On fast Construction of SAH-based Bounding Volume Hierarchies

Ingo Wald[12]

[1]SCI Institute, University of Utah [2]Currently at Intel Corp, Santa Clara, CA

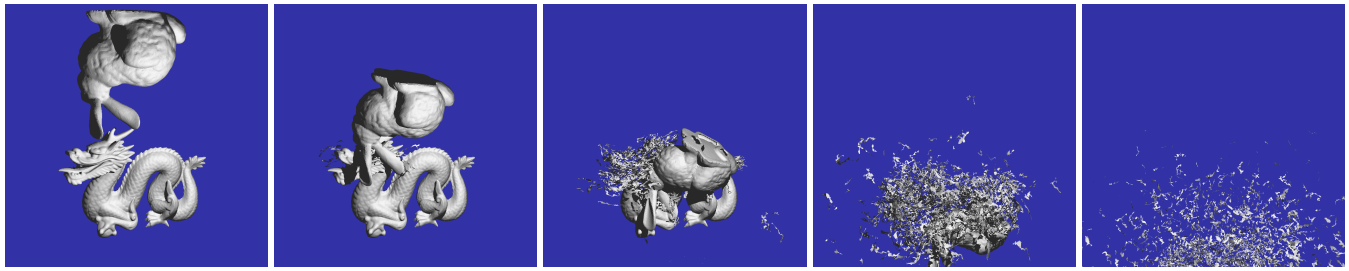

Figure 1: We present a method that enables fast, per-frame and from-scratch re-builds of a bounding volume hierarchy, thus completely removing a BVH-based ray tracer's reliance on updating or re-fitting. On a dual-2.6GHz Clovertown system (8 cores total), our method renders the exploding dragon model at around 5–10 frames per second (1024x1024 pixels) including animating the triangles, per-frame rebuilds, shading, shadows, and display. The build itself takes less than 50ms, and is nearly agnostic to the distribution of the triangles; thus, the variation in frame rate (10fps for the initial, smooth frames 5 fps for the disintegrated ones) is due only to varying traversal cost, without any deterioration in BVH quality at all (i.e., when starting with the last frame, the frame rate actually increases).

ABSTRACT

With ray *traversal* performance reaching the point where real-time ray tracing becomes practical, ray tracing research is now shifting away from faster traversal, and towards the question what has to be done to use it in truly interactive applications such as games. Such applications are problematic, because when geometry changes every frame, the ray tracer's internal index data structures are no longer valid. Fully rebuilding all data structures every frame is the most general approach to handling changing geometry, but was long considered impractical except for grid-based grid based ray tracers, trivial scenes, or reduced quality of the index structure. In this paper, we investigate how some of the fast, approximate construction techniques that have recently been proposed for kd-trees could also be applied to bounding volume hierarchies (BVHs). We argue that these work even better for BVHs than they do for kd-trees, and demonstrate that using those techniques, BVHs can be rebuilt up to 10× faster than competing kd-tree based techniques.

1 INTRODUCTION

While ray tracing has always been the method of choice for most offline rendering systems, it was long considered impractical for interactive applications, like games. With the advent of ever more capable hardware to work on—a dual-2.6GHz Clovertown desktop PC already reaches 83 GFLOPs[1] peak—and coupled with ever more refined algorithms that use this performance [13, 22], ray tracing is now approaching rates of many million rays per second; slowly reaching the stage where this technology gets interesting for truly interactive applications, like games.

Such applications, however, pose a different sort of problem in that they typically are highly dynamic in nature, which requires that the ray tracer either rebuild or update its internal data structures every frame. This now requires to look at *both* traversal performance and rebuild/update performance at the same time. Though we refer the reader to a recent survey on the topic for a more complete discussion [21], the generally accepted opinion today seems to be that the techniques that are easiest to rebuild (i.e., regular grids) are generally less efficient to traverse, and more efficient techniques (like kd-trees or BVHs) are harder to built–with the most efficient techniques requiring many seconds even for moderately complex scenes [20].

For bounding volume hierarchies, this has led to various efforts to avoid full rebuilding, and rather investigate BVH refitting [9,19,24]–potentially coupled with infrequent or asynchronous rebuilding ([9] and [6], respectively)–or to selectively restructure an existing BVH. Fast re-building from scratch has received far less attention, with one notable exception by Wächter et al [17], who's approach was originally proposed for a data structure that is most similar to 2-plane s-kd-trees [11], but which applies to traditional BVHs, too. Wächter's BIH-style build–once applied to BVHs–allows for fast building of BVHs, but leads to somewhat reduced traversal performance (also see below), because it no longer employs a surface area heuristics (SAH) for determining how to best build the hierarchy, and instead always splits at the spatial median.

For kd-trees, the set of investigated techniques is much richer, including hierarchical ray transformation schemes [18], "fuzzy" kd-trees and motion decomposition [3], and, in particular, fast "scan"-techniques that still employ a (somewhat simplified) surface area heuristic, but achieve significantly high build performance by slightly approximating a full SAH build.

For BVH, such fast from-scratch build techniques have not yet been investigated. This is surprising, since BVHs have some nice properties that make them, in fact, more amenable to these techniques than kd-trees: First, BVHs have fewer nodes than a kd-tree, so fewer operations have to be performed, anyway; second, all BVH build algorithms to date use only the triangle centroids, anyway, so there is no need for handling cases where triangles "overlap" a split plane (in a BVH, a triangle will always be on exactly one side); third, since each triangle is referenced exactly once, the total number of nodes in a BVH is bounded by $2N-1$ (where N is the number of triangles), so the build can be performed fully "in place" without any additional management of node memory.

In addition, we can expect the negative speed impact of a binned, approximate build scheme to be lower for a BVH than it is for a kd-tree: First, since we have to track each bin's exact bounds, anyway (see below), the area estimation is not approximated to the same degree; second, since BVHs are a object hierarchy–not a spatial subdivision–they never "subdivide" final than a triangle's bounding box, anyway, and thus do not suffer from not considering "perfect splits" during the construction.

[1]2 CPUs× 4 cores × 4-wide SIMD/cycle× 2.6GHz=83.2GFLOPs

In summary, we argue that the fast, approximate binning techniques recently developed for kd-trees are also applicable to fast construction of BVH from scratch; and that we can expect at least the same benefits from these techniques, while at the same time suffering less from their disadvantages. In the rest of this paper, we will first briefly review relevant related work in Section 2. We then describe in detail how the algorithm works in a single-threaded environment in Section 3, followed by a discussion on how to parallelize the algorithm on a multi-core architecture (Section 4. We then compare our technique to existing techniques based on building kd-trees and restructuring BVHs in Section 5, and conclude in Section 5.3.

2 Related Work

Research on ray tracing goes back to the 80'ies, in particular to the seminal work by Appel [1], Whitted [23], and Cook et al. [2]. For a more complete picture on ray tracing data structure trade-offs, and, in particular, fast ray tracing for animated scenes, we refer the reader to a recent survey on this topic [21]; in this section, we will cover only those techniques that are directly relevant to our technique.

BVHs for Animated Scenes have been proposed independently by both Wald et al. [19] and Lauterbach et al. [9]. Both of these systems relied on a BVH's ability to handle *deforming* models (i.e., those whose triangle count and mesh connectivity do not change over time) by simply re-fitting a BVH's technique. That BVHs are amenable to refitting is well known, and has previously been used extensively in both rendering [8] and collision detection [10,15,16].

Since refitting only will eventually lead to performance degradation if the primitives' motion becomes too severe, several techniques have been proposed that augment refitting with either periodic from-scratch rebuilding [9], asynchronous rebuilding [6], or selectively restructuring an existing BVH every frame [25]. Eventually, however,all of these techniques still rely on having a deformable scene with not-too-severe deformations; for general, unrestricted support for dynamic scenes (including varying triangle counts and/or completely random motion), the ultimate solution would be to fully rebuild from scratch every frame.

Surface Area Heuristic. Though there are many different ways of building BVHs and kd-trees, the best known method to maximize traversal (for both data structures) is to use a top-down, greedy surface area heuristics (SAH) build, in which the original scene is recursively partitioned using a greedy strategy for minimizing expected traversal cost.

Though kd-trees and BVHs have different concepts of what a "partition" is—a kd-tree recursively partitions space into two non-overlapping halves, while a BVH recursively partitions the set of primitives—the theory behind a surface area heuristic is essentially the same: Given a set of N primitives contained in a sub-tree that covers the 3D volume V, and assuming that sub-tree gets partitioned into two halves L and R with number of triangles N_L and N_R, and with associated volumes V_L and V_R, the traversal cost for this sub-tree can be estimated as

$$Cost(V \rightarrow \{L,R\}) = K_T + K_I(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R),$$

where $SA(()V)$ is the surface area of V, and K_T and K_I are some implementation-specific constants for the estimated cost of a traversal step and a triangle intersection, respectively.

Using this cost estimate, a greedy SAH build evaluated various possible partitions, selects the one with lowest cost, and recurses. For a kd-tree node, a partition is defined by an axis-aligned plane that partitions the sub-tree's volume; given the plane, the numbers N_L and N_R are defined implicitly for each plane (which requires counting them). Since triangles can overlap the plane, N_L+N_R usually is greater than N. As it can be shown that there are only $6N$ reasonable split locations [4], testing all reasonable splits is tractable, and efficient algorithms for this exist [20].

In a BVH, conversely, a "split" partitions the set of triangles into two subsets (with $N_L+N_R=N$), and the volumes V_L and V_R follows from that partitioning. As there are 2^N-2 possible partitions, testing all is intractable, existing SAH builds look at partitioning the triangles using planes like in a kd-tree; since each triangle can be in only one of the sides even if it overlaps that "split" plane, these methods typically take a single point on the primitive—such as it's bounding box's centroid—to determine on which side it is put. Note that even though that plane defines which side a triangle is on, it has no direct relationship the the actual sub-tree volumes at all, and these can or can not overlap the split plane.

Fast BVH Building. A reasonably fast, $O(N \log N)$ build scheme (which still is today's baseline in building *good* BVHs) has been proposed in [19]; however, though significantly faster in building than comparable high-quality kd-tree builders [20], build times are clearly non-interactive.

Trading BVH quality for significantly faster builds has first been proposed by Wächter et al [17]: using a spatial median split strategy instead of an SAH-based build. Consequently, his builds are much faster, but result in BVHs that are somewhat slower during rendering (see below for actual numbers).

One particularly interesting feature of Wächter's BIH build (that we will employ to some extent below) is that like in a spatial median-split kd-tree, his build strategy does not place the node into through the median of the current sub-tree's actual spatial extent; instead, he always subdivides like in a spatial subdivision: If the root node N,V is split by its center plane P, then the split for the children is not based on the actual sub-tree bounds V_L and V_R, but on the two voxels obtained by splitting V with P—the actual bounds for each BVH node are obtained only at the very end, after both sub-trees have been split. Though it is not fully clear yet why, this is not only faster, it also seems to produce slightly better trees than those built by splitting the actual sub-tree bounds. With this build strategy, the split locations for a node's children no longer depend on the outcome of the node's split itself, and thus, one can perform several splits a-priory—essentially resulting in a regular partitioning—and then "bin" each primitive into it's respective bin in a single pass, eventually saving most of the recursive partitioning steps.

This build strategy is extremely fast, but comes at the cost of no longer using a surface area heuristic—i.e., at somewhat lower performance during rendering.

Fast SAH Building for kd-trees.

Even with theoretically optimal build algorithms, the cost for evaluating lots of split planes can be very high. As it was long considered to be infeasible in real time, anyway, fast building of SAH-based data structures has not been fully investigated, yet. For kd-trees, several researchers have recently looked into this problem, and have proposed methods that strike a trade-off between build time and BVH quality, by simplifying and approximating the SAH, but keeping its principle ideas (see, e.g., the papers by Popov et al. [12] and Hunt et al. [5], and, more recently, by Shevtsov et al. [14]).

In particular, these methods use the following concepts, most of which we will use similarly: First, they ignore "perfect splits"—i.e., the process of clipping triangles to the current voxel that has to be split, in order to get tighter bounds—and only consider the triangles' axis-aligned bounding boxes (AABBs). Second, they do not test *all* potential split planes, but use the concept of "binning", in which a given number K of (usually equidistantly spaced) planes is placed into the voxel; the triangles are then, in a single linear pass, projected into the $K+1$ "bins" formed by these K planes. Thus, each bin counts the number of triangles that overlap it, and the SAH can be evaluated for the K planes that separate the bins by just knowing the triangle counts of the bins. After computing

the SAH for each of these K planes, the best one is selected, and a second linear pass over all triangles generates the actual list of triangle IDs for the left and right sub-trees. Using that algorithm not only greatly reduces the number of plane evaluations (K bin planes instead of $O(N)$ triangle bounds planes), it also avoids any sorting, and thus enables the splitting to be done in two fast $O(N)$ passes.

Though the basic concept remains the same for all of the three papers mentioned above, several further optimizations and simplifications are possible: First, the cost function itself is simplified: K_T and $\frac{K_I}{SA(V)}$ are common terms for all planes, and can be removed; and $SA(V_L)$ and $SA(V_R)$ are linear in the plane position $x(i)$, resulting in a simple $N_L * (c_{0,L} + c_{1,L}x) + N_R * (c_{0,R} + c_{1,R}x)$ (with precomputed constants $C_{0,L}, \dots$) Second, the search for a split plane can be performed along the one axis where the bounding box is widest, instead of considering all three axis in turn. Third, instead of directly increasing all bin counters overlapped by a triangle, it is possible to only mark the start and end bins for each triangle, and compute all interior bins in a final pass over the binds. Fourth, instead of projecting *all* triangles into the bins to determine the plane one can also use a sub-set of triangles and "skip" other ones. And fifth, one the number of triangles drops below the number of bin on usually reverts to a full sweep.

Binning essentially is a discretization of the SAH cost function, and can obviously lead to missing the minimum; however, all three publications have shown that surprisingly good kd-trees can be achieved with relatively few bins (even with only 8 planes in Hunt's case!). Similarly, skipping seems to work quite well, if at least a reasonably sized sub-set of triangles is used for plane selection. As such, the biggest individual factor why binning-based kd-trees are slower than the most advanced ones seems to be the negligence of perfect splits.

In addition to these algorithmic changes, careful implementation is required to achieve high performance. For example, the use of SIMD instructions, careful data layout, and very careful memory allocation schemes (typically using pre-allocated memory [14]) are required. Using such carefully designed implementations on top of various binning techniques, Popov et al. and Hunt et al. have reported build rates of 150k resp. 300k triangles per second, which allow roughly interactive builds at least for simple models.

Since modern CPUs are increasingly multi-core, parallelization can give an additional boost to build times. Parallelization was already demonstrated by Popov et al. [12], albeit they reported unsatisfactory results for their results. The first scalable results have been reported by Shevtsov et al. [14], who use two separate passes to improve scalability: In a first pass–that all threads share work in—they split the input set of triangles into sub-trees of equal size, at which stage they switch to the thread working on different subtrees in a load-balanced way. This two-stage process provides good scalability (at least for the 4 CPUs they used their experiments), but comes at the cost of not using a SAH strategy in the upper level of the kd-tree. We will follow a similar technique once we parallelize our approach.

3 Fast, binned BVH building

As can be learned from the previous section, the use of binning, sub-sampling, approximations, and parallelization have been proven very effective in accelerating the build process of SAH-based kd-trees. For BVHs, similarly fast build algorithms have not been investigated, yet.

This in fact is surprising, as all of the proposed techniques would work similarly, while some of the problems in applying these techniques to kd-trees do not apply to BVHs. In particular, a BVH with axis-aligned bounding volumes—which is what we are interested in in this paper—will *always* use a triangle's AABB; as no triangle ever gets split by a plane, there is no concept of considering "perfect splits". Since not considering perfect splits is one of the biggest sources of lower performance for binned kd-trees, not suffering from this problem bears the potential to use binning techniques without significant reductions in build quality.

In addition to that, most BVH build strategies use a single point to decide which side a primitive goes (see above), binning gets simple, and a single bin has to be updated for each primitive. We follow previous approaches in using the centroid of each primitive's AABB for this decision.

3.1 Bin setup

Like for kd-trees, we generate K bins of equal width. Since primitives are binned with respect to their AABBs' centroids, anyway, we place our bins such that they uniformly subdivide the bounding box around the centroids; resulting in a somewhat denser spacing of the bins.than if those would cover the full sub-tree bounds. Each of these K equally-sized spatial regions we call the *domain* of its associated bin.

Given K bins $B_1..B_K$, there are $K-1$ canonical ways of splitting the input set of triangles T into two halves $T_{L,j} = \{B_1..B_j\}$ and $T_{R,j} = \{B_{j+1}..B_K$ (for $j = 1..K-1$). Knowing the number of primitive n_i for each bin B_i, these $K-1$ possible partitions then have $N_{L,j}$ and $N_{R,j}$ triangles on the left and right side, respectively, with $N_{L,j} = \sum_{i=1}^{j} n_i$ and $N_{R,j} = \sum_{i=j+1}^{K} n_i$. Following previous work in neglecting all constants and common terms in the cost function, the cost for any partition j then is

$$Cost_j = A_{L,j}N_{L,j} + A_{R,j}N_{R,j}.$$

While we know the N terms, the A terms require special attention. Binning each triangle only in the bin associated with its centroid, triangles will, in general, "stick out" of the respective bin's domain, sometimes considerably so. Ignoring that effect by assuming each bin's triangles would cover exactly this bin's domain—as is done when assuming there is a linear relationship between bin position and surface area—would be a grave approximation. Thus, we have each bin track not only the number n_i of primitives that project to it, but also the bounding box bb_i (read: bin-bounds) of all those primitives. We call this bb_i the bounding box of bin i, even though it has no spatial relation to that bin's domain at all—it can be a subset, or a super-set, stick out of it, etc.

Since for AABBs the union of some bins' bounding boxes is exactly the same as the bounding box of those bins' triangles, defining $A_{L,j} = SA(\bigcup_{i=1}^{j} bb_i)$ and $A_{R,j} = SA(\bigcup_{i=j+1}^{K} bb_i)$ gives us the correct bounds of each partition's halves, without any approximation or discretization at all. Note that we track the actual $3D$ bounds for each bin, not only their $1D$ projections to the binning axis. In particular, if one or both of the halves for any given partition has a small spatial extent in the two axis perpendicular to the binning axis, this will correctly be determined, resulting in a lower cost estimate for this partition.

3.2 Initial setup

The way just described, our algorithm needs each triangle's bounding box tb_i, as well as the centroid thereof, c_i. We compute these once in the beginning of our build, and can then completely ignore the actual triangle geometry, which is no longer needed. In that setup stage, we also compute the bounds for all triangles *vb* (read: voxel bounds), as well as the bounds for all centroids, *cb*. To facilitate SSE operations throughout the code, we store the voxel bounds and centroid bounds, as well as all triangle centroids and triangle bounds, in a SIMD-friendly format of four 16-bytes aligned floats per 3D position. Thus, the setup stage requires exactly 14 SSE operations per triangle, including load/stores: three loads for the triangle's vertices, two SSE mins and maxes each for computing the triangle's bounds from those vertices; one min and max each

to grow the voxel bounds; one add and one "mul 0.5f" to compute the centroid; another min and max each for growing the centroid bounds; and finally, three SSE stores to write the triangle's bounds and centroid to memory.

3.3 Triangle-to-bin projection

The binning itself is also very efficient. Given a triangle's centroid c_i, the bin number for that triangle is

$$binID_i = \frac{K(1-\varepsilon)(c_{i,k} - cb_{min,k})}{cb_{max,k} - cb_{min,k}},$$

where cj is the centroid bounds, k is the binning axis, and K is the number of bins. The "$1-\varepsilon$" is to ensure that centroids on exactly the right bounds of cb still project to $K-1$, so no special handling is required for this case; though this also slightly shifts all other bin domains, this is not a numerical issue, as we never actually consider the bin domains, anyway.

Since all values except c_i are constant, we can pre-compute these terms, and compute the triangle's bin as

$$binID_i = k_1(c_{i,k} - k_0),$$

with $k_1 = \frac{K(1-\varepsilon)}{cb_{max,k} - cb_{min,k}}$, and $k_0 = cb_{min,k}$, followed by a float-to-int truncation (non-SSE).

3.4 Bin updates

For each bin, we track the number of primitives n_i as the bin bounds bb_i. Again, we store the bin bounds in SSE format, facilitating SSE operations. Each bin is initialized to a "negative box" $[+\infty .. -\infty]$, allowing to grow it to include a triangle bounds using one SSE and one SSE max each without having to check whether the bin is originally empty. Similarly we can later on merge the individual bin bounds without having to check for emptiness, since "growing" an AABB with a negative AABB using min/max operations does not change the original box. Of course, we can use the same trick of using only a subset of triangles for populating the bins that kd-tree builders use. This can give a slight performance improvement but introduces an additional parameter that has to be tuned to avoid BVH deterioration; we therefore by default do not use this feature even though it is implemented.

3.5 Plane evaluations

To evaluate the individual partitions' cost as described above, we have to determine $N_{L,i}$, $N_{R,i}$, $A_{L,i}$, and $A_{R,i}$. We compute these by first doing a linear pass from the left, in which we incrementally accumulate the bounds and number of triangles for the left half (exploiting the fact that $N_{L,i} = N_{L,i-1} + n_i$ etc), and storing $N_{L,i}$ and $A_{L,i}$. Then, in a second pass we do the same from the right, and evaluate the SAH for each plane. Because bins are initialized with an empty box, if one of the sides does not contain any triangles, its area gets negative; as such "empty" partitions are not allowed in a BVH, anyway—and thus, have to be rejected, anyway—this does not introduce a problem.

3.6 In-place ID list partitioning

The BVH itself is stored in two separate, continuous arrays—one for the nodes, and one for triangle ID. Since a BVH for N triangles has exactly N triangle references and at most $2N-1$ nodes, these can be pre-allocated, and no memory has to be allocated during the build process at all. Each inner node contains the bounding box, several internal flags, and the position of its child node pair's position inside the node array. Leaf nodes contain an offset into the triangle ID array, plus the number of triangles in that leaf. Thus, all of a leaf's triangles are stored in one continuous block.

With that data organization, building the BVH happens completely "in place" by re-arranging the triangle ID array—which gets initialized to—into sequences of IDs that belong to the same sub-tree: each partitioning operation operates on a sequence of IDs in that sub-tree (which we bounds in *STL*-style by a *begin* and *end* offset into the ID array), then determines the partition with minimum expected cost as described above, and then re-orders the sequence of triangle IDs in the $[begin..end)$ range such that two new sequences $[begin..mid)$ and $[mid..end)$ with the triangle IDs for the left and right sub-tree are created.

Re-arranging the IDs essentially works exactly like a *quicksort* partitioning step (with the minimum-cost partition ID working as a pivot): one iterator sweeps the triangle ID list from the left (starting with 'begin'), computes the bin ID for each triangle ID it encounters, and stops with the first triangle ID that belongs to the right. Another iterator approaches from the right until one triangle ID that belongs to the left is found; these IDs are then swapped, and the process is repeated until the two iterators meet, at "mid". Alternatively, one can also allocate a second array as temporary storage, then block-copy all IDs in $[begin..end)$ to that array, and copy them back individually to the left resp right sides of the original array.

During the ID-list partitioning, we also track the triangle bounds and centroid bounds for the left and right halves; one all this is done, BVH construction then proceeds recursively for each of these sequences, using the respective triangle bounds and centroid bounds that have just been computed.

3.7 Number of bins

So far, we have completely ignored the number of bins to be used. Previous work on kd-trees has proposed up to 1024 bins [12], but others have reported quite satisfactory results with as few as 8 bins. As argued before our binning does not introduce any approximations to the area term or any other term at all; and the only negative outcome of binning is that not all potential partitions are considered. Thus, our binning is arguable less inexact than kd-trees, so we can expect reasonably good quality even with few bins. This turns out to be true, and even as few as 4 bins produce quite reasonable results. Using more bins can produce somewhat better quality depending on the scene, but 16 bins seem to be very close to the optimum, and adding more planes hardly improves quality, if at all.

To adapt the bin size to the number of triangles, we also tested a scheme where we allocate the number of bins dynamically to $K = 4 + \sqrt{end - begin}$, but no appreciable difference over 16 bins is found, except that fewer bins are used at the leaf level, which may be faster. With only slightly more than 4 leaves at the leaf level, there is also no need to branch to a accurate sweep version when a certain threshold is reached—as usually done for kd-trees—so we use binning throughout the build process.

3.8 Termination

As usual for SAH-based builds, we terminate the recursion until either a certain threshold of triangles is reached (in our case, usually 2 or 4), until the centroid bounds becomes too small (in which case binning it would not make sense any more), or until the estimated cost is higher than the estimated cost for making a leaf. These are exactly the conditions used for non-binned builds, so nothing special has to be done.

4 Parallelization

In the way just described, the algorithm is already quite competitive with building a kd-tree, as we will outline in the results section that follows this. However, modern CPUs are increasingly parallel, so parallelizing the build is highly desirable. One often used way of doing this is to have different threads work on different sub-trees; since sub-trees are independent of each other (and no memory allocation or global is required at all), this can be done with a minimum

of inter-thread synchronization. In fact, even access to the node array can be done without synchronization at all: since we know that a sub-tree for a given $[begin..end)$ list of triangle IDs has exactly $end-begin$ triangles and requires at most $2(end-begin)$ nodes in the node array, we can pre-allocate a subset of nodes for each sub-tree, allowing each thread to allocate nodes without having to synchronize with other threads at all.

Of course, the Achilles' heel for this approach is that one has to have enough independent sub-trees to work on, which is not the case until a few partitions have been done. In particular, only one sub-tree is available for the BVH's root level, only two at the second level, etc, resulting in bad scalability. To avoid that scalability bottleneck, we have implemented two different strategies that should, in theory, load-balance in every stage.

4.1 Mixed horizontal/vertical work sharing

Instead of a sharing the work "vertically" by having each thread work on a different sub-tree of the BVH, one could also have multiple threads work on the *same* binning step, e.g., by working on separate parts of the triangle array. We call this the "horizontal" way of parallelization.

Setup. Before any actual partitions are performed, we first perform the setup phase in parallel. Each of the T thread works on one T'th of the triangles (with thread t working on triangles $[\frac{t*N}{T}..\frac{(t+1)*N}{T})$), and computes each triangles bounding box and centroid as outlined above. To avoid write conflicts to the global centroid bounds and triangle bounds, each thread tracks its own global bounds, which are then merged by thread 0 as soon as all threads have completed their setup phase.

Horizontal splitting. For the splitting itself, we split the [being..end) interval into T equally sized blocks (where T is the number of threads), and assign one block to each thread. Load balancing on fixed-sized blocks is possible, too, but requires additional synchronization, and is unlikely to yield any improvements, so we stick with this static work assignment. Since triangles from multiple threads might project to the same bins, we have each thread have its own set of bins, and merge those once all threads have finished binning. Each thread t also determines its own $N_{L,i,t}$ and $N_{R,i,t}$ sums, which are needed in the next stage.

Thread 0 then does the two sweeps over the bins (which is fast enough not to pose a scalability bottleneck), and determines the best partition. It also computes a prefix sum over all the threads' individual $N_{L,i}/N_{R,i}$ lists, resulting in $N^{(t)}_{L,i}=\sum_{i=0}^{t} N_{L,i,t}$ and $N^{(t)}_{R,i}=\sum_{i=0}^{t} N_{R,i,t}$.

Once these values are determined, all threads then perform the second sweep over their blocks of triangle IDs, determine which side the respective triangle ID belongs, and write it into the original triangle ID list. The triangle ID list offset of the left resp right ID sequence for any given thread is determined by above mentioned $N^{t}_{L,i}$ and $N^{t}_{L,i}$ prefix sums, so all binning and writing of IDs can again be performed in parallel, without any synchronization at all.

Upon finishing this horizontal list partitioning stage, thread 0 creates the respective leaf node, and starts the recursion. Eventually, the entire implementation is a sequence of code blocks that are separated by barrier's, no other synchronization primitives are required. Some of these code blocks are executed only by thread 0 (such as merging the individual threads' bounding boxes), but these are short and fast; all of the time-consuming code blocks are executed in parallel, which in theory should provide good load balancing.

Switching from horizontal to vertical. Horizontal parallelization works only high up in the tree, where the number of triangles to be binned is far higher than the number of threads. However, one can use horizontal parallelization for those splits that have many triangles, and eventually switch to vertical parallelization once the sub-trees get small. To do this, we start with horizontal parallelization, and have it proceed until a given threshold of triangles is reached; all such sub-trees are recorded in an array (with their respective begin/end interval, triangle bounds, and centroid bounds). Once all horizontal splits are performed, we switch to vertical parallelization, and proceed for each sub-tree as outlined in the previous section.

Since we cannot determine the number of sub-trees in advance, we perform dynamic load balancing on these, using an atomic counter that specifies the next sub-tree to be built. Even though we load balance dynamically, the first implementation scaled imperfectly in that stage, since the sub-trees have different sizes, and if the last sub-tree by chance is large, all other threads will run idle. We solve this by sorting the sub-trees by descending size before starting this process, which essentially resolves this problem.

In theory, the algorithm outlined above should scale near perfectly until specific hardware limitations—such as available memory bandwidth—are reached. However, on the specific hardware we used (see below) the resulting scalability was much poorer than expected, even though the machine showed near-perfect utilization ("top" showing 780+% utilization on a 8-core architecture). Though this indicates that we did indeed hit the memory bandwidth wall, we also investigated a different build alternative that is similar in spirit to parallel grid building [7], and BIH-style BVH building.

4.2 Grid-based binning

In the algorithm described in the previous section, the initial setup phase and the final vertically parallelized sub-tree phases contain almost no synchronization at all. Thus, any potential source of non-scalable synchronization would be in the horizontal splitting phase. To avoid that stage, we can also follow the same observation made by Wächter et al. [17]: when repeatedly subdividing along the spatial median, we can perform several splits s a priori, resulting in a regular grid of $2^s \times 2^s \times 2^s$ voxels. We can then bin all triangles into these 2^{3s} bins, build the resulting 2^{3s} sub-trees in parallel, and simply re-fit the boxes for the top s levels of the BVH to fit the respective sub-trees. The grid resolution can be chosen manually, or such as to reach a certain number of sub-trees per thread; we currently chose around 2–5 sub-trees per thread.

Since that binning is a single sweep over all triangles, it can be parallelized very efficiently. We again have each thread working on one T'th of the triangles, and give each thread its own grid of 2^{3s} bins to avoid write conflicts. During the binning, each thread directly writes its triangle IDs into a dynamically sized array for each bin. Once all threads have binned their triangles, thread 0 merges the bins, and determines each sub-tree's offset into the triangle ID array as outlined above, after which the threads copy their respective grid cells' ID lists into the corresponding position in the triangle IDs array.

Again, all stages of the algorithm are parallelized, except a few merge stages that only operate on a few, rather small, voxel grids. Overall, this algorithm executes far fewer barrier operations than the hybrid horizontal/vertical algorithm above, mostly because it operates in a single pass, while the algorithm outlined above is applied recursively. Like the linear BIH build, the algorithm also avoid most of the top-level split operations, saving a significant amount of operations. On the downside, however, it suffers from the same issues as the linear BIH build: no SAH is used for the top splits at all, potentially resulting in lower rendering performance. In addition, teapot-in-a-stadium scenes may break the vertical sub-tree building stage at the end, since such scenes could project most of their triangles into the same bin. This does, actually, happen in practice, requiring some manual parameter tuning to select the right number of bins.

model	#tris	sweep	BIH	binned	grid	grid	grid	grid	hybrid	hybrid	hybrid	hybrid
	#threads→	1	1	1	1	2	4	8	1	2	4	8
fairy	174K	892 ms	36 ms	85 ms	77 ms	37 ms	25 ms	27 ms	80 ms	49 ms	34 ms	44 ms
		100 %	80 %	96 %	95 %	93 %	93 %	93 %	98 %	98 %	98 %	98 %
conference	282K	1.4 s	66 ms	145 ms	134 ms	69 ms	38 ms	32 ms	139 ms	78 ms	53 ms	62 ms
		100 %	66 %	90 %	67 %	84 %	84 %	84 %	90 %	90 %	90 %	90 %
blade	1.5M	10.6 s	415 ms	1.1 s	906 ms	415 ms	245 ms	198 ms	956 ms	530 ms	352 ms	311 ms
		100 %	88 %	101 %	104 %	103 %	103 %	103 %	105 %	105 %	105 %	105 %
thai	10M	84 s	3.3 s	7.5 s	5.9 s	3.1 s	1.7 s	1.4 s	6.4 s	3.4 s	2.3 s	2.1 s
		100 %	76 %	103 %	95 %	84 %	84 %	84 %	88 %	103 %	103 %	99 %

Table 1: Absolute build times (top row for each scene), and relative traversal performance (bottom row, excluding build time, relative to the sweep build) for the various algorithms applied to four different scenes. The individual results are discussed in the text.

5 RESULTS

So far, we have intentionally omitted any performance results. We have implemented our system on a 8 core, 2.6GHz Clovertown system running inside a Dell Precision 630 box. The machine is running Linux, and achieves near-perfect scalability during rendering. The system we implemented our algorithms in is a variant of Wald et al.'s dynamic BVH system [19]. No asynchronous, partial, lazy, or on-demand builds are used, nor selective updating/restructuring, nor is any additional information passed by the application. All data structures are rebuilt from scratch for every frame, with the set of triangles and triangle vertices passed by the application as a "soup" of triangles.

Using the algorithms described above, we have three algorithms we can compare: the single-threaded binning, the hybrid parallel binning, and the grid-based binning. All of those algorithms are reasonably well optimized, even though the focus for the parallel algorithms has been on scalability, and even though each algorithm received only about one day of programming/debugging/optimization time.

To put those three algorithms into perspective, we compare them to our "gold standard" SAH sweep build outlined in [19], as well to a reasonably well optimized BIH-style build that was implemented in that code base some time ago. Both sweep and BIH-style build are single-threaded. All experiments are run with the same parameter settings, using the default parameters for the respective algorithm, and no parameter tweaking allowed.

5.1 Comparison of algorithms

As actual experiments, we have run these different build methods on a variety of typical ray tracing scenes: the Fairy forest scene used in [19], the conference scene, the blade model, and the Thai statue. Among those, the blade and Thai statue are scanned models with relatively uniform, well-behaved triangle distribution for which a median build can be expected to yield good performance. The fairy has a somewhat more complex structure, as has the conference room.

Table 1 shows, for all of these models and all of the above-mentioned algorithms, the absolute build times (in milliseconds), as well as the *relative* render performance compared to the gold standard sweep build. We have chosen absolute numbers for the build times since those depend only on the model; and relative numbers for the render performance since those depend on factors like what shading is used, how many rays are traced, etc.

Deterioration in build quality. With regard to impact on render performance, Table 1 once again shows that a spatial median BIH-build has a significant impact on render performance, in particular for models with complex triangle distributions like the conference model, where render performance drops by roughly one third. However, it is also by far the fastest for the single-threaded builds, outperforming even our binned build by about $2-2.5\times$.

The single-threaded binning yields render performance roughly comparable to the sweep build, being slightly slower in the fairy and conference scenes, but actually yielding higher render performance for the blade and Thai statue models. That it can actually yield *higher* performance than what we had declared as a "gold standard" is due to the fact that the SAH itself is only a heuristic, and the split with lowest *estimated* cost is not necessarily the one with lowest *actual* cost. Eventually, when using enough bins, the binned build converges to the same performance as achieved by the sweep build.

The hybrid build performs exactly the same partitions as the single-threaded binning (just in parallel), and thus ends up with the same render performance. The grid build uses spatial median subdivision during the grid build phase, and thus achieves somewhat worse quality near the BVH's root, even though it is not just as bad as the BIH, since it eventually reverts to an SAH in the sub-tree stage. Note that a target of 5 sub-trees per thread forces several spatial subdivisions even for a single build thread, so the single-threaded grid performance is lower than the single-threaded binning.

Absolute build time. As mentioned before, for single-threaded builds the BIH is still the fastest to build, which makes it interesting even in the face of its adverse effect on render performance. The single-threaded binning is about $2-2.5\times$ slower than the BIH build, but roughly one order of magnitude faster than the sweep build, which makes it a nice compromise between the BIH and the sweep builds. In particular, the single-threaded binning enables interactive from-scratch rebuilds for both the fairy and the conference room, at around 12 resp. 7 rebuilds per second.

Comparison to kd-tree builds. These numbers also compare quite favorably to single-threaded kd-tree builds, which Popov et al. and Hunt et al. reported at around 150,000 and 300,000 triangles per second. Even in a single thread, our binned build achieves 1.3-2 million triangles per second (1.32m for the Thai statue, 1.37 for the blade, 1.94 for the conference, and 2.05 for the fairy). This indicates that our algorithm can build a BVH between 4 and 13 times faster than similar techniques for a kd-tree, while not showing any appreciable deterioration in render performance at all.

Scalability. For both the grid-based and the hybrid parallelization schemes, scalability is far below expectations. Scalability is reasonably good for 2 threads, and still acceptable for 4, but in all our experiments, going from 4 to 8 threads gives hardly any improvement. As mentioned above, we first suspected a bug in our hybrid algorithm, and implemented the grid mostly as a reference solution, with same results. Though both algorithms required some careful tuning of the implementation to avoid scalability bottlenecks (such as sorting the sub-trees by size), their current implementations have only trivially short non-parallelized sections, which cannot explain this behavior. Also, 'top' shows almost full CPU utilization for both algorithms, which indicates that all threads are active.

Apart form actually doing more work–which is not the case–this behavior can only be explained if the actual operations get more

costly; and since the cost of arithmetic operations is constant, it can only mean that memory accesses get more costly due to cache misses or insufficient bandwidth. A closer look reveals that this is indeed the case even for a relatively small number of threads. In fact, even the setup state—in which we compute but one bounding box and centroid for each triangle—hits the memory wall very quickly. The setup phase consumes roughly 20-25% of total build time, and essentially stops scaling after 3 to 4 threads (without any synchronization at all).

On the positive side, this means that we can build a complete BVH for only around $4\times$ the cost of computing the triangle bounds, and that *no* algorithm that computes each triangle's bounding box and centroid could be more than $4\times$ faster than ours (at least on that particular hardware).

Much better scalability for building data structures has been reported by both Ize et al. [7] and Shevtsov et al. [14], but on different hardware. In particular, Ize et al. use a NUMA architecture in which each processor has its own dedicated memory (i.e., the individual processors do not have to share bandwidth). It remains to be seen whether our algorithm achieves better scalability on such a memory architecture, but that has not been investigated, yet.

Even with unsatisfactory scalability, parallelization still gives a huge benefit, allowing roughly $2\times$ faster builds than a (single-threaded) BIH build, and allowing quite interactive builds for all models except the Thai statue, and achieving rates of up to 6.4 million triangles per second for the fairy model, and 7.1 million for the Thai statue.

5.2 Combined system performance

In addition to just evaluating the build performance, we have also integrated our parallel builder into the DynBVH architecture, and evaluated the performance of the whole system. The system is fully asynchronous, meaning there is a stage in which the application computes the triangles (via interpolating from predefined keyframes), build stage in which our algorithm is being used, a render stage with tile-based dynamic load balancing, and a display stage in which the image is displayed.

The render stage is dynamically load-balanced, and usually scales very well. Updating the vertices and computing the triangle records is also parallelized, though quite simplistically so with a single *openmp* statement each. With our current graphics driver, display is a bottleneck, and will not allow for more than 20 frames per second at 1024x1024 pixels. For the build stage, we use the grid-based parallel binning, on all eight threads.

To apply our system to a case where simple re-fitting would absolutely not work, we have taken the same model that Yoon et al. used in their selective restructuring [25], which they have graciously made available to us. This scene has around 250,000 triangles that undergo quite severe motion (see Figure 1) as soon as the dragon disintegrates after being hit by the bunny.

We render the model with simple diffuse shading and shadows from one point light source. All triangles are animated on the fly and all data structures are re-built from scratch. Doing so, we achieve 5-10 frames per second including display (around 7-16 without display) for the views seen in Figure 1. Performance does drop by about a factor of two over the course of this animation; however, this is solely due to the *rendering* stage getting slower because of the chaotic triangle distribution. Build time stays roughly constant throughout the animation at around 50ms per build, irrespective of triangle distribution. Though no special tuning was performed for this scene, these results are already competitive with the ones presented by Yoon et al., even after accounting for the (severe) difference in hardware horsepower. Though with as severe differences in hardware all comparisons are apples and oranges, anyway, it seems that even after accounting for the faster hardware, we can rebuild the full BVH from scratch in about the same it takes for selectively restructuring it. However, a more detailed comparison would be very interesting, in particular if parallelization and scalability issues are taken into account.

5.3 Summary and Conclusion

In this paper, we have investigated how to build SAH-based BVHs from scratch as quickly as possible, by employing the same techniques that have proven so successful for kd-trees. While we do, in many aspects, use *exactly* the same techniques, it turns out that these seem to work at least as good for BVHs, and often even better. Consequently, we achieve single-threaded from-scratch rebuild rates (i.e., triangles per second) that are up to $10\times$ higher than comparable rates published for kd-trees.

Compared to previous BVH build algorithms, we are still slower than Wächter's BIH-style build, but achieve higher rendering performance; and at roughly the same render performance, we outperform sweep builds by roughly $10\times$.

We have also presented two different schemes for parallelizing our algorithm, both of which should scale well in theory, but both of which hit the memory wall after only around 3-4 threads. Consequently, both parallelization schemes do not scale to more than 4 threads on our current hardware setup (though they probably would on a NUMA architecture), but they achieve up to $4\times$ speedup over the single-threaded variants eventually reaching rates of several million triangles per second from-scratch rebuild performance.

Integrated into a complete ray tracer for animated scenes, our parallel binned build algorithm allows for rendering even the exploding dragon scene at around 5-10 frames per second including full rebuilds from scratch, ray tracing, shading, shadows, and display, which is quite competitive with previously published results.

In future work, we are most interested in investing the scalability of our algorithms on different memory architectures, and comparing it to the parallel grid build by Ize et al. It would also be interesting to couple our build algorithm with a Razor-like approach in which information from a scene graph is used to accelerate the plane selection process, and in which the BVH is built lazily. In particular the parallel aspects of that approach would be very interesting.

ACKNOWLEDGMENTS

We would like to thank Sung-Eui Yoon for providing the exploding dragon data set, and for doing so on very short notice. We also explicitly acknowledge Johannes Günther and Carsten Benthin, who have also worked on fast BVH construction, and who have provided insight into their respective approaches. The ray tracing system that our methods have been implemented in have been contributed to by many others, in particular, by Solomon Boulos and Peter Shirley.

This work has been funded by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions (grant W-7405-ENG-48LA-13111-PR), as well as through a visiting professorship by Intel Corp.

REFERENCES

[1] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS Conference Proceedings*, volume 32, pages 37–45, Washington, DC, 1968. Thompson Book Company. Proceedings of the Spring Joint Computing Conference (SJCC).

[2] R. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84)*, 18(3):137–144, 1984.

[3] J. Günther, H. Friedrich, I. Wald, H.-P. Seidel, and P. Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, Sept. 2006. (Proceedings of Eurographics).

[4] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[5] W. Hunt, G. Stoll, and W. Mark. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[6] T. Ize, I. Wald, and S. G. Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[7] T. Ize, I. Wald, C. Robertson, and S. G. Parker. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 47–55, 2006.

[8] T. Larsson and T. Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. In *Workshop On Virtual Reality Interaction and Physical Simulation*, pages 91–100, 2005.

[9] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006.

[10] M. C. Lin and D. Manocha. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry, 2nd Edition*, pages 787–807. CRC Press, 2004.

[11] B. C. Ooi, R. Sacks-Davis, and K. J. McDonnell. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, 1987.

[12] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[13] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH 2005).

[14] M. Shevtsov, A. Soupikov, and A. Kapustin. Fast and scalable kd-tree construction for interactively ray tracing dynamic scenes. *Computer Graphics Forum*, 26(3), 2007. (Proceedings of Eurographics), to appear.

[15] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. Cani, F. Faure, N. Magnenat-Thalmann, W. Straßer, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–82, 2005.

[16] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–14, 1997.

[17] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[18] I. Wald, C. Benthin, and P. Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 11–20, 2003.

[19] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.

[20] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, 2006.

[21] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *State of the Art Reports, Eurographics 2007*, 2007.

[22] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[23] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[24] S. Woop, G. Marmitt, and P. Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, 2006.

[25] S.-E. Yoon, S. Curtis, and D. Manocha. Ray Tracing Dynamic Scenes using Selective Restructuring. In *Eurographics Symposium on Rendering*, 2007.