On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)

Ingo Wald[†] Vlastimil Havran[◇]

[†]SCI Institute, University of Utah [◇]Czech Technical University in Prague

Abstract

Though a large variety of efficiency structures for ray tracing exist, kd-trees today seem to slowly become the method of choice. In particular, kd-trees built with cost estimation functions such as a surface area heuristic (SAH) seem to be important for reaching high performance. Unfortunately, most algorithms for building such trees have a time complexity of $O(N \log^2 N)$, or even $O(N^2)$. In this paper, we analyze the state of the art in building good kd-trees for ray tracing, and eventually propose an algorithm that builds SAH kd-trees in $O(N \log N)$, the theoretical lower bound.

1 Introduction

Over the last two decades, ray tracing has become a mature field, and a large variety of different acceleration schemes been proposed, including Octrees, Bounding Volume Hierarchies (BVHs), different variants of grids, kd-trees, etc (see, e.g., [2, 6]).

Though all these techniques have their merits, kd-trees recently seem to establish themselves as the most widely used technique. In particular since the appearance of fast – and kd-tree-based – coherent packet tracing [30, 26] and frustum traversal [17] kd-trees are increasingly believed to be the "best known method" for fast ray tracing [20]. Both concepts become particularly interesting if the kd-tree is built to minimize the number of traversal and intersection steps, which today is usually done using a heuristic cost estimate, the Surface Area Heuristic (SAH) [13]. Kd-trees have recently received lots of attention, and today are well understood in building them to be efficient, in traversing them quickly, and even in how to optimize low-level implementation and memory layout.

So far however, research on using kd-trees in ray tracing has almost exclusively concentrated on traversing them quickly, as well as on building them to *be* efficient, i.e., such that they minimize the expected number of intersections and traversal steps during rendering. The related question – the cost and complexity of building them – has been widely ignored. Construction time has historically been insignificant compared to rendering time, and was mostly ignored. However, this lack of fast construction algorithms now becomes a problem, as in particular good kd-trees take considerable time to build, and often have a time complexity of $O(N \log^2 N)$ or even $O(N^2)$. Despite growing CPU performance, this becomes problematic given the current trend towards more and more realistic – and more complex – scenes.

1.1 Contributions

In this paper, we focus on three contributions:

1. A comprehensive recap of building good kd-trees using a Surface Area Heuristic. We will not introduce any new techniques, but combine the often scattered knowledge on kd-tree construction in a coherent, concise and consistent form.
2. A discussion of three schemes for building SAH-optimized kd-trees, and an analysis of their computational complexity.
3. A algorithm that builds an SAH kd-tree in $O(N \log N)$, the asymptotic lower bound for building kd-trees.

Our emphasis is on building highly efficient kd-trees in a robust and *asymptotically* efficient way. We do not attempt fast rebuilds for interactive applications ([4, 21, 28, 12, 25]...), but instead focus on the algorithmic aspects only, and ignore any low-level optimizations—though our implementation is reasonably efficient, in case of doubt we use high-level, templated, and highly parameterizable code.

2 Building KD-Trees for Ray Tracing

Before discussing the details of our $O(N \log N)$ construction algorithm, we will first summarize the state of the art in building good kd-trees. This provides the background for the rest of the paper, and will introduce the concepts and terminology used later on.

In the following, we will consider a scene $\mathcal{S}$ made up of N triangles. A kd-tree over $\mathcal{S}$ is a binary tree that recursively subdivides the space covered by $\mathcal{S}$: The root corresponds to the axis-aligned bounding box (AABB) of $\mathcal{S}$; interior nodes represent planes $p_{k,\xi}(x) : x_k = \xi$ that recursively subdivide space perpendicular to the coordinate axis; leaf nodes store references to all the triangles overlapping the corresponding voxel. Essentially, all kd-tree construction schemes follow the same recursive scheme:

Algorithm 1 Recursive KD-tree build

```
function RecBuild(triangles $T$, voxel $V$) returns node
  if Terminate(T,V) then
    return new leaf node($T$)
  $p = FindPlane(T, V)$ {Find a "good" plane $p$ to split $V$}
  $(V_L, V_R) =$ Split $V$ with $p$
  $T_L = \{t \in T | (t \cap V_L) \neq \emptyset$
  $T_R = \{t \in T | (t \cap V_R) \neq \emptyset$
  return new node($p, RecBuild(T_L, V_L), RecBuild(T_R, V_R)$)

function BuildKDTree(triangles[] T) returns root node
  $V = \mathcal{B}(T)$ {start with full scene}
  return RecBuild($T$,$V$)
```

Obviously, the structure of a given kd-tree – i.e., where exactly the planes are placed, and when voxels are created – directly influences how many traversal steps and triangle intersections the ray tracer has to perform. With today's fast ray tracers, the difference between a "good" and a naïvely built kd-tree is often a factor of two or more [26]. For the recently proposed hierarchical traversal schemes, well built kd-trees are even reported to be up to (quote) "several times faster than a mediocre kd-tree" (see [20]).

2.1 Naïve, "spatial median" KD-Trees

Eventually, all the intelligence in a kd-tree construction scheme lies in where to place the splitting plane p, and in determining when to stop the recursion. One of the most trivial - and thus, quite often used - methods for building kd-trees is the so called "spatial median splitting", in which the dimension p_k is chosen in round robin fashion, and the plane is positioned at the spatial median of the voxel,

$$p_k = D(V) \bmod 3 \quad \text{and} \quad p_\xi = \frac{1}{2}(V_{min,p_k} + V_{max,p_k}),$$

where $D(V)$ is the current subdivision depth.

Usually, subdivision is performed until either the number of triangles falls below a certain threshold $\mathcal{K}_{minTris}$, or until the subdivision depth exceeds a certain maximum depth $\mathcal{K}_{maxDepth}$:

$$Terminate(T, V) = |T| \leq \mathcal{K}_{triTarget} \vee D(V) \geq \mathcal{K}_{maxDepth}.$$

3 The Surface Area Heuristic (SAH)

Spatial median splitting is quite simplistic, and an abundance of heuristic, ad-hoc techniques to build better kd-trees is available (see, e.g., [6, 5]. In particular techniques that maximize "empty space" – preferably close to the root of the tree – seem to be most successful. Nevertheless, applying these techniques in practice is often problematic: First, they require scene-specific "magic constants" to work well; second, in many situations different heuristics disagree on what to do, and choosing the right one is non-trivial.

To remedy this, several researchers [3, 13] [22, 6] have investigated the factors that influence the performance of hierarchical spatial subdivision, and have derived a more profound approach, the surface area heuristic (SAH). Essentially, the SAH considers the geometry of splitting a voxel V with plane p – i.e., the resulting child voxel V_L and V_R, as well as the numbers N_L and N_R overlapping these two, respectively – and estimates the *expected* cost of traversing the such-split voxel. Therefore, the SAH makes several assumptions:

(i) That rays are uniformly distributed, infinite lines; i.e., that they are uniformly distributed, and neither start, nor terminate, nor get blocked inside a voxel.

(ii) That the cost for both a traversal step and for a triangle intersection are known, and are $\mathcal{K}_T$ and $\mathcal{K}_I$, respectively.

(iii) That the cost of intersecting N triangles is roughly $N\mathcal{K}_I$, i.e., directly linear in the number of triangles[1].

Using these assumptions then allows for expressing the cost of a given configuration: For uniform distributed lines and convex voxels, geometric probability theory [18] tells us that for a ray known to hit a voxel V the conditional probability $\mathcal{P}$ of also hitting a subvoxel $V_{sub} \subset V$ is

$$\mathcal{P}_{[V_{sub}|V]} = \frac{\mathcal{SA}(V_{sub})}{\mathcal{SA}(V)}, \tag{1}$$

where $\mathcal{SA}(V)$ is the surface area of V. The expected cost $\mathcal{C}_V(p)$ for a given plane p then is one traversal step, plus the expected cost of intersecting the two children,

$$\mathcal{C}_V(p) \quad = \quad \mathcal{K}_T + \mathcal{P}_{[V_l|V]}\mathcal{C}(V_l) + \mathcal{P}_{[V_r|V]}\mathcal{C}(V_r). \tag{2}$$

3.1 Local Greedy SAH Heuristic

Expanding (2), the cost of a complete tree $\mathcal{T}$ is

$$\mathcal{C}(\mathcal{T}) = \sum_{n \in nodes} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)}\mathcal{K}_T + \sum_{l \in leaves} \frac{\mathcal{SA}(V_l)}{\mathcal{SA}(V_S)}\mathcal{K}_I, \tag{3}$$

where V_S is the AABB of the complete scene $\mathcal{S}$. The best kd-tree T for a scene $\mathcal{S}$ would be the one for which equation 3 is minimal. The number of possible trees, however, rapidly grows with scene size, and finding the globally optimal tree today is considered infeasible except for trivial scenes.

[1]In theory, adding a constant to simulate the setup cost for traversing the leaf (i.e., a leaf cost of $1\mathcal{K}_T + N\mathcal{K}_I$) should be more accurate, but—at least in our experiments—in practice is worse, probably because it punishes "flat cells" at the sides, which are often favorable in architectural scenes.

Instead of a globally optimal solution, one therefore uses a "locally greedy approximation", where the cost of subdividing V with p is computed as if both resulting children would be made leaves,

$$\begin{aligned} \mathcal{C}_V(p) \quad &\approx \quad \mathcal{K}_T + \mathcal{P}_{[V_L|V]}|T_L|\mathcal{K}_I + \mathcal{P}_{[V_R|V]}|T_R|\mathcal{K}_I \quad (4) \\ &= \quad \mathcal{K}_T + \mathcal{K}_I\left(\frac{\mathcal{SA}(V_L)}{\mathcal{SA}(V)}|T_L| + \frac{\mathcal{SA}(V_R)}{\mathcal{SA}(V)}|T_R|\right). \quad (5) \end{aligned}$$

This is a gross simplification, and tends to overestimate the correct cost, as T_L and T_R are likely to be further subdivided, and will thus have lower cost than assumed. Nevertheless, in practice this approximation works well, and – though many theoretically better approximation have been tried – so far no consistently better approximation could be found.

3.2 Automatic Termination Criterion

Apart from a method for estimating the cost of any potential split p, the SAH also provides an elegant and stable way of determining when to stop subdivision: As the cost of leaf can be well modeled as $\mathcal{C}_{asLeaf} = \mathcal{K}_I|T|$, further subdivision does not pay off if even the best split is more costly then not splitting at all, i.e.,

$$Terminate(V, T) = \begin{cases} true & ; \min_p \mathcal{C}_V(p) > \mathcal{K}_I|T| \\ false & ; otherwise \end{cases} \tag{6}$$

This local approximation can easily get stuck in a local minimum: As the local greedy SAH overestimates $C_V(p)$, it might stop subdivision even if the *correct* cost would have indicated further subdivision. In particular, the local approximation can lead to premature termination for voxels that require splitting off flat cells on the sides: many scenes (in particular, architectural ones) contain geometry in the form of axis-aligned boxes (a light fixture, a table leg or table top, ...), in which case the sides have to be "shaved off" until the empty interior is exposed. For wrongly chosen parameters, or when using cost functions different from the ones we use (in particular, ones in which a constant cost is added to the leaf estimate), the recursion can terminated prematurely. Though this pre-mature exit could also be avoided in a hardcoded way—e.g., only performing the automatic termination test for non-flat cells—we propose to follow our formulas, in which case no premature exit will happen.

3.3 Modifications and Extensions

In practice, most of the assumptions used in deriving equation 5 are at least questionable: Rays will usually *not* pass unoccluded through populated voxels; the ray density will usually *not* be uniform; the cost of the left and right half should *not* be linear (but rather logarithmic), both leaf, left, and right half should have a constant factor simulating the traversal step; memory, cache, or CPU-specific effects (SIMD) are gravely neglected; etc.

Nevertheless, in practice the basic SAH as explained above—local greedy plane selection, linear leaf cost estimate, and automatic termination criterion—often works best, and only few modifications are known to consistently yield better improvements. Among those, the most common is to favor splits that cut off empty space by biasing the cost function; if either N_L or N_R gets zero, the expected cost of the split is reduced by a constant factor. I.e., the expected costs get multiplied by

$$\lambda(p) = \begin{cases} 80\% & ; |T_L| = 0 \vee |T_R| = 0 \\ 1 & ; otherwise \end{cases} \tag{7}$$

If the ray tracer supports "termination objects" [6, 17], a similar bias can also be used for those cases where the split plane is entirely covered by triangles, which however works almost exclusively for certain architectural scenes. To keep the automatic termination criterion from getting stuck in a local minimum, it has also been reported to help if subdivision is continued for a certain number of steps even though the termination criterion would advise not to [6, 16]. This, however, has proven to be quite hard to master for general scenes. Finally, instead of only using the cost-based termination criterion some implementations additionally use the maximum depth criterion, usually to reduce memory usage.

3.4 Split Candidates and Perfect Splits

So far, we have defined the actual procedure for estimating the cost of p once N_L, N_R, V_L and V_R are known. As there are infinitely many potential planes p, one needs a more constructive approach: For any pair of planes (p_0, p_1) between which N_L and N_R do not change, $\mathcal{C}(p)$ is linear in the position x_p of p. Thus, $\mathcal{C}(p)$ can have its minima only at those - finitely many - planes where N_L and N_R change [6]. As we are only interested in these minima, we will in the following refer to these planes as "split candidates".

One simple choice of split candidates is to use the 6 planes defining the triangle's AABB $\mathcal{B}(t)$. Though this is easiest to code and fastest to build, it is also inaccurate, as it may sort triangles into voxels that the triangle itself does not actually overlap. The intuitive fix of performing some a-posterior triangle-voxel overlap test does not work, either: For small voxels it frequently happens that the voxel is completely enclosed in $\mathcal{B}(t)$, and thus no split candidate could be found at all. The accurate way of determining the candidate planes thus is to first *clip* the triangle t to the voxel V, and use the sides of the clipped triangle's AABB $\mathcal{B}(t \cap V)$ (also see [9, 6]). As this is significantly more accurate then the AABB, the candidates such produced are also often called "perfect splits" (in [7], this technique is reported to give an average speedup of 9–35%). During clipping, special care has to be taken to correctly handle special cases like "flat" (i.e., zero-volume) cells, or cases where numerical inaccuracies may occur (e.g., for cells that are very thin compared to the size of the triangle). For example, we must make sure not to "clip away" triangles lying *in* a flat cell. Note that such cases are not rare exceptions, but are in fact encouraged by the SAH, as they often produce minimal expected cost.

3.5 Accurate Determination of N_L and N_R

To compute equation 5, for each potential split p we have to compute the number of triangles N_L and N_R for V_L and V_R, respectively. Here as well, a careful implementation is required. For example, an axis-aligned triangle in the middle of a voxel should result in two splits, generating two empty voxels and one flat, nonempty cell. This in fact is the perfect solution, but requires special care to handle correctly during both build and traversal. For flat cells, we must make sure not to miss any triangles that are lying exactly *in* the flat cell, but must make sure that non-parallel triangles just touching or penetrating it will get culled (as in the latter case $t \cap p$ has zero area, and cannot yield an intersection).

Quite generally, determining N_L and N_R via a standard triangle-voxel overlap test may result in sorting triangles into a voxel even if they overlap in only a line or a point, and triangles lying *in* in the plane p may be sorted into both halves. Both cases are not actually wrong, but inefficient. Thus, the most exact solution requires to actually split T into *three* sets, T_L, T_R, T_P, the triangles having non-zero overlap for $V_L \setminus p$, $V_R \setminus p$, and p,

$$\begin{aligned} T_L &= \{t \in T | Area(t \cap (V_L \setminus p)) > 0\} && (8) \\ T_R &= \{t \in T | Area(t \cap (V_R \setminus p)) > 0\} && (9) \\ T_P &= \{t \in T | Area(t \cap p) > 0\}, && (10) \end{aligned}$$

where $V_L \setminus p$ is the part of the voxel V_L that does not lie on p (p forms one of V_L's sides). Once these sets are known, we can evaluate eq. 5 twice – once putting T_P with T_L, and once with T_R – and select the one with lowest cost (see Algorithm 2).

Algorithm 2 Final cost heuristic for a given configuration.

```
function C($P_L, P_R, N_L, N_R$) returns ($\mathcal{C}_V(p)$)
  return $\lambda(p)(\mathcal{K}_T + \mathcal{K}_I(P_L N_L + P_R N_R))$

function SAH($p, V, N_L, N_R, N_P$) returns ($\mathcal{C}_p, p_{side}$)
  $(V_L, V_R) = SplitBox(V, p)$
  $P_L = \frac{\mathcal{SA}(V_L)}{\mathcal{SA}(V)}$; $P_R = \frac{\mathcal{SA}(V_R)}{\mathcal{SA}(V)}$
  $c_{p\to L} = C(P_L, P_R, N_L + N_p, N_R)$
  $c_{p\to R} = C(P_L, P_R, N_L, N_R + N_P)$
  if $c_{p\to l} < c_{p\to l}$ then
    return ($c_{p\to L}$,LEFT)
  else
    return ($c_{p\to R}$,RIGHT)
```

4 On building SAH-based KD-Trees

In the preceding sections, we have defined the surface area heuristic, including what split candidates to evaluate, how to compute N_L, N_R, N_P, and C_V, and how to greedily chose the plane. In this section, we present three different algorithms to build a tree using this heuristic, and will analyze their performance. All three algorithms build the same trees, and differ only in their efficiency in doing that.

All construction schemes will make use of recursion, so we will need some assumptions on how that recursion behaves. In absence of any more explicit knowledge, we will use the – quite gross – assumptions that subdividing N triangles yields two lists of roughly the size $\frac{N}{2}$, and that recursion proceeds until $N = 1$. As an example, let us first consider the original median-split kd-tree: The cost $T(N)$ for building a tree over $N = |\mathcal{T}|$ triangles requires $O(N)$ operations for sorting T into T_L and T_R, plus the cost for recursively building the two children, $2T(\frac{N}{2})$. Expansion yields

$$T(N) = N + 2T(\frac{N}{2}) = \cdots = \sum_{i=1}^{\log N} 2^i \frac{N}{2^i} = N \log N.$$

Note that due to its relation to sorting, $O(N \log N)$ is also the theoretical lower bounds for kd-tree construction.

4.1 Naïve $O(N^2)$ Plane Selection

For spatial medial splitting, determining the split plane is trivial, and costs $O(1)$. For a SAH-based kd-tree however, finding the split plane is significantly more complex, as each voxel V can contain up to $6N$ potential split candidates. For each of these we have to determine N_L, N_R, and N_P. In its most trivial form, this can be done by iterating over each triangle t, determining all its split candidates C_t, and – for each – determine N_L, N_R, and N_P by computing T_L, T_R, and T_P according to Section 3.5 (see Algorithm 3).

Classifying N triangles costs $O(N)$, which, for $|C| \in O(N)$ potential split planes amounts to a cost of $O(N^2)$ in each partitioning step. During recursion, this $O(N^2)$ partitioning cost sums to

$$\begin{aligned} T(N) &= N^2 + 2T(\frac{N}{2}) = \sum_{i=1}^{logN} 2^i \left(\frac{N}{2^i}\right)^2 \\ &= N^2 \sum 2^{-i} \in O(N^2). \end{aligned}$$

4.2 $O(N \log^2 N)$ Construction

Nevertheless, $O(N^2)$ algorithms are usually impractical except for trivially small N. Fortunately, however, algorithms for building the same tree in $O(N \log^2 N)$ are also available, and widely known (see, e.g., [14, 23], the latter even including source code). Though this algorithm is sufficiently described in these publications, we will

Algorithm 3 Algorithm for naïve $O(N^2)$ plane selection

```
function PerfectSplits(t, V) returns $\{p_0, p_1, ...\}$
  $B =$ Clip $t$ to $V$ {consider "perfect" splits}
  return $\bigcup_{k=1..3}((k, B_{min,k}) \cup (k, B_{max,k}))$

function Classify(T, $V_L$,$V_R$,p) returns $(T_L, T_R, T_P)$
  $T_l = T_r = T_p = \emptyset$
  for all $t \in T$
    if $t$ lies in plane $p \wedge Area(p \cap V) > 0$ then
      $T_P = T_P \cup t$
    else
      if $Area(t \cap (V_L \setminus p)) > 0$ then $T_L = T_L \cup t$
      if $Area(t \cap (V_R \setminus p)) > 0$ then $T_R = T_R \cup t$

function NaïveSAH::Partition(T, V) returns ($p$,$T_l$,$T_r$)
  for all $t \in T$
    $(\hat{C}, \hat{p}_{side}) = (\infty, \emptyset)$ {search for best node:}
    for all $p \in PerfectSplits(t, V)$
      $(V_L, V_R)$ = split $V$ with $p$
      $(T_L, T_R, T_P) = Classify(T, V_L, V_R, p)$
      $(\mathcal{C}, p_{side}) = SAH(V, p, |T_L|, |T_R|, |T_P|)$
      if $\mathcal{C} < \hat{\mathcal{C}}$ then
        $(\hat{\mathcal{C}}, \hat{p}_{side}) = (\mathcal{C}, p_{side})$
  $(T_l, T_r, T_p) = Classify(T, V_l, V_r, p)$
  if ($\hat{p}_{side} = LEFT$) then
    return ($\hat{p}, T_l \cup T_p, T_r$)
  else
    return ($\hat{p}, T_l, T_r \cup T_p$)
```

also derive it here in detail. Our final $O(N \log N)$ will be derived from this $O(N \log^2 N)$ algorithm, and will share much of the notation, assumptions, and explanations. it can thus be best explained side by side with the $O(N \log^2 N)$ algorithm.

Since the $O(N^2)$ cost of the naïve variant is mainly due to the cost of computing N_L, N_R, and N_P, improving upon the complexity requires to compute these values more efficiently. As mentioned before, these values only change at the split candidate planes. For each such plane $p = (p_k, p_\xi)$, there is a certain number of triangles starting, ending, or lying in that plane, respectively. In the following, we will call these numbers p^+, p^-, and $p^|$, respectively.

Let us consider that these p^+, p^-, and $p^|$ are known for all p. Let us further consider only one fixed k, and assume that all p's are sorted in ascending order with respect to p_k. Then, all N_L, N_R, and N_P can be computed incrementally by "sweeping" the potential split plane over all possible plane positions p_i: For the *first* plane p_0, by definition no planes will be to the left of p_0, $p^|_0$ triangles will lie *on* p_0, and all others to the right of it, i.e.,

$$N_l^{(0)} = 0 \quad N_p^{(0)} = p_0^| \quad N_r^{(0)} = N - p_0^|.$$

From p_{i-1} to p_i, N_L, N_R, and N_P will change as follows:

1. The new N_P will be $p^|_i$; these $p^|_i$ triangles will no longer be in V_R. The triangles on plane $p^|_{i-1}$ will now be in V_L
2. Those triangles having *started* at p_{i-1} now overlap V_L.
3. Those triangles *ending* at p_i will no longer overlap V_R.

For N_L, N_R, and N_P, this yields three simple update rules:

$$N_L^{(i)} = N_L^{(i-1)} + p^|_{i-1} + p^+_{i-1} \quad (11)$$

$$N_R^{(i)} = N_R^{(i-1)} - p^|_i - p^-_i \quad (12)$$

$$N_P^{(i)} = p^|_{i-1} \quad (13)$$

To implement this incremental update scheme, for each p_i we need to know p_i^+, p_i^-, and $p_i^|$. First, we fix a dimension k. For this k, we iterate over all triangles t, generate t's perfect splits (by computing $B = \mathcal{B}(t \cap V)$, see Section 3.4), and store the "events" that would happen if a plane is swept over t: If the triangle is perpendicular to k, it generates a "planar event" $(t, B_{k,min}, |)$, otherwise it generates a "start event" $(t, B_{k,min}, +)$ and a "end event" $(t, B_{k,max}, -)$. Each event $e = (e_t, e_\xi, e_{type})$ consists of a reference to the triangle having generated it, the position e_ξ of the plane, and a flag e_{type} specifying whether t starts, ends, or is planar at that plane.

Once all events for all triangles are generated, we sort the event list E by ascending plane position, and such that for equal plane position end events precede planar events, which precede start events. For two events a and b this yields the ordering

$$a <_E b = \begin{cases} true & ; (a_x < b_x) \vee (a_x = b_x \wedge \tau(a) < \tau(b)) \\ false & ; otherwise, \end{cases}$$

where $\tau(e_{type})$ is 0 for end events, 1 for planar events, and 2 for start events, respectively.

When iterating over this $<_E$-sorted E, by construction we first visit all events concerning p_0, then all those concerning p_1, etc. Furthermore, for a given sequence of p_i-related events we first visit all ending events, then all planar events, and finally all starting events. Thus, p_i^+, $p_i^|$, and $p_i^|$ can be determined simply by counting how many events for the same type and plane one has encountered. Now, all that has to be done is to run this algorithm for every dimension k, and keep track of the best split found, $\hat{p}$ (see Algorithm 4).

Algorithm 4 Incremental sweep to find $\hat{p}$.

```
function PlaneSweep::FindPlane(T, V) returns best $\hat{p}$
  $(\hat{\mathcal{C}}, \hat{p}) = (\infty, \emptyset)$ {initialize search for best node}
  {consider all K dimensions in turn:}
  for $k = 1..3$
    {first, compute sorted event list:}
    eventlist $E = \emptyset$
    for all $t \in T$
      $B = ClipTriangleToBox(t, V)$
      if $B$ is planar then
        $E = E \cup (t, B_{min,k}, |)$
      else
        $E = E \cup (t, B_{min,k}, +) \cup (t, B_{max,k}, -)$
    sort($E$,$<_E$) {sort all planes according to $<_E$}

    {iteratively "sweep" plane over all split candidates:}
    $N_l = 0, N_p = 0, N_r = |T|$ { start with all tris on the right}
    for $i = 0; i < |E|;$
      $p = E_{i,p}, \quad p^+ = p^- = p^| = 0$
      while $i < |E| \wedge E_{i,\xi} = p_\xi \wedge E_{i,type} = -$
        inc $p^-$; inc $i$
      while $i < |E| \wedge E_{i,\xi} = p_\xi \wedge E_{i,type} = |$
        inc $p^|$; inc $i$
      while $i < |E| \wedge E_{i,\xi} = p_\xi \wedge E_{i,type} = +$
        inc $p^+$; inc $i$
      {now, found next plane $p$ with $p^+$,$p^-$ and $p^|$...}
      {move plane onto $p$}
      $N_P = p^|$, $N_R$-=$p^|$, $N_R$-=$p^-$
      $(\mathcal{C}, p_{side}) = SAH(V, p, N_L, N_R, N_P)$
      if $\mathcal{C} < \hat{p}_\mathcal{C}$ then
        $(\hat{\mathcal{C}}, \hat{p}, \hat{p}_{side}) = (\mathcal{C}, p, p_{side})$
      $N_L$+=$p^+$, $N_L$+=$p^|$, $N_P = 0$ {move plane over p}
  return ($\hat{p}$, $\hat{p}_{side}$)
```

This algorithm initializes N^0_L,N^0_R, and N^0_P differently from the way explained above. This is due to some slight optimization in when the plane is evaluated and in when the variables are updated. This optimization allows for not having to keep track of the previous plane's parameters, but otherwise proceeds exactly as explained above. Thought he explanation above is more intuitive, the code is cleaner with these optimizations applied.

As mentioned before, we have tagged each event with the ID of the triangle that it belongs to. This is not actually required for finding the best plane, but allows for using a modified "Classify" code that splits T into T_L, T_P, and T_R after the best split has been found: A triangle that ends "before" $\hat{p}$ *must* be in T_L only, and similar arguments hold for T_r and T_p. Thus, once the best plane $\hat{p}$ is found, we iterate once more over E to classify the triangles.

4.2.1 Complexity Analysis

The inner loop of the plane sweep algorithm performs $|P| \in O(N)$ calls to $SAH(\dots)$, and performs $|E| \in O(N)$ operations for computing the p^+,p^-, and $p^|$ values. There are also $O(N)$ clipping operations, and running the loop for all three dimensions just adds a constant factor as well. Similarly, the classification step after $\hat{p}$ has been found (omitted above) also cost $O(N)$. Thus, the complexity is dominated by the cost for sorting, which is $O(N \log N)$. The accumulated cost during recursion then becomes

$$T(N) = N \log N + 2T(\frac{N}{2}) = \dots = N \sum_{i=1}^{\log N} \log \frac{N}{2^i}.$$

Since $N = 2^{\log N}$, this can be further simplified to

$$\begin{aligned} T(N) &= N \sum_{i=1}^{\log N} \log \frac{N}{2^i} = N \sum_{i=1}^{\log N} \log 2^{\log N - i} = N \sum_{i=1}^{\log N} i \\ &= N \frac{\log N (\log N + 1)}{2} \in O(N \log^2 N). \end{aligned}$$

The resulting $O(N \log^2 N)$ complexity is a significant improvement over the naïve algorithm's $O(N^2)$ complexity, but is still significantly higher than the lower bound of $O(N \log N)$.

4.3 $O(NlogN)$ Build using Sort-free Sweeping

In the the previous section's plane sweep algorithm, the main cost factor in each partitioning no longer is the number of plane evaluations, but the $O(N \log N)$ cost for sorting. If that sorting could be avoided, the entire partitioning could be performed in $O(N)$, yielding a recursive cost of only $O(N \log N)$.

Obviously, this per-partition sorting could be avoided if we could devise an algorithm that would sort the event list only *once* at the beginning, and later on perform the partitioning in a way that the sort order is maintained during both plane selection and partitioning. To do this, two problems have to be solved: First, we have to take the sorting out of the inner loop of the "FindPlane" algorithm, and make it work on a single, pre-sorted list. Second, we have to devise a means of generating the two children's sorted event lists from the current node's event list without re-sorting.

As neither can be achieved as long as we sort individually for each k, we first generate one event list containing *all* events from all dimensions. This obviously requires to additionally tag each event with the dimension k that it corresponds to. As we now consider all dimensions in one loop, we keep a separate copy of N_L, N_R, and N_P for each dimension, $N_L^{(k)}$, $N_R^{(k)}$, and $N_P^{(k)}$. Then, each $e = (e_\xi, e_k, e_{type}, e_{ID})$ only affects the N's of its associated dimension e_k, and none other. For these three values, the same incremental operations are performed as in Section 4.2.

Like in the previous Section, we need to quickly determine the number of end (p^-), in-plane ($p^|$), and start (p^+) events for a given split $p = (p_k, p_\xi)$. Thus, as primary sorting criterion, we again pick the plane *position* p_ξ. Note that this is independent of dimension p_k, so planes of different dimensions are stored in an interleaved fashion. For those events with same p_ξ, we want to have them stored such that events with the same dimension (and thus, the same actual plane) lie together. For each of these consecutive events for the same plane, we then again use the same sort order as above: End events first, then planar events, then start events. Assuming that the input set is already sorted, the modified plane finding algorithm is essentially a variant of algorithm 4, in which the three iterations over k have been merged into one (see Algorithm 5).

Algorithm 5 Finding the best plane in O(N).

```
pre: E is $<_E$-sorted
function Partition::FindPlane($N$, $V$, $E$) returns $\hat{p}$
  for all $k \in K$
    {start: all tris will be right side only, for each $k$}
    $N_{L,k} = 0$, $N_{P,k} = 0$, $N_{R,k} = N$
  {now, iterate over all plane candidates}
  for $i = 0; i < |E|;$
    $p = (E_{i,p}, E_{i,k})$;   $p^+$=$p^-$=$p^|$=0
    while $i < |E| \wedge E_{i,k} = p_k \wedge E_{i,\xi} = p_\xi \wedge E_{i,\tau} = -$
      inc $p^-$; inc $i$
    while $i < |E| \wedge E_{i,k} = p_k \wedge E_{i,\xi} = p_\xi \wedge E_{i,\tau} = |$
      inc $p^|$; inc $i$
    while $i < |E| \wedge E_{i,k} = p_k \wedge E_{i,\xi} = p_\xi \wedge E_{i,\tau} = +$
      inc $p^+$; inc $i$
    {now, found the next plane $p$ with $p^+$,$p^-$ and $p^|$...}
    $N_{P,k} = p^|$, $N_{R,k}$-=$p^|$, $N_{R,k}$-=$p^-$
    $(\mathcal{C}, p_{side}) = SAH(V, p, N_l, N_r, N_p)$
    if $\mathcal{C} < \hat{\mathcal{C}}$ then
      $(\hat{\mathcal{C}}, \hat{p}, \hat{p}_{side}) = (\mathcal{C}, p, p_{side})$
    $N_L$+=$p^+$, $N_L$+=$p^|$, $N_P = 0$
  return $\hat{p}$
```

4.3.1 Splicing and Merging to Maintain Sort Order

As this partitioning depends on a pre-sorted event list E, we now have to find a way of – given E and $\hat{p}$ – computing the E_L and E_R (for V_L and V_R) without having to sort those explicitly. Though we obviously have to sort the list *once* at the beginning, during recursion we cannot afford the sorting, thus now – after each $\hat{p}$ is found – have to perform the actual classification and building of the two children's sub-lists E_L and E_R without performing any sorting.

Fortunately, however, we can make several observations:

- We can iterate over T and E several times and still stay in $O(N)$, if the number of iterations is a constant.
- Classifying all triangles to be in T_L and/or T_R can be done in $O(N)$ (see Algorithm 6).
- Since E is sorted, any sub-list of E will be sorted as well.
- Two sorted lists of length $O(N)$ can be merged to a new sorted list in $O(N)$ using a single mergesort iteration.
- Triangles that are completely on one side of the plane will have the same events as in the current node (see Figure 1a).
- Triangles overlapping p generate events for both E_L and E_R. These triangles have to be re-clipped (see Figure 1), and thus generate *new* splits that have not been in E.
- For reasonable scenes [1], there will be (at most) $O(\sqrt{N})$ triangles overlapping p.

With these observations, we can now devise an algorithm for building the sorted E_L and E_R lists.

Step 1: Classification: After $\hat{p}$ is found, for each triangle we first have to determine whether it belongs to T_L, T_R, or both (by now, we know where to put T_P). For a triangle t to be in T_L only, it must either end left of or on the split plane (i.e., $\exists e = (t, \hat{p}_k, e_\xi, -) : e_\xi \leq \hat{p}_\xi$)); or it is planar and lies left of the plane (i.e., $\exists e = (t, \hat{p}_k, e_\xi, |) : e_\xi < \hat{p}_\xi$), or the triangle is in T_P ($\exists e = (t, \hat{p}_k, \hat{p}_\xi, |)$), and $\hat{p}_{side} = LEFT$. For the right side, the criteria are symmetric; triangles fulfilling neither of these conditions must be on both sides. This leads to a simple classification algorithm: We first conservatively mark each triangle as being on both sides, then iterate once over all events, and – if that event matches any of the classification criteria above – mark its associated triangle to be only on the respective side only (see algorithm 6).

Algorithm 6 Given E and $\hat{p}$, classify triangles to be either left of, right of, or overlapping $\hat{p}$ in a single sweep over E.

```
function ClassifyLeftRightBoth($T$, $E$, $\hat{p}$)
  for all $t \in T$
    $t_{side} = Both$;
  for all $e \in E$
    if $e_{type} = - \wedge e_k = \hat{p}_k \wedge e_\xi \leq \hat{p}_\xi$ then
      $t[e_t]_{side} = LeftOnly$
    else if $e_{type} = + \wedge e_k = \hat{p}_k \wedge e_\xi \geq \hat{p}_\xi$ then
      $t[e_t]_{side} = RightOnly$
    else if $e_{type} = | \wedge e_k = \hat{p}_k$ then
      if $(e_\xi < \hat{p}_\xi \vee (e_\xi = \hat{p}_\xi \wedge \hat{p}_{side} = LEFT))$ then
        $t[e_t]_{side} = LeftOnly$
      if $(e_\xi > \hat{p}_\xi \vee (e_\xi = \hat{p}_\xi \wedge \hat{p}_{side} = RIGHT))$ then
        $t[e_t]_{side} = RightOnly$
```

Step 2: Splicing E into E_{LO} and E_{RO}: Triangles that do not overlap $\hat{p}$ contribute their events to their own side, and none to the other. Having already classified all triangles, we iterate over E again, and "splice" it by putting all events corresponding to a "left only" triangle into E_{LO}, and all those for "right only" triangles into E_{RO}; events for "both sides" triangles get discarded. Both E_{LO} and E_{RO} are sub-lists of E, and thus automatically $<_E$-sorted.

Step 3: Generating new events E_{BL} and E_{BR} for triangles overlapping p: Those triangles that do overlap $\hat{p}$ contribute (new) events to both sides. We generate these by clipping t to V_L and V_R, respectively (also see Figure 1), and put the generated events to E_{BL} and E_{BR}, respectively. Since the clipping generates new events in unknown order, neither of these is sorted.

Step 4: Merging the four strains: The events for E_L and E_R are now each scattered over two separate lists, E_{LO} and E_{BL} for E_L, and E_{RO} and E_{BR} for E_R, respectively. These now have to be merged to E_L and E_R. To do this, we first sort E_{BL} and E_{BR}. Assuming that only $O(\sqrt{N})$ triangles overlap $\hat{p}$, sorting these two lists will cost $O(|E_{LO}| \log |E_{LO}|) = O(\sqrt{N} \log \sqrt{N}) \subset O(\sqrt{N} \times \sqrt{N}) = O(N)$. Since now all E_{LO}, E_{RO}, E_{BL}, and E_{BR} are sorted, we can easily merge them to E_L and E_R in $O(N)$. Both E_L and E_R are now sorted, and recursion can proceed. Before recursing, all temporary event lists, and in particular the input event list, can be freed. Freeing that memory explicitly *before* recursing is often overlooked, but can greatly reduce the overall memory needs of the algorithm, in particular for complex models.

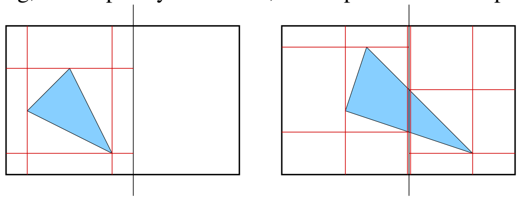
Figure 1: Triangles completely to one side of a splitting plane will maintain exactly the same set of events as without the split plane, all of which belong exclusively to the side the triangle is in. Triangles straddling the splitting plane have to be re-clipped to both sides, generating new potential split events for each side.

4.3.2 Complexity Analysis

Before we can call the recursive partitioning for the first time, we first have to build and sort the initial event list. This costs $O(N \log N)$, but has to be performed only *once*.

During recursion, in the algorithm just outlined all steps - finding the best plane, classifying triangles, splicing, new event generation, and list merging – are in the order of $O(N)$. Thus, even though there are several passes over T and E each, the total complexity of one partitioning is still $O(N)$, yielding a total complexity of

$$T(N) = N + 2T(\frac{N}{2}) = \cdots = N \log N.$$

This is once again the desired complexity of $O(N \log N)$, the same complexity as a Kaplan-style build, and the theoretical lower bound.

5 Experiments and Results

So far, we have only considered the theoretical aspects of the different build strategies, and have ignored all implementational aspects. In this section, we are going to evaluate an actual implementation of the proposed algorithms; in particular, we are going to use the implementation used in the OpenRT realtime ray tracing system [26]. As the naïve $O(N^2)$-build method becomes prohibitively expensive even for rather simple scenes (even the 802-triangle shirley 6 scene can take several minutes to build), we evaluate only the $O(N \log^2 N)$ and the $O(N \log N)$ methods.

As the implementation is used in an industrial application [10], both variants are coded in high-level C++ code and focus on production-quality robust and correctness standards. Thus, the code is poorly optimized, if at all: it uses templates, makes heavy use of STL data structures, and is highly parameterizable, to allow for setting all kinds of parameters via config files and command-line parameters, and for switching between different variants of the various formulae (for example, one can also specify non-linear leaf cost estimates, etc). All parameters have been set to the default values, which correspond to the exposition above.

If this flexibility were sacrificed by a highly optimized variant that would hard-code the default case significantly lower build times would surely be possible. Even so, such optimizations only change the implementation constants, but not the algorithms scalability in model complexity. As such, the respective $O(N \log^2 N)$-vs-$O(N \log N)$ comparisons, the (statistical) quality of the kd-trees, and the scalability in model complexity are sill valid.

For the comparisons between the $O(N \log^2 N)$ and $O(N \log N)$ algorithms, we have spend considerable care to make sure that both algorithms produce (nearly) the same kd-trees. Though both algorithms test the same planes, they do this in different order; this, together with the limited accuracy of floating point computations sometimes leads to both algorithms choosing a (slightly) different split. Overall, however, the trees computed by the two algorithms are nearly identical.

5.1 Test scenes

For our experiments, we have chosen to take test scenes from different domains. On one side, we have chosen typical standalone models like the Bunny (69k triangles), Armadillo (346k), Dragon (863k), Buddha (1.07m), Blade (1.76m), and ThaiStatue (10M) models from the Stanford Scanning Repository. Except for triangle count, these models are all very similar: all are scanned models, all have nearly equally-sized and mostly well-shaped triangles, and all are tesselations of smooth surfaces with sphere-like topology. The reason for including these models is that they are well-suited for scalability experiments, as they can be up- and downsampled in a meaningful way.

To get meaningful results on scalability with geometric complexity, we have to run our algorithms on models that have the same shape, structure, and surface complexity, but different triangle counts. This requires the same shape to be available in multiple tesselation densities, which we achieve by either downsampling via mesh-simplification (using the q-slim mesh simplification package), or via upsampling (via subdividing random triangles into four parts each). This process however works only for models that have roughly uniform tesselation to start with, as otherwise the mesh simplification software would simplify certain scene parts more than others, and thereby change the model structure.

Though good for resampling, the scanned models are well representative for scenes more commonly used in a ray tracer, like architectural models, or engineering data. Therefore, we have also added several freely available models that are commonly used in today's ray tracing papers: the ERW6 scene (802 triangles), the conference room (280k), soda hall (2.4m), and the power plant (12.5m).

5.2 Generated kd-tree statistics and build times

To facilitate easy and exact reproduction of our results, Table 1 gives some statistical properties of the kd-trees generated by our implementation. In particular, Table 1 reports for each model the total number of nodes, leaves, and non-empty leaves, as well as *expected* number of inner-node traversal steps, leaf visits, and triangle intersections. The expected number of traversals (E_T), leaf-visits (E_L), and triangle intersections (E_I) can be computed with the surface area metaphor explained for equation 3, yielding

$$\begin{aligned} E_T &= E[\#traversal\ steps] = \sum_{n \in nodes} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)}, \\ E_L &= E[\#leaves\ visited] = \sum_{n \in leaves} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)}, \text{ and} \\ E_I &= E[\#tris\ intersected] = \sum_{n \in leaves} N_n \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)}, \end{aligned}$$

where V_n is the spatial region associated to a kd-tree node n, and N_n is the number of triangles in a give leaf node n. Both $O(N \log N)$ and $O(N \log^2 N)$ produce (roughly) the same trees, so the data in Table 1 applies to both implementations.

model	tris	N_L	N_{NE}	N_{AT}	E_T	E_L	E_I	$\mathcal{C}(\mathcal{T})$
bunny	69k	338k	159k	2.57	52.3	14.7	7.1	926
armad.	346k	457k	201k	2.33	49.6	13.9	4.5	833
dragon	863k	1.39m	627k	2.56	76.5	20.8	8.4	1316
buddha	1.07m	1.85m	848k	2.61	82.7	22.4	9.7	1436
blade	1.76m	1.98m	926k	2.06	101.1	27.6	9.8	1713
thai	10m	36m	17m	2.80	74.4	20.3	7.7	1270
erw6	802	3.8k	2.7k	1.85	13.8	4.48	3.7	280
conf	280k	1.15m	679k	3.17	39.4	10.7	10.4	799
soda	2.4m	9.5m	6.3m	2.53	68.4	18.3	12.4	1275
PP	12.5m	41.8m	27m	2.89	28.3	7.97	7.5	574

Table 1: Statistical data for the generated kd-trees, for the original resolution of each model: The number of leaf nodes N_L, non-empty leaf nodes N_{NE}, the average number N_{AT} of triangles per non-emtpy leaf, the expected number of inner-node traversals $E_T = E[\#travsteps]$, leaf visits $E_L = E[\#leavesvisited]$, and ray-triangle intersections $E_I = E[\#trisintersected]$, for a random ray, where $E[X]$ denotes the expected value of event X. $\mathcal{C}(\mathcal{T})$ is the expected cost according to equation 3 ($\mathcal{K}_T = 15$ and $\mathcal{K}_I = 20$).

5.3 Build time over model size

Our theoretical analysis has made two assumptions about the behaviour of the algorithm(s): The number of stragglers in each partition step is small (in the order of $O(\sqrt{N})$), and the recursion splits each list into two equally-sized halves. To demonstrate that these assumptions in practice seem to apply—and to show how the algorithms scale in the number of primitives—we have followed the above-mentioned way of re-sampling the scanned models to a smooth scale of resolutions: using either up- or downsampling, we have generated 100 resolutions of each of the scanned models, from $40k$ triangles of the smallest resolution, to $4m$ triangles of the largest, in $40k$ steps. To abstract from secondary influences like memory performance, cache effects, and implementation constants, for each of these resolutions we have measured the total number of plane evaluations, which is a purely statistical measure. The results of these measurements is shown in Figure 2; as the difference in complexity stems from the $O(N \log^2 N)$ variant's additional sorting in each recursion step, this number is the same for both variants.

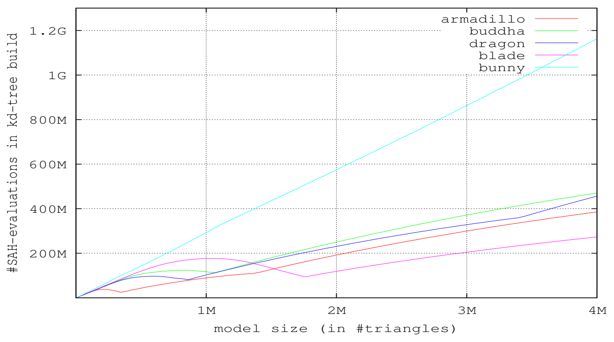

Figure 2: The number of evaluations of the "SAH" function for various resolutions of our example models.

For all tested models Figure 2 shows a noticeable peak for the down-sampled models, at around half the original size. Though we first suspected an error in implementation or analysis, this peak is, in fact, due to the simplification process used to generate the sub-sampled models: while the original meshes contain roughly equally sized and fat triangles, the simplification process creates more "slivery" triangles that have a higher chance of overlapping the split plane and generate more events.

Except for this effect, Figure 2 shows a nearly *linear* scaling in scene size. Though in fact rising with $O(N \log N)$, for as large N the impact of the log term is hardly noticeable: From $40k$ to $4m$, the $\log N$ factor contributes for a mere factor of $\frac{\log 4m}{\log 40k} = \frac{22}{15.5} \approx 1.5$, wheras the linear term makes up for two orders of magnitude.

For comparisons, it would also have made sense to depict the number of plane evaluations for the naïve $O(N^2)$ build. These however became infeasible even to compute for models with N in the range of four million; extrapolation would be possible, but would require a logarithmic scale (on which both graphs would look awkward), since for $N = 4m$ the difference between N^2 and $N \log N$ is more than five orders of magnitude.

5.4 Absolute build times

While Figure 2 has concentrated on a purely statistical measure, Table 2 then reports the absolute time for building the trees. As can be seen, the $O(N \log N)$ variant is consistently faster than the $O(N \log^2 N)$ variant, and outperforms it by a factor of $2 - -3.5$. Somewhat surprisingly, this ratio does not vary significantly for the various models, and in particular seems hardly increase with model complexity, if at all. One reason for that is that the two algorithms' complexities differ by a factor of $\log N$, which we have already observed to have a near-constant influence for such large models: Though because of that factor the speedup should rise with model complexity, even from the 69k bunny to the 10M thai statue the expected increase in speedup is a mere $\frac{\log_2 10M}{\log_2 69k} \approx \frac{2}{3}16 \approx 1.5$). In addition, both algorithms operate on huge amounts of data, and can therefore be expected to be memory bounds—not compute bound—which somewhat cancels any computational gains. Nevertheless the theoretically better algorithm manages to still consistently outperform the $O(N \log^2 N)$ variant, even though it touches was expected to have the higher implementation constants.

model	tris	build time $N \log^2 N$	build time $N \log N$	speedup
bunny	69k	6.7s	3.2s	2.1×
armadillo	346k	16s	5s	3.2×
dragon	863k	46s	16s	2.9×
buddha	1.07m	61s	21s	3×
blade	1.76m	74s	21s	3.5×
thai	10m	1,120s	430s	2.6×
erw6	804	60ms	30ms	2×
conf	282k	30.5s	15.0s	2×
soda	2.2m	228s	104s	2.2×
PP	12.7m	1,436s	559s	2.6×

Table 2: Absolute build times for the $O(N \log N)$ and $O(N \log^2 N)$ variants, for various test models. As expected, the $O(N \log N)$ variant consistently outperforms the $O(N \log^2 N)$ one by at least $2\times$, and up to $3\times$. Though we had expected this speedup to increase with model complexity, the logarithmic factor becomes nearly a constant for as large N as used in these experiments.

5.5 Relative cost of individual operations

To better understand where the two algorithms spend their time, we have also measured the time spent in the individual phases of the algorithms. In particular, for the $O(N \log N)$ variant we have measured the time spent in the initialization (including initial even list generation and intial event list sorting), as well as in the recursion, which splits into split evaluation (iterating over the event list, and evaluating the potential splits), re-clipping and resorting the straddlers, triangle classification, and list splicing and merging (out implementation interleaves both operations). For the $O(N \log^2 N)$ variant, the runtime splits into event generation (including clipping), sorting, split evaluation (list iteration and plane evaluation), and classification (including generation of the two triangle ID lists for recursion). The results of these measurements—for various of our test scenes—are given in Table 3. As can be see from this Table, the $O(N \log N)$ variant spends most of its time in list-operations (where it is most likely bandwidth-bound), while for the $O(N \log^2 N)$ re-sorting alone usually costs more than what it costs the $O(N \log N)$ variant to build the entire tree.

	bunny	buddha	thai	conf	powerplant
$O(N \log N)$ algorithm					
init time	(17)	(328)	(3610)	(75)	(5.2k)
evt-gen	< 1	9	93	3	111
evt-sort	14	290	3.2k	64	4.7k
rec.build	(375)	(2349)	(50k)	(1741)	(65k)
spliteval	84	542	11.6k	316	9.9k
classify	18	142	2.5k	78	3k
stragglers	126	386	13k	706	22k
list-ops	116	1111	19k	524	25k
total	**(395)**	**(2692)**	**(54k)**	**(1825)**	**(71k)**
$O(N \log^2 N)$ algorithm					
evt-gen	209	1045	27k	1001	36k
evt-sort	**278**	**3839**	**60k**	**1381**	**79k**
spliteval	71	427	8k	218	7.8k
classify	163	1176	21k	584	27k
total	(790)	(6856)	(125k)	(3434)	(159k)

Table 3: Time spent in the different phases of the two algorithms (in Linux timer "jiffies" of 10ms each). evt-gen: initial event list generation (including clipping if required); evt-sort: initial sorting; spliteval: split evaluation; classify: triangle classification; stragglers: reclipping and re-sorting of stragglers; list-ops: list splicing and merging. Data is given for bunny, buddha, thaistatue, conference, and powerplant. (Total higher than Table 2 due to measuring overhead).

6 Summary and Discussion

In this paper, we have surveyed and summarized today's knowledge on building good kd-trees for ray tracing. We then have described—and analyzed—three different algorithms for building such trees: A naïve $O(N^2)$ build, a more efficient $O(N \log^2 N)$ algorithm, and a variant with asymptotically optimal $O(N \log N)$ complexity.

None of these algorithms are *completely* new: In fact, the $O(N^2)$ and $O(N \log^2 N)$ algorithms are well known [14, 23], and even quite similar $O(N \log N)$ schemes have been used before: For example, for point data and range queries, similar $O(N \log N)$ algorithms are already known, both in computational geometry (see, e.g., Vaidya [24]), and also in photon mapping (see, e.g. [29]). Even in ray tracing, the $O(N \log N)$ algorithm is known to at least a few researchers for quite some time. For example, it already is at least hinted at in [6]. Nevertheless, this knowledge is not well documented, and so far has been known to but few experts in the field. Similarly, most of the details of how to actually build the three—what to consider and what to avoid—are all documented somewhere in various conferences, journals, and technical reports, but often in widely scattered form, using differnt notations and different benchmarks, and therefore again is known as lore among a selected group of experts only. As such, the main contribution of this paper is less in presenting any completely new algorithms, but rather in condensing the lore and knowledge in building good kd-trees, and in presenting, analyzing, and evaluating it in one concise, easily publication that is easily accessible to researchers interested in ray tracing. For example, the fact that the $O(N \log N)$ variant outperform the $O(N \log^2 N)$ one by (only) a factor of $2--3$ over a wide range of models may not present a significant algorithmic achievement, but knowing the options may pose in important piece of knowledge for a future ray tracing system's architect.

Apart from its direct application to building kd-trees, we believe the knowledge presented in this paper to be an important foundation for similar research also in non-directly related research, such as building good bounding volume hierarchies (to which the SAH also applies [28]), building such BVHs in an efficient manner, or in how to evaluate the cost of a ray-tracing data structure, and how to best cluster objects with a ray-tracing based cost function (see, e.g., [4]).

One issue worth mentioning is that the theoretical complexity outlined above strongly depends on the assumption of having a "well behaved" scene as one is likely to encounter in practice (see, e.g., [1]), as it is clearly possible to devise cases for which the above assumption of having—on average—less than $O(\sqrt{N})$ triangles overlapping the plane will be violated. Similarly, the complexity analysis depends on the assumption that the complexity of sorting is $O(N \log N)$, which is not necessarily true for our setting of bounded and "mostly sorted" sets of numbers. For these cases, radix sort-like algorithms exist that achieve asymptotically linear complexity [11, 19]. A binning strategy can also help in reducing the number of planes to be sorted [16]. If the sorting could be done in near-linear time, then even the theoretically $O(N \log^2 N)$ algorithm from Section 4.2 would show $O(N \log N)$ behavior.

Finally, in all our experiments we have seen that the influence of the $\log N$ term for as large N as interesting for relevant applications is but weak, and nearly a constant. Thus, in practice the relative performance of these two algorithms will mostly depend on their "constants", i.e., on how well they can be implemented.

Even at asymptotic optimal complexity, the cost for building kd-trees with these methods is still quite high, and certainly far from real-time except for trivially simple models. With the recent interest in ray tracing dynamic scenes, interest currently shifts to other data structure, like BVHs, or grids. Compared to kd-trees, these data structures have easier build- and update mechanisms that usually operate in $O(N)$. However, *if* the build time for realistic model sizes in practice is nearly linear in model complexity, then kd-trees can still be a viable option for future dynamic ray tracing systems, in particular if build time can be reduced by faster build methods and/or on-demand construction of the kd-tree [21]. In addition, much higher build performance can be achieved if certain compromises in the kd-tree's quality are being allowed: while we have only considered methods for building trees according to the best known quality standards, much higher performance can be achieved if, e.g., perfect splits are ignored, and if the cost function is sampled sparsely instead of finding *the* best local split (see, e.g., [15, 8]).

Summarily, we have shown that a viable algorithm with $O(N \log N)$ complexity exists, and that this algorithm is both simple, stable, and elegant. The presented algorithm is already being used in a production renderer, and since its introduction there has impressed through its robustness, in particular for numerically challenging cases for which several of its preceding, ad-hoc implementations had failed. The algorithm has been used extensively in many different scenes, including as large scenes as the 350 million triangle Boeing data set, for which an $O(N^2)$ approach is infeasible.

A specially optimized implementation of the presented algorithm – and which, amongst others, ignores perfect splits and only operates on the AABBs – is now also being used in a two-level approach to dynamic scenes [27] in the OpenRT system [26]. Though not originally designed for real-time rebuilds, at least for several hundred to a few thousand objects the $O(N \log N)$ SAH algorithm allows interactive rebuilds, while at the same time enabling superior ray tracing performance than its (non-SAH based) predecessor.

Acknowledgements

The authors would like to thank Alexander Reshetov for the insightful discussions about the kd-tree optimizations and construction methods used in the Intel System, and for sharing his actual construction code in the past. Carsten Benthin has provided help and support in implementation, bug-fixing, and experimentation, and has also coded several high-performance variants of this algorithm. Peter Shirley and Solomon Boulos have provided invaluable help in discussing special cases, geometric probability, cost estimates, and complexity issues. Also, we would like to thank Martin Devera for early discussions and implementations of kd-tree construction algorithms. This research work has been partially supported by MŠMT under research program MSM 6840770014, as well by the U.S. Department of Energy through grant W-7405-ENG-48.

References

[1] Mark T. de Berg, Matthew J. Katz, A. Frank van der Stappen, and J. Vleugels. Realistic Input Models for Geometric Algorithms. *Algorithmica*, 34(1):8197, 2002.

[2] Andrew Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989. ISBN 0-12286-160-4.

[3] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[4] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray Tracing Animated Scenes using Motion Decomposition. In *Proceedings of Eurographics*, 2006. (to appear).

[5] Eric Haines, editor. *Ray Tracing News*, 1987–2005. http://www.acm.org/tog/resources/RTNews/html/.

[6] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, 2001.

[7] Vlastimil Havran and Jirí Bittner. On Improving Kd Tree for Ray Shooting. In *Proceedings of WSCG*, pages 209–216, 2002.

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

[9] James T. Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon*, 2002.

[10] inView 1.4 Product Description. http://www.intrace.com/.

[11] Donald E. Knuth. *The Art of Computer Programming, Volumes 1-3*. Addison-Wesley, 1998.

[12] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[13] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[14] Matt Pharr and Greg Humphreys. *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman, 2004.

[15] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[16] Alexander Reshetov. On building good KD-Trees in the Intel Multi-Level Ray Tracing System. personal communication, 2005.

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

[18] Luis Santalo. *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002. ISBN: 0521523443.

[19] Robert Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Addison Wesley, 1998. (3rd Ed.).

[20] Gordon Stoll. Part II: Achieving Real Time - Optimization Techniques. In *SIGGRAPH 2005 Course on Interactive Ray Tracing*, 2005.

[21] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report 06-21, University of Texas at Austin Dep. of Comp. Science, 2006.

[22] K. R. Subramanian. *A Search Structure based on K-d Trees for Efficient Ray Tracing*. PhD thesis, The University of Texas at Austin, December 1990.

[23] Laászló Szécsi. An Effective Implementation of the kd-Tree. In Jeff Lander, editor, *Graphics Programming Methods*, pages 315–326. Charles River Media, 2003.

[24] Pravin M. Vaidya. An O(N log N) Algorithm for the All-Nearest-Neighbors Problem. *Discrete and Computational Geometry*, (4):101–115, 1989.

[25] Carsten Wächter and Alexander Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering*, 2006.

[26] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[27] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[28] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, (conditionally accepted), 2006. Available as SCI Institute, University of Utah Tech.Rep. UUSCI-2006-023.

[29] Ingo Wald, Johannes Günther, and Philipp Slusallek. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum*, 22(3):595–603, 2004.

[30] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).