SCI INSTITUTE
TECHNICAL REPORT

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

Ingo Wald, Vlastimil Havran[†]

UUSCI-2006-009

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

February 21, 2006

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 logN), the theoretical lower bound.

[†] Vlastimil Havran from MPI Informatik, Saarbrücken, Germany

THE UNIVERSITY OF UTAH

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

Ingo Wald
SCI Institute, University of Utah

Vlastimil Havran
MPI Informatik, Saarbrücken

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 of research, and a large variety of different schemes for acceleration ray tracing have been proposed. This includes Octrees, Bounding Volume Hierarchies (BVHs), different variants of grids, BSP trees, kd-trees, etc. (see, e.g., [Gla89, Hav01]).

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 [WSBW01, Wal04] and frustum traversal [RSH05] kd-trees are increasingly believed to be the "best known method" for fast ray tracing [Sto05]. 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) [MB90]. 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.

While we do not (yet) target interactive kd-tree building, we present a efficient, simple, and practical method for quickly building good kd-trees, which we believe have a wide range of applications.

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 ∈ T | (t ∩ V_L) ≠ ∅
  T_R = {t ∈ T | (t ∩ V_R) ≠ ∅
  return new node(p, RecBuild(T_L, V_L), RecBuild(T_R, V_R))

function BuildKDTree(triangles[] T) returns root node
  V = 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 [Wal04]. 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 [Sto05]).

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 $\mathcal{K}_{maxDepth}$:

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

[†] KD-Trees had originally been introduced as "KDB trees" [GG98], but in graphics are commonly called kd-trees.

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., [Hav01, Hai05]. 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 [GS87, MB90] [Sub90, Hav01] 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 $N\mathcal{K}_I$.

Using these assumptions then allows for expressing the cost of a given configuration: For uniform distributed lines and convex voxels, geometric probability theory [San02] tells us that for a ray known to hit a voxel V the conditional probability $\mathcal{P}$ of also hitting a sub-voxel $V_{sub} \subset V$ is

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

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,

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

3.1. Local Greedy SAH Heuristic

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

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

where $V_\mathcal{S}$ 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.

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,

$$\mathcal{C}_V(p) \approx \mathcal{K}_T + \mathcal{P}_{[V_L|V]}|T_L|\mathcal{K}_I + \mathcal{P}_{[V_R|V]}|T_R|\mathcal{K}_I \quad (4)$$

$$= \mathcal{K}_T + \mathcal{K}_I \left(\frac{\mathcal{SA}(V_L)}{\mathcal{SA}(V)}|T_L| + \frac{\mathcal{SA}(V_R)}{\mathcal{SA}(V)}|T_R| \right) \quad (5)$$

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.,

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

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. Several modifications have been proposed to compensate for this [Mac88, Hav01], but the impact of these improvements has usually been quite limited; indicating once more that the local greedy SAH is reasonably accurate.

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); memory, cache, or CPU-specific effects (SIMD) are gravely neglected; etc.

Nevertheless, in practice the basic SAH – local greedy plane selection and automatic termination criterion – often works best, and few 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

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

If the ray tracer supports "termination objects" [Hav01, RSH05], 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 remove the probability of the automatic termination criterion 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 [Hav01, Res05]. 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 [Hav01]. 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 [HKRS02, Hav01]). As this is significantly more accurate then the AABB, the candidates such produced are also often called "perfect splits" ‡. 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

‡ In [HB02], this technique is reported to give an average speedup of 9%, and up to 35% in certain cases.

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,

$$T_L = \{t \in T | Area(t \cap (V_l \setminus p)) > 0\} \quad (8)$$
$$T_R = \{t \in T | Area(t \cap (V_r \setminus p)) > 0\} \quad (9)$$
$$T_P = \{t \in T | Area(t \cap p) > 0\}. \quad (10)$$

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).

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 $t \in T$
    $(\hat{\mathcal{C}}, \hat{p}_{side}) = (\infty, \emptyset)$ {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 $(\hat{p}, T_l, T_r \cup T_p)$

Unfortunately, classifying N triangles costs $O(N)$, which, when calling it for $|C| \in O(N)$ potential split planes

amounts to a cost of $O(N^2)$ in each partitioning step. During the recursion, this $O(N^2)$ partitioning cost amounts to

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

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., [PH04, Szé03], the latter even including source code). Though this algorithm is sufficiently describe in these publications, we will 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 then 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 have been generated, we sort this event list E by ascending plane position, and such that for equal plane position the end events precede the planar events, which themselves precede the 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).

This algorithm initializes N_L^0,N_R^0, and N_P^0 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 $SAH(\ldots)$, 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

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 $i = 0; i < |E|;$
      $p = E_{i,p}$, $p^+ = p^- = p^| = 0$
      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 $(\hat{p}, \hat{p}_{side})$

by the cost for sorting, which is $O(N \log N)$. The accumulated cost during recursion then becomes

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

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^+)

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 $\hat{p}$

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.

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 [dKvV02], 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.

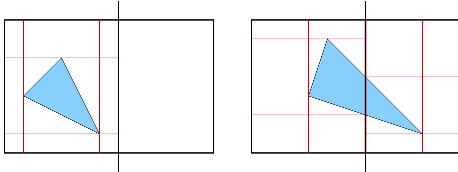

Figure 1: *Triangles completely to one side of a splitting plane will maintain exactly the same set of events as without the split plane, all of which belong exclusively to the side the triangle is in. Triangles straddling the splitting plane have to be re-clipped to both sides, generating new potential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

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, p̂)
  for all t ∈ T
    t_side = Both;
  for all e ∈ E
    if e_type = − ∧ e_k = p̂_k ∧ e_ξ ≤ p̂_ξ then
      t[e_t]_side = LeftOnly
    else if e_type = + ∧ e_k = p̂_k ∧ e_ξ ≥ p̂_ξ then
      t[e_t]_side = RightOnly
    else if e_type = | ∧ e_k = p̂_k then
      if (e_ξ < p̂_ξ ∨ (e_ξ = p̂_ξ ∧ p̂_side = LEFT)) then
        t[e_t]_side = LeftOnly
      if (e_ξ > p̂_ξ ∨ (e_ξ = p̂_ξ ∧ p̂_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.

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.

4.3.3. Experimental Validation

To validate our theoretical analysis of the $O(N \log N)$ complexity, we have performed extensive experiments, in which the number of SAH evaluations for different models have been measured. To abstract from the model's shape, we have taken several models, and generated multiple resolutions of each via up- and down-sampling: Down-sampling has been done via QSlim [Gar99]; up-sampling via randomly splitting triangles into four sub-triangles.

In particular for architectural or CAD-style models, mesh simplification is a non-trivial task, and tends to change the structure of the model (in particular, smaller triangles disappear faster than large ones). Thus, we have decided to not include such models, but have only included scanned meshes from the Stanford Model Repository. In particular, we have used the bunny, buddha, blade, armadillo, and dragon models, which cover a wide range of original model complexity (see Table 1). For each of these models, we have generated 100 reduced resolutions (in 1% steps), and have upsampled each to up to 4 million triangles. For each resolution, we have counted the number of SAH evaluations (see Figure 2).

model	#tris	exp.cost	$\#\mathcal{C}_V$ eval.'s	time (sec)
bunny	69k	82.3	19.3 M	4.8 sec
armadillo	346k	73.5	24.0 M	8.0 sec
dragon	863k	116.5	81.0 M	23.9 sec
buddha	1.07M	127.2	111.0 M	32.2 sec
blade	1.76M	151.3	94.0 M	33.7 sec
thaiStatue	10M	112.3	2,602.4 M	61.0 sec

Table 1: *Expected global cost (according to eq. 3, using $\mathcal{K}_I = 1.5, \mathcal{K}_T = 1$), number of plane evaluations to build tree, and absolute build time, for the original resolution of each of our test models.*

To give a rough impression on absolute timings, Table 1 also gives the original size for each model, as well as the absolute time to build the kd-tree on a 2.6 GHz Opteron desktop PC with 6 GB RAM. To allow for easily reproducing our results, Table 1 also gives the number of split candidates evaluated (i.e., the number of evaluations of the `SAH` function), as well as the expected global cost according to equation 3 (using $\mathcal{K}_I = 1.5$ and $\mathcal{K}_T = 1$).

In addition, Table 2 gives some statistical data on the kd-trees generated for each of the basic models, such as 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, leaf-visits, and triangle intersections have been computed with the surface area metaphor explained for equation 3. In particular, the expected number of traversal steps E_T, expected number of visited leaves E_L, and expected number of triangle intersections E_T are

$$E_T = E[\#traversal\ steps] = \sum_{n \in nodes} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)},$$

$$E_L = E[\#leaves\ visited] = \sum_{n \in leaves} \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)}, \text{ and}$$

$$E_I = E[\#tris\ intersected] = \sum_{n \in leaves} N_n \frac{\mathcal{SA}(V_n)}{\mathcal{SA}(V_S)},$$

where V_n is the spatial region associated to a kd-tree node n, and N_n is the number of trianlges in a give leaf node n.

For all tested models Figure 2 show a noticeable peak for the down-sampled models, at around half the original model size. Though we first suspected an error in our implementation, it turns out that this peak is due to the simplification process used to generate the sub-sampled models: While the original meshes contains roughly equally sized

model	N_L	N_{NE}	N_{AT}	E_T	E_L	E_I
bunny	349k	183k	2.50	52.3	14.7	7.1
armadillo	471k	274k	2.36	49.7	13.9	4.4
dragon	1.41M	812k	2.64	76.7	20.8	8.4
buddha	1.9M	1.07M	2.67	82.9	22.5	9.6
blade	1.98M	1.1M	2.16	101.1	27.6	9.9
thaiStatue	36M	19.8M	2.85	67.7	18.4	7.5

Table 2: *Statistical data describing our generated kd-trees, for the original resolution of each model: The number of leaf nodes N_L (total nodes are $2N_L - 1$), non-empty leaf nodes N_{NE}, as well as 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.*

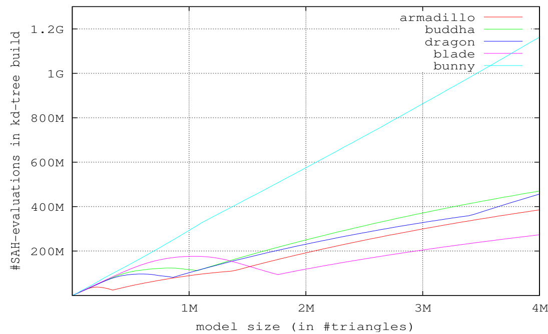

Figure 2: *The number of evaluations of the "SAH" function for various resolutions of our example models.*

and "fat" triangles (according to [dKvV02]), the simplification process automatically generates more "slivery" triangles – which have a higher chance of overlapping the split plane, and thus generate more events.

Except for this effect, Figure 2 nicely validates the expected $O(N \log N)$ behavior. Though in fact rising with $O(N \log N)$, for as large N the impact of the $\log N$ factor is hardly noticeable in the graphs. Note that the $O(N \log^2 N)$ algorithm would show exactly the same number of plane evaluations, as its higher complexity is only due to the rising cost for sorting. For comparisons, we would have liked to also show the graphs for the $O(N^2)$ build. These however became infeasible to compute for models with N in the range of four million – for which the difference between N^2 and $N \log N$ is roughly five orders of magnitude.

5. Summary and Discussion

In this paper, we have first summarized today's knowledge on building good kd-trees for ray tracing. Based on that, we 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 [PH04, Szé03], 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 [Vai89]), and also in photon mapping (see, e.g. [WGS04]). Even in ray tracing, the algorithm is known to at least a few researchers for quite some time. For example, it already is at least hinted at in [Hav01]. Still, we believe this to be the first time that this algorithm has been fully and in detail described – and theoretically analyzed – with all its peculiarities for triangular data.

Another 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., the definitions and discussion in [dKvV02]), as it is clearly possible to devise special cases for which the above assumption of – on average – having 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 [Knu98, Sed98]. A binning strategy can also help in reducing the number of planes to be sorted [Res05]. If - using any of these techniques - 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, even for as large N as used in our experiments, the difference between $N \log^2 N$ and $N \log N$ is only about an order of magnitude. 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 with all these improvements in asymptotic 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. In this paper, we have not considered low-level improvements in faster building, and have totally neglected constants factors.

Nevertheless, 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 in the spirit of [WBS03]. 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. In particular in light of the rising model complexity seen today, together with the more wide-spread use of kd-tree based ray tracers, we believe the proposed algorithm to be an interesting contribution towards making ray tracing more efficient, and more practical.

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.

References

[dKvV02] DE BERG M. T., KATZ M. J., VAN DER STAPPEN A. F., VLEUGELS J.: Realistic Input Models for Geometric Algorithms. *Algorithmica 34*, 1 (2002), 8197.

[Gar99] GARLAND M.: QSlim 2.0, 1999. University of Illinois at Urbana-Champaign,UIUC Computer Graphics Lab.

[GG98] GAEDE V., GÜNTHER O.: Multidimensional access methods. *ACM Computing Surveys 30*, 2 (1998), 170–231.

[Gla89] GLASSNER A.: *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[Hai05] HAINES E. (Ed.):. *Ray Tracing News* (1987–2005). http://www.acm.org/tog/resources/RTNews/html/.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[HB02] HAVRAN V., BITTNER J.: On Improving Kd Tree for Ray Shooting. In *Proceedings of WSCG* (2002), pp. 209–216.

[HKRS02] HURLEY J. T., KAPUSTIN A., RESHETOV A., SOUPIKOV A.: Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon* (2002). Available from http://www.graphicon.ru/-2002/papers.html.

[Knu98] KNUTH D. E.: *The Art of Computer Programming, Volumes 1-3*. Addison-Wesley, 1998.

[Mac88] MACDONALD D.: *Space Subdivision Algorithms for Ray Tracing*. M.sc. thesis, Department of Computer Science, University of Waterloo, 1988.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65.

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman, 2004.

[Res05] RESHETOV A.: On building good KD-Trees in the Intel Multi-Level Ray Tracing System. personal communication, 2005.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH).

[San02] SANTALO L.: *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002. ISBN: 0521523443.

[Sed98] SEDGEWICK R.: *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Addison=Wesley, 1998. (3rd Edition).

[Sto05] STOLL G.: Part II: Achieving Real Time - Optimization Techniques. In *SIGGRAPH 2005 Course on Interactive Ray Tracing* (2005).

[Sub90] SUBRAMANIAN K. R.: *A Search Structure based on kd-Trees for Efficient Ray Tracing*. PhD thesis, University of Texas at Austin, 1990.

[Szé03] SZÉCSI L.: *"An Effective Implementation of the kd-Tree"*. Charles River Media, 2003.

[Vai89] VAIDYA P. M.: An O(N log N) Algorithm for the All-Nearest-Neighbors Problem. *Discrete and Computational Geometry*, 4 (1989), 101–115.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003).

[WGS04] WALD I., GÜNTHER J., SLUSALLEK P.: Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum 22*, 3 (2004), 595–603.

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics).