

# Instant Ray Tracing: The Bounding Interval Hierarchy

Carsten Wächter<sup>†</sup> and Alexander Keller<sup>‡</sup>

Abt. Medieninformatik, University of Ulm, 89069 Ulm, Germany



**Figure 1:** Images from animations and interactive applications generated by our new ray tracing algorithm at interactive rates on a single processor: *Quake II* (image on the right) can be played smoothly on a dual core processor with one shadow of a point light source, reflections, and refractions.

---

## Abstract

We introduce a new ray tracing algorithm that exploits the best of previous methods: Similar to bounding volume hierarchies the memory of the acceleration data structure is linear in the number of objects to be ray traced and can be predicted prior to construction, while the traversal of the hierarchy is as efficient as the one of kd-trees. The construction algorithm can be considered a variant of quicksort and for the first time is based on a global space partitioning heuristic, which is much cheaper to evaluate than the classic surface area heuristic. Compared to spatial partitioning schemes only a fraction of the memory is used and a higher numerical precision is intrinsic. The new method is simple to implement and its high performance is demonstrated by extensive measurements including massive as well as dynamic scenes, where we focus on the total time to image including the construction cost rather than on only frames per second.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Ray Tracing

---

## 1. Introduction

Ray tracing is the core technique of photo realistic image synthesis by global illumination simulation. It also underlies many other simulation methods. Although known for long, only recently realtime ray tracing became available. Current ray tracing algorithms owe their efficiency to an additional data structure that has to be constructed from the scene geometry beforehand. So far this had taken considerable amounts of time and memory. Consequently the pre-processing amortized only for environments that are static or contain moderate dynamics.

While graphics hardware cannot efficiently handle all effects of global illumination, software ray tracing easily can compete with high end graphics hardware images synthesis for massive geometry and even the acceleration data structures of ray tracing can be used to enhance the performance of rasterization. Compared to immediate mode rendering on a rasterizer, however, the construction time and memory footprint of the acceleration data structures are prohibitive.

Only little research has been carried out concerning the efficient construction of the acceleration data structure and advances for massive geometry and/or dynamic geometry were rather moderate until recently.

We propose a hierarchical acceleration data structure for ray tracing that can be constructed much more efficiently

---

<sup>†</sup> carsten.waechter@uni-ulm.de

<sup>‡</sup> alexander.keller@uni-ulm.de

than previous approaches. The procedure is so fast that interactive ray tracing of dynamic scenes becomes available on even mono-processor systems (see the snapshots in figure 1).

For the sequel we assume the reader's familiarity with the ideas of ray tracing [Gla89, Shi00].

## 2. Principles of Accelerating Ray Tracing

When tracing huge numbers of rays for image synthesis it obviously is not efficient to intersect each ray with all objects in the scene in order to find the closest point of intersection for each ray. It amortizes to construct additional data structures that allow one to exclude most of the objects from actual intersection testing.

To obtain such a data structure either the space containing the objects can be partitioned or the list of objects can be partitioned. While it is obvious to have a hybrid, we briefly sketch the two pragmatic approaches in the following subsections.

Aside of amortizing the construction cost of the acceleration data structure, there are situations, where the additional cost of not only tracing single rays but sets of rays can amortize, too. We will discuss these issues in section 4.3.

### 2.1. Partitioning Space

The space containing the objects is partitioned into disjoint volume elements. Efficiency is obtained by enumerating the volumes intersected by a ray and then testing the objects in the volumes for intersection. A major disadvantage of space partitions is that objects often are referenced more than once, since they may have non-empty intersections with more than one of the volume elements. This results in larger memory footprint and requires a mailbox mechanism to avoid performance losses by multiply intersecting one ray with the same geometric object.

#### 2.1.1. Regular Grids

The space is partitioned into a raster of identical rectangular axis-aligned volumes. This regular structure allows for simple algorithms that enumerate the volume elements along a given ray. As each volume contains a list of the objects that it intersects, only their objects are intersected with the ray.

The memory footprint of the acceleration data structure cannot be determined in advance, because objects can intersect multiple volume elements and thus requires dynamic memory management. The data structure is constructed by rasterizing the objects. This requires variants of an object-volume element intersection routine, which is numerically unreliable due to the finite precision of floating point arithmetic. Rasterizing the bounding box of the object is numerically stable, but increases the memory footprint.

The efficiency of this straightforward approach severely suffers from traversing empty volume elements, as especially explored for massive scenes. A solution to this problem is found in hierarchical grids that allow empty space to be traversed faster while still having a moderate number of objects per volume element. Switching between the levels of the hierarchy, however, is expensive and can be achieved more efficiently by other spatially adaptive schemes.

#### 2.1.2. Binary Space Partition

The binary space partition is a hierarchical data structure. The general idea is to adaptively subdivide space by using arbitrary planes, which allows one to overcome the efficiency issues of regular grids caused by empty volumes elements. In polygonal scenes an obvious choice is to use the planes determined by the polygons themselves. However, it is not known how to do this in an optimal way and randomized algorithms are expected to yield trees of quadratic size in the number of objects in the scene [MR95, Sec.9.7].

*kd*-trees restrict binary space partitions to using only planes that are perpendicular to the canonical axes. Since all normals of the subdivision planes coincide with one of the canonical axes unit vectors, scalar products and object-volume element intersection tests become more efficient and numerically robust than in the original binary space partitions. Still, the decision of how a partitioning plane intersects an object remains a numerical issue. Along with heuristics for subdivision *kd*-trees have been used very successfully for accelerating ray tracing [Wal04, Hav01, Ben06, Kel98, RSH05].

As with all spatial partitioning schemes, objects can reside in more than one volume element. Although the number of multiple references can be effectively reduced by allowing only partitioning planes through the vertices of the objects, the number of references cannot efficiently be bounded a priori. Consequently, memory management becomes an issue during the construction of the hierarchy. Heuristics used for memory estimation and allocation only hold true for common scenes, but can be way too pessimistic for others, or even worse, can result in various reallocations if the lists need to grow during the construction phase. This results in performance losses by memory fragmentation.

### 2.2. Partitioning Object Lists

When partitioning a list of objects, each object remains referenced at most once and it becomes possible to a priori predict memory requirements. In addition each object is intersected at most once with a ray and consequently mailboxes become redundant. As an unavoidable consequence the volumes enclosing groups of objects often cannot be disjoint.

#### 2.2.1. Bounding Volume Hierarchy

Bounding volume hierarchies have been introduced in [RW80, KK86] and often are used in industry since memory

requirements a priori can be bounded linearly in the number of objects. However, limits of the classic approaches were explored in the first ray tracing chip [Hal99] and efficient software implementations [GM03, WBS06] that remained inferior to the performance of *kd*-trees.

Implementing bounding volume hierarchies does not require object-plane intersection routines. As a consequence they are simpler to implement than spatial partitioning schemes. Using axis-aligned rectangular bounding volumes avoids any numerical stability issues during construction as only minimum/maximum operations are used.

There exist heuristics for both the bottom-up and top-down construction, which have been summarized in [WBS06]. The usual heuristic is to minimize the overall volume of all volume elements. These optimization procedures are prohibitively slow and in fact it is not clear what is the most efficient construction algorithm.

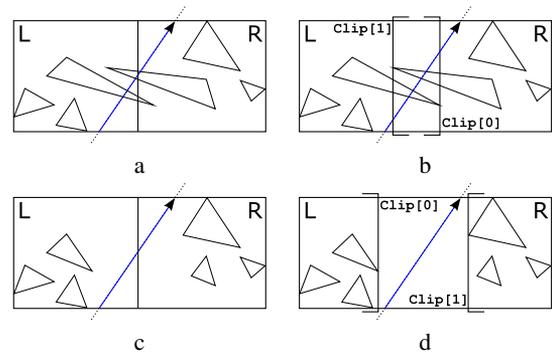
Severe performance penalties stem from the fact that opposite to binary space partitions the volume elements were not ordered. Usually all child nodes have to be intersected with a ray, because an early pruning was impossible due to a lack of a spatial order. This problem already has been identified in [FP93].

### 2.3. Summarizing the State of the Art

Certainly the simplicity, numerical robustness, and predictable memory footprint make bounding volume hierarchies most attractive for accelerating ray tracing [KK86, GM03]. However, the current performance is below what is obtained using *kd*-trees. At the price of hardly predictable memory requirements and numerical issues during the construction of the acceleration data structure, realtime performance is obtained [Wal04, Ben06, Bik05] for static and moderately dynamics scenes.

Both principle approaches of either dividing space or object lists suffer from construction routines that are far from realtime and use greedy algorithms, which certainly is another disadvantage. The most successful concept is the surface area heuristic [GS87, Hav01, Wal04, WBS06]. As it requires a fair amount of analysis of scene geometry and twiddling, the construction for a complex mesh can easily be in the range of minutes to even days [Wal04, WH06, WSB01, SBB\*06, WDS04].

While the construction times amortize for static scenes, very moderate dynamics [Wal04], or deformables only [WBS06], they are much more difficult to amortize in fully dynamic settings. Attempts to deal with fully dynamic scenes so far use regular grids [RSH00, WIK\*06] with all its disadvantages and are only efficient for scenes of moderate complexity.



**Figure 2:** Geometric primitives overlapping the splitting plane of a *kd*-tree have to be referenced in both child volume elements in a), while the bounding interval hierarchy in b) assigns an object to either the left *L* or right *R* child. Traversing a node of the *kd*-tree in c) requires to distinguish the four cases *L*, *R*, *LR*, or *RL*, whereas in the bounding interval hierarchy in d) the additional fifth case of traversing empty volume has to be considered.

## 3. The Bounding Interval Hierarchy

We propose a simple algorithm, which at the same time offers exceptional speed for both static and dynamic scenes, features much higher numerical precision, and allows one to a priori fix the memory footprint. It can be considered as the cross-over of the advantages of partitioning object lists and efficiently traversing spatial partitions.

Comparisons with two fully optimized state-of-the-art *kd*-tree based ray tracers (InView and our own implementation) show that it can outperform them for most scenes by a factor of two to even some orders of magnitudes for both total rendering time and overall memory consumption (as shown in figure 6).

### 3.1. Data Structure

Unlike classic bounding volume hierarchies [KK86, GM03], which store a full axis aligned bounding box for each child, the idea of the bounding interval hierarchy is to only store two parallel planes perpendicular to either one of *x*, *y*, and *z*-axis. Given a bounding box and the axis, the left child *L* results from replacing the maximum value along that axis by the first plane. In an analogue way the right child *R* results from replacing the minimum value by the second plane (see figure 2). Resulting zero volumes are used to represent empty children.

The inner nodes of the tree are described by the two clipping planes and a pointer to a pair of children. As this sums up to 12 bytes in total, all nodes are aligned on four-byte-boundaries. This allows one to use the lower two bits of the children-pointer to indicate the axis (00: *x*, 01: *y*, 10: *z*) or a leaf (case 11). Leaf nodes consist of a 32bit-pointer to the

---

```

typedef struct
{
  int Index;
  //lowest bits: axis (00,01,10) or leaf (11)
  union
  {
    int Items; //leaf only
    float Clip[2]; //internal node only
  };
} BIH_Node;

```

---

referenced objects and their overall number. The overhead of four bytes in the leafs (as they only use eight bytes out of the node data structure) can be resolved by a careful implementation.

### 3.2. Ray Intersection

Intersecting a ray with the bounding interval hierarchy binary tree is similar to traversing a bounding volume hierarchy. However, since the children are spatially ordered in the new data structure this can be implemented much more efficiently, since it is possible to directly access the child that is closer to the ray origin by the sign of the ray direction. The traversal thus becomes almost identical to that of a *kd*-tree, as illustrated in figure 2.

In analogy to bounding volume hierarchies, it is also possible to not intersect any child at all if the valid ray segment is between two non-overlapping children (see figure 2d). Handling this additional case is even beneficial, because it implicitly skips empty leafs. Consequently empty leafs can never be accessed and therefore do not need to be stored.

Opposite to spatial partitions, the volume elements of a bounding interval hierarchy can overlap and consequently the recursive traversal cannot stop as soon as an intersection is found. It is always necessary to test all remaining volume elements on the stack for closer intersections. However, as soon as an intersection is found, branches of the hierarchy can be pruned if they represent volume elements further away than the current intersection.

### 3.3. Construction of the Hierarchy

The key to the performance of our new data structure is the efficient construction. Assuming a split plane to be given, the algorithm is fairly simple: Each object is classified either left or right depending on which side of the plane it overlaps most. Then the two partitioning plane values of the child nodes are determined by the maximum and minimum coordinate of the left and right objects, respectively.

#### 3.3.1. A new Global Heuristic

What remains is to determine the split planes. Unlike previous approaches [GS87, Hav01, WH06, WBS06], a non-

greedy heuristic is used that is cheap to evaluate, because it does not explicitly analyze the objects to be ray traced.

As illustrated in figure 3a, we use candidate planes resulting from hierarchically subdividing the axis-aligned scene bounding box along the longest side in the middle. Note that all candidates in fact form a regular grid. If a candidate plane is outside the bounding box of a volume element to subdivide, we continue with candidates from the half, where the volume element resides (see figure 3b).

Together with the algorithm from the previous subsection, the object list is recursively partitioned and bounding boxes are always aligned to object bounding boxes. If a split plane candidate separates objects without overlap, the resulting split planes implicitly become tightly fitted to the objects on the left and right thus maximizing empty space (see figure 2d). Although the recursion terminates, when only one object is left, we define a number of objects, for which a recursion is efficient. Thus trees become flatter, memory requirements are reduced and the traversal cost is balanced against the cost of primitive intersections.

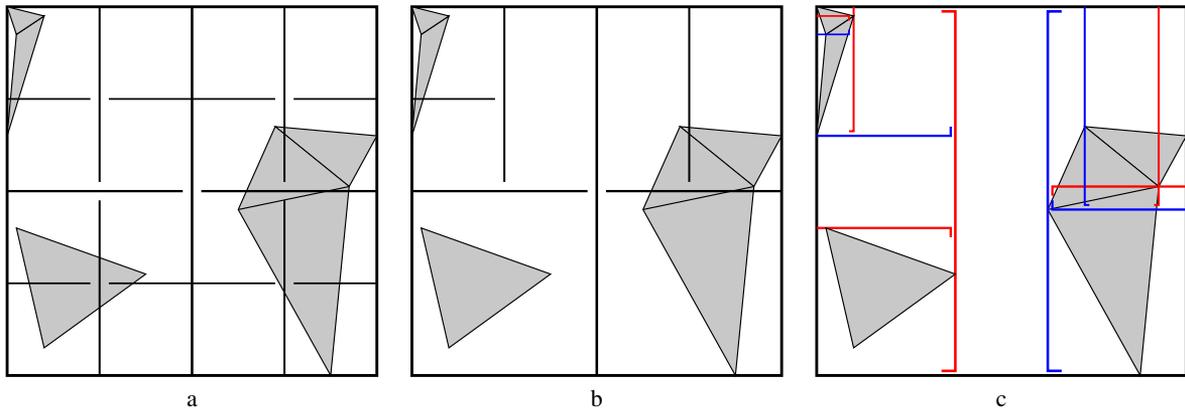
It is important to note that the split plane candidates are not adapted to actual bounding boxes of the inner nodes, but are solely determined by the global bounding box of the scene. This is the actual difference to previous approaches, which results in an adaptive approximation of the global regular grid as close as possible and bounding boxes as cubic as possible throughout the hierarchy.

#### 3.3.2. Approximate Sorting

The construction in fact is a sorting algorithm with a structure identical to quicksort and consequently runs in  $\mathcal{O}(n \log n)$  on the average. Using a bucket sorting preprocess similar to [FP93], the constant of the order can be decreased. Therefore the objects are processed in three steps:

1. The resolution of the regular grid is determined by the ratio of the scene bounding box divided by the average object size. As a spatially uniform distribution of objects is highly unlikely, this number is scaled down by a user parameter (we used the factor  $\frac{1}{6}$  for the measurements in figure 10) in order to avoid too many empty cells. Each grid cell consists of a counter that is initialized to zero.
2. One point of every object (e.g. one corner of its bounding box) is used to increment the counter in the grid cell containing that point. In the end the sum of all counters equals the number of objects. We now transform the counters to offsets by replacing each counter by the sum of all previous counters.
3. A global object index array is allocated and using the same point of every object, the objects now can be sorted into the buckets using the offsets from the previous step. For each bucket we compute the bounding box of the objects it contains.

Sorting the bounding boxes instead of the objects they contain speeds up construction by a factor of two to three. If



**Figure 3:** Illustration: a) Split plane candidates from hierarchically subdividing along the longest side of the axis-aligned bounding box. b) Used candidates and c) resulting binary interval hierarchy (red: left child L, blue: right child R).

a volume element consists of one container only, the container is replaced by the objects within. The resulting trees are very similar in rendering performance and size (see figure 10). Note that all other measurements in this paper have been performed without the approximate sorting preprocess.

This simple algorithm partitions a scene in linear time using a limited amount of memory. Even the index array can be processed in chunks.

### 3.3.3. Implementation Details

Because the bounding interval hierarchy is an object partitioning scheme, all object sorting can be done in place and no temporary memory management is required. The recursive construction procedure only needs two pointers to the left and right objects in the index array similar to the quick-sort algorithm.

On the other hand spatial partitioning schemes need to handle objects that overlap volume elements. For example the recursive *kd*-tree construction needs a vast amount of temporary data to be placed on the stack to be able to continue with backtracking later on.

We like to note that a variant of the above scheme can alleviate these inefficiencies and makes in place sorting available for *kd*-trees. The procedure requires a second array of object references that is used to keep the objects that are classified both left and right. Testing with a large number of scenes revealed that the size of this array can be chosen by a default value (a length equal to the number of objects is far more than one will ever need in 99 percent of the cases). However, because the real length of the array cannot be predicted, it might be necessary to reallocate memory. The procedure is illustrated in figures 4 and 5.

### 3.4. Construction on Demand

So far the presented framework already allows for interactive ray tracing. However, construction time and memory footprint of the acceleration data structure can be further optimized by constructing it only, where rays traverse, i.e. where geometry is "visible".

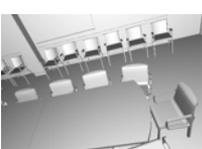
The implementation with the bounding interval hierarchy is fairly simple and especially beneficial for large scenes that feature a high depth complexity. Since all object sorting is done in place, only a flag is required to mark volume elements that have not yet been subdivided. Upon traversal of a ray, the subdivision routine is called if the flag is set. A simple optimization is to subdivide a node completely, if all objects contained in it fit into the cache (e.g. L1- or L2-cache). The on demand construction removes the classic separation of traversal and construction routines.

Using this simple extension we are able to render the Boeing 777 mesh at  $1280 \times 1024$  resolution in 3-9 minutes (depending on camera position) from scratch on a single core of an Opteron 875 2.2 GHz machine with 32GB RAM. Compared to previous approaches we only use a fraction of memory (see figure 8). The total rendering time of one view thus matches the time that InTrace's InView/OpenRT 1.4 uses to just build the acceleration data structure of the more than 350 times smaller conference room.

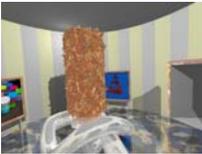
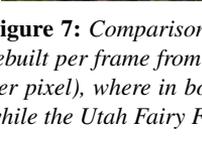
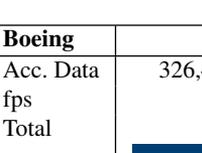
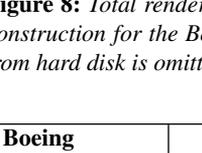
## 4. Discussion

The bounding interval hierarchy is an object partitioning scheme that benefits from the efficient traversal techniques of spatial partitioning schemes. In the sequel we point out the advantages of our approach.

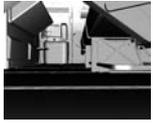


	<b>Shirley Scene 6</b>	InView	WH06	kd	BIH	on demand
	Triangles	1,380	n.a.	804	dto.	dto.
	Triangle memory	66,240	n.a.	28,944	dto.	dto.
	Acc. Data memory	115,312	n.a.	55,188	12,828	11,972
	fps	5.02	n.a.	11.17	11.99	n.a.
	Time to image (msec)	199	n.a.	89	83	87
	<b>Stanford Dragon</b>	InView	WH06	kd	BIH	on demand
	Triangles	863,334	863k	871,414	dto.	dto.
	Triangle memory	41,440,032	n.a.	31,370,904	dto.	dto.
	Acc. Data memory	26,207,404	n.a.	24,014,264	13,466,176	5,175,936
	fps	2.49	n.a.	5.92	5.98	n.a.
	Time to image (msec)	44,500	23,900	3,106	1,557	1,102
	<b>Stanford Buddha</b>	InView	WH06	kd	BIH	on demand
	Triangles	987,361	1.07M	1,087,716	dto.	dto.
	Triangle memory	47,393,328	n.a.	39,157,776	dto.	dto.
	Acc. Data memory	32,518,372	n.a.	30,566,796	17,344,944	2,719,628
	fps	3.13	n.a.	7.55	7.41	n.a.
	Time to image (msec)	53,819	32,200	3,695	1,837	705
	<b>BART Trafo. Kitchen</b>	InView	WH06	kd	BIH	on demand
	Triangles	111,116	n.a.	110,561	dto.	dto.
	Triangle memory	5,333,568	n.a.	3,980,196	dto.	dto.
	Acc. Data memory	9,989,240	n.a.	5,812,276	1,792,880	1,145,972
	fps	1.77	n.a.	4.65	1.76	n.a.
	Time to image (msec)	16,565	n.a.	871	724	770
	<b>Ward Conference</b>	InView	WH06	kd	BIH	on demand
	Triangles	964,471	n.a.	1,064,498	dto.	dto.
	Triangle memory	46,294,608	n.a.	38,321,928	dto.	dto.
	Acc. Data memory	101,627,372	n.a.	84,222,332	16,007,852	1,331,780
	fps	2.9	n.a.	9.55	4.12	n.a.
	Time to image (msec)	171,344	n.a.	11,204	1,523	630
	<b>Stanford Bunny</b>	InView	WH06	kd	BIH	on demand
	Triangles	70,027	69k	69,451	dto.	dto.
	Triangle memory	3,361,296	n.a.	2,500,236	dto.	dto.
	Acc. Data memory	6,186,288	n.a.	4,352,248	974,080	504,744
	fps	3.53	n.a.	9.9	10.2	n.a.
	Time to image (msec)	9,283	4,800	445	176	165
	<b>Car 1</b>	InView	WH06	kd	BIH	on demand
	Triangles	313,460	n.a.	312,888	dto.	dto.
	Triangle memory	15,046,080	n.a.	11,263,968	dto.	dto.
	Acc. Data memory	26,785,196	n.a.	15,093,468	4,989,168	1,271,168
	fps	3.15	n.a.	7.97	6.99	n.a.
	Time to image (msec)	39,817	n.a.	1,656	581	371
	<b>Blender Suzanne</b>	InView	WH06	kd	BIH	on demand
	Triangles	252,436	n.a.	251,904	dto.	dto.
	Triangle memory	12,116,928	n.a.	9,068,544	dto.	dto.
	Acc. Data memory	12,508,532	n.a.	12,139,800	3,707,292	2,083,020
	fps	3.84	n.a.	7.35	8.31	n.a.
	Time to image (msec)	18,260	n.a.	1,229	448	359

**Figure 6:** Comparison of the new technique and state-of-the-art kd-tree implementations, using a very simple shader and 2x2 (SSE accelerated) ray bundles (640x480 pixels, measured on a Pentium 4HT 2.8 GHz, whereas [WH06] performance data were measured on a faster Opteron 2.6 GHz). The varying number of triangles results from the VRML converter that we used to generate input for InView. Since the free version of InView is limited to  $10^6$  triangles, we removed invisible triangles from larger models for InView. Time to image measures the total rendering time for one picture, thus including (on demand) tree construction, ray tracing, and shading. The bounding interval hierarchy is clearly superior in memory and total time to image.

	<b>BART Museum (10,412 triangles)</b>	kd	BIH	on demand
	a) Average fps (primary rays only)	3.48	3.34	3.26
	Rendering time for complete animation (msec)	86,286	89,935	92,327
	b) Average fps (average 3.917 rays per pixel)	0.91	0.79	0.78
	Rendering time for complete animation (msec)	329,060	381,754	388,114
	<b>BART Museum (75,884 triangles)</b>	kd	BIH	on demand
	a) Average fps (primary rays only)	0.39	2.04	2.08
	Rendering time for complete animation (msec)	776,568	147,002	144,444
	b) Average fps (average 4.024 rays per pixel)	0.28	0.49	0.48
	Rendering time for complete animation (msec)	1,057,259	614,503	620,728
	<b>BART Kitchen (110,561 triangles)</b>	kd	BIH	on demand
	Average fps	1.45	1.96	2.17
	Rendering time for complete animation (msec)	552,207	407,460	368,903
	<b>BART Robots (71,708 triangles)</b>	kd	BIH	on demand
	Average fps	1.51	1.41	1.49
	Rendering time for complete animation (msec)	530,561	567,988	537,974
	<b>Utah Fairy Forest (174,117 triangles)</b>	kd	BIH	on demand
	Average fps	0.78	1.79	1.95
	Rendering time for complete animation (msec)	26,780	11,695	10,771

**Figure 7:** Comparison using dynamic environments (640x480 pixels, Pentium 4HT 2.8 GHz). The complete data structure is rebuilt per frame from scratch. The museum is traced using a) simple shading and b) full shading (using an average of 4 rays per pixel), where in both cases only single rays are traced. The remaining BART scenes are rendered with the simple shader, while the Utah Fairy Forest uses full shading.

Boeing	View 1	View 2	View 3	View 4	View 5	View 6
Acc. Data	326,447,848	12,748,120	15,471,692	259,261,404	50,963,768	324,602,460
fps	0.26	0.13	0.13	0.38	0.11	0.34
Total	8 min.	133 sec.	153 sec.	270 sec.	118 sec.	252 sec.
						

**Figure 8:** Total rendering times (1280x1024 pixels, single core of an Opteron 875 2.2 GHz 32GB) including on demand tree construction for the Boeing 777 data set (349,569,456 triangles amounting 12,584,500,416 bytes). Reading the triangle data from hard disk is omitted, since it heavily depends on the hard disks used (in our case 40-90 seconds loading time).

Boeing	View 1	View 2	View 3	View 4	View 5	View 6
Peak memory	1,075,418,112	538,697,728	1,054,035,968	1,267,494,912	1,078,779,904	1,190,248,448
Rendering time	5 min. 19 sec.	33 sec.	68 sec.	6 min. 45 sec.	95 sec.	5 min. 57 sec.

**Figure 9:** Rendering times (1280x1024 pixels, Pentium 4HT 2.8 GHz 2GB RAM, 349,569,456 triangles), including low level on demand tree construction and loading all necessary triangle groups from disk. The top level bucket sort preprocess, done once for all views, takes additional 53 minutes but only uses a peak 737 MBytes of RAM. The cache sizes for the preprocess and rendering were chosen to be suitable for any consumer machine offering at least 1 GB of RAM. More RAM allows for even faster rendering times (see previous figure), whereas the preprocessing step is mainly limited by the (slow and cheap) hard disk.

<b>Stanford Buddha</b>	WH06	BIH+Bucket	BIH	<b>Ward Conference</b>	BIH+Bucket	BIH
Triangles	1.07M	1,087,716	dto.	Triangles	1,064,498	dto.
fps	n.a.	94%	100%	fps	96%	100%
Construction (msec)	32,200	765	1,703	Construction (msec)	937	1,281
<b>Stanford Dragon</b>	WH06	BIH+Bucket	BIH	<b>Car 1</b>	BIH+Bucket	BIH
Triangles	863k	871,414	dto.	Triangles	312,888	dto.
fps	n.a.	93%	100%	fps	100%	100%
Construction (msec)	23,900	657	1,390	Construction (msec)	250	438
<b>Stanford Thai Statue</b>	WH06	BIH+Bucket	BIH	<b>UNC Power Plant</b>	BIH+Bucket	BIH
Triangles	10M	10,000,000	dto.	Triangles	12,748,510	dto.
fps	n.a.	94%	100%	fps	79%	100%
Construction (msec)	61,000	7,812	17,484	Construction (msec)	11,609	20,282

**Figure 10:** Comparison of the bounding interval hierarchy with/without using the bucket sort preprocess (640x480, Pentium 4HT 2.8 GHz) to numbers taken from [WH06], where a faster Opteron 2.6 GHz processor was used. fps are given relative to the "pure" bounding interval hierarchy, as several camera positions fps were averaged. The bounding interval hierarchy is the clear winner.



<b>Car 2 (542,108 triangles)</b>	kd	BIH	on demand
Triangle memory	19,515,888	dto.	dto.
Acc. Data memory	23,756,320	8,807,636	8,132,108
fps (1st Pass)	0.46	0.44	n.a.
Time to image (msec)	4,595	2,944	2,830

**Figure 11:** Comparison of the new technique and our state-of-the-art kd-tree implementation, using advanced shaders that trace single rays only (640x480 pixels, Pentium 4HT 2.8 GHz). Time to image measures the total rendering time for one picture, thus including (on demand) tree construction, ray tracing, and shading. This is a stress test for the on demand construction, because the global illumination computations requires to construct almost the full tree.

to trace sets of rays benefit from our new data structure. This is certainly due to the reduced memory bandwidth and increased cache coherency resulting from the small memory footprint and the fact that the bounding interval hierarchy always is flatter than the kd-tree. Furthermore the volume elements appear generally larger than the corresponding volumes of a kd-tree, which relaxes the conditions on coherency. Our experiments reveal that the speedup-ratio from single ray to 2x2-ray-bundle-tracing is slightly higher for the bounding interval hierarchy as compared to a kd-tree.

The frustum culling techniques introduced in [RSH05] have been successfully transferred to bounding volume hierarchies in [WBS06]. These techniques easily can be transferred to the bounding interval hierarchy by tracking the current volume element bounding box on the stack. Although our hierarchy can be updated in the fashion of [WBS06], too, our construction routine is much faster than the surface area heuristic and removes the severe restriction to meshes animated by only deformations. Then combining our techniques with the traversal techniques from [WBS06] seems to be ideal for rays that can be grouped into shafts.

Note that shaft culling techniques become problematic for diverging sets of rays as it naturally happens over the distance.

#### 4.4. Hardware Considerations

Based on the recent findings in realtime ray tracing the RPU-chip [WSS05] has been designed. While the architecture efficiently can ray trace and shade bundles of rays, it can be easily improved by our approach: The bounding interval hierarchy has a much smaller memory footprint and as an object partitioning scheme does not need a mailbox unit. Only the TPU unit has to be extended by a second plane intersection. These modifications easily can be incorporated due to the similarity of our traversal to a kd-tree traversal.

More important the construction algorithm (Sec.3.3) uses only simple operations and therefore is a very good candidate for hardware implementation.

#### 4.5. Massive Data Sets

Current data sets used in industrial applications and production rendering consist of massive amounts of geometry, which usually range from hundreds of megabytes to several gigabytes of raw data. Although we demonstrated (Sec. 3.4) that the small memory footprint of the bounding interval hierarchy allows massive scenes to be efficiently ray traced by simple means, there still may be situations, where the data does not fit into the main memory.

Along the lines of [WSB01], we implemented a minimal memory footprint renderer which is able to render pictures of the Boeing 777 using only 50 MB of RAM. If more RAM is available (we assumed 1 GB for our measurements) it is possible to render a picture from scratch in less than an hour even on a standard consumer desktop PC (see figure 9).

To achieve the minimal memory usage, we use the proposed preprocessing step to sort the triangles into buckets and store these on the hard disk. For the rendering step, a top level bounding interval hierarchy is created out of the buckets (without needing to touch any triangle). Each bucket that is intersected by a ray creates its own tree using the on demand policy. The bucket's triangles and the acceleration data structure are kept in a cache of either dynamic (able to grow until no more RAM is available) or fixed user-defined size. The bucket with the largest number of triangles defines the maximum memory footprint. Note that this results for free from the bucket sorting preprocess.

In this scenario the processing speed is determined by the speed of the hard disks. Our tree construction algorithm is so fast, that if parts of the acceleration data structure have to be flushed, they are just thrown away and rebuilt on demand.

#### 4.6. Large Objects

One might argue that the bounding interval hierarchy performance suffers when encountering a mixture of small and large geometric elements. While this is partially true, it is also true for spatial partitioning schemes.

In this situation a *kd*-tree subdivides the scene by inserting more splitting planes. This results in deeper trees, a duplication of object references, and an overall increased memory footprint. Deeper trees increase the traversal time.

The performance problem of bounding interval hierarchies in such a scenario can be spotted by the example of the BART robots (see figure 7). The scene is made up of large triangles for the streets and houses, but also features a lot of finer geometry like signs or the walking robots. As the large triangles cause large overlapping volumes in the hierarchy, an early pruning of the tree becomes impossible and more triangles per ray have to be tested.

The classic workaround in a rendering system is to subdivide large objects beforehand. In order to moderately increase memory, the objects should be divided by planes perpendicular to the canonical axis. While the memory consumption now increases similar to the *kd*-tree, it is still possible to a priori determine memory consumption.

As our approach is intended for production systems with displacement mapping and lots of geometric detail, the above discussion does not impose problems. In fact the problem only persists for low-polygon-count architectural scenarios and even older games already use 200,000 to 500,000 visible triangles per frame.

## 5. Concluding Remarks

We presented new concepts to accelerate ray tracing especially when used in fully dynamic environments or for massive data sets. Both the memory footprint and construction time of our new data structure are much smaller as compared to previous approaches. For the first time the fast construction allows for realtime ray tracing of dynamic content without restrictions to the geometry [WBS06]. It also enables the much more efficient computation of motion blur [Kel03]. The simplicity and predictability of the algorithm along with its global heuristic make it a premier candidate for a hardware implementation.

It is worth to mention that building *kd*-trees based on the global heuristic from section 3.3.1 results in a frame rate similar to the highly optimized *kd*-tree implementation used for the measurements.

First experiments using the bounding interval hierarchy with free form surfaces were very promising. We also extended the data structure from section 3.1 to represent a complete left and right interval instead of only clipping intervals. While the tree becomes even flatter the memory consumption is almost identical due to the required four floats per node. In addition the time complexity of the sequential software implementation is almost identical, too. However, a hardware implementation could benefit from the flatter trees.

## Acknowledgements

The authors would like to thank mental images GmbH for support and for funding of this research. Thanks to David Kasik from The Boeing Company for providing the Boeing 777 model data. The BRDF data are courtesy of SpheronVR AG. The Quake II original source is available from id soft.

## References

- [Ben06] BENTHIN C.: *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.
- [Bik05] BIKKER J.: *Interactive Ray Tracing*. Intel Software Network (2005).
- [FP93] FOURNIER A., POULIN P.: A Ray Tracing Accelerator Based on a Hierarchy of 1D Sorted Lists. In *Graphics Interface '93* (1993), pp. 53–61.
- [Gla89] GLASSNER A.: *An Introduction to Ray Tracing*. Academic Press, 1989.
- [GM03] GEIMER M., MÜLLER S.: A Cross-Platform Framework for Interactive Ray Tracing. *Graphiktag im Rahmen der GI Jahrestagung, Frankfurt am Main* (2003).
- [GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications* 7, 5 (May 1987), 14–20.
- [Hal99] HALL D.: The AR250-A new Architecture for Ray Traced Rendering. In *Proceedings of the Eurographics/SIG-GRAPH workshop on Graphics hardware-Hot Topics Session* (1999), pp. 39–42.

- [Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Praha, Czech Republic, April 2001.
- [Kel98] KELLER A.: *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, Shaker Verlag Aachen, 1998.
- [Kel03] KELLER A.: Strictly Deterministic Sampling Methods in Computer Graphics. *SIGGRAPH 2003 Course Notes, Course #44: Monte Carlo Ray Tracing* (2003).
- [KK86] KAY T., KAJIYA J.: Ray Tracing Complex Scenes. In *Proceedings of SIGGRAPH 86* (Aug. 1986), vol. 20/4, pp. 269–278.
- [MR95] MOTWANI M., RAGHAVAN P.: *Randomized Algorithms*. Cambridge University Press, 1995.
- [RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (2000), pp. 299–306.
- [RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics* 24, 3 (2005), 1176–1185.
- [RW80] RUBIN S., WHITTED J.: A 3-dimensional representation for fast rendering of complex scenes. In *Computer Graphics (Proceedings of SIGGRAPH 80)* (1980), vol. 14, pp. 110–116.
- [SBB\*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S.: An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2006). To appear.
- [Shi00] SHIRLEY P.: *Realistic Ray Tracing*. AK Peters, Ltd., 2000.
- [Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.
- [WBS06] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* (2006). To appear.
- [WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering* (2004), pp. 81–92.
- [WH06] WALD I., HAVRAN V.: On building fast *kD*-trees for Ray Tracing, and on doing that in  $O(N \log N)$ . *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-009* (2006).
- [WIK\*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)* (2006). To appear.
- [WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering)* (2001), S.J.Gortler, K.Myszkowski, (Eds.), Springer, pp. 277–288.
- [WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics* 24, 3 (2005), 434–444.