Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic

Warren Hunt*
University of Texas at Austin
Intel Corporation

William R. Mark†
University of Texas at Austin
Intel Corporation

Gordon Stoll‡
Intel Corporation

ABSTRACT

Construction of effective acceleration structures for ray tracing is a well studied problem. The highest quality acceleration structures are generally agreed to be those built using greedy cost optimization based on a surface area heuristic (SAH). This technique is most often applied to the construction of kd-trees, as in this work, but is equally applicable to the construction of other hierarchical acceleration structures. Unfortunately, SAH-optimized data structure construction has previously been too slow to allow per-frame rebuilding for interactive ray tracing of dynamic scenes, leading to the use of lower-quality acceleration structures for this application. The goal of this paper is to demonstrate that high-quality SAH based acceleration structures can be constructed quickly enough to make them a viable option for interactive ray tracing of dynamic scenes.

We present a scanning-based algorithm for choosing kd-tree split planes that are close to optimal with respect to the SAH criteria. Our approach approximates the SAH cost function across the spatial domain with a piecewise quadratic function with bounded error and picks minima from this approximation. This algorithm takes full advantage of SIMD operations (e.g., SSE) and has favorable memory access patterns. In practice this algorithm is faster than sorting-based SAH build algorithms with the same asymptotic time complexity, and is competitive with non-SAH build algorithms which produce lower-quality trees. The resulting trees are almost as good as those produced by a sorting-based SAH builder as measured by ray tracing time. For a test scene with 180k polygons our system builds a high-quality kd-tree in 0.26 seconds that only degrades ray tracing time by 3.6% compared to a full quality tree.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: approximation, kd-tree, error, heuristic

1 INTRODUCTION

Recent algorithmic improvements in ray tracing have made interactive and even real-time ray tracing a reality on modern processors. The fastest systems, however, are all restricted to static walkthroughs without fully dynamic geometric content. This limitation is imposed by the systems' reliance on pre-computed kd-trees that are cost optimized using the surface area heuristic (SAH) [6, 13, 8]. These data structures have generally been too expensive to re-compute each frame for all but very small models. Attempts have been made to speed up the process of optimized kd-tree construction [21] but these have met with limited success. The community has instead tended toward abandoning SAH-optimized kd-trees in favor of structures that can be rebuilt very rapidly, such as uniform grids or less-optimized trees, or that can be updated for some forms of animation rather than completely rebuilt, such as BVHs [20, 22, 12, 18]. Although such structures typically have poorer traversal performance, the improvement in build performance has been enough to overcome this disadvantage.

For many applications, a rendering system must perform at real-time frame rates and support fully dynamic scenes. If optimized kd-tree construction were sufficiently fast, fast kd-tree-based ray tracing algorithms such as MLRTA [16] could be leveraged for these applications. We present a fast scanning-based approach for constructing kd-trees that are nearly optimal according to the SAH. The approach takes a fixed number of well-chosen points at which to evaluate the SAH cost function and produces a piecewise quadratic approximation of the split-cost function from which a minimum can be obtained. The points are chosen in such a way that the difference between the approximation and the actual cost function is bounded. The scanning technique takes full advantage of SIMD operations (e.g., SSE) and has favorable memory access patterns, and in practice outperforms current approaches that use sorting. Because the approximation may be computed so quickly, for scenes with large numbers of polygons the slight increase in trace time is more than made up for by the decrease in kd-tree construction time. This algorithm assumes the kd-tree is constructed from a "soup" of axis-aligned bounding boxes (AABBs). Each AABB may contain one or more triangles or any other desired leaf geometry.

Our analysis and implementation focus on kd-trees, and the most immediate contribution is the demonstration that SAH-optimized kd-trees are a viable acceleration structure for dynamic scenes. However, the techniques described in this paper are directly applicable to the construction of other high-quality acceleration structures based on the surface area heuristic. Examples include SAH-optimized bounding-volume hierarchies [20] and B-KD trees [24, 18]. The most important conclusion from this paper is that acceleration structures for interactive ray tracers can and should be built using a good approximation to the SAH.

2 BACKGROUND

The traditional algorithm for building a kd-tree (summarized in [14]) is a greedy, top-down algorithm using the SAH to evaluate split candidates. Given a split candidate, SAH is a heuristic that takes into account both the probability of a ray hitting each child and the cost of visiting each child. The split with lowest cost across all split candidates is compared to the cost of not splitting. If the cost of splitting is less than the cost of not splitting, the current voxel is split into two children. Otherwise, a leaf node is created in the tree. Splits chosen using this cost function produce trees that have proven to be extremely effective for fast ray-tracing. The cost function $cost(x)$ [6, 13, 8] itself used by the SAH is defined as follows:

$$cost(x) = C_I + C_L(x)\frac{SA_L(v,x)}{SA(v)} + C_R(x)\frac{SA_R(v,x)}{SA(v)}$$

Where C_I is the (constant) cost of traversing the node itself, $C_L(x)$ is the cost of the left child given a split a position x and $C_R(x)$ is the cost of the right child given the same split. $SA_L(v,x)$ and $SA_R(v,x)$ are the surface areas of the left and right children respectively. $SA(v)$ is the surface area of the voxel currently being considered for splitting. The ratio $\frac{SA_L(v,x)}{SA(v)}$ and $\frac{SA_R(v,x)}{SA(v)}$ can be in-

*e-mail: whunt@cs.utexas.edu
†e-mail: billmark@cs.utexas.edu
‡e-mail: gordon.stoll@intel.com

terpreted as the probabilities of intersecting the left and right child respectively given that a ray has intersected the current node.

The split candidates are defined by the geometry that is being partitioned. In this paper we will assume that each piece of geometry is contained within an axis aligned bounding box (AABB). The upper and lower bounds of each AABB make up the list of candidate splits. In this case $C_L(x)$ and $C_R(x)$ are equal to the number of AABBs that overlap the left and right children respectively

The difficult part in evaluating the cost function for a given candidate x is in the evaluation of $C_L(x)$ and $C_R(x)$. When evaluating all split candidates, this task is simplified greatly if all of the splits are sorted with respect to their position (in one dimension) because a linear scan can keep track of the number of AABBs to the left of this split position and how many overlap it. This information is enough to derive C_L and C_R. Traditionally, the split candidates are sorted each time a new split is required, leading to an $n^2 \log n$ tree-building algorithm. A variant described by [21] and [4] does all of the sorting up front and achieves a better bound of $n \log n$. With even modest amounts of geometry, any sort is still expensive if it is to be computed each frame.

Another approach is to evaluate split candidates at arbitrary planes in space rather than at the boundaries of the AABBs [10, 9, 15]. The algorithm in this paper differs from previous approaches in this area in two primary ways: first, we choose split candidates adaptively, making more efficient use of each candidate. Second, we do not use the splits evaluated as candidates but instead as sample points for an approximation function from which a candidate is derived.

Regular grids and BVHs have recently been used in several interactive ray-tracing systems [20, 22, 12, 18]. The primary motivation for these approaches has been fast build and update time. Wald, et al. [22] discuss a grid based approach which re-builds the grid each frame. Grid acceleration structures are very fast to compute but cannot easily adapt to scenes with varied geometric density. Although major improvements have recently been made for grid acceleration structures [22], they still typically perform somewhat worse for traversal than kd-trees, particularly for less-coherent secondary rays. BVHs have also been studied for interactive ray tracing with some promising results. The key advantage for BVH based acceleration structures for real-time applications is that a fast update procedure exists. Drawbacks with this approach are that updates can only be used for a limited class of animation (e.g. deformation), and that tree quality can degrade over time. Both of these drawbacks can be addressed using full rebuilds of the tree, and rebuilding a BVH with high quality (e.g. SAH-optimized) is closely related to the problem addressed in this work.

3 Evaluating the cost function: Sorting vs. Scanning

Any algorithm that builds an acceleration structure using SAH cost must evaluate the cost function at various locations.

To evaluate the cost function at a particular location, we must know:

1. The location of the split plane (which allows us to compute the volumes of the left and right nodes, and hence the probability of hitting them)
2. How many primitives are to the left
3. How many primitives are to the right

Tasks #2 and #3 are expensive.

There are two basic algorithms for evaluating the cost function. For each, we assume that we have m AABBs at a particular node, and that we wish to evaluate the cost function at q different locations along a single axis of that node.

Sorting approach: The sorting approach consists of two phases. In the first phase, the primitives are sorted along the axis, at a cost of $O(m \log m)$. In the second phase, the cost function can be evaluated for any number of desired locations with only a single pass over the sorted data, at a cost of $O(m)$. For the kd-tree as a whole, the cost of this approach is $O(n \log^2 n)$, but by preserving and reusing the results of the top-level $O(n \log n)$ sort, the total cost can be reduced to $O(n \log n)$ [21].

Scanning approach: The scanning approach evaluates the cost function at just a single location. In its simplest form, the algorithm must be repeated to evaluate the cost function at more than one location. For each location, the algorithm loops over all of the primitives. At each primitive, it checks to see if the primitive lies to the left and/or to the right, and then increments the appropriate counter(s). Thus, to evaluate the cost function at a single location, the cost of this approach is $O(m)$. For q locations, the cost is $O(mq)$. If we set q to a constant (e.g., eight), then the approach is $O(m)$ for each node, and for the tree as a whole the cost will be $O(n \log n)$.

From asymptotic analysis, it would appear that the costs of the two approaches are equivalent. Since the sorting approach evaluates the cost function at every AABB rather than just a fixed number of locations, it would initially appear to be the better approach.

However, there are several practical advantages of the scanning approach. First, it is very simple and thus the constant factors in its cost can be very small, especially when the implementation is well-tuned (as described later in this paper). Second, the scanning approach defers more of its work to the leaf nodes.

Deferring work to the leaf nodes has two advantages. First, it may be the case that we do not need to build all of the leaf nodes. If a system builds the acceleration structure lazily (as described, for example, in [3, 17]) then it can skip a large amount of work. In contrast, the sorting approach still must do an $O(n \log n)$ sort at the top level. The results reported in this paper do not use lazy building, but we hope to report such results in a future publication. Second, the scanning approach performs well at higher levels of the tree where the data set does not fit in cache, whereas the constant factors on sorting algorithms are generally worse for such large data sets. In particular, the linear and read-only memory access pattern of the scanning approach allows the hardware prefetcher in modern CPUs to work very effectively.

4 Approximating the cost function with a few samples

The scanning approach to evaluating the cost function can perform better than the sorting approach if the number of locations at which the cost function is evaluated is small. Fortunately, a small number of locations is generally sufficient to build a high-quality kd-tree. For example, Hurley, et al. [10] found that there was very little benefit to using more than thirty-two candidate split locations.

In previous work, split planes have been restricted to lying on the locations at which the cost function has been evaluated [10, 18, 9, 15]. We show that it is possible to use a small number of evaluations of the cost function to generate an approximation of the true cost function. The final split plane is then positioned at the minimum of the approximate cost function. Thus, the location of the final split plane is not restricted to a fixed number of locations. Figure 1 illustrates this idea.

A good approximation to the full surface area heuristic must meet two criteria. First, it must significantly reduce the time required to build the acceleration structure. We assess this criteria with execution-time measurements for our algorithm. Second, it should not significantly reduce the quality of the acceleration structure.

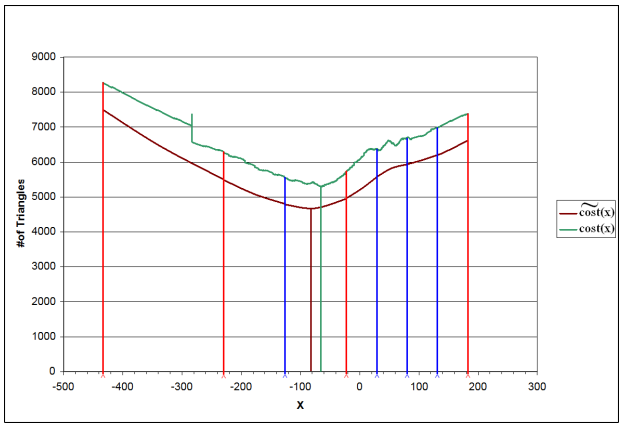

Figure 1: The split plane is placed at the minimum of a piecewise quadratic function that interpolates the sample points. Note that we have vertically displaced the approximation function from the actual function so that the details of both can be seen.

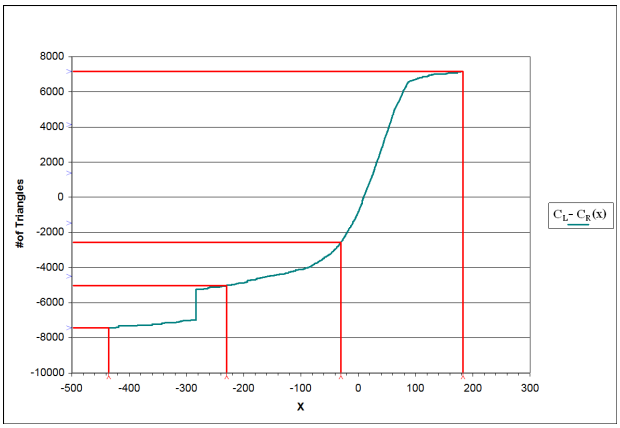

Figure 2: The initial samples as shown on $C_L - C_R$

We assess the quality of the acceleration structure in three ways: First, we measure the increase in ray tracing time that results from using our approximate tree instead of one built using the full SAH. Second, we measure the degradation in SAH tree quality according the SAH cost equation. Finally, in Appendix A we address quality mathematically, by analyzing the behavior of the SAH cost function and deriving analytical bounds on the error that results from evaluating it at a fixed number of locations.

5 Adaptively Choosing Sample Locations

Previous approaches that approximate the cost function have chosen uniformly spaced samples, as shown in Figure 2.

In this paper, we show that it is possible to achieve a better approximation to the cost function by adaptively choosing sample locations. This approach has been independently suggested (but not implemented) by Popov, et al. [15].

Figures 2 and 4 illustrate how we choose samples adaptively. In a first phase, 50% of our sample budget is used to uniformly sample the function. In a second phase, the remaining 50% of our sample budget is used to adapatively sample the function in locations where there is the greatest uncertainty about the behavior of the function.

5.1 Error bounds

For the purpose of discussing error bounds, it is useful to distinguish between the *greedy* SAH cost, and the *true* SAH cost. The true SAH cost recursively considers costs at all child nodes, but can only be evaluated after the entire sub-tree has been built. In contrast, the greedy SAH cost terminates its recursion for cost evaluation after one split, and ignores the effects of deeper tree structure.

It should be clear that our approximation is very effective in cases where the greedy cost function is smoothly varying as is typical near the top of the acceleration structure. In such smoothly-varying cases, the approximate cost function used by our algorithm closely matches the actual greedy cost function at all points. Thus, the greedy cost of the split plane chosen by our algorithm is very close to the greedy cost of the optimal split plane.

It is less obvious how well our algorithm performs in cases where the cost function has discontinuities. For example, if the portion of the scene in question contains an axis-aligned wall with many polygons, the actual greedy cost function may be a step function or hat function. This is visible in the example diagrams on the left side. In such cases, the greedy cost of the split plane chosen by our algorithm may differ significantly from the greedy cost of the optimal split plane, specifically by as much as half of the range of the cost function. Because a step function has a large high frequency component (up to infinite), any discrete sampling strategy suffers this deficiency.

However, it turns out that in such cases the *greedy* SAH cost does not correspond well to *true* (non-greedy) SAH cost if the next few child splits are chosen well. Our algorithm chooses the child splits well, for reasons that we will explain next. As a result, the degradation in the *true* SAH cost caused by our algorithm's approximation is typically small even for a cost function with discontinuities.

A key property of our algorithm is that the sample points it chooses rapidly converge towards a discontinuity. The convergence occurs both through adaptive sampling within a single node, and through samples chosen recursively in child nodes. In fact, as we will show, the larger the discontinuity is, the more rapid the convergence. The net effect of this convergence is that after a few splits, the discontinuity will be isolated into a small volume (and hence generally a small surface area). Since the true SAH cost recursively considers the product of cost with surface area, the true SAH cost penalty for non-optimal split locations near a discontinuity is generally small, even when the greedy SAH cost penalty appears to be large.

Mathematically, the convergence property of our algorithm is expressed as a guaranteed bound on the product of the cost error with the split-plane position error. That is, if there is a large bound on the error in cost, there is a small bound on the error of the location of the split plane, and vice-versa. This falls out directly from the fact that the proof provides error bounds in terms of the product of the domain and the range of the function (or segment) being approximated. Thus, for very large bounds on cost error, as occur near a discontinuity in the cost function, the algorithm will choose split planes close to the discontinuity. After a few splits, the discontinuity will be isolated in a small volume, and thus contribute very little to the *true* (non-greedy) SAH cost.

5.2 Adaptive sampling algorithm

As we have discussed, our algorithm approximates the SAH cost function by sampling it at a fixed number of locations. Instead of sampling the cost function itself, we sample all four varying inputs to the cost function (C_L, C_R, SA_L, and SA_R). By linearly interpolating each input, we are able to generate a quadratic approximation to the cost function between each pair of sample points.

It would be possible to stop at this step and choose a split plane at the minimum of the piecewise-quadratic approximation to the

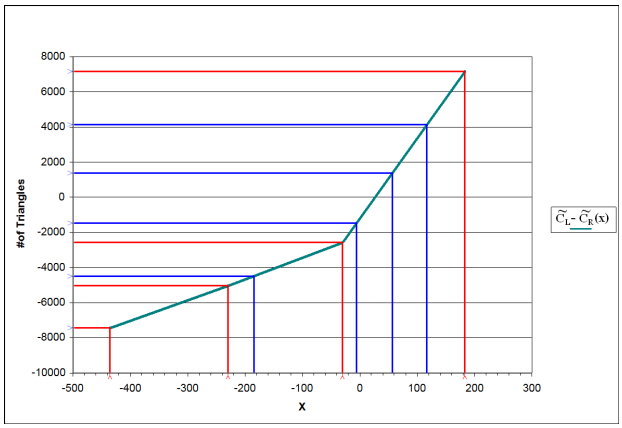

Figure 3: Use a comb sample along *range* of $C_L - C_R$ to find segments with large error.

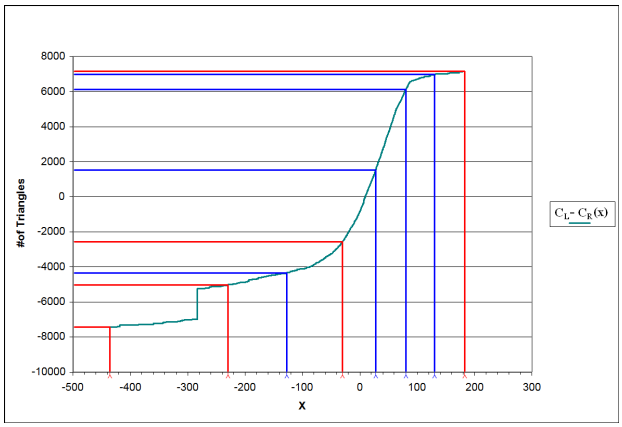

Figure 4: Sample evenly in the segments with large error.

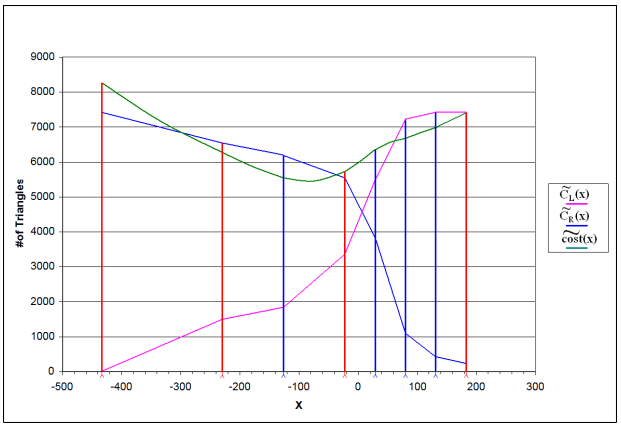

Figure 5: Approximate C_L, C_R and *cost*.

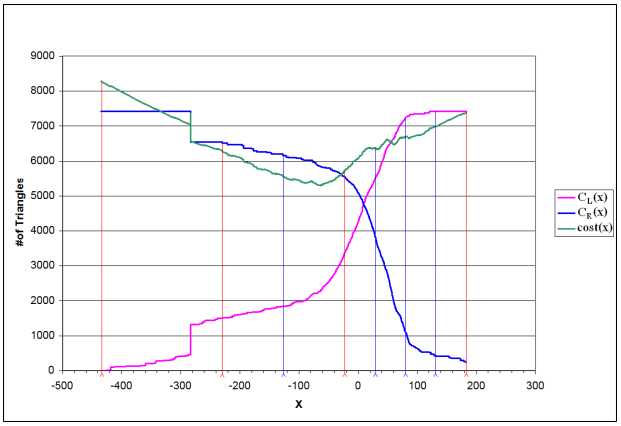

Figure 6: The actual cost function, notice the minimum is slightly off from the predicted value.

cost function. However, if the cost function is ill-behaved, the error bounds on the segment(s) that potentially contain the minimum may be loose. That is, there may be considerable uncertainty as to the actual behavior of the cost function in the vicinity of the minimum, and thus the choice of the location of the minimum may be poor.

Figure 2 illustrates this situation. Note that in this figure, rather than plotting the cost function, we plot $C_L - C_R$, which (as explained in Appendix A) is useful as a tool for error analysis due to the fact that bounds on $C_L - C_R$ simultaneously bound C_L and C_R and in turn bound $cost(x)$. In Figure 2, there is a large amount of uncertainty about the behavior of $C_L - C_R$ in the last segment.

To improve the error bounds, a second set of q samples is taken, with the locations for this second set of samples chosen adaptively based on the information from the first set of samples. We choose the sample locations using a simple algorithm which we illustrate graphically in Figures 2–6.

We want to place additional samples in those segments for which there is a large change in $C_L - C_R$ within the segment. A simple but effective mechanism for choosing these new sample locations is to create n bins for the values that $C_L - C_R$ may take, and then require an additional sample within a segment each time that $C_L - C_R$ crosses a bin boundary within that segment. This is equivalent to taking q additional samples regularly along the *range* of $C_L - C_R$, as illustrated in Figure 3.

Once it has been decided how many of the additional samples are allocated to each of the original segments, these additional samples are positioned such that they are evenly spaced within their respective segments (Figure 4).

The end result of the two sampling steps is a piecewise quadratic approximation to the cost function, using $2q$ samples. As shown in Appendix A, the product of cost error and position error has an $O(1/q^2)$ bound between adjacent pairs of samples with this adaptive sampling technique. This is important because it governs the rate of convergence for iterations of the sampling process. In this context, scans within children may be considered iterations.

To choose the split plane location for the node, we consider the local minima of each of the $2q-1$ piecewise quadratic segments, and place the split plane at the overall minimum. As with any kd-tree builder, if the estimated cost of splitting at this location is greater than the cost of not splitting, a leaf node is formed.

6 Implementation

6.1 Implementation of scan

As mentioned earlier, our algorithm for choosing a split plane does not sort the bounding boxes, but instead scans over them one or more times to accumulate various information.

In the simplest implementation of the algorithm, each evaluation of the cost function at a sample point requires a scan over the bounding boxes. Thus, evaluating the cost function at five sample points along the x-axis would require five separate scans over the bounding boxes. During a single scan, two quantities are computed:

- C_L – Number of primitives that lie partially or completely to the left of the sample point along the relevant axis.
- C_R – Number of primitives that lie partially or completely to the right of the sample point along the relevant axis.

To improve efficiency, it is possible to evaluate the cost function at multiple sample points in a single scan. For example, on modern machines with 4 x FP32 register SIMD support (e.g., SSE), it is possible to evaluate four sample points in one pass using inner-loop code of the form:

```
C_R += ((AABB.upperbound > sampleLocation) ? 1 : 0);
C_L += ((AABB.lowerbound < sampleLocation) ? 1 : 0);
```

where all of the variables are 4-wide vectors and all of the operators are 4-wide vector operators. The conditionals are implemented using masks rather than branches.

It is possible to evaluate the cost function at even more sample points in a single scan by maintaining additional pairs of containers for the C_L and C_R values associated with the additional sample points. Since each scan must read every AABB once, by combining scans we reduce the number of memory accesses. This mechanism is not limited to combining scans for a single axis – it may also be used to combine the scans for the X, Y, and Z axes. These scan-combining optimizations are most important at the top of the acceleration structure, where the AABBs do not fit in cache and the scans become memory-bandwidth bound. Note, however, that a minimum of two scans is required, one for the first set of samples, and another for the adaptive samples.

Toward the bottom of the acceleration structure, the decision as to how many samples to evaluate in a single scan becomes more complex. Once all of the AABBs fit in cache, the implementation on modern machines may become compute dominated. In this case, scan combining is still beneficial, but only up to the point where all of the temporary variables fit in registers. On today's 64-bit x86 architectures, for example, 16 samples may be evaluated in a single scan without any register spilling. Note that a scan organized in this way performs no writes to memory at all until it completes.

The compactness and high speed of the scanning algorithm make the approach fast even for large amounts of geometry. The scanning approach also lends itself to a parallel implementation where each CPU gets an allocation of the AABBs over which to evaluate the splits. Once each CPU has completed its evaluation, a gather operation can be used to combine the results. We have not yet implemented this form of parallelization.

6.2 Exact evaluation of SAH near leaf nodes

Near the bottom of the tree the number of AABBs in a node is small and the SAH cost function is generally much less smooth. Under these conditions approximating the SAH is less effective, and it is advantageous to evaluate it at the start and end of every AABB. Fortunately, exact evaluation is inexpensive for a small number of AABBs.

After experimenting with various sorting algorithms, we concluded that a modified version of our vectorized scanning algorithm performed better than any standard sorting algorithm for a small number of AABBs. Rather than choosing the sample points uniformly, we generate a sample point for the start and end of each AABB in the node, and use our scanning algorithm to evaluate the cost function at each sample point. The results from this algorithm are equivalent to the standard sorting approach. Asymptotically this algorithm is $O(m^2)$ in the number of AABBs, but for small m it is faster than the $O(m \log m)$ sorts that we have tried.

6.3 Implementation of overall algorithm

There are a number of ways in which our approach can make a tradeoff between tree-construction time and tree quality. The following are the mechanisms for adjustment, along with the values we use for them:

- Number of uniform samples per axis at each node: 8
- Number of adaptive samples per axis at each node: 8
- Node size below which we switch to evaluating SAH for all axes vs. evaluating just along longest axis. We report three sets of results:
  - *oneaxis*: always use one axis
  - *hybrid*: use one axis for more than 1024 AABBs, all axes for less or equal
  - *allaxes*: always use all axes
- Node size below which we switch to exact SAH evaluation: $\leq$36 AABBs
- Relative node weight – i.e. SAH cost of intersecting a kd-tree node vs. cost of intersecting an AABB in a leaf node: one-to-one
- Empty space bonus – Hurley, et al. [10] showed that it is useful to weight the SAH more heavily in favor of cutting off empty space: We use a cost weighting factor of 0.85 for any node with an empty sibling.

6.3.1 Additional optimizations

A kd-tree builder must perform two tasks at each node. First, it must choose a split plane (and evaluate any associated cost functions). We refer to this first task (which has already been discussed, although not by this name) as *splitting*. Second, it must create two lists of child nodes based on the chosen split plane. We refer to this second task as *sifting*. Note that for a kd-tree, a single AABB may be copied into both children, so sifting is more complex for a kd-tree than a B-KD tree [24, 18].

Our sifting implementation uses two important optimizations. First, memory is managed carefully, with the left child overwriting the parent storage, and the right child allocated linearly from a heap. Second, we copy pointers to AABBs instead of the actual AABBs. Although this strategy necessitates a pointer de-reference when reading AABB bounds, the performance improvement resulting from reducing the size of memory copies outweighs the performance penalty of pointer de-referencing.

7 RESULTS

We have implemented a SIMD-vectorized (using SSE) version of our kd-tree building algorithm as a stand alone package and evaluated it both for build performance and tree quality. We then implemented the same algorithm within the Razor system [17] in order to evaluate the practical effect of tree quality on render time. The results in Table 1 demonstrate that our algorithm's kd-tree build performance is competitive with other interactive acceleration structure builds and that the quality of the resulting kd-trees is almost as good as that produced by a sorting method.

Although the Razor system normally uses a lazy kd-tree builder, it was modified to force a full (non-lazy) tree build to make the measurements presented in this paper. Rendering time was measured with the system configured to use large numbers of both primary rays (4x super sampling) and secondary rays (area light sources).

Because Razor has some fixed per-ray overhead for multiresolution ray tracing, a non-multi-resolution ray tracer would likely see somewhat larger percentage changes in rendering times than those shown in Table 1. This is one of the reasons we also report tree quality using SAH cost.

Several of the results in Table 1 are outliers deserving further discussion. First, the `Courtyard` scene has an especially low SAH cost (i.e. high tree quality). We attribute this behavior to the fact that the scene is composed of 4×4 axis aligned instances of a single sub-scene. Second, the `Soda Hall` scene has high SAH cost (i.e. low tree quality) and a fast build time. We believe this behavior is caused by the large wall polygons in the scene which cause the SAH to terminate earlier, producing larger leaf nodes. Finally, Table 1 is missing rendering time results for the `Armadillo` and `Soda Hall` scenes due to memory-consumption issues in the current implementation of Razor.

All of the results we present are for builds from an AABB "soup". Preliminary experiments indicate that lazy building and the use of pre-existing hierarchy (e.g., from a scene graph) can be combined with the algorithm presented in this paper to yield further dramatic improvements in build times. We expect to report these results in a future publication. It should be noted that the algorithm presented in this paper benefits strongly from lazy building, because most of its time is spent on nodes near the bottom of the tree (fewer than 100 elements), which are the ones most readily culled in a lazy build.

We have not yet exhaustively experimented with all of the parameters in our algorithm. In particular, we have not carefully evaluated the impact of changing the number of fixed and adaptive samples (currently eight and eight respectively), because in an optimized implementation these settings cannot be changed arbitrarily. The current settings provide a good balance between build time and tree quality, but changing them would allow additional tuning of this tradeoff.

Our algorithm switches to exact evaluation of the SAH for nodes below a certain size (currently 36 AABBs). This cutoff was chosen to lie at the point where the cost of $O(m^2)$ brute-force search grew to be greater than that of the $O(m)$ two-pass adaptive sampling algorithm.

Figure 7: The Courtyard Scene (character models ©2003-2006 Digital Extremes, used with permission)

Figure 8: The Daz Fairy Scene

8 Discussion

We have set out to demonstrate that SAH based acceleration structures – and in particular kd-trees – are a viable acceleration structure for interactive and real-time ray-tracing of dynamic scenes. With a build time of 0.26 seconds for a 180k triangle model, interactive performance is already a reality. We believe that kd-trees and other SAH-based acceleration structures will turn out to be the most effective acceleration structure for real-time ray-tracing applications. Build times can be improved even further with additional parallelization, low level optimizations and the future use of additional information, such as pre-sorted geometry or scene graph hierarchy during construction. The advantages of high-quality acceleration structures are especially important for ray tracers that trace large numbers of secondary rays, since in these systems it is especially favorable to increase build time slightly in order to decrease traversal time.

Another important feature of this algorithm is that it interacts favorably with lazy building of the acceleration structure. In a lazy system with n pieces of geometry of which only m, $m << n$ pieces of geometry are touched, the size of the tree produced is $m \log m$. However, any system that sorts must sort the entire n pieces of geometry for the first split, leading to an $n \log n$ build algorithm. Since our approach only does linear work at each step, it only takes $n + m \log m$ time to build the entire tree. (The n accounts for the initial pass.) This asymptotic improvement in runtime may provide a substantial improvement in the build times for lazy systems.

9 Acknowledgments

Peter Djeu made changes to the Razor code base that were needed to make accurate rendering-time measurements and to allow the use of new test scenes. Jim Hurley and Bob Liang supported and encouraged this work. We thank the rest of the Razor team (Peter Djeu, Rui Wang, Ikrima Elhassan) for building the Razor infrastructure that made this work possible, and Don Fussell for early conversations about kd-tree building and ray tracing. The anonymous reviewers, Peter Djeu, and Paul Navratil provided careful and detailed feedback that was very helpful in revising this paper.

Scene						
	Hand [2]	Bunny [1]	Fairy Forest [2]	Courtyard	Armadillo [1]	Soda Hall
Geometric Information						
Quads	7,510	0	78,201	141,657	0	0
Triangles	835	69,451	17,715	37,574	345,944	1,510,775
Total Polygons	8,345	69,451	95,916	179,231	345,944	1,510,775
Build Time in Seconds						
All Axes	0.024	0.25	0.30	0.69	1.41	5.14
Hybrid	0.022	0.23	0.27	0.63	1.14	4.50
One Axis	0.012	0.11	0.14	0.26	0.69	1.59
Build speed in polygons/second						
All Axes	348,000	278,000	320,000	260,000	245,000	294,000
Hybrid	379,000	302,000	355,000	284,000	303,000	336,000
One Axis	695,000	631,000	685,000	689,000	501,000	950,000
Tree Quality, Measured as SAH Cost						
All Axes	70.0	76.0	53.6	39.3	63.8	452
Hybrid	70.0	76.0	59.6	39.9	73.3	450
One Axis	72.0	95.0	69.0	43.7	84.6	489
Increase in Render Time Over Optimal Tree						
All Axes	0.33%	1.63%	3.19%	3.31%	-	-
Hybrid	0.89%	1.63%	3.77%	3.60%	-	-
One Axis	7.90%	7.72%	4.35%	3.60%	-	-

Table 1: Build Performance Measurements, using one core of a 2.4 GHz Intel Core 2 Duo processor ("Conroe"), with 4MB of L2 cache

REFERENCES

[1] Stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[2] Utah 3D animation repository. http://www.sci.utah.edu/~wald/animrep.

[3] Sigal Ar, Gil Montag, and Ayellet Tal. Deferred, self-organizing bsp trees. In *Eurographics 2002*, 2002.

[4] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[5] Andrew Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[6] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5), 1987.

[7] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3), September 2006. (Proceedings of Eurographics) (to appear).

[8] Vlastimil Havran and Jiri Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, February 2002.

[9] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On fast construction of spatial hierarchies for ray tracing. In *Proc. 2006 IEEE Symp. on Interactive Ray Tracing*, 2006.

[10] Jim Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov. Fast ray tracing for modern general purpose CPU. In *Graphicon 2002*, 2002.

[11] Thomas Larsson and Tomas Akenine-Möller. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. Technical Report MDH-MRTC-92/2003-1-SE, MRTC, February 2003.

[12] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Technical Report 06-010, Department of Computer Science, University of North Carolina at Chapel Hill, 2006.

[13] J. David MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface*, pages 152–63, 1989.

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

[15] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH kd-trees. In *Proc. 2006 IEEE Symp. on Interactive Ray Tracing*, 2006.

[16] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005. (SIGGRAPH 2005).

[17] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An architecture for dynamic multiresolution ray tracing. Technical Report 06-21, University of Texas at Austin Department of Computer Sciences Technical Report, 2006.

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

[19] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE symp. on parallel and large-data visualization and graphics*, 2003.

[20] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, (conditionally accepted, to appear), 2006.

[21] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proc. 2006 IEEE Symp. on Interactive Ray Tracing*, 2006.

[22] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of ACM SIGGRAPH 2006, to appear).

[23] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proc. of Eurographics 2001).

[24] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Proc. of Graphics Hardware 2006*, September 2006. (to appear).

A Error Bounds

In order to bound the error of our approximation of the SAH cost function we first bound the error on each of its components. The focus of the rest of this section is providing error bounds for the piecewise linear approximations of the monotone functions C_L and C_R.

A.1 Linear Approximation

For an arbitrary integrable function f and an approximation $\widetilde{f}$, we may define the error of the approximation over domain (a,b) to be $\int_a^b |f-\widetilde{f}|$. If f is a monotone function and $\widetilde{f}$ is a linear approximation of f over (a,b) with $f(a)=\widetilde{f}(a)$ and $f(b)=\widetilde{f}(b)$, the maximum error of the approximation (according to this metric) is $\frac{|b-a||f(b)-f(a)|}{2}$. We do not provide proof of this simple theorem here, but Figure 9 should provide some intuition as to why it is true. We refer to this bound as the linear approximation bound (of a monotonic function).

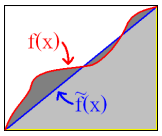

Figure 9: Monotonicity guarantees that the function does not leave the box. The area between the function and its linear approximation cannot be more than $\frac{1}{2}$ the area of the box, giving us our bounds.

A.2 Uniformly Spaced Linear Approximation

A uniformly spaced piecewise linear approximation provides a better error bound for monotone functions. When approximating a monotone function f we may take n evenly spaced points $x_i, i \in [0,n)$ **s.t.** $x_0=a$ and $x_{n-1}=b$. Let $\widetilde{f}$ be the piecewise linear approximation of f such that $\forall x \in [0,n) f(x)=\widetilde{f}(x)$ and $\widetilde{f}$ is linear between all x_i, x_{i+1} pairs. The following bound holds given the assumptions: f is an integrable monotone function, $n \geq 2$ and (a,b) is a non-empty range.

$$
\begin{aligned}
error(\widetilde{f}) &= \int_a^b |f-\widetilde{f}| \\
&= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} |f-\widetilde{f}| \\
&\quad \{\text{the linear approximation bound}\} \\
&\leq \sum_{i=1}^{n} \frac{|x_i-x_{i-1}||f(x_i)-f(x_{i-1})|}{2} \\
&\quad \{\text{f is monotone}\} \\
&\leq |\sum_{i=1}^{n} \frac{|x_i-x_{i-1}|(f(x_i)-f(x_{i-1}))}{2}| \\
&\quad \{x_i-x_{i-1}=\frac{b-a}{n-1}\} \\
&= \frac{(b-a)}{2(n-1)}|\sum_{i=1}^{n}(f(x_i)-f(x_{i-1}))| \\
&\quad \{f(x_i)\text{is a telescoping sum}\} \\
&= \frac{(b-a)|f(b)-f(a)|}{2(n-1)}
\end{aligned}
$$

This bound improves over the linear approximation bound by a factor proportional to $\frac{1}{n}$. We will refer to it as the evenly spaced piecewise linear approximation bound (of a monotone function).

A.3 Adaptive Linear Approximation

Although the evenly spaced piecewise linear approximation provides an improved error bound for $\widetilde{f}$, it provides no information about the locality of error. Each segment (x_i, x_{i+1}) contains some fraction α_i of the overall error in our piecewise linear approximation but we have no guarantee that any one segment doesn't contain most or all of the error. By adding at most n additional samples (using our adaptive method), we may ensure that each segment contains proportional to $\frac{1}{n}$ of the overall error.

Using the piecewise linear approximation bounds, adding $\lfloor n\alpha_i \rfloor$ evenly spaced points to a segment reduces its error proportional to $(\lfloor n\alpha_i \rfloor + 1)^{-1}$. It should be noted that two additional samples are avaliable in each segment: the endpoints. This accounts for the addition, rather than the subtraction, of 1 in this reduction of error. If the error was initially $\alpha_i \frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2(n-1)}$ after the additional samples it becomes $\alpha_i \frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2(n-1)(\lfloor n\alpha_i \rfloor+1)}$. This quantity is proportional to $\frac{(x_{i+1}-x_i)|f(x_{i+1})-f(x_i)|}{2n^2}$ providing us an error bound on each segment proportional to $\frac{1}{n^2}$. We will refer to this bound combined with the evenly spaced linear approximation bound as the adaptive linear approximation bound. $\sum_{i=1}^{n} \lfloor n\alpha_i \rfloor \leq \sum_{i=1}^{n} n\alpha_i = n$ proves that this process takes at most n additional samples overall.

A.4 Bounds for the Cost Function

Recall the SAH cost function:
$cost(x) = C_I + C_L P_L(x) + C_R P_R(x)$

Because C_L and C_R are monotone, we may sample them adaptively and achieve the adaptive linear approximation bound. C_I, $P_L(x)$ and $P_R(x)$ are computable directly and have no error. The error bounds for the cost function are derived here.

$$
\begin{aligned}
error(cost(x)) &= error(C_L)P_L(x) + error(C_R)P_R(x) \\
&\quad \{P_L, P_R \in [0,1]\} \\
&\leq error(C_L) + error(C_R) \\
&\quad \{f(x_i)\text{evenly spaced linear approximation bound}\} \\
&= \frac{(b-a)|C_L(b)-C_L(a)|}{2(n-1)} + \frac{(b-a)|C_R(b)-C_R(a)|}{2(n-1)} \\
&= \frac{(b-a)(|C_L(b)-C_L(a)|+|C_R(b)-C_R(a)|)}{2(n-1)}
\end{aligned}
$$

Slightly more work is required to achieve the adaptive bound for $cost(x)$ because we are choosing only one additional set of points and must bound error for both C_L and C_R. Since C_L and $-C_R$ are both monotone increasing, so is their sum. Also, since the range of $C_L - C_R$ is equal to the sum or the ranges of C_L and C_R, any bound for $C_L - C_R$ is also a bound for C_L and for C_R. Using the adaptive bound for $C_L - C_R$ we can show that the error between any two samples over $C_L - C_R$ is of the order $\frac{(b-a)(|C_L(b)-C_L(a)|+|C_R(b)-C_R(a)|)}{n^2}$. The same bound holds for the error between any two samples on C_L and C_R. Since adding the error of C_L and C_R together only increases overall error by a factor of two and P_L and P_R are ≤ 1. The error between any two points in $cost(x)$ is also order of $\frac{(b-a)(|C_L(b)-C_L(a)|+|C_R(b)-C_R(a)|)}{n^2}$. This completes the error analysis for using at most $2n$ samples with our adaptive scheme.