# Benchmarking a K-D Tree Construction Algorithm for GPU

Cuebeom Choi, Ziwen Ye

November 2, 2020

**Abstract**

*In this report, we discuss our implementation and benchmarked results of a GPU based k-d tree construction algorithm based on [1]. Our implementation achieves an average of* $5.17\times$ *speedup over a reference serial implementation. We discuss our implementation approach, the data structures used, and analyze our benchmarked performance.*

## I. Background

Ray tracing is a commonly used method to render photo-realistic texture in graphics applications, which heavily rely on a spatial data structure to accelerate ray-primitive intersections. Among the various space partitioning algorithms including grid, octree, bounding volume hierarchies (BVH) and k-d trees, k-d trees are widely used in many commercial products. [2] Past work has shown that k-d tree built from a surface area heuristic (SAH) has superior performance over other criteria, but building a SAH k-d tree is usually expensive [2], [3].

As CPU and GPU are evolving into incredibly parallel computing units, it's possible to have GPU based parallel k-d tree construction algorithms that can be applied in real-time ray tracing. In a sequential SAH k-d tree construction, the sequential algorithm builds a k-d tree in a depth first order by evaluating the SAH cost and accordingly splitting the nodes. To determine the best partition, it computes SAH costs for all possible partition planes [2]. In a sequential version, this is generally done by sweeping the sorted list. But in a parallel version, this can be achieved by data-parallel primitives scan or segment scan. After finding the best splitting planes, we subdivide the nodes and distribute the associated triangles into their child nodes. Specifically, we split triangles that cross the splitting plane and shrink their AABBs (axis aligned bounding box). This computation benefits from parallel sorting.

## II. k-d tree

### A. Algorithm overview

k-d trees are well established space-partitioning data structure for organizing points in a k-dimensional space. As an acceleration structure, it's widely used in various graphics applications, including ray-triangle intersection tests in ray tracing, nearest photo queries in photo mapping, and nearest neighbor search in point cloud modeling and particle-based fluid simulation [1]. A k-d tree is essentially a binary tree in which every leaf node stores the following information: the start and end index of the triangles it contains in the sorted triangle vector, the number of triangles, and the bounding box of all triangle's AABB. In this report, we focus on a real-time parallel k-d tree construction algorithm for graphics application.

### B. Serial Implementation

Given mesh objects consisted of triangles meshes, our algorithm takes in an array of triangle meshes and outputs a binary tree where each child leaf node contains information mentioned above. Our k-d tree construction scheme follows a conventional algorithm [7] which builds a k-d tree in a greedy, top down manner by recursively splitting the current node into two sub nodes as follows:

1) Evaluate the SAH cost for all splitting plane candidates in k dimensional space.
2) Pick the optimal candidate with the least cost and split the node into two child nodes.
3) Sort triangles in the node and distribute them to the child nodes [1].

The SAH cost function is defined as

$$SAH(x) = C_{ts} + \frac{C_L(x)A_L(x)}{A} + \frac{C_R(x)A_R(x)}{A} \qquad (1)$$

where $C_{ts}$ is the constant cost of traversing the node itself, $C_L(x)$ is the cost of the left child given a split position x, and $C_R(x)$ is the cost of the right child given the same split position. $A_L(x)$ and $A_R(x)$ are the surface areas of the left and right child. A is defined as the surface area of the node. Note that $C_L(x)$ and $C_R(x)$ can only be evaluated after the entire tree is built. Rather than seeking a globally optimal solution, existing algorithms use a locally greedy approximation which treats the child nodes as leaf nodes [1]. In this scenario, $C_L(x)$ and $C_R(x)$ is the same as number of primitives contained in left and right node.

### C. Computation Cost and Parallelization

The most expensive computation in the process of constructing a k-d tree is to compute SAH cost when finding the best partitioning plane. The optimal sequential SAH k-d tree construction algorithm as mentioned above performs linear-time sorted-order coordinates sweep to compute the SAH to find the bet partition plane. This achieves $O(n \log n)$ efficiency

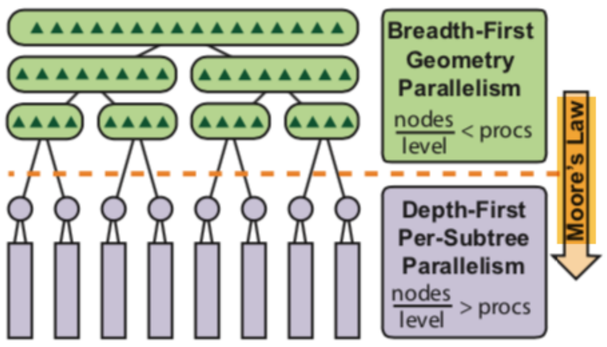
Breadth-First Geometry Parallelism
nodes/level < procs
Moore's Law
Depth-First Per-Subtree Parallelism
nodes/level > procs

**Figure 1:** Parallel k-d tree patterns [3].

because it takes three sweeps over each of the $O(n)$ splitting candidates for each of the $O(\log n)$ level of the k-d tree. [9]

We first take a look at the parallel k-d tree patterns as shown in Figure 1. For nodes at top levels of the tree where the number of nodes is less than that of the cores, multiple cores cooperate and lead to a breadth-first stream process that organizes triangles into nodes at the current level. After the number of nodes exceed the number of cores, each subnode tree can be processed by each core separately. Various past work have addressed the problem of parallel k-d tree construction. Some work uses a single thread to create the top levels of the tree until each node can be mapped to each core in a multi-core system. [9] However, this can lead to load imbalance since at top levels of the tree, each node may have a large difference in the number of triangles it contains. To address load imbalance, Shevstov el al. [10] proposed a shared memory architecture with multi-core CPUs. The algorithm first divides space into several balanced sub-regions and then builds sub-trees in each region in parallel and DFS order. However, in this report, we seek for a parallel model with GPU. The above algorithms do not map well into GPU architecture because number of threads in GPU is much greater than that in CPU. The other problem with Shevstov's algorithm is that it degrades the construction quality when rendering since it does not follow the SAH heuristic.

To maximally exploit the GPU's streaming architecture for parallelizing k-d tree construction while maintaining relatively high construction resolution, we adapted algorithms proposed by Zhou et al. [1]. We first build the tree in a BFS order. By following a BFS construction, we take advantages of the parallelism in GPU because the number of threads doubles from the preceding step at each tree level. To further exploit the large scale parallelism in GPU, we parallelize over geometric primitives instead of nodes at upper levels of the tree. This method is more efficient since the number of nodes in upper levels is relatively small. Moreover, the workload may be imbalanced among nodes since the number of primitives may differ significantly. To further reduce computational cost for node splitting, we set a threshold and distinguish nodes between large and small nodes based on the number of triangles a node contains [1]. For large nodes at upper tree level, we adopt two inexpensive cost estimation methods: median splitting and "empty space maximizing"[8] which will be explained later in details. For small nodes near bottom of the tree, we want to have a more exact evaluation of the cost to maintain relatively high reconstruction resolution, and so we store geometric primitives in nodes as bit masks and use bitwise operations to evaluate costs and sort primitives.

### D. *Data Dependency*

This algorithm is highly parallel across all levels of the tree. In the upper level of the tree, since we are parallelizing over primitives, each thread is independent, so no inter-communication is required. For small nodes, across nodes, there may be a global read to the same triangle when the splitting plane intersects a triangle, but no global write happens.

## III. Implementation

Our algorithm follows multiple stages in the construction of the tree. After initializing the root node, the algorithm follows two processes; first, it processes large nodes, and then it goes through and processes the small nodes. After the large and small node processes complete, we reorganize the result of the earlier processes and compute the final tree. We now describe each stage in greater detail.

### A. *Initialization*

During initialization, we first convert the input file into a triangle soup. For our implementation, we decided to take `.obj` files as input for our 3D representation, and we convert this into an array of triangles. Using the array of triangles, we construct the root node which we will split in later stages, and compute the AABBs of every triangle in parallel on the GPU. As mentioned in class, our array of triangles (and most arrays in our implementation) follow the structure of arrays principle, as it is more efficient to hold a structure of arrays over an array of structures.

### B. *Large node stage*

Next, once we have computed the AABBs of every triangle, we then perform the large node stage. This stage takes all large nodes (defined as all nodes with $\geq T$ triangles) and continuously reduces and splits them until there are no more large nodes left over multiple rounds. To perform this in an efficient manner, we implemented a *chunklist* structure and a multi-step reduction pass as described by Zhou et al. that allows us to exploit parallelism [1]. To start, we first initialize the chunklist at the beginning of every pass by separating triangles in every large node in

**Algorithm 1:** Multi-step reduction pass

```
Function ReducePass (
    in: chunklist, operation op
    out: list of reduced values red_vals)
begin
    parallel for $c \in chunklist$ do
        reduce in parallel over triangles in $c$
    parallel for $n \in nodelist$ do
        segmented reduce in parallel over
          chunks that correspond to $n$
end
```

**Algorithm 2:** Large node stage

```
Function ProcessLarge (
    in: root_node, an initial node w/ all triangles
    out: small_nodes, list of small nodes,
        node_list, a list of list of large nodes)
begin
    active_nodes ← [root_node]
    new_nodes ← new list

    while !active_nodes.empty() do
        node_list.append(active_nodes)
        chunk active_nodes into chunklist

        // calculate boundaries
        ReducePass(chunklist, min, min_bound)
        ReducePass(chunklist, max, max_bound)

        // split and separate triangles
        parallel for $n \in active\_nodes$ do
            for sides of $n$ do
                if side has $C_e$ empty space do
                    cut off empty space
            split node into $n_\ell$, $n_r$ along median
            add $n_\ell$, $n_r$ to new_nodes
        parallel for $k \in chunklist$ do
            parallel for triangle $t \in k$ do
                place and clip $t$ into correct child
                  node $n_\ell$, $n_r$
        // count triangles
        ReducePass(chunklist, +, num_triangles)
        parallel for $n \in new\_nodes$ do
            if $n$.numTriangles $< T$ do
                new_nodes.remove(n)
                small_nodes.add(n)
        active_nodes = new_nodes
    end
end
```

groups of size $N$. Using this structure, the goal of the multi-step reduction pass (outlined in Algorithm 1) is to efficiently a reduction over all triangles in a node. To do so, we first perform a reduction over every chunk individually, as all triangles in a chunk are guaranteed to be in the same node. Then, we perform a segmented reduce on this output. As the name implies, segmented reduce performs a reduction over specified segments in the input sequence; for our purposes, we segment by node, allowing us to efficiently reduce triangles over a node. This multi-step reduction pass is preferable to a bigger, single-step segmented reduce as Sengupta et al. found that a large segmented reduce is about 3× slower than the multi-step reduction [4].

Now we discuss the overall large node stage. For every pass, we are given an input of active large nodes that we will split. As mentioned earlier, we first chunk the list of triangles. Using the multi-step reduction pass, we first reduce and compute the bounding box for each node. To do so, we simply use the min and max operators over each of the triangle's AABBs to compute a node's bounding box. Next, we go over each of the computed bounding boxes in parallel and perform a pruning step. For each side of a node's bounding box, we check for empty space; if a side has more than some ratio $R_e$ that is empty along that axis, we cut off the empty space. After having done that, we then split the node along the middle of the longest axis. Using this split information, we sort and separate the triangles from the node into the two child nodes that are formed with this split, clipping any triangles that are in both nodes. Finally, to finish, we use the multi-step reduction pass to now count the number of triangles in each node and we filter out the nodes that fall under the large node threshold. For this reduction pass, we simply add one for each of the triangles we reduce over. An overview of this stage is seen in Algorithm 2.

## C. Small node stage

After the large node stage is complete, we now process all of the small nodes. Unlike the large stage, most of the parallelization here is done over nodes. We note that because we limit the number of triangles in a small node to $T$, the workload per node is relatively the same. Next, we proceed in two steps. First, we perform a preprocess step. In the preprocess step, we construct all possible splitting point candidates for each small node and compute the triangle sets for both sides of the possible splitting points in parallel. We perform this step to prepare for the next step, where we compute the SAH for every node and attempt to split. Similarly to the large node step, we continuously operate over a list of active nodes (which starts off as the list of all small nodes) until no nodes are active. We define an active node as one that has a splitting candidate $p$ such that the SAH is reduced. If we find that the SAH only increases with any potential split, this means that the current node should be a leaf.

As mentioned earlier in section 2.3, in order to achieve the SAH computation efficiently, we employ the use of bit masks and bitwise operations. At every pass, we go over all active nodes in parallel. For each active node, we take the bitset of the set of triangles in this node. Next, we look at the small root of this node. The small root is the uppermost small node ancestor in the tree. We use this small

**Algorithm 3:** Small node stage

```
Function ProcessSmall (
    in: small_nodes, a list of small nodes
    out: node_list, a list of list of all nodes)
begin
    active_nodes ← small_nodes
    new_nodes ← new list

    // preprocess
    parallel for $n \in small\_nodes$ do
        n.splitList ← all split candidates for n
        parallel for $s \in n$.splitList do
            s.left ← set on left of the split
            s.right ← set on right of the split

    // compute the SAHs and split
    while !active_nodes.empty() do
        node_list.append(active_nodes)
        parallel for $n \in active\_nodes$ do
            $t \leftarrow n$.triangleSet
            $r \leftarrow n$.smallRoot
            $SAH_0 \leftarrow |t|$
            for valid $s \in r$.splitList do
                $C_L \leftarrow |t \cap s.\text{left}|$
                $C_R \leftarrow |t \cap s.\text{right}|$
                compute areas $A_L$, $A_R$
                compute $SAH_s$ with equation (1)
            $s_{\text{opt}} \leftarrow \text{argmin}_s\ SAH_s$
        if $SAH_{s_{\text{opt}}} \geq SAH_0$ then
            n is a leaf node
        else
            split n with $s_{\text{opt}}$
            appending new nodes to new_nodes
            sort triangles into new nodes
        end
        active_nodes = new_nodes
    end
end
```

root and go over its precomputed splitting points. For each of these splitting points, we know that we also precomputed the triangle sets for both sides of these splitting points. Using a bitwise AND, we can then compute the triangle sets for the split of the current active node using bitset of the triangles in this node and the bitsets of the triangle sets that split the small root. Using this triangle information and an easily computable area calculation, we can easily calculate the SAH of this split. From here, we can minimize over all splitting points and evaluate the minimal SAH, determining whether to split or not. The overview of this stage is covered in Algorithm 3.

## D. Traversal

Finally, we compute a final preorder traversal. Zhou et al. needed to compute the preorder traversal of the k-d tree because their ray-tracer implementation was stack-based, and thus required a specific ordering. Although we did not implement their ray-tracer, we decided to port over this preorder traversal as well to get an accurate point of comparison with their

**Algorithm 4:** The complete algorithm.

```
Function ConstructKDTree (
    in: triangles, a list of all triangles
    out: kd_tree, tree in preorder traversal)
begin
    node_list ← new list
    small_nodes ← new list
    root ← new node(triangles)

    ProcessLarge(root, small_nodes, node_list)
    ProcessSmall(small_nodes, node_list)
    kd_tree ← PreorderTraversal(node_list)
end
```

data. We will not discuss the preorder traversal in great detail; simply, their implementation performs two passes. First, it performs a bottom-up phase to compute the size of each subtree, allocates a proper tree, then performs a top-down phase to store pointers in a preorder format. This is done by examining each level in the tree in parallel. We note that the *node_list* that has been passed to both ProcessLarge and ProcessSmall keeps this dept level information since we append full levels of nodes to the list for each inner loop we perform in both stages.

## E. Overall algorithm and implementation detail

Using all the stages in the algorithm, our final algorithm simply performs the stages in order, as seen in Algorithm 4. For our implementation, we built off Choi et al.'s C++ implementation of a parallel CPU k-d tree construction algorithm. This code was readily available on Choi's Github, and was easily modifiable for our purposes. Using this baseline, we implemented the specific stages and processes using CUDA as well as the thrust library for many of the operations and helpful data structures. The thrust library provides many of the crucial operations in the multi-step reduction pass and also has helpful data structures (primarily, the `device_vector` data structure). As mentioned throughout the algorithm description, there is a lot of possible parallelization in the algorithm. Because nodes, triangles, and splits can all be done in parallel, all stages of the algorithm lend well to a CUDA implementation. For example, the chunked data structure lends well to thread blocks, as we can compute a reduce over thread blocks of size $N$. This allows this algorithm to take advantage of the GPU to compute k-d trees with great speed.

## IV. Results

### A. Test suite and methodology

For our test data, we decided to test on GHC machines with a serial and parallel CPU implementation from Choi as well as our parallel implementation. For the serial implementation, we tested using the CPU, while we ran our parallel results with NVIDIA

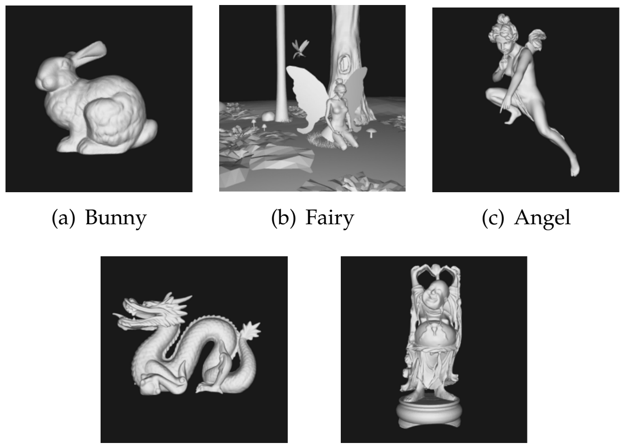

**Figure 2:** Images of the models in our test suite.

| Model: | Bunny | Fairy | Dragon | Happy | Angel |
|---|---|---|---|---|---|
| # Triangles | 69k | 172k | 203k | 293k | 474k |
| Serial (ms) | 73.8 | 177.0 | 216.4 | 317.6 | 538.5 |
| Parallel (ms) | 25.1 | 62.4 | 39.6 | 46.0 | 71.2 |
| **Speedup** | **2.95** | **2.85** | **5.55** | **6.9** | **7.58** |

**Table I:** K-D tree construction results for a serial implementation compared to our GPU implementation

GeForce RTX 2080 GPUs. We measured performance based on the K-D tree construction time, and we tested the results on mesh models consisting of 69k to 474k triangles. Due to time constraints, we did not measure ray-traversal time for the two construction results, so we could not measure frame rate and reconstruction resolution for our parallel implementation.

## *B. Benchmark*

We first compare our CUDA parallel K-D tree construction time with single thread CPU construction time. As shown in Table 1, we achieved a speed up ranging from 2.85x to 7.59x. As the number of triangles increases, the speedup of our parallel implementation also increases. We do see that the result of Fairy seems to be an outlier, as it has less speedup than Bunny despite possessing more triangles which should benefit from parallelization more. We speculate that this is because the distribution of geometric primitives in fairy are rather sparse. Looking at Figure 2.b, we observed that Fairy is an entire scene instead of an isolated object. This may make the BFS streaming process during the large node stage more time-consuming. We then plotted K-D tree construction time vs. number of triangles in the model for parallel implementation with data for Fairy removed. As we see in Figure 3, We have a near-linear relationship between the two variables. This was expected because the increase in model complexity and the file format lends to longer build times simply because of the increase in the problem size.

| Model: | Bunny | Fairy | Dragon | Happy | Angel |
|---|---|---|---|---|---|
| Init | 0.51 | 0.55 | 0.63 | 0.71 | 0.76 |
| Large Nodes | 72.8 | 79.1 | 70.0 | 74.1 | 70.0 |
| Small Nodes | 21.9 | 15.4 | 20.8 | 23.3 | 22.1 |
| Others | 4.82 | 4.98 | 8.57 | 1.88 | 7.12 |

**Table II:** Instrumentation results of different components in K-D tree construction

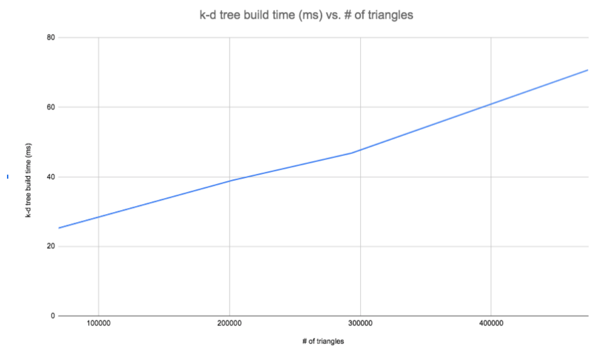

**Figure 3:** Plot of K-D tree construction vs. number of triangles in the model for parallel implementation

We also instrumented different components in our k-d tree construction implementation as shown in Table 2. We observed that nearly $70 - 80\%$ of our construction time is denoted to large node process despite the large parallelism we applied. Since we already substituted expensive SAH evaluation with inexpensive median splitting and "empty space maximizing", we speculate that the high cost comes from the reduction step. Lastly, we believe that our GPU implementation is more suitable for large k-d tree constructions than the multi-core CPU implementation. To test our choice, we compared our GPU implementation with Choi's in-place parallel algorithm running on 8 threads. As seen in Table 3, the building time is relatively the same for models with small number of triangles. But as the number of triangles increases, our GPU implementation has a much greater speedup over the parallel CPU algorithms. We believe that the overhead of the GPU implementation from launching kernels, transfering memory, etc. is a big factor for smaller models, resulting in negligible speedup for Bunny and Fairy, whereas the speedup for larger models is much better due to the complexity of the models.

## V. Reflection and future work

We decided to pick this particular implementation of k-d tree construction because we believed it was the right scope for us to implement for this project. Originally, we decided to implement the work done in [2] because we thought that the ideas in the paper were more novel and seemed more interesting to implement. However, because of both the COVID-19 situation and the vague, unclear language of the paper, we were unable to finish a complete implementation and decided to implement [1] instead.

| Model: | Bunny | Fairy | Dragon | Happy | Angel |
|---|---|---|---|---|---|
| # Triangles | 69k | 172k | 203k | 293k | 474k |
| CPU (ms) | 26.8 | 70.1 | 93.5 | 148.6 | 318.5 |
| GPU (ms) | 25.1 | 62.4 | 39.6 | 46.0 | 71.2 |
| **Speedup** | **1.07** | **1.12** | **2.36** | **3.23** | **4.47** |

**Table III:** K-D tree construction results for a parallel CPU implementation on 8 threads compared to our GPU implementation

Because of these issues, we were also unable to also complete the ray-tracer algorithm, which would have allowed us to benchmark and test the frame rate and reconstruction resolution. We wish that under better circumstances, we would be able to implement and test the ray-tracing algorithm as well.

In the future, we would like to incorporate the ideas from [2] into our GPU implementation here. Despite the unclear language of the paper, we believe that it has some interesting ideas that we could have ported over into this algorithm but could not have a chance to do. For example, we think that the parallel event sort is something that we could have ported, but we did not successfully complete this reach goal. Additionally, incorporating ideas of more recent works would also be quite interesting to do had the scope of this project been larger.

## VI. Division of work

We both did equal amounts of work on this project.

## References

[1] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5), December 2008.

[2] Zhefeng Wu, Fukai Zhao, and Xinguo Liu. Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, page 71–78, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, page 77–86, Goslar, DEU, 2010. Eurographics Association.

[4] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, page 97–106, Goslar, DEU, 2007. Eurographics Association.

[5] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 235–246, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 467–478, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[8] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[9] W. Hunt, W. R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 81–88, 2006.

[10] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, 2007.