Parallel SAH k-D Tree Construction for Fast Dynamic Scene Ray Tracing

Byn Choi Rakesh Komuravelli Victor Lu Hyojin Sung Robert L. Bocchino
Sarita V. Adve John C. Hart
University of Illinois, Urbana-Champaign
{bynchoi1, komurav1, victorlu, sung12, bocchino, sadve, jch}@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 new, faster multicore algorithms for building precise SAH-optimized kd-trees. Our best algorithm makes a tradeoff between worse cache performance and higher parallelism to provide up to 7X speedup on 16 cores, using two different kinds of parallelism models, without degrading tree quality and rendering time.

1. Introduction

Visual computing is becoming one of the killer application classes for emerging multicore client systems. This surge in computational power enables the improved photorealism and simplified software development of ray tracing to replace the approximations and time-consuming special-case programming of rasterization-based rendering (until recently the only choice for real-time image synthesis) [4, 10, 13, 18, 23]. At the heart of an efficient real-time ray tracing algorithm is a spatial data structure to accelerate geometric queries, such as the popular k-D tree, shown by some to be optimal [11]. It organizes space as a hierarchy of axis-aligned splitting planes, with each plane dividing a node's region of space in two. These splitting planes hence refine geometry into smaller and smaller axis-aligned bounding boxes. When used for ray tracing, rays missing a splitting plane need not intersect any scene geometry on its other side, and rays intersecting a splitting plane can check scene geometry on its front side first and avoid other-side intersection if a front-side intersection is found.

For static scenes where the k-D tree can be constructed off-line, there has been enormous progress in parallelizing ray tracing to approach real-time speeds. As computer games and virtual environments become more dynamic and social, with unpredictable user-generated content that may include self-scanned avatars, construction of the k-D tree must occur on-line and could pose a critical bottleneck to real-time rendering. Sequential k-D tree construction algorithms are too slow and current parallel approaches for construction sacrifice k-D tree quality, which degrades ray tracing performance.

The position of splitting planes is the key to the quality and rendering acceleration of the k-D tree. For ray tracing, the surface area heuristic (SAH) is widely agreed to be the best heuristic for choosing the splitting plane [9, 14]. SAH selects splitting planes that bound the largest number of scene triangles with a region of least surface area. This accelerates ray tracing since rays that miss a node's region need not intersect its geometry, and rays are more likely to miss a region with less surface area.

Construction of an SAH k-D tree therefore starts with determining which axis and the position along that axis of the splitting plane that minimizes the SAH metric for the space represented by the root node (the geometries from the entire scene) and splits the geometries along this plane to make two children nodes. The algorithm recursively follows this procedure for the children, building a tree in depth-first order. A node becomes a leaf when the SAH indicates it is no longer profitable to split it; leaf nodes each contain a small list of the triangles in the region of space they represent.

An obvious source of parallelism in this algorithm is in the work for independent nodes of the tree (node-level parallelism). Once enough nodes are created, their sub-trees can all be efficiently built in parallel. The challenge arises when working in the upper levels of the tree where there is insufficient node-level parallelism. The main parallelism here is in the processing of the geometries to determine the SAH optimal plane (geometry-level parallelism). Unfortunately, this has been hard to scale as demonstrated by previous work in Section 3 and explained in Section 4. State-of-the-art parallel algorithms therefore either ignore or approximate SAH evaluation for upper levels of the tree at the expense of tree quality and potential rendering performance degradation (Section 3). As the number of cores in a multicore system is expected to increase with every generation of Moore's law, we expect an increase in the number of tree levels that must be processed using such approximations before sufficient node-level parallelism becomes available. This results in increasing, and as we show, unacceptable rendering time degradation.

In this work, we propose two new algorithms for k-D tree construction that exploit both geometry and node-level parallelism in different ways. The first, called *nested*, began as an incremental and relatively straightforward task parallelization of the sequential algorithm. It evolved into a nested-parallel version that parallelizes the SAH computation within a node to nest the geometry-level parallelism within the node-level parallelism.

The second algorithm, called *in-place*, was developed in response to an early analysis of nested, that showed a bottleneck in the movement of the geometries into different containers when splitting a node into its children. The in-place version eliminates the movement of the geometries by a more fundamental change to the sequential algorithm. The key idea is to process all the geometry in a given tree level in-place in a unified manner in one phase, using all the cores available for that phase. There is no geometry movement from level to level; rather, the location of the geometry in the tree is recorded along with the geometry itself.

The in-place algorithm has worse cache and sequential behavior than the best current sequential algorithm, and the nested version still requires a lot of geometry data movement. However as the number of nodes increases up to as many as sixteen, both quickly outperform both the sequential approach and previous parallel approaches. Both algorithms employ geometry parallelism to compute SAH more accurately and faster than existing parallel approaches. The nested algorithm combines node and geometry parallelism whereas the in-place algorithm also performs both but separately. The nested algorithm builds a tree in depth-first order whereas the in-place approach builds breadth-first.

Our results focus on the performance of our algorithms for the upper levels of the tree. These upper levels are critical for parallel performance and scalability as they need to divide the problem into enough subtrees that each core can process independently. We used 16 cores, and computed to depths of 4, 6 and 8, generating up to 256 subtree jobs that can then be independently processed by 16 or more cores efficiently with load balancing. For these depths we achieved speedups of up to 7.1X for the nested algorithm and 4.8X for the in-place algorithm, on 16 cores, relative to the best sequential code.

To our knowledge, these results represent the first parallel speedups on the upper levels of k-D tree construction using precise SAH. In contrast, the best previously reported speedups for a 16 core system is about 2.5X on a GPU using a spatial median instead of SAH for the upper-tree nodes, which generates lower quality k-D trees [24]. Further discussions and results are provided in Section 9.

Both algorithms were implemented solely for MIMD multicore processing. While the in-place algorithm required significant programming complexity and scaled well but short of the nested approach, it does provide valuable insight into the non-intuitive tradeoffs between cache performance and parallelism. Interestingly, the self-relative speedup of the in-place algorithm is high and so the relative tradeoff between the in-place and nested algorithms remains an open question, making both algorithms valuable at this point. In the conclusion we look forward to further work on these algorithms that include SIMD parallelism available through ever widening vector instructions and graphics co-processors.

2. Background

Ray tracing is a photorealistic rendering method that traces paths of light backwards from the eye bouncing off one or more objects in the scene to a light source. Recursive Whitted-style ray tracing casts a ray through each pixel in the scene, and at its intersection point casts a ray toward the light source to detect shadows, a ray in a carom bounce direction for mirror reflection, and through the surface in a Snell-ratio direction for refraction [22]. Since reflected and refracted rays can themselves reflect and refract, this process generates a tree of rays for each pixel. Multiple perturbed rays are often cast from each point for antialiasing, motion blur, soft shadows and glossy reflection [17]. Non-local effects such as color bleeding, caustics and sub-surface scattering can also be ray traced, by additional ray trees from the light source. Hence ray tracing renderers typically need to intersect $10^6 \sim 10^9$ rays with $10^4 \sim 10^7$ triangles, in some cases many more.

Spatial data structures avoid testing all-pairs of ray-triangle intersections. The three most common choices are grids, bounding volumes hierarchies and k-D trees. Uniform 3-D grids create a rectilinear array of cells each containing a list of the geometry that intersects the cell. The cells a ray intersect can be simply enumerated, and only the geometry intersecting these cells need be intersected against the ray. Grids work best when scene triangles are similarly sized, and a k-D tree was shown to accelerate a GPU ray tracer $8\times$ faster than a 3-D grid [6].

A bounding volume is a simple shape surrounding a large number of triangles, such that a ray missing the bounding volume quickly indicates it misses all of the triangles. Clusters of bounding volumes can themselves be contained in a bounding volume forming a hierarchy. The bounding volume hierarchy often follows a modeler's natural scene hierarchy, which makes them better suited for animator-designed scenes than for scanned objects or user-designed ad hoc objects.

The k-D tree accelerates ray intersections with varying sized and unorganized geometry. The construction of a k-D tree is often top-down, finding a median or other optimum planar division of space. For ray-triangle intersection, this optimum is guided by the surface area heuristic [9, 14]

$$\mathrm{SAH}(P) = C_T + C_I \left(\frac{A_L}{A} N_L + \frac{A_R}{A} N_R \right), \qquad (1)$$

where P is a splitting plane candidate, C_T, C_I are the relative costs of node traversal and plane intersection, A, A_L, A_R are the surface areas of the current node's region and its portion to the left and right of P, and N_L, N_R are the number of the node's triangles to the left and right of P.

The splitting plane P might itself intersect triangles, in general as many as $\sqrt{N}$ where N is the number of triangles in a node's region. Such triangles would be included in both the left and the right children of the node, so $N_L + N_R \geq N$.

3. Related Work

Wald and Havran [21] describe an optimal $O(n \log n)$ 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 5 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 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 [12, 16, 19] 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 (almost) all n of the triangles are processed for each of the $\log n$ levels.

There have been many attempts to parallelize 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 [1, 12, 16], limiting 4-core speedup to only 2.5.

Shevtsov et al. [19] 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 as much as 30%. They did not report a construction time speedup, but they did report a 4-core speedup of 3.5 for a construction combined with rendering, which includes millions of k-D tree traversals.

Kun Zhou et al. [24] 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[1], 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. Our 16-core CPU precise-SAH algorithms outperform the speedups of $1.8 \sim 2.65\times$ they achieved for their 16-core GPU versions, when compared to a single core CPU.

Both of these very recent approaches sacrificed SAH evaluation in favor of medians in the upper levels until a level where subtrees can be processed independently by each core. The next section shows that this approach will degrade k-D tree rendering performance in the near future, prompting the development of new parallel algorithms for precise-SAH k-D trees, such as the ones described in the remainder of this paper.

4. Parallel Patterns for k-D Trees

Software patterns emerge from recurring program designs [7], and have evolved to include parallel programming [15]. Fig. 1 illustrates patterns for parallel k-D tree construction that emerge from the analysis of previous work.

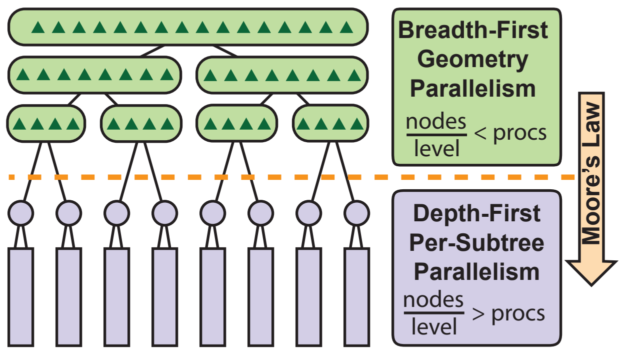

Breadth-First Geometry Parallelism
nodes/level < procs
Depth-First Per-Subtree Parallelism
nodes/level > procs
Moore's Law

Figure 1. 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 [12, 16, 19, 21] all share this same pattern at the top of their hierarchies (as do breadth-first GPU constructions based on median finding [8, 24]), which can be efficiently accelerated, even for precise SAH, by a geometry-parallel approach.

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.

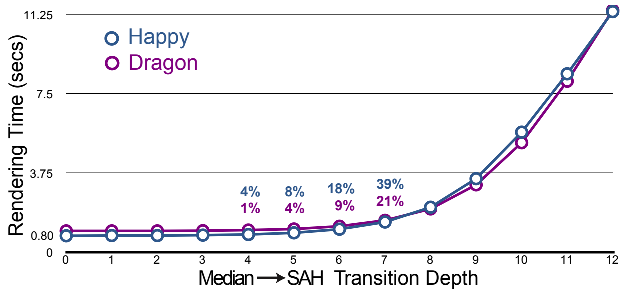


Figure 2. Rendering times v. the k-D tree transition level where SAH is used instead of triangle count median. (Spatial median similarly degrades rendering time [24].) The depth of the k-D tree is 30. The horizontal transition axis corresponds to the depth of the dashed (orange) line in Fig. 1 for the best existing parallel k-D tree construction algorithms [19, 24]. The percentage of rendering performance degradation is indicated above levels 4-7.

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 [19] or the spatial median [24]. Fig. 2 shows the rendering time (on a single core) of an implementation of the algorithm in [19] (Section 8 gives details on the experimental methodology including the input models and system used). We see that using median splitting planes for upper levels in a k-D tree does not adversely affect tree quality until about level 6; beyond the sixth level, it starts to degrade the rendering time drastically. As the number of cores continue to double every 18 to 24 months, we will soon reach the point where existing parallel k-D tree construction algorithms produce increasingly worse k-D trees for ray tracing. In contrast, all the algorithms described in the rest of this paper paper compute precise SAH at all levels of the tree for high tree quality and rendering performance.

5. State-of-the-Art Sequential Algorithm

We begin by summarizing the best known sequential algorithm for precise SAH k-D tree construction [21]. As shown in Algorithm 1, 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 a list of triangles, a presorted list of events (each representing an axis-aligned bounding edge of a triangle, described later), and an axis-aligned bounding box representing the space covered by the node. For the root node, this bounding box is the minima and maxima of each of the coordinates of the triangle vertices. For a deeper k-D tree node, this bounding box is refined by intersection with its 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 [11, p. 57]. Each event contains three members: the 1-D *position* of the event along the axis, the type (START or END) of the event, and a reference to the triangle generating the event.

[1] . . . measured via a driver that disabled processors but not peak memory bandwidth.

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 preprocess and also require $O(n \log n)$ time.

Algorithm 1: Sequential $O(N \log N)$ KD-Tree Construction

```
BuildTree($T$, $E_{x,y,z}$, $\square$) returns $Node$
/* $T$ - Triangles, E[axis] - sorted events for
   each axis, $\square$ - Node extent                  */
$C \leftarrow \infty$
foreach $axis' \in \{x, y, z\}$ do
    FindBestPlane($E[axis'], \square$) $\rightarrow (pos', C', i')$
    if $C' < C$ then
        $(C, pos, axis, i) \leftarrow (C', pos', axis', i')$
if $C > C_I \times |T|$ then return $Node(T)$ // leaf
/* Split triangles, events and bounding box  */
ClassifyTriangles($T$, $E[axis]$, $i$)
FilterGeom($T, E, pos, axis$) $\rightarrow (T_L, E_L, T_R, E_R)$
Subdivide $\square$ into $\square_L, \square_R$ at $pos$ along $axis$.
$Node_L \leftarrow$ BuildTree($T_L, E_L, \square_L$)
$Node_R \leftarrow$ BuildTree($T_R, E_R, \square_R$)
return $Node(pos, axis, Node_L, Node_R)$
```

```
FindBestPlane($E[axis], \square$) returns $(pos', C', i')$
$C' \leftarrow \infty, S \leftarrow$ surface area of $\square$
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
    $C \leftarrow C_T + C_I(n_L \frac{S_L}{S} + n_R \frac{S_R}{S})$ ;    // compute SAH
    if $C < C'$ then $(pos', C', i') \leftarrow (e_i.\text{pos}, C, i)$
    if $e_i$.type is START then incr $n_L$
return (pos', C', i')
```

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

```
FilterGeom($T, E$) returns $(T_L, E_L, T_R, E_R)$
foreach $\triangle \in T$ do
    if $\triangle$.Lbit then $T_L$.append($\triangle$)
    if $\triangle$.Rbit then $T_R$.append($\triangle$)
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 $(T_L, E_L, T_R, E_R)$ // $E_L, E_R$ sorted
```

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.[2] The SAH evaluation at each event occurs in constant time, demanding the instant availability of the number of triangles n_L, n_R to the left and right of the current splitting plane. Hence n_L and n_R are maintained and updated as the sweep passes each event in each axis's sorted list. Triangles that intersect the splitting plane are considered on both sides. When the (left-to-right) splitting plane sweep passes an END event, one less triangle is on its right side, and when it passes a START event, one more triangle is on its left side.

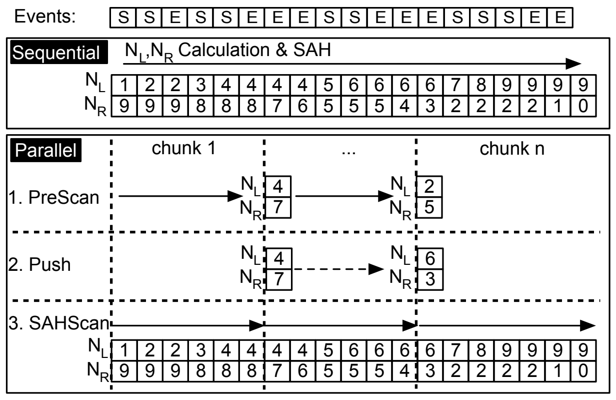
The next two phases divide the triangle and event lists into left and right lists of triangles and events. CLASSIFYTRIANGLES sweeps over the triangles, marking them as left or right (or both if they intersect the splitting plane). FILTERGEOMETRY divides the triangle and event lists into two portions, duplicating the splitting-plane straddling triangles and their events, and maintaining the sorted order of the events for each axis.

6. Nested Parallel Algorithm

As Figure 1 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 (per-node 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.

Expressing node-level parallelism is relatively straightforward in lightweight task programming environments such as Cilk [3] or Intel's Thread Building Blocks (TBB) [5] that allow recursive creation of light-weight tasks that are load balanced through a work stealing algorithm. (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.



Events: S S E S S E 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 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 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 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 2 1 0

Figure 3. Parallel SAH

6.1 FINDBESTPLANE

Figure 3 depicts how `FindBestPlane` works. Given an array of events (the top row of boxes, S=START E=END), the serial "1

[2] In Algorithm 1, FINDBESTPLANE evaluates SAH at each event position. When triangles are coplanar with the splitting plane or multiple events share the same position, Wald and Havran describe special case rules to find the least SAH evaluation without computing SAH for all events that share this single position [21]. We implemented these optimizations, but found that they did not provide much benefit; therefore, for simplicity, we do not include them here. (As discussed in Section 8, our sequential version outperforms the state-of-the-art manta builder that incorporates the above optimizations.)

thread" 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.

We parallelize `FindBestPlane` with a parallel prefix style operation, in three 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 `ScanSAH` 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.

6.2 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 parallel computation across the events; however, as we find in our experiments, it incurs significant false sharing which makes it not profitable to parallelize.

6.3 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. Thus, 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 starting location in the new arrays where it needs to insert its set of events. 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. (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 by the ClassifyTriangles function.)

7. In-Place Parallel Algorithm

One major drawback to the state-of-the-art sequential Algorithm 1 in Section 5 is that the division and distribution of triangle and event lists from a node to its two children requires a lot of data movement (and slight data growth for triangles intersecting the splitting plane, proportional to the square root of the number of triangles in the node). Since the parallel version in Section 6 essentially follows the structure of the sequential algorithm, it does not address this shortcoming either.

To avoid 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. Our experiments revealed that triangles usually belong to a single node (most don't intersect splitting planes) and in the worst case belong to no more than five nodes for typical tree depths of eight, for the scenes used in this paper.

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

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 in 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 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 has to modify the array structure. In order to preserve the ordering, one must move elements in separate arrays in groups.
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 did not focus on SIMD or GPUs in this paper.

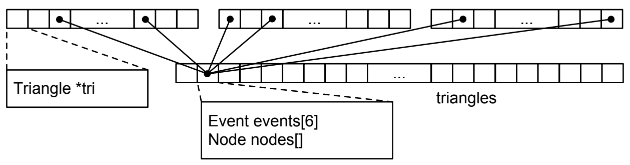


Figure 4. Data Structures used in in-place algorithm

7.1 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. 6 and the in-place approach is that the latter is constructed breadth first, which makes available more geometry parallelism per task. The in-place approach processes the entire triangle stream and updates all the nodes of the current level whereas the node-parallel version switches between triangle processing and node update phases. Hence it is a good choice for the geometry-parallel upper levels of k-D tree construction, and should terminate when the number of nodes at the current level meets or exceeds the number of processing cores, at which point subtrees can be constructed independently by each processor.

Algorithm 2 outlines this approach. It describes a breadth-first process, calling the current level's nodes "live," and consider each for an SAH-guided split. It consists of four main phases:

FindBestPlane Expanded from the SAH phase in Sec. 6, this phase considers all live nodes in parallel instead of just the current node. This phase outputs a splitting plane for each live node that doesn't become a leaf.

Newgen This phase extends the tree by one level, creating two child nodes for each split live node.

ClassifyTriangles This phase updates each triangle's node list to reflect membership in the child nodes generated by NEWGEN.

Fill This phase occurs at the very end, outside the main loop, to construct for each leaf node the triangles that belong in its region.

7.2 Parallelization

We verified that FINDBESTPLANEand CLASSIFYTRIANGLES phases together account for most of the build time and so we focus on parallelizing these two phases.

As did the nested parallel approach, we too use the parallel scan operators to compute FINDBESTPLANE, but instead of a single pair of n_L, n_R lists, we maintain multiple pairs of lists, one for each live node. Each of these lists is updated by the parallel scan only when the event's triangle includes that list's node. Hence each node's n_L, n_R list contains a count of the node's triangles left and right of a splitting plane placed at each event.

At the CLASSIFYTRIANGLES SPLIT stage all of the information needed to update the nodes membership of each triangle object can be found locally, yielding a completely parallel process where each thread operates independently on a subsection of the triangle array.

8. Implementation and Methodology

For our experiments, we use four widely used test models (Fig. 5) of varying geometric complexity, with triangle count ranging from 70K to 1M, in order to verify performance across a spectrum of problem sizes.

For our experiments, we constructed k-D trees to a transition depth of 4, 6, and 8, to evaluate each algorithm's effectiveness at enabling SAH-based parallel construction of the top levels of a tree. We assume that the remainder of the precise-SAH k-D tree will be generated by assining each node's remaining subtree to its own processing core for independent construction using existing sequential precise-SAH k-D tree construction algorithms.

All parallel tree construction time speed-ups are reported relative to measurements obtained using our optimized sequential implementation of Alg.1. Tree construction times for our implementation were compared to those generated by Manta (Table 1) for the Happy input to verify that our implementation is comparable to the state-of-the-art at maximum tree depths of 4, 6, and 8.

Algorithm 2: Outline of the in-place algorithm

```
Data: List of triangles (tris) in the scene
Result: Pointer to the root of the constructed kd-tree
live ← {root ← new kdTreeNode() };
foreach $\triangle \in T$ do
    $\triangle$.nodes ← {root};
while nodes at current level < cores do
    // FindBestPlane phase (84.84% of time)
    foreach $e \in E[x] \cup E[y] \cup E[z]$ do
        foreach node ∈ e.$\triangle$.nodes do
            SAH ← CalculateSAH($e$);
            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 $\triangle \in T$ do
        oldNodes ← $\triangle$.nodes ;
        clear $\triangle$.nodes ;
        foreach node ∈ oldNodes do
            if no node.bestEdge found then
                // leaf node
                insert $\triangle$ in node.triangles ;
            else
                if $\triangle$ left of node.bestEdge then
                    insert node.left in $\triangle$.nodes ;
                if $\triangle$ right of node.bestEdge then
                    insert node.right in $\triangle$.nodes ;
    live ← nextLive;
// Fill phase (0.52% of time)
foreach $\triangle \in T$ do
    foreach node in $\triangle$.nodes do
        insert $\triangle$ in node.triangles ;
return root
```

	Max Tree Depth	4	6	8	15
Bunny	Our builder	0.13	0.19	0.24	0.59
	Manta's builder	0.43	0.61	0.79	1.46
Angel	Our builder	0.89	1.25	1.65	3.33
	Manta's builder	3.59	5.21	6.65	11.05
Dragon	Our builder	1.56	2.23	2.90	6.03
	Manta's builder	5.06	7.19	9.09	14.94
Happy	Our builder	2.00	2.81	3.67	7.53
	Manta's builder	6.62	9.41	11.85	19.40

Table 1. Comparing sequential construction times (in seconds) of our builder to Manta's builder

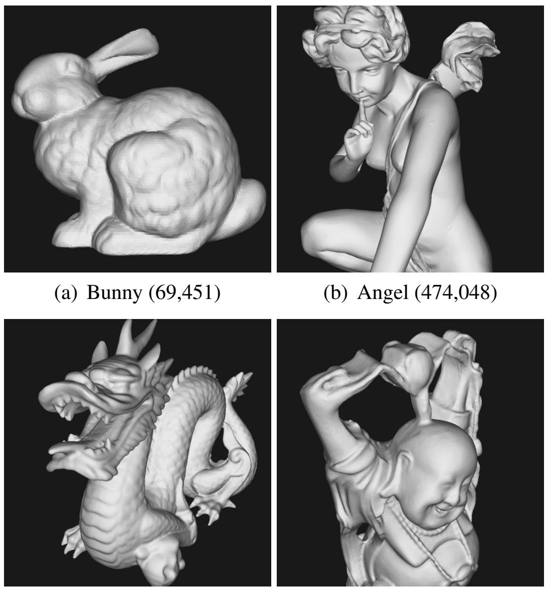
All rendering time measurements were obtained using the state-of-the-art Manta shared-memory multicore ray-tracer [2, 20]. The k-D trees produced by our construction algorithms were written to a file and then read into Manta using its built-in read-from-file function. The times for writing and reading the k-D tree to/from disk were not included in any of the measurements. Table 2 both demonstrates the scalability of the Manta ray-tracers as well as verifies the quality of the tree produced using our construction algorithms.

Number of Threads	1	2	4	8	16
Rendering Time	3.61 s	2.00 s	1.15 s	0.58 s	0.30

Table 2. Time to render (excluding build time) fig.5(d) at the original size of 1024×1024, with varying number of threads. Rendered using a tree constructed up to a maximum tree depth of 15 by our sequential builder

All phases in the nested parallel algorithm were parallelized, except for the CLASSIFYTRIANGLES phase. Specifically, we observed that the time taken by the CLASSIFYTRIANGLES phase was small and there was enough false sharing to not gain any benefit from parallelism. All phases in the in-place algorithm were parallel except FILL and NEWGEN because they occupied a very small portion of the overall computational time.

All experiments were run on a 24-way (four six-core CPUs - Intel© Xeon© E7450 @ 2.4 GHz) machine with 48 GB of RAM. The operating system is RedHat Enterprise Linux 4.1.2-44. All code was compiled using GCC 4.1.2 with -O3 flag. All parallel code was written using the Intel Threaded Building Blocks library.



(a) Bunny (69,451) (b) Angel (474,048)

(c) Dragon (871,306) (d) Happy (1,087,474)

Figure 5. Test models with triangle counts included in parenthesis. The images shown here are down-sized versions of the actual images (1024×1024) produced for measuring rendering time. The Bunny, Dragon, and Happy models are courtesy of the Stanford 3D Scanning Repository. Angel is courtesy of the Georgia Tech Large Geometric Model Archive).

9. Experimental Results

9.1 Overall Results

Figure 6 shows our overall results. Each graph plots the speedup of the nested parallel (dotted curves) and in-place parallel (solid curves) algorithms for increasing number of cores for our four inputs, for a specific depth of the tree. As mentioned in Section 8, we focus on performance for the upper levels of the tree and show results for depth = 4, 6, and 8 in parts (a), (b), and (c) respectively. In particular, depth=8 potentially provides 256-way parallelism for the subsequent phase exploiting node-level parallelism. All speedups are reported relative to the best sequential version for the corresponding depth as discussed in Section 8.

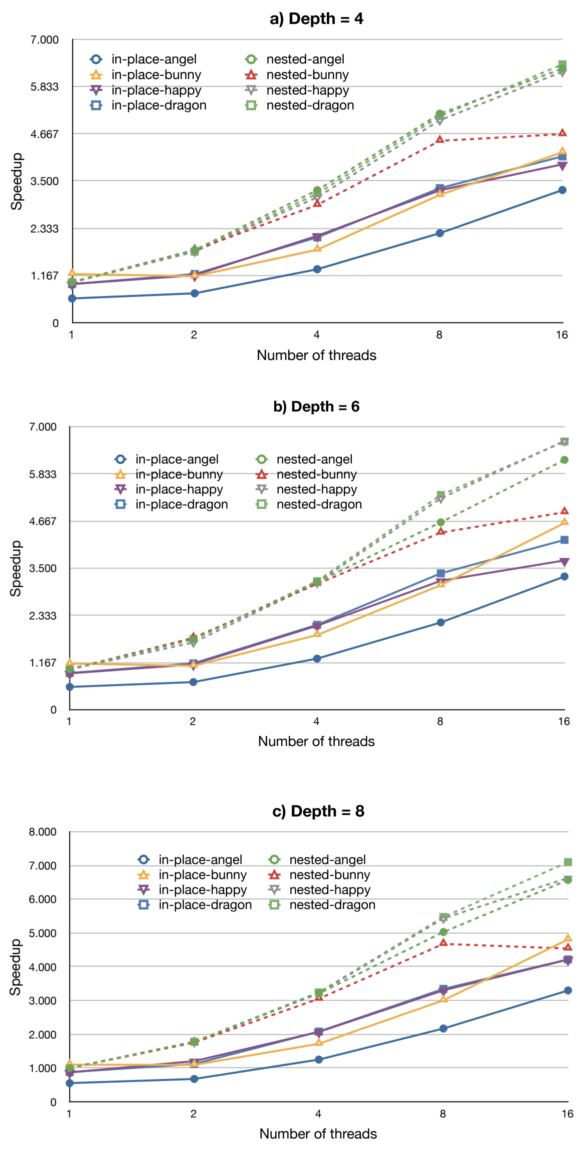


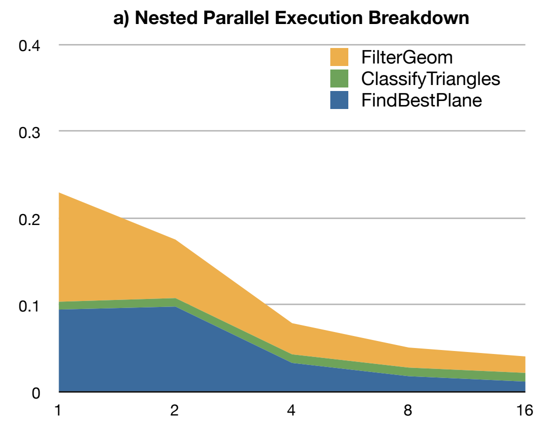
Figure 6. Speedup of the nested and in-place parallel algorithms relative to the best sequential algorithms for four inputs and tree depths = 4, 6, and 8 in parts (a), (b), and (c) respectively.

The figure shows that the in-place algorithm achieves speedup up to 4.8X and the nested algorithm achieves speedup up to 7.1X on 16 cores. These represent the first parallel speedup results for the upper levels of the kd-Tree construction using the high quality precise SAH-heuristic. Recall that the previously best known results for 16 cores show 1.8 to 2.65X speedup on a GPU, using median based heuristics, and for a full depth tree that includes node-level parallelism in the lower levels [24].

Surprisingly, the nested algorithm performs much better than the in-place algorithm (up to a factor of 2, for angel). We next analyze these results further to gain insight into the sources of speedup and limits to parallelism in both these versions.

9.2 Analysis

To further understand the performance of our algorithms, Figures 7 (a) and (b) plot the components of execution time for different core counts for the nested and in-place algorithms respectively.



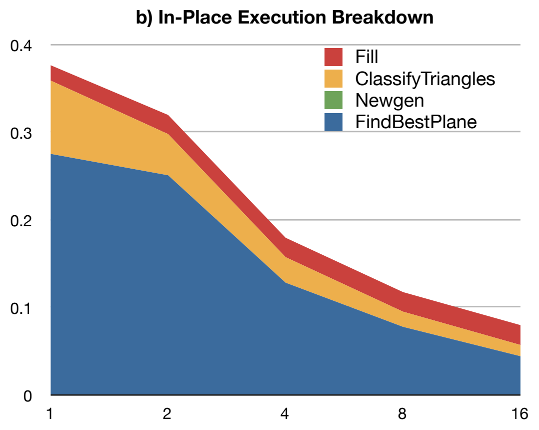


Figure 7. Contribution of different components of the (a) nested and (b) in-place parallel algorithms.

For the nested algorithm, the figure shows the breakdown of execution time for the computation for the root node (i.e., the first level of the tree) for a representative input, angel. Since this version exploits nested node- and geometry-level parallelism, the computation for nodes in the subsequent levels proceeds asynchronously and it is difficult to isolate the time devoted to a particular function. Recall that the nested algorithm with one core also represents the breakdown for our best sequential algorithm.

For the in-place algorithm, the figure shows the breakdown of execution time for the equivalent operation, i.e. computation for the root node of the tree for the same input. Our detailed results show that this behavior is representative of other levels as well.

Focusing on the nested version, we find that both the FindBestPlane and FilterGeom phases scale well. In FindBestPlane, the sequential version does not need to do the PreScan and so there is no speedup in that part from 1 to 2 cores, thereby limiting the overall speedup (however, the phase speeds up well after 2 cores).

For ClassifyTriangles, we found that parallelizing it produced worse results than the sequential version. This is because each task is sweeping through its chunk of contiguous events (along the splitting plane axis) and then updating the membership of the corresponding triangles (the left or right bits). Since the triangles are not in the same order as the events (the ordering would be different for different axes), this results in many random updates and significant false sharing, removing any performance benefit from parallelism. While this does not hinder speedup for small numbers of cores, we find that this phase could start to matter for larger number of cores (an element not seen in the in-place algorithm).

Focusing on the in-place algorithm, we find some intersting behavior. The main reason for the in-place algorithm to do worse is that it does a lot more work in the FindBestPlane phase for a single core. By design, it does much less for ClassifyTriangles, but this advantage is not enough to overcome the added work in the former phase for small numbers of cores. In particular, in the FindBestPlane phase, the in-place algorithm suffers from two problems: (1) it makes full sweeps over the entire geometry at every level versus the nested algorithm which splits the geometry at each level potentially resulting in better cache behavior per task. (2) The membership structures that it accesses contain node indexes rather than more compact bits of the nested version creating further cache misses.

However, for larger number of cores, we find that the ClassifyTriangles phase of the nested version is starting to dominate whereas the in-place version still scales. (The fill part of the in-place occurs once at the end and can be amortized with deeper levels and parallelized.) It remains to be seen if this self-relative scalability of the in-place algorithm might translate into better overall speedup with larger number of cores. Further, the interaction of SIMD instructions and the streaming nature of our algorithms also remains to be explored.

Finally, we note that all our algorithms (including the best sequential) assume a sorted array of events as input – parallel creation of this input is outside the scope of this paper, but also needs to be explored.

10. Conclusion

We have shown that existing parallel algorithms for k-D tree construction do not scale in quality, and soon will generate k-D trees that will degrade rendering performance when run on 16 or more cores. We presented two new algorithms for generating k-D trees that scale better, and are specifically focused on the geometry parallelism needed to generate enough sub-trees to allow each core to process the remaining tree components independently. To our knowledge, these algorithms provide the best known speedups for precise SAH based high quality kd-tree construction, relative to a sequential case that is better than the best publicly available code.

Our in-place data structure avoids the data movement that has plagued other parallel ray tracing acceleration structures. This algorithm invoked an interesting tradeoff. It yielded worse cache hierarchy performance for higher parallelism. The result is an algorithm that does much worse than the previous sequential best, but scales well enough to compensate for this initial handicap. Whether it will outperform the nested algorithm for larger number of cores remains an open question.

We demonstrated both algorithms on a variety of models ranging from tens to hundreds of thousands of polygons, showing that fast and real-time k-D tree construction is feasible for such large models, and in the future for self-scanned avatars and other user-created dynamic content.

We implemented our approaches using a task-parallel shared memory multicore processing enabled by Intel's Threaded Building Blocks. Several of the operations, especially the prefix-sum scan operations, map well to data parallel processing and the wide vectors available on upcoming CPU and GPU architectures should provide even further improvements in acceleration and scalability. The streaming memory operations available on a GPU could overcome cache problems we encountered during our geometry-parallel streaming of triangle and event data over the breadth of a k-D tree level. For maximum GPU efficiency these algorithms would need to avoid conditional program flow that especially occurs during the in-place processing of each triangle's nodes membership list which varies in length.

In conclusion, we believe our work opens up new possibilities for precise SAH based full kd-tree construction where previous literature had virtually abandoned the use of precise SAH for the increasingly dominant "upper" tree levels.

Acknowledgments

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

[1] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland U., 2006.

[2] J. Bigler, A. Stephens, and S. Parker. Design for parallel interactive ray tracing systems. In *Proc. Interactive Ray Tracing*, 2006.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[4] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. Graphics Interface*, pages 203–209, 2006.

[5] I. Coporation. Intel threading building block tutorial. 2009. URL `http://softwarecommunity.intel.com/isn/downloads/softwareproducts/pdfs/301132.pdf`.

[6] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proc. Graphics Hardware*, pages 15–22, 2005.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] A. Godiyal, J. Hoberock, M. Garland, and J. C. Hart. Rapid multipole graph drawing on the gpu. *(Proc. Graph Drawing), LNCS*, 5417:90–101, 2008.

[9] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE CG&A*, 7(5):14–20, 1987.

[10] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. Toward a multicore architecture for real-time ray-tracing. In *Proc. MICRO*, pages 176–187, 2008.

[11] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. URL `http://www.cgg.cvut.cz/ havran/phdthesis.html`.

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

[13] D. Luebke and S. Parker. Interactive ray tracing with cuda. Technical report, NVIDIA, 2008.

[14] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–65, 1990.

[15] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[16] S. Popov, J. Gnther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of sah kd-trees. In *Proc. IEEE Sym. Interactive Ray Tracing*, pages 89–94, 2006.

[17] L. C. Robert L. Cook, Thomas Porter. Distributed ray tracing. *(Proc. SIGGRAPH), Comp. Graph.*, 18(3):137–145, 1984.

[18] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *(Proc. SIGGRAPH), ACM Trans. Graph.*, 27(3):#18, 2008.

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

[20] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. Parker. An application of scalable massive model interaction using shared memory systems. In *Proc. EG Sym. on Par. Graph. and Vis.*, 2006.

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

[22] T. Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, 1980.

[23] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *(Proc. SIGGRAPH), ACM Trans. Graphics*, 24(3):434–444, 2005.

[24] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *(Proc. SIGGRAPH Asia), ACM Trans. Graph.*, 27(5):#126, 2008.