Booth [1990] later formalized SAH. Walter et al. [2008] used SAH to build trees using a bottom-up, node-merging approach, but this approach requires long execution times. Recently, Gu et al. [2013] demonstrated a real-time, multi-threaded CPU approximation to Walter et al.'s agglomerative clustering algorithm. While showing impressive results, we show in our results, using their provided source code, that our top-down algorithm running on a multi-threaded CPU can build and trace scenes faster.

Aila et al. [2013] extended the SAH metric by proposing additional quality metrics for tree construction and, hence, improved ways to measure ray-tracing performance. They introduced two terms—the first term accounted for the fact that many rays start or terminate inside the scene, whereas SAH assumes they do not. They called the first term end-point overlap (EPO); it takes into account the area of the surfaces within each node. Second, they showed how to model SIMD performance by taking into account the number of leaf nodes intersected by a ray, using their leaf-count variability (LCV) term. LCV is computed as ray tracing is performed, which makes it a good measure for explaining performance, but impractical for BVH construction.

The construction of BVHs typically follows a top-down approach where a bounding volume of the entire object is split into two child volumes. These child volumes are recursively split, and before splitting, the SAH is used to estimate the cost of each potential split. While this type of exhaustive search can generate trees with very low SAH cost, it can take a very long time, so faster methods are often used. A popular approximation is binned SAH [Wald 2007; Wald 2012], which limits the number of potential split planes to a fixed number.

To further improve performance and utilize the parallel capacity of GPUs, Lauterbach et al. [2009] presented a technique called linear BVH (LBVH), which constructed a BVH by first generating a Morton code for each primitive, then using a parallel GPU algorithm to sort them, and finally recursively bucketing primitives based on the bits in their Morton codes. HLBVH [Pantaleoni and Luebke 2010] improved this technique by using a two-level hierarchical sort that used the upper bits of the Morton code to do an initial sort. Pantaleone et al. [2010] used a similar two-level build approach in their stream-based out-of-core BVH construction algorithm. Garanzha et al. [2011a] simplified the bookkeeping for the HLBVH algorithm and used work queues and binary search. Karras [2012] improved parallel construction time of this group of algorithms by creating node indices and keys using a binary radix tree that allowed creation of connections between parent and child nodes. A related hierarchical GPU-based approach is presented by Garanzha et al. [2011b]. In their work, a hierarchical grid is computed over the scene and used to construct the BVH using SAH.

Triangle splitting is an important technique to handle difficult scenes with a wide variety of triangle sizes. Havran and Bittner [2002] presented the idea of split clipping, where the bounding box of an object is split to reduce empty overlap between object bounding boxes and *kd*-tree nodes. Ernst and Greiner [2007] applied a similar concept to BVHs by splitting triangle bounding boxes in a preprocess, before using a typical BVH construction pass. Stich et al. [2009] and Popov et al. [2009] proposed similar ideas, where primitives are considered for splitting into children during BVH construction, which resulted in tighter bounding boxes on a larger range of triangles than previous approaches.

Further performance enhancement can be achieved by optimizing existing trees. Kensler [2008] presented a method of improving a BVH by locally rearranging nodes or using tree rotations. Bittner et al. [2013] also refined existing BVHs by selecting expensive SAH nodes for optimization, removing them, and then reinserting their children at locations with minimal cost. Karras and Aila [2013] selected groups of nodes in treelets and performed an exhaustive search for the optimal treelet in parallel on GPUs.

## 3. Background BVH Generation

As background, we first review the surface-area heuristic (SAH) [Goldsmith and Salmon 1987; MacDonald and Booth 1990], which is used extensively in spatial data structure generation. The SAH cost for a bounding volume hierarchy (BVH), with similar notation as Karras and Aila [2013], is

$$C_I \sum_{n \in I} \frac{A(n)}{A(\text{root})} + C_L \sum_{n \in L} \frac{A(n)}{A(\text{root})} + C_T \sum_{n \in L} \frac{A(n)}{A(\text{root})} N(n). \tag{1}$$

This formula expresses the expected cost of traversing a random ray through the BVH, such that the ray does not terminate inside the scene geometry. The set of internal nodes is denoted by $I$, and $L$ is the set of leaf nodes. The function $A$ computes the surface area of a node's bounding volume and the function $N$ represents the number of triangles in a leaf node. The constants $C_I$ and $C_L$ are the traversal costs of an internal node and a leaf node, respectively, and $C_T$ is the cost for intersecting a triangle. Karras and Aila use $C_I = 1.2$, $C_L = 0$, and $C_T = 1$.

To determine the SAH cost of a node, $n$, we use the standard formulation, where we again use a similar notation as Karras and Aila [2013], i.e.,

$$C(n) = \begin{cases} C_I A(n) + C(n_l) + C(n_r), & n \in I, \\ C_T A(n) N(n), & n \in L. \end{cases}$$

The left and right child nodes are denoted $n_l$ and $n_r$, respectively. Note that the first formula is an expression of splitting $n$ into a left and a right child, while the second represents the cost of making a leaf node of the triangles.

## 4. Our Implementation of Sweep SAH

The sweep SAH BVH algorithm was introduced by MacDonald and Booth [1990], and one often uses a top-down, greedy approach to build such trees. Sweep SAH is commonly used as a comparison algorithm due to its high-quality trees. However, most implementations seem relatively slow. In this section, we will adapt a partitioning trick from kd-tree building to BVHs and then describe a very efficient implementation.

Sweep SAH is a top-down recursive algorithm that, at each recursion, tries to partition a set of primitives into two subsets that minimize the surface area heuristic. The initial set to be partitioned is all primitives in the scene, and recursion stops when a partition cannot improve the cost of the tree. The SAH metric is minimized by sweeping over primitives along the $x$-, $y$-, and $z$-axis. For this sweep to work, primitives need to be sorted along each coordinate axis. Typical implementations do this by sorting the primitives along the axis to test before each sweep. However, we have discovered that this is unnecessary work.

In the spirit of previous work on kd-tree building [Wald and Havran 2006; Zhou et al. 2008; Wu et al. 2011], we sort all primitives *once* along each coordinate axis before any recursion takes place and keep these three arrays sorted within each subset during recursion. This allows us to improve performance without sacrificing tree quality, which is in contrast to the binned SAH approach [Wald 2007], where quality is often reduced. The sweep part of the algorithm simply performs a sweep over the correctly sorted array, so no additional sorting is required. Once the partition that minimizes the SAH metric is found, we need to ensure that all three arrays are correctly partitioned and sorted within the two subsets.

Without loss of generality, we assume that $x$ is the coordinate axis along which we chose to partition the primitives. This means that the $y$- and $z$-arrays need to have the same primitives in each subset as the $x$-array, but ordered by the $y$- and $z$-axis within each subset. We do this by flagging all triangles depending on which side of the pivot they are on along $x$. Then, using this flag, a partition of the primitives in $y$ and $z$ is performed, while preserving order. Partitioning is a fast and simple operation that runs in $O(n)$ time. This is in contrast to any comparison-based sorting algorithm, which would take $O(n \log n)$ time at best. Thus, the sweep recursion is improved from running in $O(n \log^2 n)$ time to instead execute in $O(n \log n)$ time.

In addition to the algorithmic improvements, we have found that many parts of the SAH algorithm lends itself well to vectorization and threading. Instruction-level parallelism and SIMD is exploited during SAH minimization by sweeping over multiple triangles at the same time. In our implementation, we successfully utilize 8-wide AVX2 instructions for the sweep. Thread-level parallelism is achieved by branching off the two subsets as new thread tasks at each recursion. Threads are then coordinated using a work list with task information. Since each subset can be processed independently, little synchronization is required. Additionally, the initial sorting along each coordinate axis can be performed using a parallel sorting algorithm. We currently use a radix sorter, where each axis $(x, y, z)$ is sorted in a separate thread. Further details relevant to the implementation are presented in Section 6.

## 5. Bonsai BVH Algorithm

The basic idea of our approach to rapidly building bounding volume hierarchies (BVHs) is illustrated in Figure 2. Very briefly, the triangles are partitioned into groups

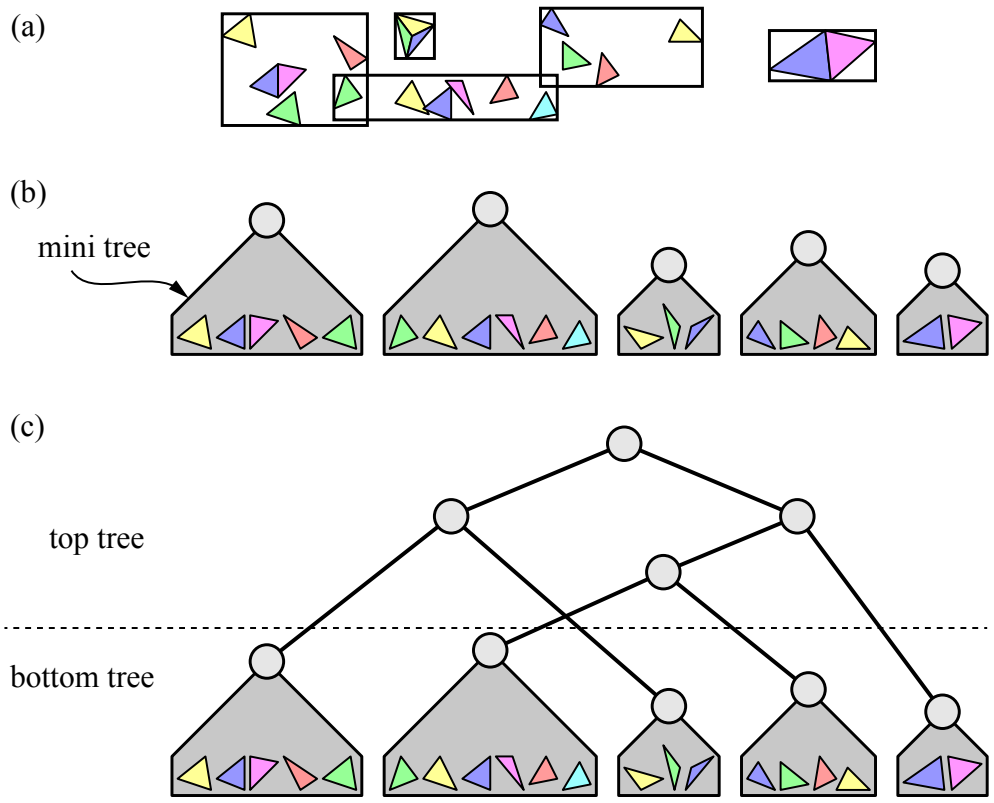


**Figure 2.** Illustration of our BVH tree builder in two dimensions. (a) In an initial pass, groups of triangles are generated. (b) For each group of triangles, an SAH-optimized mini tree is built using the algorithm in Section 4. (c) The top tree is built using an SAH-optimized builder as well. Pruning is *not* shown in this illustration.

with a user-defined size, and then a *mini tree* is built for each group, and finally, the mini trees can be seen as leaf nodes in a top-tree build. If a bounding box is computed for each triangle group as part of the grouping, then the top tree and all the mini trees can be built in parallel. To improve tree quality further, we have developed a novel mini tree pruning algorithm (Section 5.4), which can be applied before the top tree is built. However, the pruning algorithm is dependent on the mini trees, and therefore, the top tree must be built after the mini trees. We make extensive use of mini trees and pruning, and hence decided to call the entire algorithm *Bonsai*.

The Bonsai algorithm is summarized by the following list of operations:

1. Compute the midpoint for each triangle.

2. Mini tree selection: split the set of midpoints hierarchically into groups of triangles.

3. Use efficient implementation of sweep SAH (Section 4) to build a mini tree per triangle group.

4. [optional] Mini tree pruning, i.e., find and optimize mini trees with subtrees that cause less optimal ray-tracing performance.

5. Top-tree construction using the mini trees as leaves.

These five steps are described in more detail in the following subsections.

### 5.1. Compute Midpoints

Initially we loop over all triangles, where the *midpoint*, i.e., the center point of a triangle's axis-aligned bounding box, is computed for each triangle. Computing midpoints maps well to both thread- and instruction-level parallelism.

### 5.2. Mini Tree Selection

The purpose of the second step is to find a number of relatively small groups of triangles, where the triangles of each group is spatially coherent.

It is common to use triangle bounding-box midpoints to determine the sorted order of triangles in the $x$-, $y$-, and $z$-dimensions as well as to determine whether a triangle is to the left or to the right of a plane, but also to determine to which bin a triangle belongs. However, in many algorithms, the bounding boxes of triangles enlarge the bins or left and right bounds, and then a hierarchical top-down split follows using these boxes. We have found that it is considerably faster to use only the triangle midpoints rather than the minimum and maximum of the bounds of all vertices. In

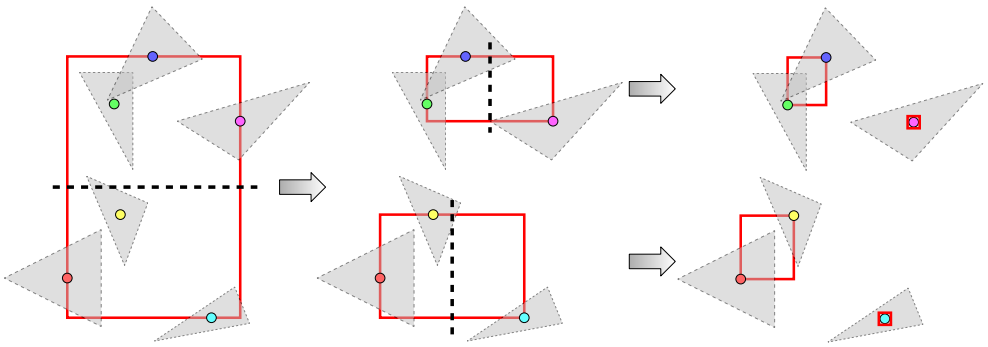

**Figure 3.** Our hierarchical split uses the triangle midpoints (colored circles) to generate triangle groups using a top-down approach. In each step, the bounding box of the triangle midpoints in the set is computed and used in the next split. We always split along the longest axis and in the middle.

addition, long sliver triangles are simply treated as points, which avoids problems where long boxes do not become subdivided.

In Figure 3, we illustrate our hierarchical split using triangle midpoints. The bounding box of the current set of triangle midpoints is calculated, and the set of triangle midpoints are simply split into two subgroups using the center point of the longest axis of the parent box. Each subgroup computes its own bounding box of the triangle midpoints, and then the hierarchical split continues until fewer or equal to $N$ triangles are located in each triangle group. The threshold $N$ is a parameter that can be chosen for a particular platform, depending on cache sizes, etc. Again, we achieve a high computational efficiency by exploiting thread-level parallelism at each hierarchical subdivision and instruction-level parallelism when computing the triangle midpoint bounds.

### 5.3. Mini Tree Construction

In the third step of our mini tree BVH algorithm, we compute an SAH-optimized subtree, called a *mini tree*, for the triangles in each group using our implementation of sweep SAH (Section 4). In theory, any method, such as, for example, LBVH [Lauterbach et al. 2009], HLBVH [Pantaleoni and Luebke 2010; Karras 2012], and binned SAH [Wald 2007], could be used here. However, our sweep SAH implementation results in the same tree quality as the greedy, top-down sweep SAH [MacDonald and Booth 1990], and it is important to generate high-quality trees for the mini trees in order to get good overall tree quality. It is also possible to introduce triangle-splitting techniques [Ernst and Greiner 2007; Stich et al. 2009; Karras and Aila 2013] here, but this is beyond the scope of our work and is something we want to investigate in the future.

When building mini trees, we get even better hardware utilization and thread-level parallelism compared to using our implementation of sweep SAH (Section 4) for all triangles in the scene, since each mini tree is built with only one thread each. In addition, situations like sorting the three index arrays for a full sweep SAH build, using only three threads are avoided. As long as there are mini trees to build, all threads will have completely parallel tasks to process.

### 5.4. Bonsai Pruning

Mini tree pruning is a novel technique that we introduce in order to recover tree quality lost due to potentially poorly chosen mini tree triangle groups in the selection algorithm (Section 5.2). Since the mini tree selection does not take SAH into account, the separation of large triangles from groups of smaller triangles will be rather arbitrary. So even though each mini tree is SAH optimized, the initial choice of triangles for a mini tree may be quite poor. As a result, both the top tree and the mini trees may suffer from reduced tree quality. Although, for some scenes, such as Hairball, midpoint-split works surprisingly well as a BVH build heuristic.

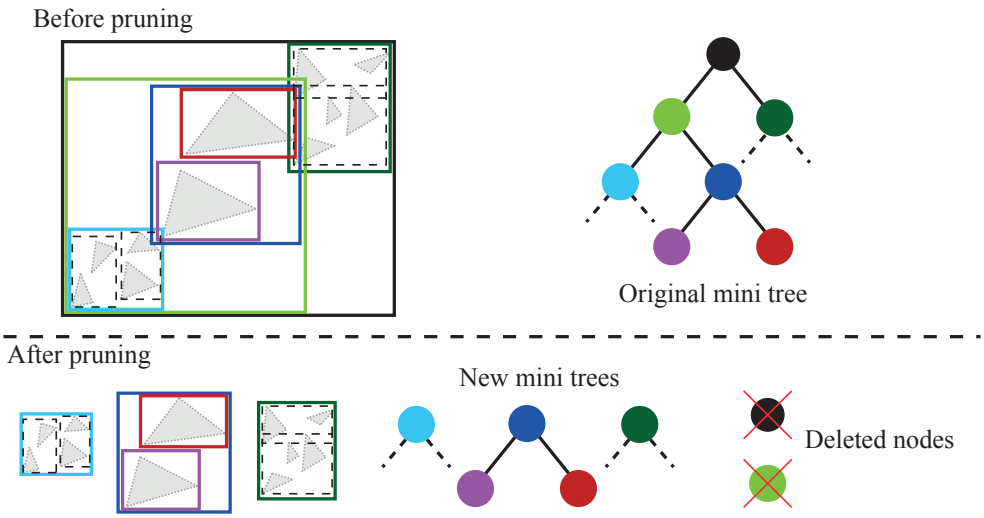


**Figure 4.** In mini trees, whose root box has a surface area larger than a threshold, our pruning algorithm performs a depth-first search in the mini tree to find better nodes to use as mini tree root nodes. Nodes that have a surface area smaller than the threshold (light blue, dark blue, and dark green) are found, and their subtrees are selected as new mini trees. The old mini tree root node (black) and all nodes between it and the new mini tree root nodes (light green) were parts of a tree created by poor mini tree selection and can be safely deleted. The remaining nodes (light blue, dark blue, and dark green) are used as new mini tree root nodes in the subsequent top-tree build.

Our pruning algorithm searches for mini trees that have a surface area larger than some user-defined threshold, $T$. For each such mini tree, a depth-first traversal searches for the first nodes that are smaller than $T$, and such nodes become new mini trees that are used in the top-tree construction (Section 5.5). All nodes between the found nodes and the mini tree root node are deleted, and the remaining nodes are added back as mini trees for the top-tree construction. This is illustrated in Figure 4. By pruning mini trees, we will find misplaced triangles (or entire misplaced subtrees) and just add them to the top-tree index list as mini tree roots. The effect is that difficult (large) triangles or difficult regions of a mini tree will be pushed up in the hierarchy and rebuilt with the top-tree builder among equally sized nodes. The threshold value, $T$, is simply a fraction of the average surface area of all the original mini tree root boxes. We present results with $T = 0.1$ and $T = 0.01$.

The pruning algorithm relies solely on the already computed mini trees (e.g., it does not split triangles or build any new data structures), and all it really does is traversing a mini tree at most once, so its addition to the overall build time is very small for most scenes. There are exceptions to all rules, and the Hairball is one such case. Since it is evenly tessellated and has an even distribution of triangles,

the variance of mini tree root node areas is small, and thus Hairball build-time is quite sensitive to pruning. For future work, we plan to use a fraction of the standard deviation of the mini tree root box areas as a threshold in order to avoid this.

It could be worth mentioning that mini tree pruning is not exclusive to the mini tree BVH algorithm. Any already constructed BVH could potentially benefit from pruning. As an example, pruning could start at any node with less than a user-defined number of triangles referenced by the sub tree.

### 5.5. Top-Tree Construction

The top-tree construction is similar to sweep SAH, except that now we have a set of mini trees, each with an axis-aligned bounding box and a fully built subtree, and we need to build the top part of the tree based on these. Also, while performing the sweeps, there is no need to weigh in the SAH cost to the parent surface area as is done in normal sweep SAH, since the top-tree nodes will always be split as far as possible. Building the top tree can be done using any appropriate method. We use our implementation of sweep SAH (Section 4).

## 6. Implementation

We have implemented both our sweep SAH (Section 4) and Bonsai BVH (Section 5) with as much focus as possible on both thread-level and instruction-level parallelism.

The first consideration to achieve good parallelism is data layout. We store our vertex data in three arrays of vertices where each vertex has four values, $[x, y, z, 0]$. A triangle is composed of the $i$th vertices from the three arrays where $i$ is the triangle index. This layout maps well to SIMD execution when computing midpoints and triangle bounds. The midpoints are stored in three float arrays, one for each dimension. We keep the triangle bounding boxes in an array of eight values per bounding box, $\mathbf{b} = [x_{min}, y_{min}, z_{min}, 0, x_{max}, y_{max}, z_{max}, 0]$, and all bounds are arranged in an array as $[\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}]$, where $n$ is the number of triangles in the scene. With this layout, each triangle bounding box can be loaded into a single 256-bit AVX2 register.

In our implementation of sweep SAH, denoted SweepSAH from now on, each recursion in the sweep algorithm spawns a new thread task to the left child, while the right child is constructed using the current thread. The sweep loop from left to right operates on eight triangle indices per iteration, that is, eight triangle bounds are loaded and accumulated to eight potential left-side bounding boxes. The left to right sweep is quite similar. One situation when SIMD instructions are not used is the sorted order-preserving index partitioning described in Section 4. Memory reads and writes while partitioning are simply too scattered to benefit from SIMD instructions, and the partitioning algorithm does not have good SIMD features. However, partitioning consumes only a small part of the total run time, and so is generally not a problem.

In Section 5.1(midpoint computation), we compute four midpoints per loop iteration using 256-bit AVX2 registers. Even though triangle vertices are loaded in order to compute the midpoints using the triangle bounding boxes, it does not pay off to save the bounding boxes to memory at this stage of the algorithm. A serial loop operating on independent data lends itself well to thread-level parallelism, and our threading scheduler simply assigns the available hardware threads to 1024-sized segments of the loop range. Once a thread finishes computing its segment, it is assigned a new segment of 1024 iterations of the loop.

The mini tree selection (Section 5.2) is designed in a recursive fashion, where we spawn new thread tasks at each recursion. We compute the bounding boxes around the midpoints using 8-wide AVX2 registers and the 8-wide min and max intrinsic functions. Since we work with midpoints and not bounding boxes, we can read eight of them per iteration and find the min and max of all eight midpoints using only six SIMD instructions (one min and one max AVX2 instruction per dimension). Without 8-wide AVX2, the same computation would require 16 min and max operations per dimension.

Triangle bounding boxes are computed and stored during mini tree construction (Section 5.3). The reason why this is not done earlier is that now it is known which triangles belong to which mini tree, and all data necessary is gathered and computed in pre-allocated thread local memory. Operating in thread local memory improves caching and removes false sharing between threads. Each mini tree is built by only one thread, and so there are no idle threads as long as there are mini trees left to construct. This is slightly different from SweepSAH, since there is no need to spawn new thread tasks while recursing down the tree.

It is not straightforward to map the Bonsai pruning algorithm to SIMD instructions, since the algorithm basically just traverses the constructed mini trees. The average surface-area computation of mini tree root nodes benefits from SIMD instructions but is a tiny part of pruning. However, the traversal has thread-level parallelism just as SweepSAH and mini tree selection, and in addition, each mini tree can be pruned in parallel. The only synchronization is when a new mini tree root is found, and the new mini tree root node bounds are written to the top-tree's bounds array.

The top tree (Section 5.5) is built in a similar manner as SweepSAH.

## 7. Results

All our results have been generated on a Macbook Pro laptop with Iris Pro 5200 integrated graphics processor. More specifically, the CPU is a 4950HQ, which has eight hardware threads at 2.6 GHz. All BVH building is done entirely on the CPU cores, while ray tracing is done using both the CPU cores and the GPU. More precisely, BVH traversal and triangle intersections are done using the GPU while shading com-

putations are done using the CPU. Note that for all our comparisons, we use a path tracer, which means that the rays are highly incoherent after a few bounces. Our baseline algorithm for comparison is the efficient implementation of sweep SAH as described in Section 4. This method is denoted SweepSAH; note that it generates the same high-quality trees as a standard sweep-based SAH BVH algorithm. Furthermore, we compare to binned SAH [Wald 2007], referred to as binSAH, to Bonsai (Section 5), and to two versions of Bonsai P (Section 5.4), where Bonsai P is our algorithm with pruning. For binSAH, we use Intel's Embree 2.2 implementation of binned SAH BVH. However, Embree's fastest BVH builder is designed to build 4-wide trees, but we only compare to 2-wide trees, since our GPU traversal is faster for these trees. Although older versions of Embree implement binary tree-builders, those implementations were not as fast, and we found it fairest to modify the faster 4-wide builder to construct 2-wide trees. We also compare to approximate agglomerative clustering (AAC), using the authors' source code [Gu et al. 2013], where we used both the high-quality (HQ) and the low-quality version (LQ), where the latter is faster, but generates lower-quality trees. The provided source code is single threaded, so to generate fair build times, giving AAC the benefit of the doubt, we have divided the single-threaded performance by four, since we have four hardware cores and because the authors claim linear speed-up with number of cores.

Our first contribution in terms of results is to show the results of our implementation of the sweep SAH algorithm (Section 4). While it is difficult to compare against others' implementations of the same algorithm, we simply note that the Hairball often takes at least 15× longer to generate [Karras and Aila 2013; Bittner et al. 2013] than when using our implementation. In fairness, there are differences in CPUs and likely also in the ambition level of optimization for sweep SAH. Since we will release our source code, we believe that this is a small but important contribution; the community will get access to a highly efficient implementation of sweep SAH for BVHs.

All major results are shown in Figure 5 for 14 different scenes, where Bonsai and Bonsai with pruning (Bonsai P in the table) were generated with a maximum mini tree size of 512 triangles and Bonsai P* with a maximum of 4096 triangles. The pruning threshold constants are 0.1 for Bonsai P and 0.01 for Bonsai P*.

For Bonsai, ray tracing performance ranges from 75% up to 95% compared to SweepSAH. With Bonsai P, ray tracing performance is increased to range between 92% to 105%, with an average of 98.5%. The most difficult scene to build for Bonsai P and P* is Dragon, which with P* is ray traced at 93% performance compared to SweepSAH. The other scenes are in the range 97% to 108% for P* and the total average ray tracing performance is 101.5% to that of SweepSAH. Depending on the scene, the increase in build time for pruning can range from very little (Bentley) to nearly double (Hairball). Bonsai build times vary with different mini tree sizes, and without loss of ray tracing performance, improved build times can be made for

| | Arabic City | | Battlefield | | Bentley | | Conference | | Crown | | Dragon | | Fairy Forest | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Triangles | 411,563 | | 64,256 | | 2,346,676 | | 331,177 | | 4,868,924 | | 7,349,988 | | 174,117 | |
| | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace |
| SweepSAH | 67.3 | **100%** | 8.9 | **100%** | 560 | **100%** | 57.8 | **100%** | 1585 | **100%** | 3074 | **100%** | 29.9 | **100%** |
| binSAH | 46 | 91% | 9.5 | 92% | 271 | 89% | 38.7 | 93% | 584 | 90% | 948 | 89% | 24.5 | 94% |
| AAC HQ | 76.8 | 84% | 13.2 | 75% | 502 | 69% | 60.9 | 91% | 1033 | 73% | 1624 | 67% | 34.7 | 76% |
| AAC LQ | 32.3 | 72% | 6.2 | 69% | 209 | 67% | 23.7 | 87% | 415 | 71% | 632 | 66% | 14 | 79% |
| Bonsai | **19** | 81% | **3.3** | 78% | **117** | 90% | **16.3** | 86% | **251** | 90% | **403** | 92% | **8.8** | 87% |
| Bonsai P | 20.2 | 96% | 3.7 | 94% | 119 | 94% | 16.9 | 96% | 278 | 96% | 415 | 92% | 9.5 | 96% |
| Bonsai P* | 23.6 | **100%** | 4.3 | **101%** | 133 | **97%** | 19.7 | **98%** | 315.4 | **97%** | 463 | **93%** | 11 | **97%** |

| | Hairball | | Italian City | | Kalabsha | | Sala | | San Miguel | | Sibenik | | Sponza | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Triangles | 2,850,000 | | 368,322 | | 4,542,545 | | 395,725 | | 7,842,744 | | 79,306 | | 262,266 | |
| | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace | Build | Trace |
| SweepSAH | 590 | **100%** | 57.3 | 100% | 1603 | 100% | 67.9 | 100% | 2386 | **100%** | 10.5 | 100% | 51.4 | 100% |
| binSAH | 275 | 81% | 40.71 | 92% | 571 | 96% | 43.3 | 98% | 947 | 94% | 10.7 | **100%** | 32.3 | 100% |
| AAC HQ | 625 | 49% | 68.3 | 87% | 954 | 82% | 76.6 | 97% | 1649 | 91% | 15.9 | 84% | 50.4 | **105%** |
| AAC LQ | 276 | 50% | 29 | 82% | 367 | 74% | 32 | 93% | 688 | 87% | 6.6 | 84% | 21.5 | **105%** |
| Bonsai | **119** | 95% | **17.3** | 75% | **267** | 81% | **19.3** | 91% | **409** | 78% | **4.3** | 94% | **12.6** | 90% |
| Bonsai P | 156 | **100%** | 18.5 | **101%** | 271 | **102%** | 20 | **104%** | 424 | **106%** | 4.9 | **100%** | 13.4 | **105%** |
| Bonsai P* | 203 | **103%** | 21.5 | **108%** | 298 | **108%** | 23 | **103%** | 478 | **107%** | 5.8 | **108%** | 15.4 | **108%** |

**Figure 5**. Build times are in milliseconds and ray-tracing (path tracing) performance is relative to SweepSAH. Note that the fastest build time is marked with bold text, and so are the two fastest ray-tracing performance numbers per column. The binned SAH algorithm is from Intel's Embree 2.2 and uses a modified version of its fast BVH4 builder to construct binary trees.

some scenes by selecting other mini tree sizes. However, with no a priori knowledge regarding which scene would benefit from which mini tree size, we have found that a size of 512 is a reasonable compromise across the test scenes. Bonsai P and P* build times are also affected by the pruning constants, and we base our choices on empirical observations with regard to both ray-tracing performance and build times. It is actually not pruning itself that adds to the build time, but it is the increased number of leaf nodes for the top tree, caused by pruning, that affects build time. Simply put, more leaf nodes result in more work for the top tree. This effect can be seen in Figure 6, where we show the timings of each step of the Bonsai and the Bonsai P algorithms, relative to Bonsai build times of the San Miguel scene.

We found that scenes with a uniform distribution of finely tessellated triangles with roughly the same size are more sensitive to the pruning algorithm. This is because we use a fraction of the average surface area of mini tree root nodes as a threshold. If there is little variance among mini tree root nodes sizes, then too many mini trees may be larger than the threshold, and they simply get over pruned. However, if a scene has a larger variance in triangle sizes and triangle distribution, then it is

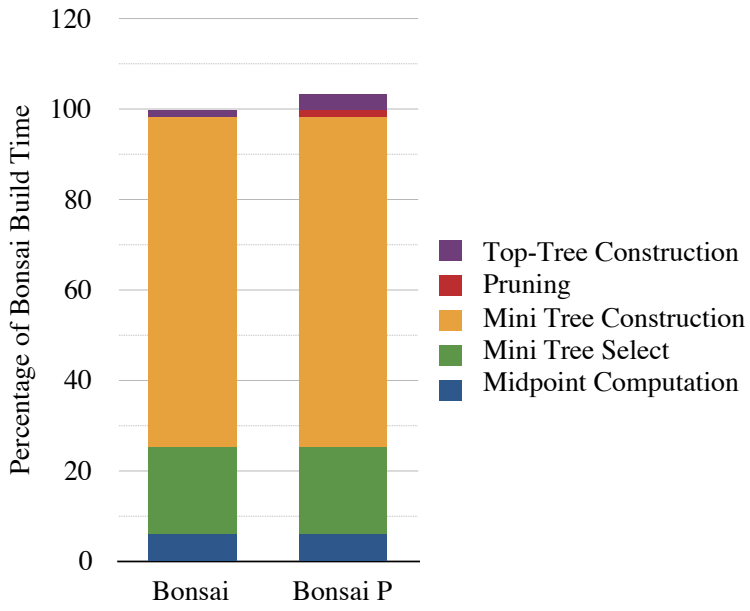


**Figure 6.** While pruning increases ray-tracing performance significantly, it also has an adverse effect on BVH build times. In the bar graph, we show the time of each step of the Bonsai and Bonsai P algorithms relative to Bonsai's full build time. We chose San Miguel as a representative scene. Even though pruning (red) does not take up much more than 1% of the total build time, the top-tree build time (purple) increases from 1.5% of the total build time without pruning to 3.3% with pruning.

| | Build Times | SAH Cost | RT Performance |
|---|---|---|---|
| SweepSAH | 100% | 100% | 100.0% |
| binSAH | 61% | 106% | 92.6% |
| AAC HQ | 101% | 106% | 80.8% |
| AAC LQ | 42% | 108% | 75.2% |
| Bonsai | **25%** | 112% | 85.7% |
| Bonsai P | 27% | 101% | 98.5% |
| Bonsai P* | 32% | 99% | **101.5%** |

**Figure 7.** Average build times, SAH costs, and ray-tracing performance compared to SweepSAH. The algorithms with the best build times, respectively, best ray-tracing performance are highlighted.

likely that just a smaller number of all the mini trees have a surface area larger than the threshold, and these mini trees are exactly the ones that need to be pruned.

Average build times, SAH costs, and ray-tracing times for our 14 test scenes are presented in Figure 7. Build performance for Bonsai compared to SweepSAH is on average 4× faster and compared to binSAH a little more than 2× faster. Bonsai, Bonsai P, and P* all have faster build times than AAC LQ and, in addition, a significantly higher ray-tracing performance. The SweepSAH implementation is on par with the build times of AAC HQ, but ray tracing is often much faster for SweepSAH. Note that we use a full-path tracer, with highly incoherent ray distribution after a few bounces, for all comparisons, and we measure wall clock rendering time, while Gu et al. [2013] used 16 diffuse rays and counted BVH node traversal and intersections tests in their paper. As can be seen, our algorithm is significantly faster than the others at building and, at the same time, it has the best ray-tracing performance.

Although it is very difficult to make comparisons to algorithms executed on completely different hardware, it might be worth noting, that on average, in the cases where we share test scenes, our Bonsai BVH build times are less than 1.4× the build times of the GPU BVH algorithm by Karras and Aila [2013]. Note that the latter was executed on an NVIDIA GTX Titan, which has 10× more FLOPS than the CPU cores that we use. We have similar behavior with Bonsai P, where the shared scenes have an average build time less than 1.6× to that of Karras and Aila.

It is well known that SAH does not correlate perfectly to ray-tracing performance. However, it is generally assumed that better ray-tracing performance can be the result when SAH cost is lowered. Figure 7 shows the average SAH costs for all the tested algorithms and Figure 8 shows the SAH cost details across all scenes. Although it is not possible to deduce any clear conclusions on how SAH cost reflects ray-tracing performance among different algorithms (AAC HQ often has a lower SAH cost than

| | Arabic City | Battlefield | Bentley | Conference | Crown | Dragon | Fairy Forest |
|---|---|---|---|---|---|---|---|
| SweepSAH | 113.5 | 33.9 | 4.2 | 53.6 | 36.6 | 25.5 | 47.8 |
| binSAH | 119.0 | 36.4 | 6.1 | 57.5 | 37.5 | 26.3 | 47.6 |
| AAC HQ | 111.0 | 41.5 | 6.1 | 49.8 | 41.8 | 29.1 | 54.5 |
| AAC LQ | 124.5 | 41.8 | 6.1 | 50.9 | 41.6 | 28.2 | 53.5 |
| Bonsai | 129.1 | 39.4 | 6.0 | 63.9 | 37.5 | 26.0 | 49.9 |
| Bonsai P | 109.3 | 34.1 | 5.9 | 55.9 | 36.8 | 25.9 | 46.6 |
| Bonsai P* | 108.2 | 33.9 | 5.8 | 55.4 | 36.7 | 26.1 | 46.3 |
| | Hairball | Italian City | Kalabsha | Sala | San Miguel | Sibenik | Sponza |
| SweepSAH | 620.5 | 90.7 | 10.4 | 53.9 | 95.6 | 74.9 | 121.3 |
| binSAH | 663.4 | 96.2 | 10.8 | 54.2 | 96.7 | 74.2 | 122.9 |
| AAC HQ | 767.1 | 86.9 | 8.2 | 52.3 | 88.5 | 80.2 | 109.5 |
| AAC LQ | 785.8 | 97.5 | 8.0 | 52.0 | 89.0 | 82.2 | 112.7 |
| Bonsai | 640.5 | 107.6 | 10.3 | 58.3 | 111.0 | 79.3 | 136.5 |
| Bonsai P | 610.7 | 86.2 | 10.1 | 51.2 | 92.6 | 71.6 | 114.2 |
| Bonsai P* | 595.9 | 85.2 | 8.6 | 51.6 | 93.0 | 71.0 | 113.4 |

**Figure 8.** SAH costs, calculated using Equation (1), of all scenes and across all algorithms. Traversal cost constants used are $C_I = 2$, $C_L = 0$, and the triangle-intersection cost constant is $C_T = 1$.

SweepSAH but most of the time also lower ray-tracing performance), there seems to be a clear intra-algorithmic correlation for both Bonsai and AAC, where reduced SAH costs correlate well to improved ray-tracing performance.

## 8. Conclusions and Future Work

Bonsai is a highly efficient and simple-to-implement algorithm for building bounding volume hierarchies. When measuring ray-tracing performance, our algorithm, supplemented with the pruning optimization technique, meets or comes close to, but in many cases surpasses, the tree quality of SweepSAH. We have shown that Bonsai maps well to both thread-level and instruction-level parallelism. An observation is that it may well be that CPU hardware is a better fit to construct high-quality bounding volume hierarchies than GPU hardware. This is based on the comparisons of BVH build times between Bonsai (with and without pruning) on a laptop CPU and Karras and Ailas [2013] GPU BVH algorithm. The latter is executed on an NVIDIA GTX Titan with 10 times more compute capabilities, while Bonsai with pruning only takes 60% longer to execute on a laptop CPU.

### Acknowledgements

Thanks to the whole Advanced Rendering Technology group at Intel, and especially to Charles Lingle for help with internships and to Jacob Munkberg for help in the beginning of this

project. Thanks to Veronica Sundstedt for the Kalabsha temple model and to Timo Aila for the Italian and Arabic City models. Thanks also to Carsten Benthin for sharing information about Embree.

Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. Per Ganestam and Michael Doggett are funded by ELLIIT and the Intel Visual Computing Institute.

## References

AILA, T., KARRAS, T., AND LAINE, S. 2013. On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics*, ACM, New York, 101–107. URL: http://dl.acm.org/citation.cfm?id=2492056. 25

BITTNER, J., HAPALA, M., AND HAVRAN, V. 2013. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum, 32*, 1, 85–100. URL: http://onlinelibrary.wiley.com/doi/10.1111/cgf.12000/abstract. 26, 34

CLARK, J. H. 1976. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM, 19*, 10, 547–554. URL: http://dl.acm.org/citation.cfm?id=360354. 24

ERNST, M., AND GREINER, G. 2007. Early Split Clipping for Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, 73–78. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4342593. 25, 30

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and Faster HLBVH with Work Queues. In *High-Performance Graphics*, ACM, New York, 59–64. URL: http://dl.acm.org/citation.cfm?id=2018333. 25

GARANZHA, K., PREMOZE, S., BELY, A., AND GALAKTIONOV, V. 2011. Grid-based SAH BVH construction on a GPU. *The Visual Computer, 27*, 6-8, 697–706. URL: http://link.springer.com/article/10.1007%2Fs00371-011-0593-8. 25

GOLDSMITH, J., AND SALMON, J. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications, 7*, 5, 14–20. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4057175. 24, 26

GU, Y., HE, Y., FATAHALIAN, K., AND BLELLOCH, G. 2013. Efficient BVH Construction via Approximate Agglomerative Clustering. In *High-Performance Graphics*, ACM, NY, 81–88. URL: http://dl.acm.org/citation.cfm?doid=2492045.2492054. 24, 34, 37

HAVRAN, V., AND BITTNER, J. 2002. On Improving KD-Trees for Ray Shooting. In *Winter School on Computer Graphics*, 209–217. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.6503. 25

KAJIYA, J. T. 1986. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, vol. 20, 143–150. URL: http://dl.acm.org/citation.cfm?doid=15886.15902. 24

KARRAS, T., AND AILA, T. 2013. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *High-Performance Graphics Conference*, ACM, New York, 89–99. URL: http://dl.acm.org/citation.cfm?doid=2492045.2492055. 26, 30, 34, 37, 38

KARRAS, T. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 33–37. URL: http://diglib.eg.org/handle/10.2312/EGGH.HPG12.033-037. 25, 30

KAY, T. L., AND KAJIYA, J. T. 1986. Ray Tracing Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, vol. 20, 269–278. URL: http://dl.acm.org/citation.cfm?doid=15886.15916. 24

KENSLER, A. 2008. Tree Rotations for Improving Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, 73–76. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4634624. 26

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum, 28*, 2, 375–384. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01377.x/abstract. 25, 30

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer, 6*, 3, 153–166. URL: http://link.springer.com/article/10.1007%2FBF01911006. 24, 26, 30

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 87–95. URL: http://dl.acm.org/citation.cfm?id=1921493. 25, 30

PANTALEONI, J., FASCIONE, L., HALL, M., AND AILA, T. 2010. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Transaction on Graphics, 29*, 3, 37.1–37.10. URL: http://dl.acm.org/citation.cfm?doid=1778765.1778774. 25

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*, 2nd ed. MKP, San Francisco, CA. URL: http://www.pbrt.org/. 24

POPOV, S., GEORGIEV, I., DIMOV, R., AND SLUSALLEK, P. 2009. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *High-Performance Graphics*, ACM, New York, 15–22. URL: http://dl.acm.org/citation.cfm?doid=1572769.1572772. 25

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 80)*, vol. 14, 110–116. URL: http://dl.acm.org/citation.cfm?doid=800250.807479. 24

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics*, ACM, New York, 7–13. URL: http://dl.acm.org/citation.cfm?doid=1572769.1572771. 25, 30

WALD, I., AND HAVRAN, V. 2006. On Building Fast Kd-trees for Ray Tracing, and on Doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, 61–69. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4061547. 27

WALD, I. 2007. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, 33–40. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4342588. 24, 25, 27, 30, 34

WALD, I. 2012. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics, 18*, 1, 47–57. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5669303. 25

WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. 2008. Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, 81–86. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4634626. 24

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Communications of the ACM, 23*, 6, 343–349. URL: http://dl.acm.org/citation.cfm?doid=358876.358882. 24

WU, Z., ZHAO, F., AND LIU, X. 2011. SAH KD-tree Construction on GPU. In *High-Performance Graphics*, ACM, New York, 71–78. URL: http://dl.acm.org/citation.cfm?doid=2018323.2018335. 27

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-Time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics, 27*, 5, 126:1–126:11. URL: http://dl.acm.org/citation.cfm?doid=1409060.1409079. 27

## Author Contact Information

Per Ganestam
per.ganestam@cs.lth.se

Rasmus Barringer
rasmus.barringer@cs.lth.se

Michael Doggett
michael.doggett@cs.lth.se

Tomas Akenine-Möller
tomas.akenine-moller@cs.lth.se

P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller, Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 3, 23–42, 2015
http://jcgt.org/published/0004/03/01/

| | | | |
|---|---|---|---|
| Received: | 2015-01-26 | | |
| Recommended: | 2015-07-15 | Corresponding Editor: | Wenzel Jakob |
| Published: | 2015-09-22 | Editor-in-Chief: | Marc Olano |