Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies

(revised version)

INGO WALD, SOLOMON BOULOS, and PETER SHIRLEY

University of Utah

Fig. 1. Screenshots from an animated 180,000 triangle scene with moving dragonfly, fairy, and plants. At 1024×1024 pixels the animated scene is ray traced at roughly 3.7 frames per second on a dual-2.6 GHz Opteron desktop PC including shadows and texturing.

The most significant deficiency of most of today's interactive ray tracers is that they are restricted to static walkthroughs. This restriction is due to the static nature of the acceleration structures used. While the best reported frame-rates for static geometric models have been achieved using carefully constructed kd-trees, this paper shows that bounding volume hierarchies (BVHs) can be used to efficiently ray trace large static models.

More importantly, the BVH can be used to ray trace deformable models (sets of triangles whose positions change over time) with little loss of performance. A variety of efficiency techniques are used to achieve this performance, but three algorithmic changes to the typical BVH algorithm are mainly responsible. First, the BVH is built using a variant of the "surface area heuristic" conventionally used to build kd-trees. Second, the topology of the BVH is not changed over time so that only the bounding volumes need be re-fit from frame to frame. Third, and most importantly, packets of rays are traced together through the BVH using a novel integrated packet-frustum traversal scheme. This traversal scheme elegantly combines the advantages of both packet traversal and frustum traversal, and allows for rapid hierarchy descent for packets that hit bounding volumes, as well as rapid exits for packets that miss. A BVH-based ray tracing system using these techniques is shown to achieve performance for deformable models comparable to that previously available only for static models.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*; I.3.3 [**Graphics Systems**]: Picture/Image Generation—*Display algorithms*

1. INTRODUCTION

Recent trends in computer architecture and model complexity, along with a desire for improved visual realism, have spurred researchers to consider ray tracing as an alternative to Z-buffering, and ray tracing has since been demonstrated to be a viable method for a wide class of interactive applications [Muuss 1995; Parker et al. 1999; Wald 2004; Reshetov et al. 2005]. Until recently these demonstrations have largely been restricted to static scenes; ray tracing *dynamic* scenes has not been able to yield such high frame rates.

This paper is a revised version of our SIGGRAPH 2006 submission, which got conditionally accepted at ACM Transactions on Graphics. Meanwhile, this document is available via a Technical Report by the SCI Institute, University of Utah, under TR number UUSCI-2006-023.

The reason ray tracing is problematic for dynamic scenes is that fast ray tracers use precomputed spatial search structures to achieve interactive frame rates. For most data structures, rebuilding these for each frame is too expensive for any but relatively small models [Wald et al. 2003]. Ray tracing's historical failure to deal with dynamic scenes is a major limitation because they are important for a large class of applications such as games and simulation [Mark and Fussell 2005]. Because of this importance, several concurrent approaches for ray tracing dynamic scenes are currently being pursued, for example [Wald et al. 2006; Lauterbach et al. 2006; Stoll et al. 2006; Günther et al. 2006; Carr et al. 2006].

In this paper we use a bounding volume hierarchy (BVH) [Rubin and Whitted 1980] to interactively ray trace a particular type of dynamic scene: *deformable scenes*. A deformable scene is one whose triangles move, but no triangles are split, created, or destroyed over time. An example of such a scene is shown in Figure 1 where two meshes deform and change position within an animated polygonal environment. The entire scene is ray traced with a single BVH whose topology is constant for the whole animation. This approach was motivated by the successful use of constant-topology BVHs for collision detection between deformable objects [van den Bergen 1997; Schmidl et al. 2004]. In contrast to spatial subdivision structures such as the kd-tree, the BVH subdivides the object hierarchy, and a given object hierarchy is more robust over time than a given subdivision of space. As a result, a BVH can be quickly updated between frames thus avoiding a complete per-frame rebuilding phase [Larsson and Akenine-Möller 2003]. However, a barrier to exploiting the BVH's advantages for dynamic scenes is that their performance on static scenes has lagged far behind that achieved using kd-trees [Wald 2004; Reshetov et al. 2005].

In this paper, we demonstrate how BVHs can be used for fast ray tracing of static models by using many of the same techniques developed for kd-trees including careful tree construction, SIMD programming, and the use of ray packets. In addition, we propose a novel traversal algorithm that benefits from ray packets that are much larger than the 4-ray packets commonly used (e.g., [Wald et al. 2001; Lauterbach et al. 2006]). The combination of these techniques allows a BVH to be competitive with a kd-tree even where kd-trees perform their best. By using a constant topology BVH that is only refit each frame, we can naturally extend our system to handle dynamic scenes. We empirically show — for at least a wide class of dynamic scenes — that the rendering performance of using a constant topology is close to the "optimal" case of a BVH built specifically for that frame. Our approach does not require knowledge of all frames of an animation, so it should be applicable to models driven by physics or user-interaction.

2. BACKGROUND

Ray tracing was used for rendering at least as early as the classic work by Appel [1968], but was introduced in its modern form by Whitted [1980]. To speed up intersection, hand-constructed bounding volume hierarchies (BVHs) [Clark 1976] were the first spatial efficiency structure used for ray tracing [Rubin and Whitted 1980; Whitted 1980]. While a BVH partitions objects, various schemes for partitioning space soon became more popular for ray tracing [Cleary et al. 1983; Glassner 1984; Kaplan 1985; Jansen 1986; Arvo and Kirk 1989]. Kirk and Arvo [1988] speculated that the best efficiency scheme varied with object characteristics and advocated a heterogeneous software architecture. The first modern, systematic investigation of the various efficiency schemes was conducted by Havran [2001] in his dissertation. He concluded that kd-trees [Bentley 1975] were probably the best, and BVHs were by far the worst data structures for ray tracing.

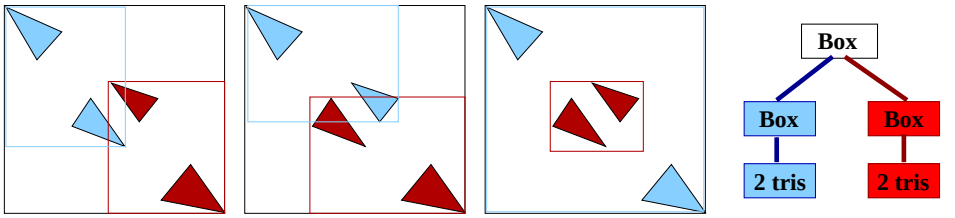

Fig. 2. Different BVHs for 4 triangles. The siblings are allowed to spatially overlap (unlike spatial subdivision). Other possibilities include splitting to size 1 and 3 triangle list, and recursively splitting lists of 2 or 3 triangles.

Interactive ray tracing. In the last decade a number of interactive ray tracing systems have been developed on a variety of architectures. Wald used kd-trees to achieve interactivity on both single PCs and clusters of PCs [Wald 2004]. His implementation was released as part of the OpenRT system, which we use as a comparison baseline in this paper. Interactive ray tracing has also been demonstrated on a variety of platforms including supercomputers [Parker 2002], FPGAs [Schmittler et al. 2002; Woop et al. 2005], GPUs [Purcell et al. 2002; Foley and Sugerman 2005; Carr et al. 2006], and the Cell [Minor et al. 2005].

Bounding volume hierarchies. BVHs are trees that store a closed bounding volume at each node. In addition, each internal node has references to child nodes, and each leaf node also stores a list of geometric primitives. The bounding volume is guaranteed to enclose the bounding volumes of all its descendants. Each geometric primitive is in exactly one leaf, while each spatial location can be in an arbitrary number of leaves. A variety of shapes have been used for the bounding volumes [Weghorst et al. 1984], with axis-aligned boxes a common choice. An example of different BVHs for a small model are shown in Figure 2.

Building BVHs. Goldsmith and Salmon [1987] used a cost model to optimize the bottom-up construction of BVHs. Top-down builds have also been used that split at the spatial median [Kay and Kajiya 1986] or the object median to force a balanced tree [Smits 1998] as often done in spatial databases [Guttman 1984]. Both top-down and bottom-up builds have also been used for collision detection applications [Larsson and Akenine-Möller 2005]. For kd-trees, a greedy top-down build based on Goldsmith and Salmon's cost model has been shown to be quite effective [Havran 2001; Hurley et al. 2002; Wald 2004]. A similar build has been used for BVHs [Müller and Fellner 1999; Mahovsky 2005], but has not resulted in performance competitive with kd-tree implementations even once improvements in hardware speeds are accounted for. Ng and Trifonov [2003] investigated randomized BVH construction, but only found modest improvements over other techniques.

Traversing BVHs. The BVH has a very simple recursive intersection routine. For leaves the ray is first tested for intersection with the bounding volume, and when positive, the list of triangles is tested. For internal nodes, when the ray hits the bounding volume, its two children are recursively called. Unlike spatial subdivision schemes, the two children are not spatially ordered, so the second child must be tested even when the first is hit. Haines [1991] and Mahovsky [2005] proposed schemes that reduced the numbers of tests by attempting to order the tests in at least some cases. Ray packets have been used with BVHs [Mahovsky 2005] and have resulted in speedups up to a factor of two relative to single rays. Other optimizations on serial efficiency for BVH traversal include different memory layouts [Smits 1998], faster ray-box overlap tests [Mahovsky and Wyvill 2004; Williams et al. 2005], and early exits for shadow rays [Smits 1998].

Dynamic models. Ray tracing for dynamic models has historically received little attention. Most research on animated sequences stresses exploiting the coherence within successive frames to reduce the number of rays to be traced [Gröller and Purgathofer 1991; Adelson and Hodges 1995]. The earliest paper directly related to animated ray tracing is the "space-time ray tracing" approach proposed by Glassner [1988], which used a heavy-weight data structure for batch ray tracing of known animation sequences. Parker et al. [1999] kept animated objects out of the overall acceleration structure, and intersected those separately. This allowed for animating several objects, but does not scale well. Reinhard et al. [2000] used an updateable grid data structure. Their method allows for a wide range of dynamic behavior but its efficiency is limited by the overall performance of the grid.

For hierarchical rigid-body deformations, Lext et al. [2001] proposed a two-level rapid reconstruction scheme. Though their scene update time is insignificant, their overall speed was not interactive. This idea was applied for kd-trees [Wald et al. 2003] and extended to more general animations, but was too costly except for small scenes. For point based models, Adams et al. [2005] used a deforming BVH of spheres. Larsson and Akenine-Möller [2003] proposed a method to incrementally update the BVH in sub-linear time. However, their performance for static models was low compared to kd-tree based systems. Carr et al. [2006] ray trace deformable geometry images using a balanced BVH on a GPU. They achieve good performance for small models, but their method does not yet scale well to large models. Concurrently with our work, several approaches to ray tracing dynamic scenes have been proposed for kd-trees [Stoll et al. 2006; Günther et al. 2006], grids [Wald et al. 2006], and BVHs [Lauterbach et al. 2006; Carr et al. 2006]. We compare these concurrent approaches to our own towards the end of this paper.

3. INTERACTIVE RAY TRACING WITH BVHS

We optimize our system using the two main strategies that have made kd-trees dominant: improving the structure of trees via cost functions (Section 3.1), and using a novel approach to tracing coherent packets of rays during tree traversal (Section 3.2).

3.1 Building effective BVHs

A general BVH can have an arbitrary branching factor and can use any type of bounding volume. Though there are many cases where oriented bounding volumes or high branching factors could be advantageous, we have pursued a "simplest is best" strategy and thus use only binary BVHs with axis-aligned bounding boxes (AABBs).

As with any tree-based search, the construction of a hierarchy for ray tracing fundamentally influences performance. For example, Figure 2 shows three of the seven different ways to partition a set of 4 triangles into a hierarchy of two subtrees. Each of these will result in different runtimes. The runtimes of kd-tree based implementations is greatly improved by careful tree construction [Havran 2001]. That construction is based on a greedy algorithm with the "surface area heuristic" (SAH) cost function [MacDonald and Booth 1989; Havran 2001]. Interestingly, the (SAH) cost function is derived from analysis first done by Goldsmith and Salmon [1987] for the optimization of BVHs. This section reviews the reasoning behind the SAH and shows how it can be applied to ray tracing using BVHs. As discussed in Section 2, various authors have advocated dividing objects evenly for a balanced tree, dividing space evenly, and using an SAH to minimize expected cost.

For BVHs, Goldsmith and Salmon [1987] developed a simple expression for the expected execution time of a random ray that hits the root node's bounding volume. They

use the well-known result from geometric probability that the probability of interacting with a particular node is the ratio of the surface area of that node's bounding volume to the surface area of the bounding volume of the root node [Santalo 2002]. This leads to the following global cost estimate of a tree:

$$T = \sum_{b \in \text{Internal Nodes}} 2\frac{A(b)}{A_{\text{root}}} T_{\text{AABB}} + \sum_{b \in \text{Leaf Nodes}} \frac{A(b)}{A_{\text{root}}} N(b) T_{\text{tri}}, \tag{1}$$

where T_{AABB} is the time to test a ray and an AABB for intersection, T_{tri} is the time to compute a ray-triangle intersection, $A(b)$ is the surface area of a node's bounding box, and $N(b)$ is the number of triangles in the list for leaf node b (see [Wald and Havran 2006]).

Macdonald and Booth [1989] developed a cost expression similar to Equation 1 for kd-trees. They argued empirically for a greedy top-down tree building strategy that recursively attempted to find the best two-leaf tree possible. A similar greedy strategy can be applied for BVHs: when building a BVH top-down, each recursive construction step consists of *partitioning* a set S of triangles into two subsets S_1 and S_2, and subdivided recursively until S is considered small enough to be made a leaf. In each step one chooses the partition that minimizes the cost as defined by Equation 1 for a two-leaf tree using that partition:

$$T = 2T_{\text{AABB}} + \frac{A(S_1)}{A(S)} N(S_1) T_{\text{tri}} + \frac{A(S_2)}{A(S)} N(S_2) T_{\text{tri}}, \tag{2}$$

where $A(S)$ is the area of the bounds of the triangles in set S, and $N(S)$ is the number of triangles in set S. Though very similar to the cost function used for kd-trees, there are several important differences. In particular, for a kd-tree there are $O(N)$ reasonable split candidates for splitting a node into two halves (see, e.g., [Wald and Havran 2006]), and the surface areas of left and right child are given by a function that is linear in the position of a potential split plane's position. For a BVH, however, there are $O(2^N)$ possible ways of partitioning N triangles into two subsets S_1 and S_2, and the surface areas of each is a complex function of the partitioning. Thus for BVHs it is not straightforward to apply the linear incremental splitting techniques used in kd-tree construction.

Instead of trying to find the *optimal* partition, we attempt to find a *good* partition by using a set of candidate axis-aligned planes to partition the triangles. For each such plane (we will discuss the choice of planes below), each triangle could be either left of, right of, or overlapping the plane. Since it is not obvious which set an overlapping triangle should be put into, we currently only consider the centroids of the triangles' bounding boxes. For every such generated partition, we exactly evaluate Equation 2, and select the partition with minimal expected cost.

For choosing sets of partitioning planes, we have investigated three schemes. First, we have used sets of evenly spaced planes in each axis. Second, we have used the sides of all the bounding boxes of the triangles as done in some kd-tree builds. Finally we have used the planes through the centroids of all the triangles. We found it somewhat surprising that the overall ray tracing speed resulting from these schemes is very similar. For the centroid-based method, there is a simple sub-quadratic build method which we detail in Algorithm 1 (which has an average-case complexity of $O(N \log^2 N)$). Investigating even faster build methods in the spirit of [Wald and Havran 2006] is an interesting avenue of future work; for the results in this paper, however, this was not required, as we only build the BVH once, and building a BVH for typical scenes of several hundred thousand triangles can be done in a few seconds (see Table I).

Algorithm 1 Centroid-based SAH partitioning

```
function partitionSweep(Set S)
  bestCost = T_tri*|S| {cost of making a leaf}
  bestAxis = -1, bestEvent = -1
  for axis = 1 to 3 do
    sort S using centroid of boxes in current axis

    {sweep from left}
    set S1 = Empty, S2 = S
    for i = 1 to |S| do
      S[i].leftArea = Area(S1) {with Area(Empty) = ∞}
      move triangle i from S2 to S1
    end for

    {sweep from right}
    S1 = S, S2 = Empty
    for i = |S| to 1  do
      S[i].rightArea = Area(S2)
      {evaluate Equation 2}
      thisCost = SAH(|S1|, S[i].leftArea, |S2|, S[i].rightArea)
      move Triangle i from S1 to S2
      if thisCost < bestCost  then
        bestCost = thisCost
        bestEvent = i
        bestAxis = axis
      end if
    end for
  end for

  if bestAxis = -1 then {found no partition better than leaf}
    return make leaf
  else
    sort S in axis 'bestAxis'
    S1 = S[0..bestEvent); S2 = S[bestEvent..|S|)
    return make inner node with axis 'bestAxis';
  end if
end
```

scene	ERW6	Conference	Fairy (1st frame)	Stanford Buddha	Soda Hall
#triangles	804	282,664	174,117	1,087,716	2,169,132
build time (s)	0.014	5.06	2.8	20.8	53.2

Table I. Times on a 2.6GHz Opteron for building an SAH-based BVH from scratch using Algorithm 1 for various different-sized scenes. Though not fast enough for interactive rebuilds, the build times are acceptable because the BVH is built only once and as a preprocess.

Performance Impact of an SAH-based BVH. The impact of using an SAH build versus the more commonly used object median and spatial median builds is shown in Table II. As can be seen, for small packet sizes the impact of a good SAH build can be significant, and can provide speedups of up to $2\times$ over spatial median, and up to $6\times$ over object median. For larger packet sizes, our combined packet/frustum traversal scheme (described in the next section) provides most of the speedup, and consequently lessens the impact of a good build. This in fact is quite fortunate, as it implies that this traversal scheme suffers less from not-as-optimal BVH builds. In particular, if an originally well-built BVH will get deformed during an animation, its deformed shape can significantly differ from its original build –

but if the difference between a good an a mediocre build is small, then the performance impact of deforming a BVH will be small as well.

Though Table II implies that for our packet traversal scheme an SAH build hardly matters, the SAH has one additional advantage that is particularly important for animations: the SAH cost function in practice tends to cluster nearby objects, and partitions different clusters into different subtrees. For typical scenes, a given cluster of triangles often belongs to the same logical part of the model; triangles in that cluster then often deform similarly to each other, and are advantageous to put into the same BVH subtree. A SAH build will usually tend to do just that, even though there is no guarantee that it will.

We did not include a Goldsmith-Salmon [1987] bottom-up build as this method has been shown to be greatly inferior to other strategies in practice [Havran 2001; Mahovsky 2005].

Scene	2×2 packets			16×16 packets		
	object median	spatial median	SAH	object median	spatial median	SAH
erw6	0.9	1.4	3.6	27.0	32.4	42.6
Conference	0.2	0.5	1.2	5.7	7.6	10.5
Soda Hall	0.3	0.8	1.9	5.5	8.8	12.3

Table II. Performance of object median build, spatial median build, and centroid-based surface area heuristic (SAH) built for three static scenes. Numbers are frames per second for 1024×1024 pixels on a 2.6 GHz Opteron CPU, and depend on the fast BVH traversal method explained in Section 3.2. For small packet sizes, using the SAH provides a substantial performance improvement of up to $2\times$ and $6\times$ over spatial median and object median, respectively. For larger packet sizes, our traversal algorithm provides the majority of the performance.

3.2 Efficient Packet-/Frustum Traversal for BVHs

As shown in the previous section, a well constructed hierarchy can have a substantial impact on performance, but the traversal algorithm has a much higher impact. For kd-trees the highest performance gains have been achieved using packets [Wald et al. 2001] and frusta [Reshetov et al. 2005][1]. In this section, we show how these concepts can also be applied to BVHs in a straightforward manner. In fact, this leads to a new traversal scheme that not only combines these two concepts in a simpler way, but which also allows for several additional optimizations not available for previous traversal schemes. In particular, this algorithm builds on several concepts — large packets of rays, ordered traversal, an early frustum exit test, a first hit early descent, first active ray tracking, and a SIMD packet test of last resort — all tightly integrated into a single unified traversal step.

Packet traversal. A standard BVH traversal proceeds in a recursive fashion: a ray hitting an interior node tests its two children's bounding volumes for overlap, and each child is traversed recursively if it is hit:

```
BVH_Traverse(Node n, Ray r)
{
   if (n.isLeaf()) return intersectTrianglesIn(n);
   if (r overlaps n.child[0]) BVH_Traverse(n.child[0]);
   if (r overlaps n.child[1]) BVH_Traverse(n.child[1]);
}
```

The generalization of this algorithm to packets of rays is straightforward: both children are checked in turn, and get traversed if hit by *any* of the rays in the packet:

[1]The concept of frusta in ray tracing has also been used in [van der Zwaan et al. ; Genetti et al. 1998].

```
BVH_Traverse(Node n, Packet ray[Nrays])
{
   if (n.isLeaf()) return intersectTrianglesIn(n);
   if (any r in ray[] overlaps n.child[0]) BVH_Traverse(n.child[0]);
   if (any r in ray[] overlaps n.child[1]) BVH_Traverse(n.child[1]);
}
```

Note, however, that the optimizations described below let us benefit from much larger packets of rays than typically used for kd-tree packet traversal, making us use much larger packets than the 4-ray packets used by Wald et al.[2001] and Lauterbach et al. [2006].

For efficiency reasons, this recursive scheme can also be transformed into an iterative scheme as is done for kd-trees (see, e.g., [Wald 2004]): for nodes hit by the packet, one of the child nodes is pushed on the stack, and iteration proceeds with the other one. If a node is missed by the packet, the next node is taken off the stack, and iteration continues.

Ordered traversal. One of the biggest deficiencies of BVHs is that a kd-tree can guarantee a strict front-to-back traversal (and thus, can avoid traversing parts of a hierarchy that are occluded by other parts in front), while a BVH cannot. To increase the likelihood of traversing the children in front-to-back order (and thus reduce redundant operations on occluded parts), we determine the order the children are tested from properties of the rays; this optimization has been employed by several researchers (see, e.g., [Mahovsky 2005]).

To implement this *ordered traversal* scheme, our BVH nodes store two fields: the dimension n_{axis} in which its two children are furthest apart, and an int n_{first} specifying which of the children should be traversed first by a ray traveling along axis n_{axis}. During runtime the traversal order is determined by `xor`'ing the node's order bit with the first ray's n_{axis} direction sign. In contrast to kd-tree packet tracers we do not need to guarantee that all rays in a packet have the same sign bits. If the children are tested in the "wrong" order there is an efficiency penalty but no error. Not having to guarantee the same signs avoids many special cases, and thus greatly simplifies the overall implementation. In particular, we can base the entire packet's traversal order only on the direction signs of its first ray. In practice we have found this to work well for both primary and secondary rays [Boulos et al. 2006].

```
BVH_Traverse(Node n, Packet ray[Nrays])
{
   firstChild = sign(ray[0].direction[n.axis]) ^ n.first;
   if (n.isLeaf()) return intersectTrianglesIn(n);
   if (any r in ray[] overlaps n.child[firstChild])
         BVH_Traverse(n.child[firstChild]);
   if (any r in ray[] overlaps n.child[1-firstChild])
         BVH_Traverse(n.child[1-firstChild]);
}
```

Early hit test. In a standard packet-based kd-tree traversal all rays are tested at each tree node, albeit 4 at a time [Wald et al. 2001]. For a BVH, however, not all rays in the packet need to be tested: if *any* of the rays during the packet-box intersection reports a positive intersection, we can immediately enter this subtree without considering *any* of the remaining rays. When rays are coherent this avoids many redundant intersections, usually testing an entire packet — of usually 8×8 or 16×16 rays — using just a single test. Standard kd-tree traversals cannot easily use this optimization, as they have to compute the entry and exit distances to the box even for all remaining rays [Wald 2004].

Early miss exit. The early hit test can greatly accelerate the traversal for cases in which the first tested ray actually hits the box. However, if the packet misses the box, we would still have to test all of the rays in the packet to find that none of them hits the box. This is a typical deficiency of packet-traversal algorithms, as the cost of the (quite common) full-miss case is linear in the number of rays in a packet.

For this case, however, we can employ the same idea that Reshetov et al. have proposed in their MLRT traversal [2005]. Using interval arithmetic, we can compute an approximate (but conservative) packet-box overlap test, and can immediately skip all individual traversal steps if this conservative test already indicates missing the box. To use this scheme, we first perform the first active ray's overlap test as described above. If this ray overlaps, we immediately descend, and do not perform any interval arithmetic test. If the first hit test did not yield a valid intersection, we perform the overlap test based on the packet's precomputed minima and maxima direction components, and, if that test fails, can return a miss without having to perform any further ray-box tests.

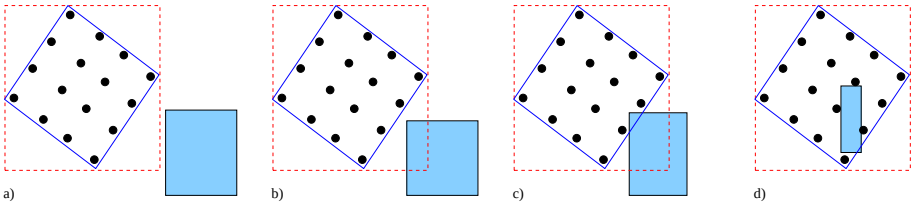

Fig. 3. Four ways a packet can miss a subtree of a BVH. Points denote rays; blue lines are a "perfect" bounding frustum; red lines show a conservative bounding frustum as used by the interval arithmetic test; the blue box denotes the bounding volume. a) a wide miss, where even the conservative frustum misses the bounding volume. b) the packet fully misses the bounding volume, but the (conservative) frustum test cannot detect it. c) Even the perfect frustum overlaps the node, but none of the rays does. d) the node is very small (as for highly tesselated scenes) but falls in-between the raster of rays. Only the first one is handled correctly with an interval arithmetic frustum technique; the second one is a deficiency of the conservative interval arithmetic test, the third and fourth are general deficiencies of any frustum method. Our traversal scheme handles all four cases correctly.

In contrast to Reshetov et al. [2005] we *only* use the frustum to conservatively *discard* subtrees; we will never let the outcome of a frustum test lead us into descending without further tests. One reason for this decision is that the interval arithmetic is conservative by design (also see the discussion in [Reshetov et al. 2005]). The second and more important reason is that even *if* the packet's frustum overlaps the node's bounding volume, there is no guarantee that any of the rays will actually hit the box as well – in particular for large scenes with small triangles, entire subtrees of the BVH can fall in-between the raster of the rays in the packet (see Figure 3). By only using the test to discard subtrees, we avoid this major problem of frustum traversal methods for highly complex scenes[2].

Packet test of last resort. As just described, we use the frustum test only to discard *guaranteed* misses, and perform a more accurate test if that is not the case. If neither early hit nor frustum exit test was conclusive, we revert to intersecting all the remaining rays in the packet until we find the first one that hits, or until we determine that all miss. In that way, we can use the frustum's quick reject capabilities for full misses, but never traverse any node that is not hit by at least one ray in the packet.

[2]A pure frustum traversal scheme that would also descend if the frustum has an overlap with the box was also tested, but was inferior in performance.

First active ray tracking. The first hit descend allows for descending as soon as any ray in a packet hits the box they all descend. However, the ray that hits the box may not be the first ray in the packet. In particular, if a ray has missed a node, it is guaranteed to also miss all descendents of this node, so checking it again further down in that subtree is redundant. We take advantage of this by tracking the index of the first ray that has not yet missed an ancestor and use this ray for the first hit test. Knowing the first ray that hit the parent we have a much higher chance that the first ray tested actually hits the box, and furthermore avoid re-testing those rays that already missed an ancestor's box.

Leaf traversal. By having all rays in a packet descend once the first ray hits the parent, we can often replace N (packet size) ray-box tests with a single one. This comes at the cost of some rays descending into the hierarchy even though they did not hit the parent. This does not hurt during descent, but could lead to an increased number of ray-triangle intersections once such rays reach a leaf node. This can be avoided by not intersecting *all* rays with the triangles once a leaf is reached; instead, one can perform an additional ray versus leaf-node test, and can exclude rays that do not hit the box from ray-triangle intersections[3].

SIMD implementation. Both early hit and early exit test have a constant cost independent of the size of the packet. However, if neither early hit nor early exit test was conclusive, the packet test of last resort is linear in the number of (missing) rays, and can thus be expected to be costly. Therefore, we do not intersect individual rays, but use a SIMD approach in which four rays are intersected in parallel. To allow for using SIMD extensions to compute ray-box tests in parallel, data must be properly arranged. For each inner node, we need to store information about the bounding volume, the node traversal order, and a reference to the child nodes or triangle list. This information can be stored in a 32 byte record:

```
#pragma align(32)
struct BVHNode {
  float box_min[3]; // 16 byte aligned
  union {
    int firstChildNodeID;  // for inner nodes
    int firstTriangleID;   // for leaf nodes
  };
  float box_max[3]; // 16 byte aligned
  short num_triangles; //  0 flags inner node
  unsigned char axis; // ordered traversal axis
  unsigned char first;// first node to be traversal along axis
};
```

Similarly, we have to align our packets of rays, and consequently use packet sizes that are powers of two, though any multiple of four would be possible. Once the data is properly organized, the SIMD implementation is fairly straightforward. For each node, we use the slabs algorithm [Kay and Kajiya 1986] for computing ray-box overlap, but perform this test for four rays in parallel. Once a leaf is reached, we use the SIMD triangle test described by Wald [2004] to test 4 rays with the same triangle in parallel. To take advantage of the fact that we have much larger packets, we also employ a frustum test in the triangle test (as also done by [Dmitriev et al. 2004; Reshetov et al. 2005]), to quickly reject full misses. Using the four virtual corner rays of the bounding frustum, this test in SIMD costs roughly

[3]Of course, this additional test can be implemented very efficiently in SIMD.

as much as a SIMD ray-triangle test, but can usually decide around 30–50% of the packet-triangle tests without having to consider any individual ray. We also tested a SIMD version of the Möller-Trumbore test [1997], but its performance was slightly inferior.

Although the use of SIMD operations for the packet test of last resort is helpful, it is not a central part of the overall design. The main benefits of using ray packets are algorithmic in nature, and will persist even in a non-SIMD implementation. We have not created a non-SIMD implementation of our system, but believe such an implementation would not be considerably slower than ours. This is because the packet-BVH optimizations benefit little from testing rays four at a time. The main benefits of SIMD are probably in the ray-triangle routine, which caps the potential of SIMD to the fraction of time spent there.

Combining the individual concepts. The concepts just explained – large packets, ordered traversal, first hit descent, frustum exit, active ray tracking, and SIMD packet test of last resort – can be combined to be mutually supportive; each technique either strengthens another one, or counters another technique's deficiencies (such as a frustum offering the early reject that the first hit cannot do, or the packet test of last resort that counters a frustum's frequent false hit decision). The combined algorithm nevertheless is surprisingly simple, and can be coded in a few dozen lines of code.

We show the code as two routines: Algorithm 2 combines the frustum exit, first hit exit, and SIMD packet test described above. It returns the first ray in the packet that hits the box, or an "end of packet" marker in case none of the rays hits. This "findFirst" operation is then being employed in the recursive traversal routine described in Algorithm 3, which includes the iterative BVH traversal including ordered traversal and first active ray tracking.

Algorithm 2 Pseudo-code for the fast packet/box intersection. Note that both "full hits" (i.e., first ray that hits parent also hits box) and "full misses" (i.e., a covering frustum misses the box) are very cheap, and have a constant cost independent of packet size. Only for rays partially hitting the box do we need to perform more than the first two cheap tests.

```
{Compute ID of first ray hitting AABB box}
{'parentsFirstActive' is ID of first ray hitting current box's parent}
function findFirst(ray[Nrays], int parentsFirstActive, AABB box)
 {First:  Quick 'hit' test using 'first' ray}
 if ray[parentsFirstActive] intersects box then
  {first one hits → packet hits...}
  return parentsFirstActive
 end if

 {Second:  Quick 'all miss' test using frustum or interval arithmetic}
 if frustum(ray[0..Nrays]) misses box then
  return Nrays {all rays miss}
 end if

 {Neither quick test helped, test all rays}
 for i = parentsFirstActive ..  Nrays do
  if ray[i] intersects box then
   return i {all earlier ones missed}
  end if
 end for
 return Nrays {all rays have missed}
end
```

Algorithm 3 Pseudo-code of our packet-based BVH traversal.

```
function traverse(ray[Nrays])
  node=root; firstActive = 0; {Initialize recursion}
  while true do
   {Find ID of first ray hitting node}
   firstActive = findFirst(ray,node->box,firstActive);
   if firstActive < Nrays then
     if node is inner node then
       firstChild = traversalOrder(node,ray);
       stack.push(firstActive,node.child[1-firstChild]);
       node = node.child[firstChild];
       continue
     else
       determine rays actually active in that leaf
       intersect all triangles in node
     end if
   end if
   if stack.empty() then
     return
   end if
   (node,firstActive) = stack.pop();
  end while
end
```

Packet type-specific optimizations. Packets for different kinds of rays have different properties. For example, primary rays share the same origin, and are bounded by their corner rays; shadow rays often share the same origin, but have no concept of corner rays; secondary rays may not even share the same origin; and some packets do have the same direction signs while others don't. Currently, both the traversal and intersection functions are templated in a way such that the template parameters specify whether the packet has common origin, corner rays, or is a shadow packet. Corner rays are used only for the triangle intersection, where a triangle can be skipped if all the corner rays miss the triangle at the same side [Dmitriev et al. 2004; Reshetov et al. 2005]. Note, however, that these optimizations are not restricted to primary rays: for secondary rays, some "virtual" corner rays can easily be computed for that purpose [Wald et al. 2006]; for shadow rays the triangle test can then even be accelerated for the case where the full frustum hits the triangle, as then only a distance test has to be performed, as no barycentric hit coordinates are required.

During traversal and box intersection, only interval arithmetic is used, and all rays are handled the same. Since some operations like the ray box intersection and interval arithmetic get simpler if the signs are known, we compute the signs at the beginning of the traversal loop, and can use a somewhat faster box intersection if the signs are equal. Even if they differ, we do not have to split the packet, and only lose a few percent of performance due to a somewhat slower intersection routine. The signs are checked once at the beginning of traversal, and two separate traversal routines are then being called, depending on whether the signs match or not. A more detailed discussion of using secondary rays in our system can be found in Boulos et al. [2006].

3.3 Traversal performance

To demonstrate that our packet tests can gain efficiency over tracing every ray, we measured the performance of our algorithm on the test scenes shown in Figure 4. The results of this experiment are given in Table III, and show several interesting results. First, that our

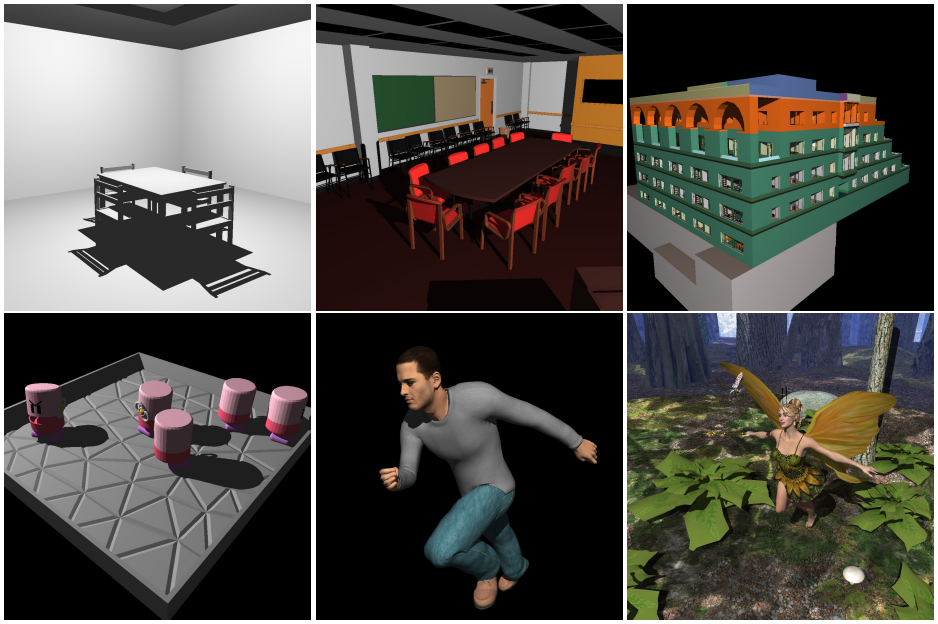

Fig. 4. The scenes used for our experiments. From left to right: ERW6 (800 triangles, static), Conference (280,000, static), Soda Hall (2.5M, static), Toys (11,000, animated), Ben (78,000, animated), complete Fairy Forest (180,000, animated; also see Figure 1). With pure ray casting (without shading), these scenes render at 42.6, 10.5, 12.3, 23.7, 15.6, and 6.1 frames per second (fps) at 1024×1024 pixels, respectively. Including shading, shadows, and textures, they still render at 15.2, 4.8, 9.5, 10.5, 8.53, and 2.16 fps.

technique performs best for much larger packets of rays than the 4-ray packets commonly used by, e.g., [Wald et al. 2001; Lauterbach et al. 2006]. Second, that the algorithm's optimal packet size is quite robust over a wide range of scenes of different nature and complexities: except for very small scenes or scenes with large polygons — for which even 16×16 packets can be beneficial — 8×8 and 16×16 rays usually perform similarly well, and significantly better than other resolutions. Thus, though further parameter tweaking can further improve performance, in practice we usually use 8×8 rays.

	2×2	4×4	8×8	16×16	32×32	best speedup vs. 2×2
erw6	4.9	15.1	32.2	42.6	36.7	10.7×
conf	1.8	5.3	10.2	10.5	7.0	5.8×
soda	2.7	7.4	12.6	12.3	7.7	4.6×
toys	5.4	14.1	23.3	23.7	16.7	4.4×
runner	5.0	11.5	16.4	15.6	10.5	3.3×
fairy	1.5	3.9	6.4	6.1	4.0	4.3×

Table III. Runtimes in frames/sec for ray casting at 1024×1024 on one 2.6 GHz Opteron as packet size is varied. Animated scene performance is given as an average over the course of the animation and includes update time.

Impact of packet size. When comparing the 8×8 or 16×16 ray performance to the baseline performance of 2×2 rays (the smallest possible packet size in a SIMD-based system), Table III demonstrates that the improvements of our method are mostly of an algorithmic nature, showing a significant speedup of up to an order of magnitude. Note that the 2×2 case is already a highly optimized, SIMD-enabled packet variant as proposed in Lauterbach et al. [2006]. A comparison to a single-ray variant could be expected to be even

more dramatic, but could not be done as our system is based on a SIMD implementation, for which at least 4 rays have to be considered at a time.

This performance increase for larger packets can best be explained by investigating the impact of the early hit and early exit optimizations used by our algorithm. For the chosen set of test scenes, Table IV gives the relative fraction of cases in which an 16×16 ray traversal step can be terminated with constant cost via an early hit or via an early miss, as well as the average number of SIMD ray-box intersections that have to be performed if neither of the early exits could be employed. As can be seen, in around half of the cases can we exit immediately after the first test, and the frustum test handles the majority of the remaining cases. In only 5% to 30% of the cases we have to test the remaining rays, and even then only a fraction of the rays have to be considered[4].

scene	(A) early hit exits	(B) frustum exits	(C) last resort packet test	avg SIMD tests in (C)
erw6	52.3%	42.9%	4.8%	31.7
conference	51.9%	35.3%	12.8%	22.8
soda hall	49.5%	27.5%	23.0%	32.8
toys	49.7%	32.2%	18.1%	22.7
runner	44.1%	25.3%	30.6%	20.6
fairy	49.1%	30.2%	20.7%	19.9

Table IV. Relative number of cases where our algorithm can immediately exit after the first test, after the second test, and during the loop over all rays, respectively, and the average number of rays tested in the latter case (for 16×16 rays per packet).

Table V shows that these cheap exits greatly reduce the total number of SIMD 4-ray-box tests. In this table, we have counted a interval arithmetic test to cost as much as a ray-box test, which is roughly the case in our implementation. Under this assumption, Table V shows that our traversal can save up to an order of magnitude in ray-box tests, which roughly corresponds to the performance increase seen above. Again, note that the baseline 2×2 variant already performs 4 times less ray-box tests than a single-ray variant.

scene	erw6 static	conference static	soda hall static	toys 1st frame	runner 1st frame	fairy 1st frame
2×2 brute force ray-box tests	4,201	12,021	8,688	4,041	4,728	14,501
16×16 clever ray-box tests	148	890	2,129	462	1,102	1,781
interval tests	31	91	72	32	41	116
sum	179	982	2,202	494	1,143	1,898
ratio (2×2:sum)	23.4	12.2	3.9	8.2	4.1	7.6

Table V. Number of SIMD ray-box tests (in thousands) for a brute force 2×2 packet traverser, and for our algorithm with 16×16 rays per packet, for a 1024×1024 image. The number of ray triangle tests stays about the same for both methods. As can be seen, our traversal greatly reduces the total number of box tests.

Comparison to kd-tree and grid. At least for static scenes, we can now compare our method to competing methods, in particular to OpenRT, MLRT, and coherent grid traversal (a discussion of dynamic scene performance will follow in Section 6.2). As shown in Table VI, the performance achieved by our technique is at least competitive with all these

[4]Note that since early hit and early exit optimizations react to mutually exclusive traversal cases, the relative number of cases does not change if the order of the two tests is reversed.

methods; usually being within roughly a factor of two of the performance reported for MLRT, and consistently faster than OpenRT in all experiments we have performed so far.

Scene	#tris	MLRT static Xeon 3.2GHz w/HT	OpenRT static Opteron 2.6GHz	Frustum Grid static Opteron 2.6GHz	BVH static Opteron 2.6GHz
erw6	800	50.7	6.6	17.8	32.5
conf	280k	15.6	3.9	3.8	9.3
soda	2.5M	24	6.2	6.4	11.1
toys (1st frame)	11k	n/a	9.7	23.6	30.5
runner (1st frame)	78k	n/a	8.8	11.0	21.4
fairy (1st frame)	180k	n/a	3.6	1.8	7.7

Table VI. Performance in frames/sec. (at 1024×1024 pixels including simple shading) for OpenRT, coherent grid traversal, MLRT, and BVH. For toys, runner, and fairy, a single time step has been used. MLRT performance data is taken from [Reshetov et al. 2005], and corresponds to a Xeon 3.2 GHz with Hyperthreading; all other data was gathered on a 2.6 GHz Opteron machine. MLRT data was not available for toys, runner, or fairy. OpenRT data is newer than in the MLRT paper, and is for a 4×4-ray packet implementation with SIMD frustum culling.

4. APPLICATION TO DYNAMIC SCENES

In the previous section, we have shown that a well-built BVH – when using a proper traversal algorithm – can achieve performance for static scenes that is competitive to today's fastest kd-tree and grid based schemes [Reshetov et al. 2005; Wald et al. 2006]. In this section we show how this can also be used to ray trace deformable models.

The most general way to handle deformable models would be to build a new BVH for each frame. Although building a new data structure from scratch for every frame has recently been shown to be feasible for uniform grids [Wald et al. 2006], for more complex hierarchical data structures like kd-trees or BVHs this is currently infeasible — even with the fast SAH build described in Section 3.1 and Algorithm 1, build times for our BVH are still too high for per-frame rebuilds (c.f. Table I).

For applications where all animation poses are known in advance, we could also store one pre-built BVH for each time step. This however would preclude our approach from being applied to truly interactive settings, in which this information is not available.

Alternatively, we could incrementally change the tree, but how to maintain a well-built SAH by doing this is still a matter of research. To avoid that complexity we try to build a tree whose topology (hierarchy) does not change over time, but whose AABB coordinates do change. An example of such a change for a small tree is shown in Figure 5. This idea of using a constant-topology tree that is only refit every frame has been proposed by several researchers before (e.g. [Larsson and Akenine-Möller 2003]), and is also being used in concurrent research [Lauterbach et al. 2006].

Fast BVH updates. After having decided to not change the topology, all that has to be done after the triangles' vertex positions have moved is to "update" (or "refit") the bounding volumes to reflect the changed primitive positions. The procedure for refitting the BVH after the geometry has been deformed is straightforward, and is shown in Algorithm 4: we recursively traverse the hierarchy, first recompute the child nodes' bounding volumes, and then refit the parents' boxes as well.

Because we use an ordered traversal, the flag indicating which of the three axes is "dominant" also needs to be updated with the box positions. To do this, we determine the axis in

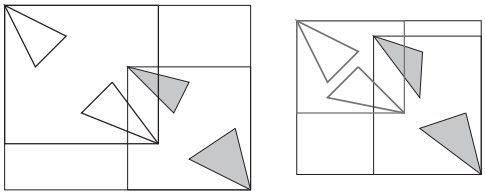

Fig. 5. When the objects move, a BVH can keep the same hierarchy, and only needs to update the bounding volumes. Though the new hierarchy may not be as *good* as the old one, it will always be *correct*; for all but some worst-case examples, even severe deformations did not significantly deteriorate the BVH quality. By considering different primitive positions during the build, we can also make sure that the chosen BVH will be reasonably good for scenes in which a good hierarchy is not apparent from a single pose.

which the centroids of the two child nodes are furthest apart. Using the axis in which the boxes have minimum overlap has also been tested, and can be used just as well. Once this axis is determined, the traversal order is determined by the coordinates of the box centroids along this axis. Updating this ordered traversal information ensures that the BVH traversal order will remain predominantly front-to-back even if subtrees switch sides. For example, if two subtrees enclose two different characters that move past or around each other, the traversal order will automatically be corrected.

Algorithm 4 BVH update after triangles move

```
function UpdateBBoxes(Node node)
  if node is leaf then
    bounds(node) ← boundsOf(triangles(node));
  else
    (c0, c1) = children(node);
    UpdateBBoxes(c0);
    UpdateBBoxes(c1)
    node.bounds = ∪{ bounds(c0), bounds(c1) }
    recompute ordered traversal information
  end if
end
```

Choosing the BVH to be deformed. Though we have explained how to deform a constant-topology BVH, we have not yet explained where the original BVH topology being deformed is coming from. In our experiments, simply building the BVH over the first pose of the object has shown to usually work quite well, so this is what we do by default.

As we will discuss below, for most scenes this simple strategy will work well. For applications like games, characters are usually modeled in a "rest pose" — similar to the "daVinci" pose — which is then animated. As this rest pose exposes exactly the intrinsic hierarchy of the character, this would be the perfect pose to build the BVH over.

Building the BVH over time. Though we have just argued that for most applications it is sufficient to build the BVH over the first frame (or rest pose, if available), there might be examples in which this is not possible. For example, an animation might start with the character having his hands clasped over his head, in which case the BVH might not separate the head and hands. In this case, building the BVH for the first frame would result in an inefficient BVH once the character moves into a pose with its hands in its pockets.

On the other hand, even if the first frame's pose is unsuitable for deformation, it is quite unlikely that *all* poses of the animation are similarly bad for deformation. As such, if we can at least sample the space of valid poses (without having to know all of the frames in advance), we have a good chance of finding a reasonably good topology for deformation.

To find such a BVH that works well for all poses, we first need a way to evaluate how good a BVH is for a certain deformation. Given this metric, we can measure how well a certain hierarchy behaves over a set of different poses. While we could evaluate this by actual ray tracing, we instead estimate a BVH's quality via Equation 1. This estimate is more straightforward than determining a "fair" camera path for cost evaluation, is inexpensive to compute, and has shown to have a high correlation to the real ray tracing cost.

Given a set of M example poses of the model, we can compute a BVH for each of these poses, getting M distinct candidate BVHs. For each of those candidate BVHs, we can compute their maximum (or average) cost over the example poses, and can select the one with least cost. In practice we have found that using lowest maximum cost results in slightly better frame rates than using lowest average cost.

Alternatively we can build a BVH that chooses the best *partition* based on all known deformations at each recursive partitioning step. For example, instead of first building M individual BVHs first, we compute M distinct partitions during the recursive SAH build, and select the partition with least cost before recursively building the resulting two subtrees. Since this function has to evaluate different poses of the model in each partitioning step, this best split over time can be quite expensive. For all the models we tested, the results were usually the same as with the simpler method of constructing a BVH for each time step. Whether there are models that would benefit from the more sophisticated build method is unknown. There may, however, be more extreme cases where this method is required, so we have decided to sketch it here even though we do not generally use it. Extending techniques that analyze and decompose the space of possible poses, such as proposed by Günther et al. [2006], would be straightforward to apply as well.

5. EXPERIMENTS AND EVALUATION

Having described both the static as well as dynamic-specific ingredients of our system, we can now evaluate its performance on several animated scenes. In particular, we quantify build and update times, and the achieved ray tracing performance for a variety of different scenes, as well as demonstrate empirically that the build over time can help, but that the first-frame BVH usually is sufficient. We will also demonstrate that our algorithm is quite robust to deforming BVHs, and explain why this is the case.

5.1 Test Scenes

For this evaluation, we have chosen a set of different models designed to shine light on different aspects of our system (see Figure 6). In particular the fairy, toys, and marbles scenes have been modeled explicitly for this purpose.

Runner. Though not the simplest in triangle count, the "runner" model is the simplest of our test scenes in that it is a typical example of a "single model" deformable mesh. It is a 80,000 triangle animated *Poser* figure and has a reasonable range of deformations.

Toys. The runner consisted only of a single deformable mesh, for which deforming a single BVH might be expected to work well. However, typical interactive applications use multiple animated objects at the same time. These usually show a totally different dynamic

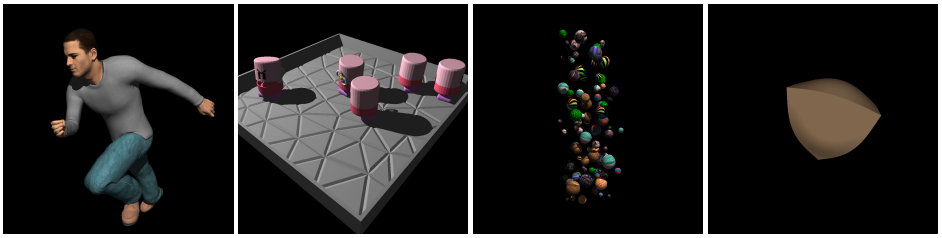

Fig. 6. The runner, toys, marbles, and BART scenes used for stressing the dynamic aspects of our method.

behavior: they run around each other, are sometimes close and sometimes far apart from each other, etc. To stress such an example, we have modeled the "toys" scene, in which some animated wind-up toys run incoherently around among each other, bump into each other, and even jump over each other, thereby frequently changing their relative position and orientation. At 11K triangles, this scene has a lower triangle count than the runner, but exposes a more complicated dynamic behavior.

Marbles. To stress behavior for non-hierarchical motion like a particle system, we also modeled another scene – "marbles" – in which a set of 110 marbles is falling into an (invisible) glass cylinder, there performing a rigid-body simulation. Though having even less triangles than the toys scene (8.8K), the motion in this scene is non-trivial.

BART. The "BART" scene is actually only a part of the original "museum" scene specified in the BART benchmark (cf. [Lext et al. 2000a]). While the original scene also contained some static geometry that would not be a problem for our approach, the given subset of it – first used by Lauterbach et al. [2006] – concentrates on the problematic part: a set of triangles interacting in a way that intentionally destroys any possible structure over the animation. Though the model is available in several resolutions, we have only tested one such resolution: the most detailed one of 64K triangles.

Fairy Forest. To also include a more practically relevant stress case, we have taken a free modeling program (DAZStudio) and have created a scene composed of a total of nearly 180,000 animated triangles (Figure 7). The scene consists of a skinned 80,000 triangle fairy model dancing through a forest made up of trees and animated ferns. Additionally, a dragonfly — also with skinned body and flapping wings — flies around the fairy. The scene also makes heavy use of textures (30 MB) thus stressing the impact of shading cost.

The fairy forest is considerably less extreme in dynamics than the BART scene, but was intentionally designed to provide a stress case that could also arise in a practical application. It intentionally stresses several issues: it is quite complex, and with 180,000 triangles in the complexity range of today's games; it is almost fully animated (including plants and trees), but no knowledge about the modeling hierarchy is provided to our algorithm; the surrounding geometry with tall trees and background objects stresses some "teapot in a stadium" situations and makes hierarchy construction non-obvious[5]. The fairy's wings are opening and closing while she dances, and the dragonfly (itself being animated) is flying

[5]Though the different sizes of dragonfly, fairy, and surroundings are not a problem for a hierarchical data structure like a BVH, the fact that fairy and dragonfly are surrounded by the forest does: since fairy and dragonfly are *inside* the forest, the build algorithm cannot find splitting planes that would separate the fairy and dragonfly from the surroundings. That knowledge would be available in the modeling hierarchy, but is not available to our method.

all around the fairy at varying distances. The latter is intended to break the BVH by leading to badly enlarged bounding boxes higher up in the tree.

Fig. 7. Fairy-forest test scene: An animated dragonfly (a) and a dancing fairy (b), placed into a typical game environment with background textures and animated foreground geometry (c). The resulting scene (d) consists of 180,000 animated triangles, and is rendered with textures, shading, and shadows, at 2.2 fps at 1024×1024 resolution.

5.2 Performance Results

Except for the BART example, our method works quite well for all of these examples. This robustness has been somewhat surprising to us – as we had expected all but the runner to be stress cases – but will be explained below. Before going into a detailed discussion, we briefly summarize the performance one can observe when applying our method to the varying test scenes. The original build times for both a single build as well as for the build over time, as well as the update time for each model, are given in Table VII.

scene	#tris	single BVH build	build over time	triangle update	AABB update
toys	11k	0.13s	1.20s	0.96ms	0.47ms
runner	78k	1.26s	10.80s	6.88ms	4.76ms
fairy	180k	3.24s	31.40s	15.89ms	12.91ms

Table VII. BVH build times for one BVH, and for a build over time with 10 candidates, as well as per-frame update time split into triangle update and bounding box refitting.

Runner. As expected, the runner example works quite well. Updating both triangles and BVH only costs around 11ms (see Table VII). Though the hierarchy deforms, most of the deformation is in the legs and arms moving forward and backward. This can be handled well with the ordered traversal. Figure 8 shows the model for time step 14, for both the BVH built over the first frame, as well as for the best BVH over time. As can be seen, the SAH build already extracts a useful hierarchy even when built over the first frame only, and 15.3 frames per second are achieved (1024×1024 pixels, primary rays without shading). Though using the best BVH over time can further improve performance to 16.2 fps, this modest gain was less than expected. Adding shadows to this scene does not cost a lot: for the view given in Figure 6, the frame rate drops from 15.6 to 8.5 frames per second when turning on shading, textures, and shadows.

Toys. Even though the floor on which the toys are moving has some surrounding welt on the sides (which we expected to hinder the SAH in finding a good clustering), the SAH extracts a good model hierarchy that remains reasonably efficient even under deformation. When building over time, the first split is somewhat unintuitive in that it first separates the

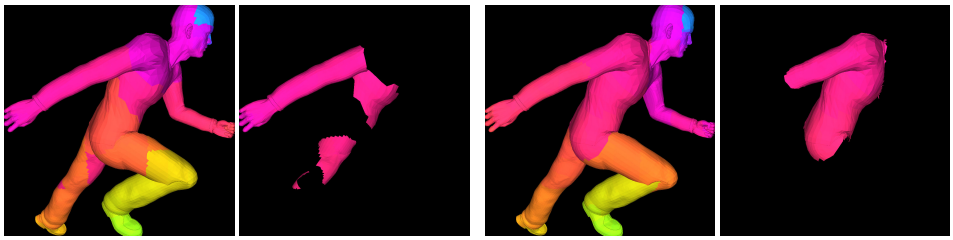

Fig. 8. Color-coded hierarchy for the runner at frame 14, showing both the full model as well as the subtree containing the torso. Left: A deformed BVH originally built for frame 0. Right: Best BVH over time. The best tree over time does find a better partitioning, but even the first frame's BVH shows a reasonably good hierarchy, which – though with significantly more overlap of different subtrees – is reasonably good for our algorithm.

larger ground object (including the railing and lower parts of the toys) from the upper parts of the toys. This could be avoided if our algorithm were allowed to exploit knowledge about the scene hierarchy. After that initial chop, the SAH quickly clusters the individual toys' halves into subtrees, which subsequently leads to a good hierarchy after all.

Even when only deforming the first frame's BVH, we achieve a performance of 10.5 frames per second (on average) on a single Opteron 2.6 GHz CPU (at 1024×1024 pixels) including shading and shadows. The impact of using a built-over-time BVH for all frames is, like for the Runner scene, relatively small, improving performance by less than 20% even over the worst individual frame's BVH we could find (see Figure 9).

Marbles. Though originally believed to be a stress case for incoherent motion, the marbles scene works surprisingly well even under deformation. Including shading, shadows, and textures, the pose shown in Figure 6 renders at 16.2 fps (1024×1024 pixels on a single Opteron CPU) for the build over time, and at 20.6 fps for the first frame's BVH. As with the previous scenes, the difference between the build over time and the best/worst performance for that frame is small.

BART. Not unexpectedly, the BART scene can not be handled well with our approach. By design this scene does not have any intrinsic model hierarchy at all, which therefore cannot be found by our build method. Though each static pose in itself is not a problem (with over 10 fps for the first frame), any individual frame's BVH deteriorates quickly during deformation. Though this thoroughly breaks our algorithm, we do not consider it a major limitation: first, it was to be expected; and second, we believe such artificial test cases to be quite rare in practical applications. When using the build over time the *average* performance is actually higher than that reported by [Lauterbach et al. 2006]; though a few frames are still slow, the average is still high since most of the frames are fast. Nevertheless, the slow frames cause a high variation in frame rate (with some frames costing more than a second), and the worst-case performance is still bad.

Fairy Forest. In contrast to BART, our algorithm handles the fairy forest surprisingly well. Without shading and shadows – but with full animation – the five example frames in Figure 1 render at 6.0–6.5 frames per second on a single CPU. This includes the time to recompute all of the 180k triangles and 100k vertices every frame, as well as the time to refit the bounding volumes and recompute the triangle acceleration structure required for the triangle test. Including shading and shadows, the performance drops as expected. For a fast ray tracer even a simple shader computing only a dot product can reduce performance by more than half [Reshetov et al. 2005]. Additionally, our shader interpolates shading

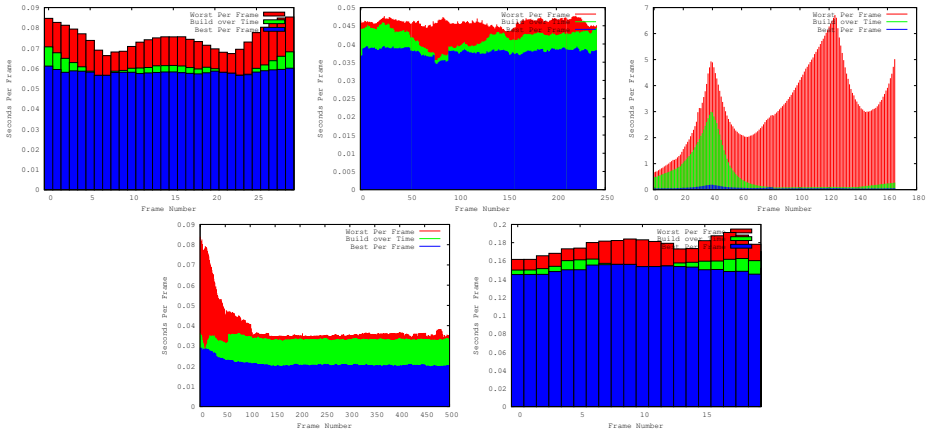

Fig. 9. Impact of BVH deformation for the runner, toys, BART, marbles, and fairy scenes. One BVH has been built for each time step, and has been deformed to every other time step. For each time step these graphs show the worst performance of *any* of these BVHs, the performance of the best BVH over time, as well as the performance achieved by a BVH explicitly built for that time step. As can be seen, for all tested scenes except the BART model, taking the worst BVH of *any* other time step is only around 20-30% slower than a custom-built BVH.

normals, computes textures coordinates, touches megabytes of textures, and shoots shadow rays. Finally, the model contains multiple distinct object silhouettes, which can lead to incoherent shadow packets when connecting all primary rays' hit points to the same point light source. Nevertheless, including shading, textures, and shadows we still achieve 2.0 to 2.3 frames per second on a single CPU, and 3.4 to 4.0 frames per second once we enable the second CPU in our PC.

5.3 Impact of BVH Deformation

The experiments just performed demonstrate empirically that:

(1) BVH refits are small, and do not greatly influence run time. Thus, performance including BVH refit does not significantly differ from static model performance.

(2) The intentionally bad BART model breaks the refitting approach, resulting in a significant performance deterioration during deformation.

(3) Except for BART, all other models worked well with the deformation approach.

(4) Build over time did *not* significantly improve the performance, meaning that simply deforming a BVH built for the first frame was not significantly worse than the best BVH over time.

The last of these items is the most interesting one. To investigate it further, we have also taken several poses for each of the models, and have generated one BVH for each of these poses, as well as the best BVH over time. Each of these BVHs has then been deformed to each of the sampled poses, and the best and worst frame rate have been recorded for each pose. As can be seen from Figure 9, except for BART deforming works well for *any* initial pose encountered in our experiments: the best BVH over time can avoid BVH deterioration to a certain degree, but even the worst BVH generated by any of the time steps usually is less than 20-30% slower than a custom built BVH. As the scenes deform significantly over time, this small impact of BVH deformation on runtime at first was quite surprising.

5.4 Influence of Traversal Method of Deformation Robustness

The reasons for the smallness of this negative impact of BVH deformation are twofold: First, as mentioned in Section 3.1, the SAH used to build the BVH tends to automatically cluster logically connected parts of the model. For typical models where this is the case, the resulting BVH will therefore not deteriorate badly if the model gets animated, as logically connected parts usually behave similarly during animation. Of course, this argument *only* holds if there is some reasonable hierarchy exposed in the model, and the method can break if no such hierarchy is given, or is intentionally hidden in artificial worst-case scenes.

Second, as shown in Section 3.3 for large packets our algorithm is less susceptible to non-optimal BVH builds. In particular, if boxes become large — as happens for a box whose two children move away from each other — the speed impact is small: the enlarged parent box is likely to be traversed cheaply with the first hit test, and one of the children is likely to be culled cheaply by the frustum exit test. The only real case of bad deterioration is excessive overlap of many previously separated subtrees.

If the model does not contain any intrinsic structure at all – such as the BART model – then all bounding volumes will deteriorate. In that case, even if the traversal steps are still cheap the traversal will have to test many boxes, intersect many triangles, and yield bad performance. If at least some structure is available, however, the structures close to the leaves will remain intact, and only the boxes higher up in the hierarchy will grow large. These can be traversed quickly as described above.

This effect can also be seen in Figure 10, which shows the same plots as in the previous section (for the fairy model only, the other models yield similar graphs), but with varying packet size. As can be seen, the impact of BVH deformation is significant for packets of 2×2 rays – showing a factor of $2\times$ between best and worst BVH – but becomes significantly smaller for increasing packet size. For larger packets of 16×16, the traversal algorithm hides most of the impact of the BVHs deformation, leaving an impact of only 30% between best and worst BVH performance.

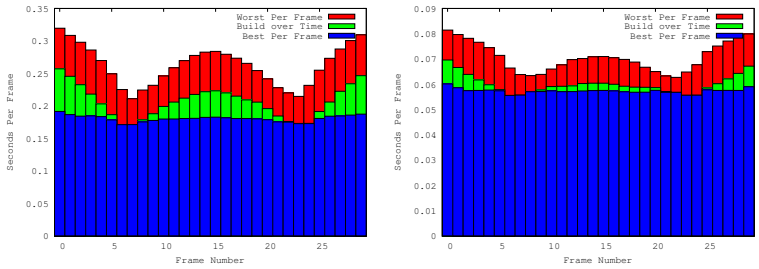

Fig. 10. Impact of BVH quality for 2×2, 4×4, and 16×16 ray packets (for the runner model only, other models exhibit similar behavior). Each graph shows the best and worst performance for any of the time step's deformed BVHs, as well as the best BVH over time performance. As can be seen, the difference between best and worst BVH can be significant for small packets, but diminishes for larger packet sizes.

6. DISCUSSION

In the previous sections, we have shown that our approach can deliver frame rates for static scenes that are comparable to the best known alternative techniques, and that this level of performance can also be sustained for a wide range of dynamic scenes. In this section, we discuss potential limitations of our method, as well as its relation to alternative acceleration

structures like kd-trees [Reshetov et al. 2005] and grids [Wald et al. 2006], as well as to several – concurrently developed – approaches for ray tracing animated scene [Wald et al. 2006; Stoll et al. 2006; Günther et al. 2006; Lauterbach et al. 2006].

6.1 Limitations

There are several limitations to our current approach. It is limited to deformable scenes, so only triangle positions can be changed. Thus applications with primitives such as adaptive meshes or particle systems with births and deaths could be a problem. Because we assume some reasonably smooth space of poses, our BVH might not be efficient for some models.

One such example was the BART scene, for which no apparent intrinsic hierarchy is available, and where any individual BVH deteriorates quickly during deformations. Such scenes cannot be handled by our approach, and are better handled by approaches that do not make any assumptions on the deformations, like a uniform grid [Wald et al. 2006]. Similarly, it is possible to devise scenes that can break the BVH even in the static case: if the scene consists only of long, skinny, and diagonal triangles spanning the entire scene's bounding box, no reasonable BVH hierarchy can be built, and the render time will be linear in the number of triangles[6].

For scenes composed of multiple objects we currently assume that all these objects are known in advance, and that the first pose exhibits enough of the natural hierarchy of the model to be found by the SAH build; or that their positions can at least be sampled if this is not the case. In practice, however, many interactive applications consist of several independent models that are moving incoherently, and no advance knowledge at all of how many of these models will be in the scene, nor where these will be, at any time. In that case the upper levels of the hierarchy that group the scene's individual objects would have to be rebuilt. Though we believe this is feasible, no implementation is available yet that would confirm this belief. Mark and Fussell [2005] have argued that future rendering systems would probably be coupled much more closely to scene graphs; in that case, part of the model's intrinsic hierarchy would be accessible through that scene graph, and exploiting this information could be highly advantageous in particular for such scenes.

6.2 Comparison to Alternate Approaches

Though fast ray tracing until recently was restricted to kd-trees, grids and BVHs have recently proven to be similarly effective. Not only do all three data structures allow for fast ray tracing, but at least one approach towards dynamic scenes has been shown for each of those data structures.

Comparison to other traversal methods. The grid, BVH, and kd-tree mostly differ in their traversal method. The coherent grid traversal [Wald et al. 2006] is arguably the most extreme in that it uses a *pure* frustum traversal, which does not consider individual rays at all during traversal (only the bounding frustum is being considered). As a result, the grid can benefit most from coherence – though doubling the number of rays in a given frustum eventually requires more ray-triangle intersections, the traversal cost is not affected at all. On the other side, the grid will therefore suffer more from incoherent rays, and has to invest more effort into making sure the packets are coherent. For similar reasons, the grid suffers

[6] A set of such scenes has been generated by Alexander Reshetov to find worst-case scenes for a kd-tree. These scenes are equally bad for both BVHs and kd-trees, though a k-t tree could in theory be optimized for that case.

more from higher geometric density, as all triangles overlapping the frustum may need to be intersected, even if those fall "in between" the raster of rays (also see Figure 3).

A pure packet traversal as used for kd-trees [Wald et al. 2001] and BVHs [Lauterbach et al. 2006] does not suffer from increased geometric density as much as a pure frustum traversal, but also cannot take as much benefit from coherence if available. This gap between frustum and packet traversal methods is being bridged by hybrid approaches like ours or Reshetov's MLRT. Though both are hybrids, MLRT is a two-stage process: MLRT traverses frusta in the upper levels of the hierarchy, and eventually switches to packet-traversal once this is considered advantageous. Though high performance is reported for this approach, the exact point when to switch from one method to the other is not obvious. In our method no such separation into different phases is required, as the advantages of packets and frusta are combined in each individual traversal step. On the other hand, the automatic adjustment of packet sizes used by the MLRT approach might also be beneficial for our system. Note, however, that our traversal method is not restricted to BVHs, but can similarly be applied to kd-trees as well.

Dynamic Scenes. For ray tracing dynamic scenes, four alternative approaches are currently available: Razor [Stoll et al. 2006], Coherent Grid Traversal [Wald et al. 2006], kd-tree Motion Decomposition [Günther et al. 2006], and Dynamic BVHs (ours and [Lauterbach et al. 2006]). The kd-tree based Motion Decomposition approach is the most restricted of these approaches, as it requires advance knowledge of the animation, but can achieve good results if this is available. The other extreme is the Coherent Grid Traversal [Wald et al. 2006], which makes no assumptions at all on the kind of deformation, and therefore can be applied even to scenes where the BVH cannot be applied. In a direct performance comparison our BVH is consistently faster than the Grid for all our test scenes except the BART scene (see Table VIII).

Scene	#tris	Frustum Grid dynamic	BVH (1st frame BVH) dynamic	BVH (best over time) dynamic
runner 1st frame	78k	11.7	21.8	18.8
toys 1st frame	11k	26.9	33.6	29.6
marbles 1st frame	9k	28.6	35.0	35.0
BART 100th frame	65k	12.5	0.1	8.3
fairy 1st frame	180k	1.9	7.1	6.4

Table VIII. Performance in frames/sec. (ray casting only, 1024×1024 pixels, one 2.6 GHz Opteron CPU) for the grid and BVH. Grid performance includes grid rebuild; BVH performance is given for both build over time as well as for the respective time step's BVH (timings excluding BVH build time, but including BVH update). For all scenes but BART, the BVH is also faster than the grid. Again with the exception of BART – for which the best over time is almost two orders of magnitudes faster than the deformed first frame – the first frame's BVH usually is within around 20% of the build over time.

Razor [Stoll et al. 2006] presents an interesting approach to ray tracing dynamic scenes with a kd-tree, but so far is only available as a non-interactive proof-of-concept prototype. Lauterbach's BVH and our method both build on deforming BVHs, and thus have the same restrictions in supported kinds of animation. While no direct comparison is available, we assume that our method is the faster one, as Lauterbach's BVH uses a pure 4-ray packet traversal, whereas Table III has shown that our traversal can provide up to a factor of 10 compared to such a pure 2×2 packet traversal.

6.3 Secondary Rays

So far, our experiments have mostly concentrated on primary rays only. For secondary rays, the BVH so far has shown to be surprisingly robust. For hard shadows, we follow the usual way of connecting the 8×8 or 16×16 primary rays to the same point light source to form a coherent ray packet. This method was already used in [Wald et al. 2001], and is also being used in the grid-based system. However, compared to both the kd-tree and the grid we have found that our BVH traversal is much more robust to packets with (some) incoherent rays. As discussed above, a pure frustum traversal suffers badly once the ray coherency drops, and a single incoherent ray can significantly widen the packet's frustum. To avoid this, a frustum-based approach has to split incoherent packets into several coherent ones.

The BVH, in contrast, has shown to be surprisingly stable for secondary rays (see Boulos et al. [2006] for a more detailed evaluation). If a packet contains an incoherent ray, the frustum will get large, and the early frustum reject can suffer. The first hit descent and first active tracking however will still work; and the number of triangle intersections are but moderately affected. In that example, such a packet would be at most twice as costly (due to a reduced culling efficiency of the frustum test, as well as a moderately increased number of visited leaves), whereas in the grid such a packet performs lots of additional triangle intersections, and consequently hurts much more.

In all of our experiments so far, hard shadows, soft shadows, reflections, refractions, etc could be supported by simply putting all rays of the same type into one packet (i.e., one reflection packet, one refraction packet etc). Though tracing more rays obviously decreases the frame rate, the "felt performance" for secondary rays in our experience is not significantly slower than for primary rays [Boulos et al. 2006]. In fact, there are several optimizations for secondary rays that we do not employ yet – such as terminating shadow rays on the *first* hit instead of at the closest. Though we have so far been quite content with our method's behavior for secondary rays, no exact performance breakdown for the individual ray types has been performed yet, and cases can probably be constructed for which the performance for secondary rays would deteriorate badly (such as Monte Carlo path tracing or a glassy sphere flake model).

7. CONCLUSION

In this paper, we have presented two main contributions: a novel ray packet traversal scheme for BVHs, and a method for BVH construction that allows many deformations of the same scene to reuse the same hierarchy. Taken together, these two techniques allow for ray tracing animated models at a performance that is competitive to the fastest published ray tracing performance for static models. Our approach combines ordered traversal, packet traversal, SIMD computations, early BVH hits, and MLRT-style early exits. This combination turns out to be so natural that the full implementation including all these concepts can be written up compactly.

Future work. To help compare different approaches, the ray tracing community needs a set of animation benchmarks similar in spirit to the static SPD database developed by Haines [1987]. There is some movement in this direction [Lext et al. 2000b] but more is needed. To address the limitation to deformable scenes, incremental trees that can change the number of leaves is worth investigating. To make the BVH more general, oriented bounding primitives could be helpful. There are several optimizations that could improve our run times. First, there are algorithmic techniques such as marking shadow rays once

they are occluded to prevent redundant traversal and intersection, exiting once all shadow rays are occluded [Smits 1998], and the same optimizations for architectural scenes that are being used in MLRT [Reshetov et al. 2005]. Second, there is considerable room for low-level optimizations. Although we already use SIMD extensions, most of the code is written for flexibility, simplicity, and portability, and makes use of templated high-level C++ code. We believe there is potential in further optimizing this code if more aggressive and architecture-specific coding were performed. Our algorithm could benefit from powerful hardware architectures such as IBM's Cell processor [Minor et al. 2005]. If our algorithm maps well to the Cell, which we believe it will, it could run at $10\times$ the speeds reported here and ray tracing on commodity game consoles might finally come into reach. Whatever kind of platform they will run on, future ray tracing systems also have to spend more attention on secondary rays, in particular on Cook-style ray tracing effects; though a first proof-of-concept implementation for our BVH has already been done [Boulos et al. 2006], a much deeper investigation of the relation of packets and secondary effects is overdue. Finally, adapting our system to interactive modeling applications or games would be the ultimate test of whether ray tracing could become an everyday interactive technique.

Acknowledgments

The toys and marbles animations were modeled by Andrew Kensler. The fairy animation has been created using DAZ Studio; the software and base models have been graciously provided by DAZ Productions (http://www.daz3d.com). All animations are available via the Utah Animation Repository (http://www.sci.utah.edu/~wald/animrep). We would like to thank Alexander Reshetov, Gordon Stoll, Bill Mark, Carsten Benthin, Thiago Ize, Steven G Parker, Aaron Knoll, Andrew Kensler, Johannes Günther, Heiko Friedrich, Christian Lauterbach, Sung-Eui Yoon, and Dinesh Manocha for insight into, and discussions about, their respective systems. Jesse Dylan Lacewell, Dave Edwards, and Joe Kniss are currently involved in an extended evaluation of secondary rays. Ingo Wald was supported by the State of Utah Center of Excellence program and the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions under grant W-7405-ENG-48. Peter Shirley was supported by NSF 03-06151 and the Utah Centers of Excellence Program. Solomon Boulos was supported by the Barry M. Goldwater Scholarship.

REFERENCES

ADAMS, B., KEISER, R., PAULY, M., GUIBAS, L. J., GROSS, M., AND DUTRÉ, P. 2005. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum 24,* 3 (Sept.), 677–684.

ADELSON, S. J. AND HODGES, L. F. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE CG&A 15,* 3, 43–52.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. *SJCC*, 27–45.

ARVO, J. AND KIRK, D. 1989. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, A. S. Glassner, Ed. Academic Press, San Diego, CA.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18,* 9, 509–517.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2006. Interactive distribution ray tracing. Tech. Rep. UUSCI-2006-022, SCI Institute, University of Utah.

CARR, N., HOBEROCK, J., CRANEH, K., AND HART, J. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface (submitted)*.

CLARK, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19,* 10, 547–554.

CLEARY, J., WYVILL, B., BIRTWISTLE, G., AND VATTI, R. 1983. A Parallel Ray Tracing Computer. In *Proceedings of the Association of Simula Users Conference*. 77–80.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

FOLEY, T. AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of HWWS*. 15–22.

GENETTI, J., GORDON, D., AND WILLIAMS, G. 1998. Adaptive supersampling in object space using pyramidal rays. *Computer Graphics Forum* 17.

GLASSNER, A. 1988. Spacetime ray tracing for animation. *IEEE CG&A 8,* 2, 60–70.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE CG&A 4,* 10, 15–22.

GOLDSMITH, J. AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE CG&A 7,* 5, 14–20.

GRÖLLER, E. AND PURGATHOFER, W. 1991. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics*. 103–113.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. In *Proceedings of Eurographics 2006*. (to appear).

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*. 47–57.

HAINES, E. 1987. A proposal for standard graphics environments. *IEEE CG&A 7,* 11, 3–5.

HAINES, E. 1991. Efficiency improvements for hierarchy traversal in ray tracing. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 267–272.

HAVRAN, V. 2001. Heuristic Ray Shooting Algorithms. Ph.D. thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

HURLEY, J. T., KAPUSTIN, A., RESHETOV, A., AND SOUPIKOV, A. 2002. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon*.

JANSEN, F. 1986. Data structures for ray tracing,. In *Proceedings of the Workshop in Data structures for Raster Graphics*. 57–73.

KAPLAN, M. 1985. The uses of spatial coherence in ray tracing. In *ACM SIGGRAPH '85 Course Notes 11*.

KAY, T. AND KAJIYA, J. 1986. Ray tracing complex scenes. In *Proceedings of SIGGRAPH*. 269–278.

KIRK, D. AND ARVO, J. 1988. The ray tracing kernel. In *Proceedings of Ausgraph*. 75–82.

LARSSON, T. AND AKENINE-MÖLLER, T. 2003. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. Tech. Rep. MDH-MRTC-92/2003-1-SE, MRTC. February.

LARSSON, T. AND AKENINE-MÖLLER, T. 2005. A dynamic bounding volume hierarchy for generalized collision detection. In *Workshop On Virtual Reality Interaction and Physical Simulation*. 91–100.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Tech. Rep. 06-010, Department of Computer Science, University of North Carolina at Chapel Hill.

LEXT, J. AND AKENINE-MÖLLER, T. 2001. Towards Rapid Reconstruction for Animated Ray Tracing. In *Proc. of Eurographics*. 311–318.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2000a. BART: A benchmark for animated ray tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden. May.

LEXT, J., ASSARSSON, U., AND MÖLLER, T. 2000b. BART: A benchmark for animated ray tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden. May.

MACDONALD, J. D. AND BOOTH, K. S. 1989. Heuristics for ray tracing using space subdivision. In *Proceedings of Graphics Interface*. 152–63.

MAHOVSKY, J. 2005. Ray Tracing with Reduced-Precision Bounding Volume Hierarchies. Ph.D. thesis, University of Calgary.

MAHOVSKY, J. AND WYVILL, B. 2004. Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *JGT 9,* 1, 35–46.

MARK, W. AND FUSSELL, D. 2005. Real-time rendering systems in 2010. Tech. Rep. 05-18, Computer Science, University of Texas. May.

MINOR, B., FOSSUM, G., AND TO, V. 2005. TRE : Cell broadband optimized real-time ray-caster. In *Proceedings of GPSx*.

MÖLLER, T. AND TRUMBORE, B. 1997. Fast, minimum storage ray triangle intersection. *JGT 2,* 1, 21–28.

MÜLLER, G. AND FELLNER, D. 1999. Hybrid scene structuring with application to ray tracing. In *Proceedings of International Conference on Visual Computing*. 19–26.

MUUSS, M. 1995. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*.

NG, K. AND TRIFONOV, B. 2003. Automatic bounding volume hierarchy generation using stochastic search methods. In *Mini-Workshop on Stochastic Search Algorithms*.

PARKER, S. 2002. Interactive ray tracing on a supercomputer. In *In Practical Parallel Rendering*, A. Chalmers and E. Reinhard, Eds.

PARKER, S. G., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B. E., AND HANSEN, C. D. 1999. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*. 119–126.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*. 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering*. Brno, Czech Republic, 299–306.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *Proceedings of ACM SIGGRAPH*. 1176–1185.

RUBIN, S. AND WHITTED, T. 1980. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*. 110–116.

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

SCHMIDL, H., WALKER, N., AND LIN, M. 2004. CAB: Fast update of OBB trees for coll. det. between articulated bodies. *JGT 9,* 2, 1–9.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*. 27–36.

SMITS, B. 1998. Efficiency issues for ray tracing. *Journal of Graphics Tools 3,* 2, 1–14.

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. 06-21, University of Texas at Austin Dep. of Comp. Science.

VAN DEN BERGEN, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *JGT 2,* 4, 1–14.

VAN DER ZWAAN, M., REINHARD, E., AND JANSEN, F. Pyramid clipping for efficient ray traversal. In *Rendering Technique '95, Proceedings EG Workshop on Rendering*.

WALD, I. 2004. Realtime ray tracing and interactive global illumination. Ph.D. thesis, Saarland University.

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

WALD, I. AND HAVRAN, V. 2006. On building good kd-trees for ray tracing, and on doing this in O(N log N). Tech. Rep. UUSCI-2006-009, SCI Institute, University of Utah. available at http://www.sci.utah.edu/~wald.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (to appear).* (Proceedings of ACM SIGGRAPH).

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

WEGHORST, H., HOOPER, G., AND GREENBERG, D. 1984. Improved computational methods for ray tracing. *ACM TOG 3,* 1, 52–69.

WHITTED, T. 1980. An improved illumination model for shaded display. *CACM 23,* 6, 343–349.

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An efficient and robust ray-box intersection algorithm. *JGT 10,* 1, 49–54.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of SIGGRAPH*. 434–444.