High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Parallel SAH k-D Tree Construction

Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart

University of Illinois at Urbana-Champaign
sds@cs.illinois.edu

Abstract

The k-D tree is a well-studied acceleration data structure for ray tracing. It is used to organize primitives in a scene to allow efficient execution of intersection operations between rays and the primitives. The highest quality k-D tree can be obtained using greedy cost optimization based on a surface area heuristc (SAH). While the high quality enables very fast ray tracing times, a key drawback is that the k-D tree construction time remains prohibitively expensive. This cost is unreasonable for rendering dynamic scenes for future visual computing applications on emerging multicore systems. Much work has therefore been focused on faster parallel k-D tree construction performance at the expense of approximating or ignoring SAH computation, which produces k-D trees that degrade rendering time. In this paper, we present two new parallel algorithms for building precise SAH-optimized k-D trees, with different tradeoffs between the total work done and parallel scalability. The algorithms achieve up to 8× speedup on 32 cores, without degrading tree quality and rendering time, yielding the best reported speedups so far for precise-SAH k-D tree construction.

1. Introduction

We foresee an evolution of visual experiences into shared online visual simulations whose user-generated content (including self-scanned avatars) changes dynamically and unpredictably. Unlike modern videogames, which achieve lush visual effects through heavy precomputation of predefined content, the real-time rendering, meshing, and simulation of dynamic content will require the rapid construction and update of hierarchical spatial data structures. For example, these spatial data structures are well known rendering accelerators for both ray tracing [WHG84] and rasterization [GKM93], and form integral components of recent parallel real time ray tracers [WSS05, CHCH06, SCS*08, LP08, GDS*08]. However, existing parallel algorithms designed to rapidly build dynamic spatial hierarchies will soon face a serious roadblock as processor parallelism continues to grow.

The previous work summarized in Sec. 2 and the emergent pattern analyzed in Sec. 3 reveal that parallel hierarchy construction algorithms load balance well when the frontier of hierarchy nodes needing processing exceed the number of parallel processors, but struggle with the initial stages of construction when the hierarchy contains too few nodes. Some parallel approaches suffer reduced throughput at these initial levels [Ben06, PGSS06, HMS06], whereas others use alternative subdivision heuristics that can reduce hierarchy quality [SSK07,ZHWG08,LGS*09]. Fig. 1 shows that as processor parallelism continues to scale up, the number of initial steps in parallel hierarchy construction grows, and current subdivision heuristic sacrifices made to maintain throughput cause increasing degradation in tree quality and ultimately rendering rates.

This paper presents two new parallel algorithms for improving throughput when constructing these initial upper levels of a k-D tree. The first algorithm, “nested,” is a depth-first task parallelization of sequential k-D tree construction that nests geometry-level parallelism within the node-level parallelism for these upper level nodes. The second algorithm, “in-place,” builds the upper nodes of the hierarchy breadth-first, one level at a time, storing in each triangle the node(s) it belongs to at that level. This reduces geometry data movement and allows an entire level’s nodes to be computed across a single data parallel geometry stream.

These new algorithms regain throughput without sacrificing spatial hierarchy quality, as measured by rendering performance gains. They compute a precise surface area heuristic (SAH) that subdivides geometry into regions of small surface area that contain many triangles [GS87, MB90]. Spatial hierarchies formed by subdividing at the spatial me-

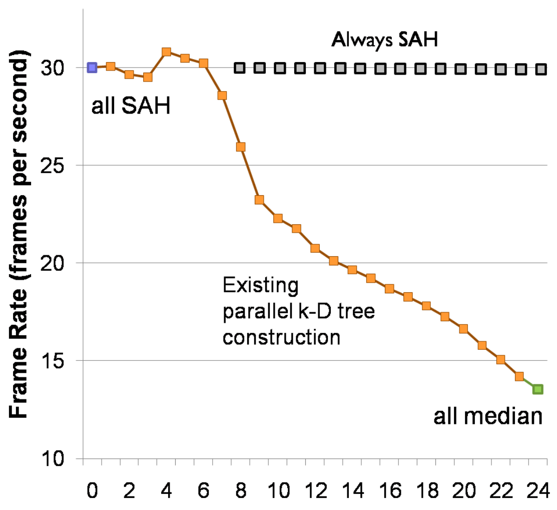

Figure 1: *Degradation in hierarchy quality using spatial median vs. precise-SAH to select splitting planes for the upper level nodes. The vertical axis indicates the rendering rate in normalized frames per second for ray tracing the fairy scene on the Dunnington machine described in Sec. 7 (the performance of the best configuration is normalized to a target of 30 fps). The horizontal axis indicates the depth at which current parallel kD-tree construction algorithms switch from using a spatial median to using SAH. This switch occurs when the depth approximately equals* $\log_2$ *number of processors. As the number of processors continue to double biannually, the hierarchies generated by existing parallel algorithms eventually degrades rendering performance, whereas the rendering rate remains constant for our (fully SAH) parallel k-D trees.*

dian (e.g., the octree) [ZHWG08] or at the object median (e.g., into children of approximately equal numbers of primitives) [SSK07] can be computed faster than SAH but the resulting hierarchies render slower than SAH hierarchies, as shown in Fig. 1 for the spatial median.

Hierarchy quality can be further justified by the relationship between hierarchy rendering time and construction time. Recent renderers that focus on real-time direct ray tracing of dynamic content currently experience about a 1:1 rendering-to-construction speed ratio. For these, some approaches justify a degraded hierarchy quality that increases the rendering time by a corresponding decrease in hierarchy construction time, and given a few processors, the upper-level nodes may not even incur a quality degradation. Such a relationship might continue as the triangle count grows but only to a ceiling level on the order of one REYES-micropolygon triangle per pixel since frame rates and display resolutions remain fairly constant. As processor parallelism nevertheless continues to grow, we will see increased global illumination Monte-Carlo effects and hundreds of rays per pixel which would cause the rendering-to-construction speed ratio to grow to 100:1 such that even a 1% degradation in rendering rate could not be tolerated by a hierarchy construction acceleration.

Our implementation is designed to measure the efficiency and throughput of the *parallelism* of our approach, as opposed to the raw *performance* of SAH k-D tree construction and rendering. For example, we compute SAH directly at the endpoints of triangle extents in each direction, and do not implement "binned SAH" approximations or "split clipping" triangle subdivision which would affect raw performance but their impact on scalability results from less work for binning [WBS07] and similar but greater dynamic growth in per-level triangles for split clipping. We believe our parallel construction algorithms are general enough to permit both enhancements in a production enviroment. We similarly focus on the construction of k-D trees, but believe our parallel algorithms can also be adapted to bounding volume hierarchies (BVHs). BVHs can be constructed and maintained more efficiently [WBS07, WIP08, LGS*09] but k-D trees better accelerate ray tracing [Hav00] which would make them the preferred choice for high rendering-to-construction speed ratio applications.

Sec. 7 examines the results of our two approaches on a 32-core shared-memory CPU platform for five input models, indicating scalability of these difficult upper levels up to depth 8. For these configurations, the algorithms achieve speedups of up to 8X, with in-place outperforming nested for two input models and vice versa for the other three. A deeper analysis of the scalability of the two algorithms reveals that while nested performs less work overall, in-place has better parallel scalability and is likely a better choice for future machines with larger core counts. To our knowledge, these results represent the first multicore speedups on the upper levels of k-D tree construction using precise SAH, and the best parallel approach for working with these levels in general.

2. Related Work

Wald and Havran [WH06] describe an optimal sequential $O(n \log n)$ SAH k-D tree construction algorithm that initially sorts the geometry bounding box extents in the three coordinate axes, peforms linear-time sorted-order coordinate sweeps to compute the SAH to find the best partitioning plane, and maintains this sorted order as the bounding boxes and their constituent geometries are moved and subdivided. We describe this algorithm in more detail in Section 4 and use it as our baseline state-of-the-art sequential algorithm. Our contribution is to develop a parallel approach that produces the same k-D tree as this sequential algorithm but at a much higher level of performance.

Some have accelerated SAH computation by approximation, replacing the initial $O(n \log n)$ sort with an $O(n)$ binned radix sort along each axis, and interpolating the SAH measured only between triangle bins [PGSS06, HMS06, SSK07] for both sequential and parallel acceleration. Even with a binned approximate sort, the k-D tree construction cost nevertheless remains $O(n \log n)$ since all n of the triangles are processed for each of the $\log n$ levels.

Many have worked on parallel SAH k-D tree construction. Several versions use a single thread to create the top levels of the tree until each subtree can be assigned to each core in a 2- or 4-core system [Ben06, PGSS06, HMS06], limiting 4-core speedup to only 2.5×.

Shevtsov et al. [SSK07] also implemented a 4-core parallel SAH k-D tree builder, but used a parallel triangle-count median instead of SAH to find splitting planes at the top levels of the tree, which degraded k-D tree quality by 15%. They did not report a construction time speedup, but they did report a 4-core speedup of 3.9 for a construction combined with rendering, which includes millions of k-D tree traversals. This algorithm was also used for Larrabee's real-time ray tracer [SCS*08], which reports the real-time construction of a 25MB k-D tree of a 234K triangle scene rendered with 4M rays and similar scalability for total time-to-render.

Kun Zhou et al. [ZHWG08] built k-D trees on the GPU, using a data-parallel spatial median algorithm for the upper levels of the tree, to a level where each node's subtree could be generated by each of the GPU's streaming processors. Their 128-core GPU version achieved speedups of $6 \sim 15\times$ over a single-core CPU, and of $3 \sim 6\times$ over 16-cores of the GPU for scenes ranging from 11K to 252K triangles. Their speedups improved for larger models, but their SAH and median approximations degraded the k-D trees and corresponding rendering times of these larger models, by as much as 10% for scenes over 100K triangles. Like Zhou et al.'s GPU algorithm, both our nested and in-place algorithms use scan primitives for data parallelism, but our new algorithms compute SAH precisely at all levels and propagate information differently from level to level.

Several authors have also examined the construction of dynamic bounding volume hierarchies. Wald et al. [WBS07] explore BVH maintenance for dynamic scenes for real-time rendering, showing them to be faster to construct but slower to render than similar k-D tree approaches. Wald [Wal07] describes a binned SAH BVH approach using "horizontal" and "vertical" parallelism, which resembles the node and geometry parallelism described in the next section, and reports CPU bandwidth limitations (as does our results section).

Lauterbach et al. [LGS*09] constructed a dynamic BVH on the GPU, using a breadth-first approximated SAH computation using GPU work queues optimized for SIMD processing by compaction. Similar to previous k-D tree approaches, they observe low utilization for the upper-level nodes and instead sort along a space filling curve to organize the upper levels into a linearized grid-like structure that serves effectively as a flattened spatial median tree.

3. Parallel Patterns for k-D Trees

Software patterns emerge from recurring program designs [GHJV95], and have evolved to include parallel programming [MSM04]. Fig. 2 illustrates patterns for parallel k-D tree construction that emerge from the analysis of previous work.

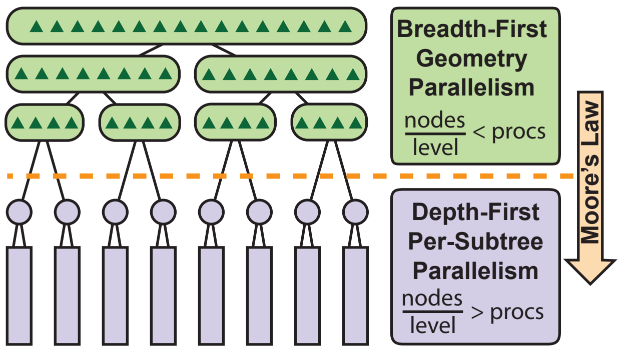
Breadth-First Geometry Parallelism
$\frac{\text{nodes}}{\text{level}}$ < procs
Moore's Law
Depth-First Per-Subtree Parallelism
$\frac{\text{nodes}}{\text{level}}$ > procs

Figure 2: *Parallel k-D Tree Patterns. Each level of the upper (green) portion of the tree has fewer nodes than cores, so multiple cores must cooperate on node creation leading to a breadth-first stream process that organizes all of the triangles into the current level's nodes. When the number of nodes at a level meets or exceeds the number of cores, then each node's subtree can be processed per core independently. The dashed dividing line (orange) where the number of nodes equals the number of processors descends one level every 1.5 to 2 years, indicating that the upper (green) pattern will eventually dominate k-D tree construction.*

The initial phases of a breadth-first top-down hierarchy construction consist of cases where large amounts of geometry need to be analyzed and divided among a few nodes. These cases suggest an approach where scene geometry is streamed across any number of processors whose goal is to analyze the geometry to determine the best partition, and categorize the geometry based on that partition. Previous serial and parallel versions of this streaming approach to SAH computation [WH06, PGSS06, HMS06, SSK07] all share this same pattern at the top of their hierarchies (as do breadth-first GPU constructions based on median finding [ZHWG08, GHGH08]), which can be efficiently parallelized by the techniques discussed in this paper.

Once the hierarchy has descended to a level whose number of nodes exceeds the number of cores or threads, then a node-parallel construction with depth-first traversal per node becomes appropriate. Here each subtree is assigned to a separate thread and is computed independently. Even on the GPU this parallelism is independent in that it needs no interprocessor communication, though the processes would run in SIMD lock step. If the subtrees vary in size, then load balancing via task over-decomposition/work stealing or other methods can be employed.

The most recent parallel SAH k-D tree construction algorithms ignore SAH in the top half of the tree, instead using the triangle count median [SSK07] or the spatial median [ZHWG08]. We see from Figure 1 that using median splitting planes for upper levels in a k-D tree degrades tree quality and rendering times significantly. In contrast, all the algorithms described in the rest of this paper compute precise SAH at all levels of the tree for high tree quality and rendering performance.

4. State-of-the-Art Sequential Algorithm

We begin by summarizing the best known sequential algorithm for precise SAH k-D tree construction [WH06]. Algorithm 1 shows that it finds the best SAH splitting plane for each node by an axis-aligned sweep across each of the three axes. It takes as input three pre-sorted lists (one per axis) of "events" (edges of the axis-aligned bounding box, one pair per triangle), and an axis-aligned bounding box representing the space covered by the node. The bounding box of the root node consists of the per-coordinate minima and maxima of the triangle vertices. For a descendant node, this bounding box is refined by intersection with the node's ancestry of splitting planes.

This single-thread sequential version builds a k-D tree in depth-first order, as revealed by the tail recursion. It achieves its $O(n \log n)$ efficiency due to its three axial sweeps through $E[axis]$ that compute SAH for each of the $O(n)$ events for each of the $O(\log n)$ levels of the k-D tree.

The SAH need only be evaluated at each *event* where the sweep encounters a new triangle or passes the end of a triangle [Hav00, (p. 57)]. Each event contains three members: its 1-D *position* along the axis, its type (START or END), and a reference to the triangle generating the event.

The three event lists $E[x], E[y], E[z]$ are each provided in position sorted order, and when two events share the same positions, in type order, where START $<$ END. These three sorts are a pre-process and also require $O(n \log n)$ time.

The algorithm consists of three phases. The first phase, FINDBESTPLANE, determines the axis, position, and corresponding event index of the splitting plane yielding the lowest SAH cost over the events in E. FINDBESTPLANE evaluates SAH at each event position (redundantly computing SAH even for coplanar events). The SAH evaluation at each event utilizes the triangle counts n_L, n_R to the left and right of the current splitting plane, which are maintained and updated as the sweep passes each event in each axis' sorted list. The SAH computation utilizes constants C_I, the cost of ray intersection, and C_T, the cost of traversal. Triangles that intersect the splitting plane are added to both sides. When the splitting plane sweep passes an END event, one less triangle

Algorithm 1: Sequential k-D Tree Construction

```
BuildTree($E_{x,y,z}$, $\square$) returns $Node$
/* $E[axis]$ - sorted events, $\square$ - Extent                    */
$C \leftarrow \infty$ ;                                     // SAH cost
foreach $axis' \in \{x,y,z\}$ do
    FindBestPlane($E[axis']$, $\square$) $\rightarrow (pos', C', i')$
    if $C' < C$ then $(C, pos, axis, i_{split}) \leftarrow (C', pos', axis', i')$
if $C > C_I \times |E[axis]|$ then return Leaf Node
ClassifyTriangles($E[axis]$, $i_{split}$)
FilterGeom($E, pos, axis$) $\rightarrow (E_L, E_R)$
Subdivide $\square$ into $\square_L, \square_R$ at $pos$ along $axis$.
$Node_L \leftarrow$ BuildTree($E_L, \square_L$)
$Node_R \leftarrow$ BuildTree($E_R, \square_R$)
return $Node(pos, axis, Node_L, Node_R)$

FindBestPlane($E[axis]$, $\square$) returns $(pos', C', i')$
$C' \leftarrow \infty, S \leftarrow$ surface area of $\square$, $n_L \leftarrow 0, n_R \leftarrow \frac{|E[axis]|}{2}$
foreach $e_i \in E[axis]$ do
    if $e_i$.type is END then decr $n_R$
    let $S_L, S_R$ be surface areas of $\square$ split at $e_i$.pos
    $C \leftarrow C_T + C_I(n_L \frac{S_L}{S} + n_R \frac{S_R}{S})$ ;          // SAH
    if $C < C'$ then $(pos', C', i') \leftarrow (e_i.\mathrm{pos}, C, i)$
    if $e_i$.type is START then incr $n_L$
return $(pos', C', i')$

ClassifyTriangles($E[axis]$, $i_{split}$)
/* Lbit, Rbit cleared for every $\triangle$ by prev. sweep          */
for $i \leftarrow 0 \ldots i_{split}$ do
    if $e_i$.type is START then set $E[axis][i]$. $\triangle$ .Lbit
for $i \leftarrow i_{split} \ldots |E[axis]| - 1$ do
    if $e_i$.type is END then set $E[axis][i]$.$\triangle$.Rbit

FilterGeom($E$) returns $(E_L, E_R)$
foreach $axis \in \{x,y,z\}$ do
    foreach $e \in E[axis]$ do
        if $e$.$\triangle$.Lbit then $E_L[axis]$.append($e$)
        if $e$.$\triangle$.Rbit then $E_R[axis]$.append($e$)
return $(E_L, E_R)$ // $E_L, E_R$ sorted
```

is on its right side, and when it passes a START event, one more triangle is on its left side.

The next two phases divide the event lists into (not necessarily disjoint) subsets left and right of the splitting plane. CLASSIFYTRIANGLES sweeps over the triangles, marking them as left or right, or both if they intersect the splitting plane. FILTERGEOMETRY divides the event lists into two portions, duplicating the splitting-plane straddling events, and maintaining the sorted order of the events for each axis.

5. Nested Parallel Algorithm

As Figure 2 illustrates, an obvious source of parallelism comes from independent nodes in the tree. Given two children of a node, the sub-trees under each child can be built indepedently (node-level parallelism). The problem with solely pursuing this approach is the lack of parallelism at the top levels of the tree. Unfortunately, at the top levels of the tree, each node has a larger number of events than at the bottom; the lack of node-level parallelism at these levels becomes a severe bottleneck. To alleviate this problem, we exploit a second source of parallelism: we parallelize the work on the large number of events (triangles) within a given node, referred to as geometry-level parallelism. Thus, our parallel algorithm nests two levels of parallelism. This is similar to the nested parallelism popularized by the NESL programming language [BHC*93, Ble95].

Expressing node-level parallelism is relatively straightforward in lightweight task programming environments such as Cilk [BJK*95] or Intel's Threading Building Blocks (TBB) [Int09] that allow recursive creation of light-weight tasks that are load balanced through a work stealing task scheduler. (We use TBB for our code.)

Within the computation of each node, we again use lightweight tasks to parallelize each of the major functions in the sequential computation (Algorithm 1) – FINDBESTPLANE, CLASSIFYTRIANGLES, and FILTERGEOM – as follows.

5.1. FINDBESTPLANE

Figure 3 depicts how FINDBESTPLANE works. Given an array of events (the top row of boxes, S=START E=END), the sequential "1 thread" box shows how FINDBESTPLANE in Algorithm 1 proceeds. The left-to-right sorted axis sweep maintains a running count of N_L and N_R, immediately incrementing N_L for each START event, and decrementing the next N_R for each END event. Recall that some triangles straddle the splitting plane and are counted in both N_L and N_R, and this post-decrement processing of END events accounts for such triangles. The remaining values needed for SAH evaluation are constants and $O(1)$ surface area computations. Hence as each event is processed, the current N_L, N_R counts generate the current SAH, which is compared against the previous minimal SAH to determine the minimal SAH splitting plane at the end of the sweep.

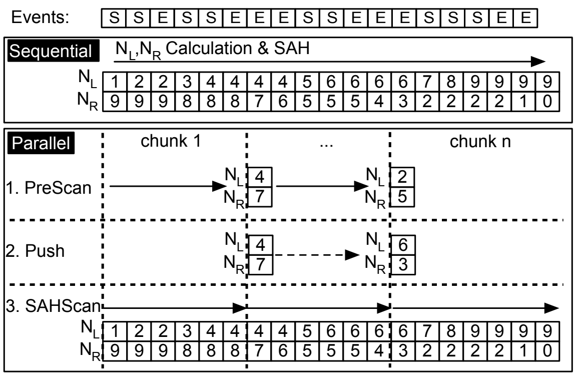
Events:
S S E S S E E E S S E E E S S S E E
Sequential
N_L,N_R Calculation & SAH
N_L 1 2 2 3 4 4 4 4 5 6 6 6 7 8 9 9 9 9
N_R 9 9 9 8 8 8 7 6 5 5 5 4 3 2 2 2 1 0
Parallel
chunk 1
...
chunk n
1. PreScan
N_L 4
N_R 7
N_L 2
N_R 5
2. Push
N_L 4
N_R 7
N_L 6
N_R 3
3. SAHScan
N_L 1 2 2 3 4 4 4 4 5 6 6 6 7 8 9 9 9 9
N_R 9 9 9 8 8 8 7 6 5 5 5 4 3 2 2 2 1 0

Figure 3: *Parallel SAH.*

We parallelize FINDBESTPLANE using a parallel prefix style operation [HS86], with three sub-phases: `PreScan`, `Push`, and `SAHScan` as illustrated in the lower (parallel) box of Fig. 3. We first decompose the event list into n contiguous chunks, allocating one chunk per task. For the `PreScan` phase, each of $n-1$ tasks counts the number of START and END edges in its corresponding chunk. (The last chunk need not be PreScanned.) Next, a single thread executes the `Push` phase, adding the total N_L, N_R of previous chunks to the current chunk totals, yielding correct N_L, N_R values at the beginning of each chunk. (In a typical parallel prefix, this is also done in parallel, but we did not find that necessary for the relatively few cores in our system.) For the final `SAHScan` phase, each of the n tasks processes its corresponding chunk, propagating its starting N_L, N_R values through the chunk and computing the minimum SAH value for its chunk. A final (sequential) reduction yields the minimum SAH across all n chunks.

5.2. CLASSIFYTRIANGLES

The CLASSIFYTRIANGLES phase classifies whether a triangle will fall into the left and/or right child of the current node, depending on its position with respect to the splitting plane. We can parallelize this phase by sweeping through the event array corresponding to the splitting plane axis, finding the corresponding triangle index for the event, and updating the right or left membership bit of the triangle. This is conceptually a parallelizable computation across the events; however, we found that it incurs significant false-sharing making it not profitable to parallelize. Our experiments reported in Sec. 7, therefore, do not parallelize this phase.

5.3. FILTERGEOM

The FILTERGEOM phase divides (for each of x, y, and z axes) one big array of events into two smaller arrays, duplicating some entries corresponding to plane straddling triangles, while preserving the sorted ordering from the original. On the face of it, this splitting with potential duplication of geometries into two sorted arrays of unknown length may appear to have limited parallelism (the length of the new arrays is currently unknown because some triangles may need to be duplicated). However, we can use the same observations as for parallelizing the FINDBESTPLANE phase here. We map the above to a parallel prefix style computation again, performing a parallel `PreScan`, a short sequential `Push`, and a parallel `FilterScan`. The parallel `PreScan` determines how many triangles in its chunk need to go to the left and right arrays. The `Push` accumulates all of the per-chunk information so that each chunk now knows how many triangles to its left will enter each of the two new arrays. This gives each chunk the correct starting location in the new arrays. All chunks can thus proceed in parallel to update their own independent portions of the two new arrays, creating a fully sorted pair of arrays in parallel in the `FilterScan` phase. (Note that the information about whether an event goes to the left or right new array is obtained from the *Lbit* and *Rbit* flags of the triangle corresponding to the event, as set in the CLASSIFYTRIANGLES phase.)

6. In-Place Parallel Algorithm

One major drawback of the state-of-the-art sequential Alg. 1 in Sec. 4 is that the division and distribution of triangle and event lists from a node to its two children require a lot of data movement. Worse yet, there exists a slight growth in the aggregate working set size due to triangles intersecting the splitting plane, which is proportional to the square root of the number of triangles in the node [WH06]. Since the parallel version in Sec. 5 essentially follows the structure of the sequential algorithm, it inherits these problems as well.

In an attempt to eliminate the cost of this data movement, we developed a new "in-place" algorithm. This algorithm is based on the insight that, although each node can contain many triangles, each triangle belongs to a small number of nodes at any given time during the construction of the top-levels of the tree. Our experiments revealed that triangles usually belong to a single node (most don't intersect splitting planes) and even in the worst case they belong to no more than eleven nodes for the tree depth of eight for the inputs used in this paper.

Our "in-place" algorithm overcomes the expense of data movement by letting the triangles keep track of which of a level's nodes they belong to. This is in contrast to the previous approach that required nodes to keep track of which triangles they contained. When FILTERGEOM processes each level, it moves triangle and event data from the parent node into its two child nodes. In "in-place," we instead update the "membership" of each triangle.

Zhou et al. [ZHWG08] employ an analogous strategy of keeping events (split candidates) in-place during a small node precise SAH construction phase, but the strategy relies on a bit mask representation of triangle sets which is only feasible for small numbers of triangles and is hence only viable for lower level construction. In contrast, our approach keeps events in-place throughout top level construction as well.

This new approach has the following implications:

1. The triangle data structure and the axial event elements are not moved in memory. Instead, the triangle's "nodes" membership field is updated.
2. A post-process at the end of k-D tree construction is necessary to produce the output in a desired format, which involves scanning the entire array of triangles and collecting them into appropriate node containers.
3. Since event elements remain fixed in memory, no re-sorting of any form is necessary at any stage.
4. Triangles can be more easily organized in a struct-of-arrays instead of an array-of-structs for a more cache-friendly memory access pattern. This particular optimization is not as easily applicable in the previous nested parallel algorithm due to the FILTERGEOM phase that mutates the array structure. The ordering must be preserved at the object granularity, which is difficult to achieve with the array of objects in struct-of-arrays format.
5. The in-place algorithm operates one level of the tree at a time, with sweeps on the entire array (instead of chopping the array into increasingly smaller pieces). This type of access pattern incurs worse cache behavior but is arguably more amenable to SIMD instructions and GPUs – this tradeoff remains to be studied since we do not focus on SIMD or GPUs in this paper.

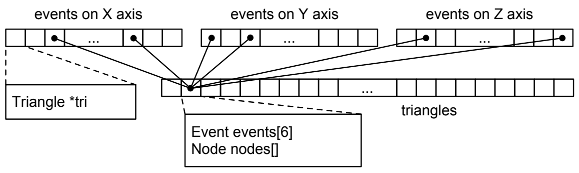

Figure 4: *Data structures used in the in-place algorithm.*

6.1. Algorithm

The algorithm operates on the data structure shown in Fig. 4. The three axial event arrays hold the events in position sorted order, and each event includes a pointer to the triangle that generated it. Each element of the triangle array contains pointers to the six events it generates, and a list of the current level's nodes to which it belongs.

One of the major differences between the nested-parallel approach in Sec. 5 and the in-place approach is that the latter is constructed in a breadth-first search manner, which makes more geometry parallelism available to tasks. The in-place approach processes the entire triangle stream and updates all the nodes of the current level, whereas the nested-parallel version switches between geometry processing and node construction phases. Therefore, it is a good choice for the geometry-parallel upper levels of k-D tree construction, and it should terminate when the number of nodes at the current level meets or exceeds the number of processing cores. From that point, subtrees can be constructed independently in parallel by each processor.

Alg. 2 outlines this approach. Current level's nodes are called "live," and each of them are considered for an SAH-guided split. It consists of four main phases:

FINDBESTPLANE Expanded from the FINDBESTPLANE phase in Sec. 5, this phase considers all live nodes in parallel instead of just one node. This phase outputs a splitting plane for each live node that is not to become a leaf.

NEWGEN This phase extends the tree by one level, creating two child nodes for each split live node. The decision to extend the tree is made dynamically since the SAH-based k-D trees are usually sparse.

CLASSIFYTRIANGLES This phase updates each triangle's node list using the next generation nodes created in NEWGEN.

FILL This phase occurs once at the very end of the tree-building process, outside the main loop. It is essentially a glue phase that translates the generated tree into the format of the trees generated by the sequential and the nested parallel algorithms.

Algorithm 2: Outline of the in-place algorithm.

```
Data: List of triangles (T) in the scene
Result: Pointer to the root of the constructed kd-tree
live ← {root ← new kdTreeNode() };
foreach △ ∈ T do
    △.nodes ← {root};
while nodes at current level < cores do
    // FindBestPlane phase (84.84% of time)
    foreach e ∈ E[x] ∪ E[y] ∪ E[z] do
        foreach node ∈ e.△.nodes do
            SAH ← CalculateSAH(e, node.extent);
            if SAH is better than node.bestSAH then
                node.bestEdge ← e ;
                node.bestSAH ← SAH ;
    // Newgen phase (0.04% of time)
    nextLive ← {};
    foreach node ∈ live do
        if node.bestEdge found then
            nextLive += (node.left ← new kdTreeNode()) ;
            nextLive += (node.right ← new kdTreeNode()) ;
    // ClassifyTriangles phase (14.60% of time)
    foreach △ ∈ T do
        oldNodes ← △.nodes ;
        clear △.nodes ;
        foreach node ∈ oldNodes do
            if no node.bestEdge found then
                // leaf node
                insert △ in node.triangles ;
            else
                if △ left of node.bestEdge then
                    insert node.left in △.nodes ;
                if △ right of node.bestEdge then
                    insert node.right in △.nodes ;
    live ← nextLive;
// Fill phase (0.52% of time)
foreach △ ∈ T do
    foreach node in △.nodes do
        insert △ in node.triangles ;
return root
```

6.2. Parallelization

As shown in Alg. 2, FINDBESTPLANE and CLASSIFYTRIANGLES phases together account for virtually all the build time. Therefore, we focused on parallelizing these two phases.

As in the nested-parallel algorithm, we employ the parallel prefix operators to compute FINDBESTPLANE. However, instead of a single pair of n_L, n_R, we maintain a list of pairs, one for each live node. In the nested algorithm, the goal of FINDBESTPLANE was to find one best plane that splits the given node. However, in the in-place algorithm, the end goal is to find a best plane for each live node.

CLASSIFYTRIANGLES phase is fully-parallel, since all of the information needed to update the node membership of each triangle object is found locally. Therefore, each thread can operate on a subsection of the triangle array in isolation.

7. Results

Methodology and metrics. We demonstrate the algorithms using the five test models shown in Fig. 5 for triangle counts varying from 60K to 1M. We measured the performance of the geometry parallel construction of the top eight levels of the tree, which on completion yields 256 subtree tasks that can be processed independently in parallel.

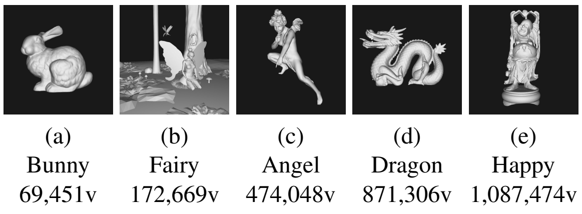

Figure 5: *Test models with triangle counts. Bunny, Dragon, and Happy courtesy of Stanford U., Angel courtesy of Georgia Tech, and Fairy courtesy of U. Utah.*

We performed experiments on the two machines shown in Table 1, which we refer to by Intel's product codename "Beckton" and "Dunnington." Both machines run CentOS 5.4. Beckton represents the state of the art, while results obtained using Dunnington are used to show how the algorithms exploit increased resources on new generations of machines (e.g., larger caches and memory bandwidth). We did not utilize Beckton's hyperthreading capability as we experimentally concluded that there were no significant advantages. We compiled the executables with GCC 4.1.2 with `-O3 -funroll-loops -fomit-frame-pointer` flags and linked against Intel TBB 2.2.

We present results in terms of speedup, measured both in absolute and self-relative terms. Absolute speedup numbers are measured using, as a 1× baseline, our optimized implementation of the sequential algorithm (Alg. 1), which outperformed Manta's sequential k-D tree builder [SBB*06]. We report self-relative speedups solely to understand parallel scalability of the algorithms. These use the single-thread runs of the parallel nested and in-place implementations as their 1× baseline. These single-thread versions do the same "work" as the parallel versions, including the unnecessary

Processor	Xeon E7450 (“Dunnington”)	Xeon X7550 (“Beckton”)
Microarchitecture	Core	Nehalem
Core Count	24	32
Socket Count	4	4
Last-level Shared Cache Size	12 MB (L2)	18 MB (L3)
Frequency	2.4 GHz	2.0 GHz
Memory Bandwidth	1x	9x
Memory Size	48 GB	64GB

Table 1: *Experimental Setup*

		Nested		In-Place	
Model	Best-serial	1-core	32-core	1-core	32-core
Bunny	0.304	0.455	0.068	0.512	0.050
Fairy	0.737	1.10	0.146	1.50	0.116
Angel	2.16	3.09	0.337	6.98	0.387
Dragon	3.75	5.50	0.654	8.63	0.744
Happy	4.67	6.89	0.835	11.8	0.951

Table 2: *Running times, in seconds, on Beckton.*

prescan portions of the parallelized phases. For reference, Table 2 lists running times, in seconds, for the best-serial, nested, and in-place algorithms on the Beckton machine, which also clarifies the difference between best-serial algorithm performance and one-core parallel algorithm performance.

Performance on state-of-the-art machine. Fig. 6 shows the absolute speedups of nested (left) and in-place (right), measured on the Beckton machine. Nested achieves nearly 8x speedup on Angel and in-place reaches 7x on Fairy. These represent the best parallel speedup for the upper levels of precise-SAH k-D tree construction to date.

The absolute speedup plot shows that for smaller Bunny (scanned) and Fairy (gaming, varying-sized triangle) inputs, in-place performs better than nested, whereas nested outpeforms in-place on larger (scanned, uniform-sized triangles) inputs. The performance of both algorithms saturates as the number of cores increase. Nested gives increasing performance up to 20 threads, whereas in-place gives increasing performance through 24 threads. In fact, nested's performance degrades significantly from the peak in all cases. Thus, although nested outperforms in-place for three out of five cases on the evaluated machine, the results indicate that in-place is more scalable. We next investigate in more detail the scalability of the two algorithms and the implications for future machines.

Scalability and performance on future machines. Fig. 7 shows the self-relative speedup of nested (left) and in-place (right) over our five inputs. This metric removes the impact of the increased amount of work done in the parallel algorithms (compared to the best sequential algorithm). By fixing the amount of work done across different thread counts, it provides us with a deeper insight on how effectively each algorithm exploits parallelism. The higher the self-relative speedup, the higher the potential for future larger machines with more cores to mitigate the cost of the increased work with increased parallelism. To further understand the effectiveness of the two algorithms in exploiting additional resources in new generations of machines (e.g., larger caches and memory bandwidth), we show self-relative speedups for

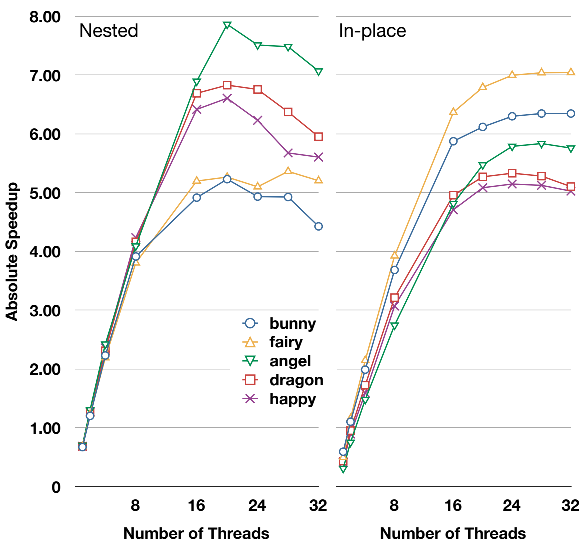

Figure 6: *Absolute speedup of the nested and in-place parallel algorithms for five inputs on the Beckton machine.*

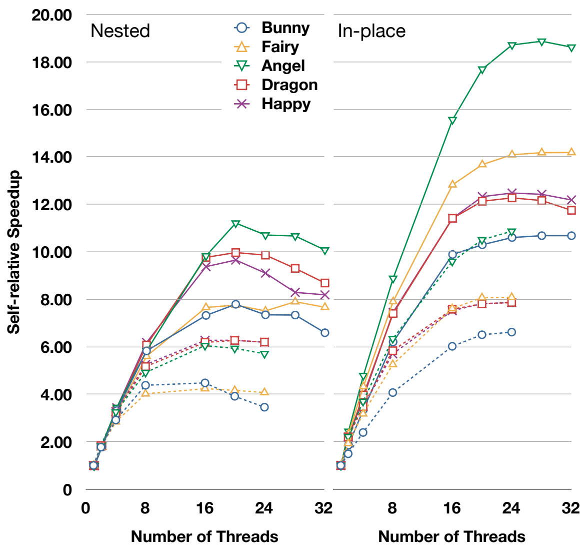

Figure 7: *Self-relative speedup of the nested and in-place parallel algorithms for five inputs, on the Beckton (solid lines) and Dunnington (dashed lines) machines.*

both the newer Beckton (solid line) and the older Dunnington (dashed lines) machines.

The figure immediately shows that in-place is more effective at exploiting parallelism than nested for all inputs on both machines. Although both algorithms perform better on the newer machine, in-place is better able to exploit the resources of the newer machine. Fig. 8 quantifies this effect by showing the ratio of the best speedup of in-place relative to nested for both machines (> 1 implies that in-place is faster). The figure clearly shows that for the two inputs where in-place starts out better on the older machine, its performance advantage increases further on the new machine. Conversely, for the cases where nested starts better, its performance advantage reduces on the new machine. Although in-place performance does not yet catch up with nested on the new machine for these cases, the following analysis shows that it is likely that in-place will continue to show higher scalability than nested in newer machines, potentially outperforming it for all cases.

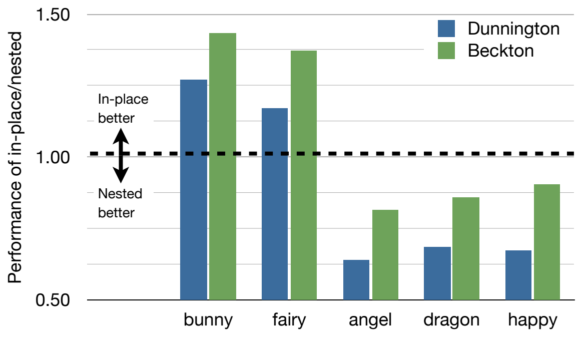

Figure 8: *Performance of in-place relative to nested on the Dunnington and Beckton machines, on all five inputs (> 1 means in-place is better).*

The main bottleneck to scalability for nested is its hard-to-parallelize CLASSIFYTRIANGLES phase. Amdahl's law [Amd67] states that the theoretical maximum speedup attainable using N threads for a program whose parallelizeable fraction is P is given by $1/((1-P)+(P/N))$. Table 3 indicates these maximum absolute (and self-relative) speedups for nested, based on measurements of the fraction of the execution time spent in CLASSIFYTRIANGLES on the Beckton machine.

For example, nested achieves close to 8x absolute speedup on Angel using 20 threads, whereas Table 3 indicates the theoretical maximum speedup of the nested algorithm is slightly less than 10.1x using 20 threads. Thus, nested is already seeing most of its theoretical maximum speedup. The degradation beyond that point is likely due to the increased communication and parallelization overhead with larger number of threads that is not mitigated enough by the increased parallelism.

The in-place algorithm, on the other hand, does not suffer

Input	24 threads	32 threads	∞ threads
bunny	11.5 (14.2)	12.9 (16.5)	21.0 (33.1)
fairy	11.6 (14.4)	13.1 (16.8)	21.6 (34.4)
angel	10.1 (13.5)	11.2 (15.6)	16.7 (29.6)
dragon	9.4 (13.4)	10.3 (15.5)	14.7 (29.3)
happy	9.4 (13.2)	10.3 (15.2)	14.8 (28.1)

Table 3: *Theoretical maximum absolute (and self-relative) speedups achievable by the nested algorithm, based on parallelizable fraction on the Beckton machine.*

from such a bottleneck since it does not contain any significant sequential portion. The performance saturation at larger core counts seen in in-place is likely due to limited system resources; e.g., cache size and memory bandwidth. To investigate this hypothesis, we ran our experiments with all threads scheduled in as few sockets as possible (the default scheduler spreads the threads among the sockets) – this had the positive effect of more cache sharing for smaller input sizes and the negative effect of reduced available pin bandwidth for larger input sizes. We found that the performance of our algorithms was indeed sensitive to the thread placement, showing both the above positive and negative effects (detailed results not shown here).

In summary, we believe that higher core counts coupled with larger caches and memory bandwidth in future machines will allow in-place to continue seeing performance improvements. The performance scalability for nested, however, is likely to be limited by its serial bottleneck.

8. Conclusion

We have presented and analyzed a pair of algorithms designed to address the lack of scalability and/or lack of quality in the upper levels of spatial hierarchies construction. Using our prototype implementations, we showed that our two algorithms, nested and in-place, can achieve speedups of up to 8x and 7x, respectively, over the best sequential performance on a state-of-the-art 32-core cache-coherent shared-memory machine. To our knowledge, these algorithms provide the best known speedups for precise SAH-based high quality k-D tree construction, relative to a sequential case that is better than the best publicly available code.

Each algorithm outperforms the other on some of our inputs for the current state-of-the-art machine, but the in-place approach showed better scalability. Using data obtained from two machines that are a product generation apart, we show that in-place is more effective in harnessing the additional system resources of new machine generations (e.g., cache size and memory bandwidth) than nested. We showed that nested is limited in scalability by a sequential Amdahl's law bottleneck. Overall, we conclude that the in-place algorithm has more potential to scale in future generation multicore hardware.

An interesting future research topic is a GPU implementation of the in-place algorithm. The streaming nature of the in-place algorithm makes it more amenable to a GPU's SIMD-style computation model than the nested algorithm's inherent recursive approach. We are currently investigating various ways to map in-place onto a GPU, and are not aware of any prior work on fully precise SAH based high quality k-D tree construction on the GPU platform.

Another topic for further research centers around the bandwidth limitations of hierarchical data structures identified here and by previous publications, both on CPU and GPU platforms. We have identified some optimizations for the current implementations to improve locality, but these remain to be fully explored.

Acknowledgments Thanks to Ingo Wald for helpful discussions. This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation.

References

[Amd67] AMDAHL G. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. Spring Joint Computer Conf.* (1967), pp. 483–485.

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[BHC*93] BLELLOCH G. E., HARDWICK J. C., CHATTERJEE S., SIPELSTEIN J., ZAGHA M.: Implementation of a Portable Nested Data-parallel Language. In *Proc. Symp. on Principles and Practice of Parallel Programming* (1993), pp. 102–111.

[BJK*95] BLUMOFE R. D., JOERG C. F., KUSZMAUL B. C., LEISERSON C. E., RANDALL K. H., ZHOU Y.: Cilk: An Efficient Multithreaded Runtime System. 207–216.

[Ble95] BLELLOCH G. E.: *NESL: A Nested Data-Parallel Language*. Tech. rep., Pittsburgh, PA, USA, 1995.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In *Proc. Graphics Interface* (2006), pp. 203–209.

[GDS*08] GOVINDARAJU V., DJEU P., SANKARALINGAM K., VERNON M., MARK W. R.: Toward a Multicore Architecture for Real-Time Ray-Tracing. In *Proc. Intl. Symp. on Microarchitecture* (2008), pp. 176–187.

[GHGH08] GODIYAL A., HOBEROCK J., GARLAND M., HART J. C.: Rapid Multipole Graph Drawing on the GPU. In *Proc. Graph Drawing* (2008), pp. 90–101.

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-buffer Visibility. In *Proc. SIGGRAPH* (1993), pp. 231–238.

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] HUNT W., MARK W. R., STOLL G.: Fast k-D Tree Construction with an Adaptive Error-Bounded Heuristic. In *Proc. Interactive Ray Tracing* (2006), pp. 81–88.

[HS86] HILLIS W. D., STEELE JR. G. L.: Data Parallel Algorithms. *CACM 29*, 12 (1986), 1170–1183.

[Int09] INTEL: *Intel (R) Threading Building Block Manual*, 2009. Document number 315415-001US.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LP08] LUEBKE D., PARKER S.: *Interactive Ray Tracing with CUDA*. Tech. rep., 2008.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing using Space Subdivision. *Visual Computer 6*, 3 (1990), 153–65.

[MSM04] MATTSON T. G., SANDERS B. A., MASSINGILL B. L.: *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proc. Interactive Ray Tracing* (2006), pp. 89–94.

[SBB*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S.: An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *Proc. EG Symp. on Parallel Graphics and Vis.* (2006), pp. 19–26.

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph. 27*, 3 (2008), 18:1–18:15.

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast k-D Tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404.

[Wal07] WALD I.: On Fast Construction of SAH based Bounding Volume Hierarchies. In *Proc. Interactive Ray Tracing* (2007).

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[WH06] WALD I., HAVRAN V.: On Building Fast k-D Trees for Ray Tracing and On Doing That in $O(NlogN)$. In *Proc. Interactive Ray Tracing* (2006), pp. 61–69.

[WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved Computational Methods for Ray Tracing. *ACM Trans. Graph. 3*, 1 (1984), 52–69.

[WIP08] WALD I., IZE T., PARKER S. G.: Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes. *Computers & Graphics 32*, 1 (2008), 3–13.

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph. 24*, 3 (2005), 434–444.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time k-D Tree Construction on Graphics Hardware. In *Proc. SIGGRAPH Asia* (2008), vol. 27, pp. 1–11.