

Ray Tracing with the Single Slab Hierarchy

Martin Eisemann, Christian Woizischke, Marcus Magnor

Computer Graphics Lab, TU Braunschweig

Email: {eisemann,magnor}@cg.tu-bs.de, christian@woizischke.de

Abstract

Bounding volume hierarchies have become a very popular way to speed up ray tracing. In this paper we present a novel traversal and approximation scheme for bounding volume hierarchies, which is comparable in speed, has a very compact traversal algorithm and uses only 25% of the memory, compared to a standard bounding volume hierarchy.

1 Introduction

Ray tracing is one of the fundamental methods in computer graphics for image synthesis and has been studied for more than three decades. While ray tracing has been used mainly for offline rendering in the early days, recent advances in this field allow for real-time rendering as well [19, 27].

The biggest gain in efficiency is achieved by using acceleration data structures which lower the complexity of tracing rays from linear to logarithmic on the average. For a long time, kD-trees were supposed to be the best choice for these acceleration data structures [27, 22]. But recently Bounding Volume Hierarchies (BVH) and hybrids have become increasingly more popular, due to their versatility and lower memory footprint [15, 25, 10, 28, 9].

However for rather complex scenes with millions of triangles, even when using BVHs, memory consumption can become a limiting factor. Approximation schemes exist [17, 3], but usually include a computational overhead.

In this paper we present a new hierarchical acceleration data structure, which is comparable in performance to highly optimized BVH ray tracers, but that uses only 25% of the memory of a BVH, allowing for more complex scenes to be rendered. We will show that this data structure can be easily integrated into existing ray tracing systems and has a very simple and efficient traversal algorithm similar to kD-trees, without any of the drawbacks from

previous hybrid schemes.

The remainder of the paper is organized as follows. In Section 2 we give a short overview of common acceleration data structures to speed up ray tracing. We assume that the reader is familiar with the basic principles of ray tracing [8, 24, 20]. Section 3 introduces our new acceleration data structure, whose performance is evaluated in Section 4. We conclude our paper in Section 5.

2 Related Work

It is obvious that intersecting every ray with every object in the scene is barely efficient. Acceleration data structures can be used to exclude most of the objects from actual intersection testing. Large amounts of research has been done on finding efficient representations for these acceleration data structures for ray tracing. Good surveys can be found in [1, 24, 31] and [30]. For brevity we can only mention some seminal and recently published work in this field.

The broader class of these data structures can be divided into two main classes: *Space partitioning* and *object lists partitioning*, which we will briefly describe in the following.

2.1 Space Partitioning

The space of the scene, containing all objects, is subdivided into disjoint volume elements (voxels). Efficiency is achieved by testing the ray in an intelligent way against these elements and the contained geometry, preferably in a front-to-back order, if the element was pierced by the ray. While able to achieve high performance, some disadvantages are inherent. Multiple references to the same object are needed if it overlaps with more than one volume element. This requires an intelligent mailboxing to prevent multiple intersection tests with the same object and induces larger memory footprints.

Space partitioning schemes are also generally more complicated to update, e.g., if dynamically changing scenes need to be rendered.

Uniform Grids by Fujimoto *et al.* [5] divide the scene into regular, or uniform, disjoint but connected voxels. Using a fast 3DDDA algorithm a ray can be sent through this grid and one needs to test only the objects contained inside the pierced voxels. Its simplicity is also attractive for ray tracing dynamic scenes [29] as the creation of the grid can be achieved by fast rasterization of the objects. However, this scheme suffers severely from traversing empty voxels if the scene contains irregularly distributed objects. This can be alleviated by introducing hierarchical grids or macro cells, but switching between these levels is expensive. Grids are hardly usable for more complex scenes, as these require finer grids in order to limit the number of objects in the voxels, which on the other hand introduces a larger traversal overhead plus very high memory consumption.

Octrees have been introduced as a space partitioning scheme by Glassner [7]. The scene is enclosed by a voxel which is recursively subdivided into eight disjoint child nodes of equal size. This acceleration scheme adapts better to the scene content than the uniform grid. But adaptation and rendering is comparably slow to other acceleration data structures. Therefore octrees are usually only used for special ray tracing applications as in [14], but seldomly in real-time ray tracers for arbitrary scenes.

Binary Space Partitioning schemes recursively subdivide space into two half-spaces using arbitrary planes [2, 12, 6]. If the splitting planes are chosen carefully, this effectively reduces the drawbacks of the formerly mentioned acceleration data structures. However, it is not known how to choose these planes in an optimal way.

kD-trees restrict the splitting planes to be orthogonal to the world coordinate axes. This way traversal becomes easier, less computationally intensive and numerically more robust. Even though kD-trees are arguably the fastest way to ray trace static scenes, [27, 31, 26, 21], they suffer from the lack of adaptability for dynamic scenes, plus they incur an a-priori unknown memory footprint. Even though single nodes in a kD-tree allow for a very

compact representation, overall memory consumption can become several times as high as object lists partitioning schemes [25].

2.2 Object Lists Partitioning

Instead of subdividing space, one could also subdivide the objects into different, possibly overlapping, bounding volumes (BV). Rays are first tested against these BVs and only if a hit is found, then they are tested against the contained geometry. Efficiency is gained, as testing a ray against these BVs is simpler than the triangle tests.

Bounding Volume Hierarchies, first proposed by Rubin and Whitted [23] and Kay and Kajiya [13], carry this idea further by creating a hierarchy of bounding volumes. The bounds of every node in this tree are chosen so that it exactly bounds all the nodes in the corresponding subtree and every leaf node exactly bounds the contained geometry. If a ray misses a BV in this tree structure, the whole subtree can be skipped. Tracing rays in front-to-back order becomes a little bit more difficult with BVHs when compared to space partitioning schemes, but the advantages often outweigh the drawbacks: memory requirements are always directly proportional to the number of objects in the scene, each object is referenced exactly once and consequently no mailboxing is necessary. BVHs are efficient (it has been shown by several authors [10, 15, 28, 9], that BVHs can become almost as efficient as kD-Trees) and versatile when it comes to updating a BVH, e.g. for dynamic scene changes. These advantages have made them very attractive for real-time ray tracing during the last years [28, 9, 11, 15].

Memory Reduction Even though BVHs already need only a relatively low amount of memory compared to other acceleration data structures (kD-trees may need up to $4\times$ the amount of a BVH [25, 9]), several approaches exist which reduce the memory consumption even further. Mahovsky and Wyvill [17] investigated a hierarchical scheme for encoding BVHs that reduces the storage requirements by 63%-75% but introduces a computational overhead which can be alleviated by tracing bundles of rays, if possible. Cline *et al.* [3] take a similar approach. They also use a hierarchical encoding scheme, com-

pressing a node to 12 bytes, plus a high branching factor of 4 and implicit encoding of the child pointer due to a heap like data structure, typically a BVH node needs 32 bytes. Their approach is slower than a standard BVH and is limited to an object based median split technique for creating the BVH, which is known to be one of the slowest techniques for ray tracing.

Hybrid techniques have been thoroughly investigated during the last years which try to combine the benefits of spatial and object list partitioning. Most of them are a deviation of the Bounding Slab Hierarchy by Kay and Kajiya [13]. Originally at least six bounding planes were used (the standard BVH uses this scheme, where the bounding planes are perpendicular to the world coordinate axes), so that they formed a closed hull around the object. Most hybrid approaches store a set of two parallel planes that partition the current node’s bounding volume into two, potentially overlapping, halves. By saving the current active ray interval, one is able to estimate a hit or miss of a ray with these sparsely represented bounding volumes. The data structure looks like a kD-Tree but still has all the benefits from a BVH. This kind of hybrid was independently developed by several researchers [25, 10, 32, 33]. While Wächter *et al.* [25] focused on fast construction of their bounding interval hierarchy, Woop *et al.* [32] showed a hardware implementation of a similar structure, called b-kD tree. The DE-tree in [33] is very similar to the b-kD tree but includes a wide object isolation to prevent larger objects from plugging the hierarchy, by keeping them at higher levels. Havran *et al.* [10] adapted a version of the skd-tree, originally proposed by Ooi *et al.* [18], which is similar to the one proposed in [25]. As most of the other approaches have difficulties dealing with empty spaces and need to include empty nodes, Havran *et al.* extended them to H-trees, which is an skd-tree with an additional node type, the bounding node, a simple six-sided bounding volume, to cut away empty spaces. Even though this solves the empty spaces problem, it complicates the traversal and creation of the hierarchy, as one has to deal with different kinds of nodes.

In the following we present an approach that does not have this empty spaces problem as we always carve away the side which is most beneficial for the surface reduction of the nodes, plus due to the use

of a single type of node, the traversal is much simpler. It uses only 8 Bytes per node, i.e. 25% of the memory needed by a standard BVH without introducing any efficiency drawbacks. Often it is even faster than a standard BVH.

3 The Single Slab Hierarchy

We propose a simple yet efficient acceleration data structure for ray tracing which offers favorable speed and very low memory consumption. The single slab hierarchy (SSH) is a binary object partitioning scheme, resulting in a binary tree. Similar to BVHs a ray traverses this tree in a top-down fashion. If it does not intersect one of the nodes, the whole subtree can be skipped (see Sect. 3.4). Instead of representing BVs with all six planes we make use of the enclosing characteristics of BVHs to conservatively estimate a BV by representing it with only one bounding plane. One can think of this as carving away parts of the BV in every subdivision step.

3.1 Properties of BVHs

The reason why BVHs are effective is that whenever a ray $\vec{r} = \vec{o} + t \cdot \vec{d}$ misses the bounding box of a node, the complete subtree can be skipped during the traversal. This enclosing property can be useful in two ways for our SSH. First, we can define active ray segments t_{in} and t_{out} where a ray enters and leaves the bounding box. Given two BVs B^1 and B^2 and corresponding ray segments $t_{in}^1, t_{out}^1, t_{in}^2$ and t_{out}^2 for an arbitrary ray which pierces both BVs then $t_{in}^1 \leq t_{in}^2 \leq t_{out}^2 \leq t_{out}^1$ if B^1 encloses B^2 , see Fig.1. This property is essential for any hybrid BVH method, which approximates a node by less than its six sides.

Another beneficial property concerning efficiency is that the BVs of the children of a node will share some of the bounding slabs with its parent. In fact, one can prove that at least six or more out of the twelve planes by which the two children are defined, will coincide with the bounds of the parent node. Testing these against a ray is redundant. Interestingly if a bounding plane of a child node is not equal to the same plane of its parent, it will be in its sibling. Or in other words, every plane which bounds the ancestor node will also bound at least one of its child nodes. So if one tries to approximate

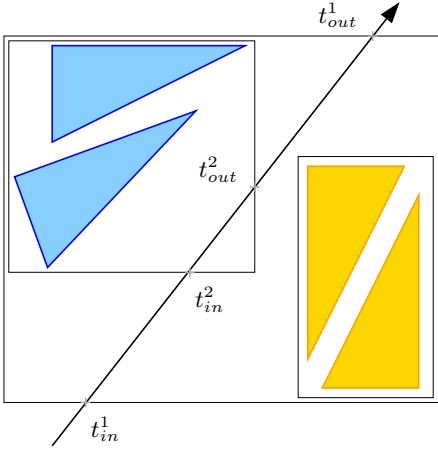


Figure 1: The enclosing property of BVHs ensures that a ray always pierces first, or at least at the same time the ancestor node before it can intersect the children, and will always leave the children nodes before or at the same time it leaves the ancestor node.

BVs by representing them with less than its usual six planes it is crucial not to choose the same bounding planes for both children. Otherwise redundant intersection tests are the consequence. Methods like [32, 10, 33] do not pay attention to this or do not provide the possibility to change it.

3.2 Data Structure

Instead of storing the complete six-sided Bounding Box for every node in our tree, the idea of the single slab hierarchy is to use only one single plane perpendicular to either the x, y or z -axis. In pre-

```
#pragma align(8)
struct SSHNode{
    float plane;
    union{
        int firstChildNodeID; //inner nodes
        int firstTriangleID; //leaf nodes
        // bit 0..1: axis (x,y,z) or leaf
        // bit 2: interior left or right
        // bit 3..4: traversal axis
    }
};
```

Figure 2: 8 byte representation of the SSH node.

vious hybrid approaches the axis of the splitting plane, used for subdividing the object list into the two child nodes, was the same axis as the one used for bounding the nodes. As described in 3.1, we observed that restricting ourselves to this behaviour prohibits flexibility and therefore efficiency, as this might lead to problems with empty spaces where either empty nodes [25] or special nodes [10] have to be inserted in order to deal with this. Havran *et al.* reported performance loss of more than an order of magnitude if one does not deal with these cases. We define an outer and inner half space for each of the bounding planes, to define, on which side the interior of the node is placed. This gives us the needed flexibility to carve away empty space from any side of the bounding box. Therefore no additional nodes need to be added. This is also depicted in Fig. 3.

A complete SSH node contains the position of the bounding plane, a pointer to a pair of children and some additional flags. It can be represented with just 8 bytes (see Fig. 2). Note the resemblance to kD-Tree nodes [26].

We use the lower two bits to indicate the axis (00: x , 01: y , 10: z) to which the bounding plane is perpendicular to or whether it is a leaf node (case 11). One bit indicates whether the included geometry is to the left or right of the bounding plane; the other side is empty respectively. We also use two additional bits to save the traversal axis to define which node a ray should visit first during traversal, according to its signs. For a four byte integer we still have 27 bits left for encoding the index, so up to 134 million triangles are encodable or even more if we allow more than one triangle per leaf node, note that this number has to be predefined before construction and is not saved explicitly per leaf node.

3.3 Construction of the Hierarchy

Our construction is in fact quite similar to BVH construction methods. All the usual construction schemes can be applied such as spatial median cut or surface area heuristic (SAH). Therefore our SSH is also a binary tree. To find the optimal bounding plane for a node we make use of the SAH [16]. The probability P of an arbitrary ray \vec{r} , piercing the root node B^0 , to hit an interior node B^i is exactly the ratio of the surface areas

$$P(\vec{r} \cap B^i | \vec{r} \cap B^0) = \frac{S(B^i)}{S(B^0)} \quad (1)$$

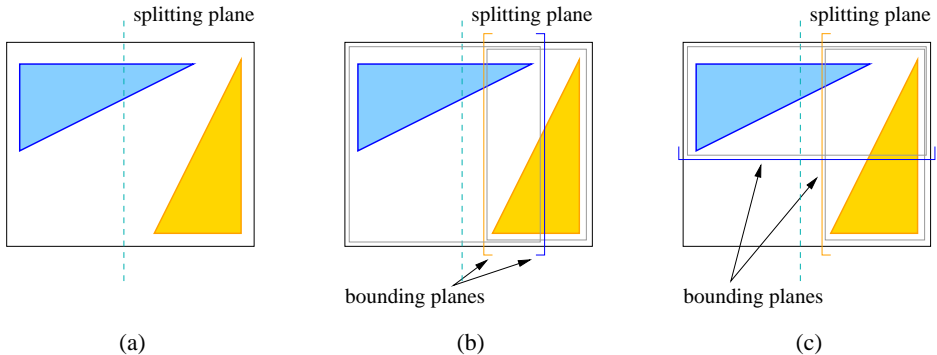


Figure 3: Bounding Plane adjustment. (a) Ancestor volume containing all objects is split along the x-axis (splitting plane depicted by the dashed line). (b) Resulting volumes if splitting and bounding planes are set to the same axis. (c) Our approach can choose the best plane for each of the children, resulting in smaller bounding boxes. The approximated bounding volumes in gray and the chosen bounding planes in blue and orange are shown slightly displaced for better readability.

To find the optimal bounding plane for a node, we use the calculated bounds of its ancestor node. For every side of the BV we set the bounding plane as tight as possible to the included objects. We then calculate the new surface area and keep that bounding plane which reduces the surface area the most (see Fig. 3 and 5).

3.4 Traversal

Intersecting a ray with the single slab hierarchy resembles the traversal of a BVH, but with the complexity of kD-Tree traversal. In each traversal step we maintain the active ray segment $[t_{in}, t_{out}]$, which depicts the entry and exit point of the ray and the current node. This is first initialized to $[0, \infty)$, then clipped to the scene bounding box and updated during traversal. For each traversed node we simply calculate the distance to its bounding plane and compare it to the active ray segment. Depending on the interior bit, which encodes on which side of the bounding plane the geometry is, and the ray direction we can create a very simple intersection algorithm (see Fig. 4).

We increase the likelihood of traversing the children in front-to-back order by including an ordered traversal scheme, as proposed in [17], which determines the order in which the children of a node should be tested. We also implemented the algorithm presented in Fig. 4 as an iterative version, by using a stack to save the nodes and ray segments.

3.5 Extension to dynamic scenes

In fact, our single slab hierarchy shares many properties with BVHs. Therefore we can easily refit our single slab hierarchy to adapt it to changes in the geometry. This can be done in a similar way as described in [32] or [28]. A simple bottom-up approach can be applied using trivial min/max operations to adapt the bound of the nodes. The structure of the hierarchy remains unchanged. This is sufficient for most coherent animations, e.g. skinned meshes.

To handle arbitrary movements one could easily adopt the method of Ize *et al.* [11] where the BVH is refitted while one core of a multi-core PC constantly regenerates the hierarchy and switches it once it is finished.

4 Results and Discussion

We implemented our presented SSH scheme into a SIMD Ray Tracer which uses 2×2 -ray-bundle-tracing to exploit ray coherence and a BVH-based ray tracer following [15] for comparison, including 2×2 -ray-bundle-tracing, a spatial median cut scheme for construction, ordered traversal and using 32 byte of data per node.

Both are state-of-the-art and achieve interactive frame rates on a single desktop PC. For our test scenes we evaluated only the first intersection and used a simple eyelight shader. We are aware of

```

// reverse is true if the ray direction is negative,
// t_hit is the distance to the currently closest intersection
bool intersectSSH(Ray& ray, mask reverse[3], SSHNode* node,
                 float& t_hit, float& t_near, float& t_far)

int axis = node->getSlabAxis();
float t = (float(node->plane) - ray.origin[axis]) / ray.dir[axis];

if (node->geometryIsLeft()){
    t_near = (reverse[axis] | (t <= t_near)) ? t_near : t; // increase t_near
    t_far = (reverse[axis] & (t < t_far)) ? t : t_far; // decrease t_far
}
else {
    t_near = (reverse[axis] & (t > t_near)) ? t : t_near; // increase t_near
    t_far = (reverse[axis] | (t >= t_far)) ? t_far : t; // decrease t_far
}

if((t_near > t_far) || (t_near > t_hit)){
    return false;
}

if(node->isLeaf()){
    return intersect geometry;
}

traverse children;

```

Figure 4: Traversal scheme for the single slab hierarchy.

methods which exploit ray coherence to a higher degree, as in [28, 22], but for a fairer comparison, we did not use them. For testing we used a PC equipped with an Intel Core 2 Quad Q6600 2.4GHz. Only one core was used for the time measurements. Our current subdivision scheme is a spatial median cut [24, 15] which provides good rendering performance for almost all scenes. The nodes are subdivided until each leaf node consists of only one triangle. Note that the structure of the BVH and the SSH is essentially the same, only the representation of the nodes has been changed.

We have tested this system on scenes with varying complexity to measure the performance and memory consumption of our approach. As one can see from Table 1 the performance of the SSH is comparable to a state-of-the-art BVH ray tracer, even faster in some cases, while only 25% of the memory is needed for the acceleration data structure. This memory saving is important because with even more complex models the scene might not fit into main memory anymore if using kD-Trees or BVHs. Built times are slightly higher for the SSH as we need to search for the best bounding

plane, but this is rather negligible. The number of traversed nodes and triangle intersections is larger for the SSH as only approximated BVs are used in essence, but this is alleviated by the simpler traversal algorithm. Only two times more nodes are traversed with the SSH than with a complete BVH, but the intersection test is approximately six times less expensive. Due to the approximation more triangles have to be intersected, but as one can see this is usually only one more per ray. Results are listed in Table 1.

5 Conclusion

We have presented a new acceleration data structure with applications to real-time ray tracing. By relaxing the standard bounding volumes to a single slab we achieved memory savings of 75% compared to standard BVHs without loss in efficiency. And even more sophisticated variations, like the Bounding Interval Hierarchy [25] use at least 50% more memory compared to our approach. This allows for rendering of much more complex models without using out-of-core techniques.

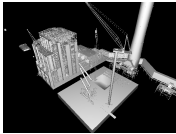
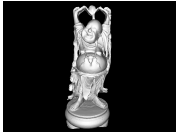

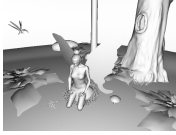
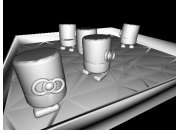

Method	N_T	N_I	T_C	T_R	$s(T_R)$	Mem
Scene - Power Plant - 12,748,510 triangles, 640 × 480						
 BVH	43.55	4.24	28.83s	3.31s	1.0×	778.11MB
SSH	80.23	5.34	28.9s	2.73s	1.22×	194.53MB
Scene - Happy Buddha - 1,087,716 triangles, 640 × 480						
 BVH	6.45	0.63	1.93s	0.54s	1.0×	66.39MB
SSH	14.14	1.53	1.98s	0.59s	0.91×	16.60MB
Scene - Dragon - 871.414 triangles, 640 × 480						
 BVH	10.47	0.95	1.55s	0.84s	1.0×	53.19MB
SSH	23.21	2.11	1.57s	0.89s	0.94×	13.30MB
Scene - Fairy 1st frame - 174,117 triangles, 640 × 480						
 BVH	26.52	2.53	0.30s	2.06s	1.0×	10.63MB
SSH	45.51	4.23	0.31s	1.70s	1.21×	2.66MB
Scene - Toys - 11.141 triangles, 640 × 480						
 BVH	29.57	1.31	0.02s	2.08s	1.0×	0.68MB
SSH	46.85	3.24	0.02s	1.57s	1.33×	0.17MB
Scene - Bunny - 69,451 triangles, 640 × 480						
 BVH	8.77	0.55	0.09s	0.66s	1.0×	4.24MB
SSH	18.01	1.31	0.10s	0.65s	1.03×	1.06MB

Table 1: Comparison of our new technique (SSH) with a BVH implementation following [15], using a simple shader and 2×2 (SSE accelerated) ray bundles. Measurements have been made on an Intel Core 2 Quad Q6600 2.4GHz with 4GB of main memory. Only one core was used for the time measurements. N_T is the average number of traversed nodes per ray, N_I is the average number of ray-object intersections per ray, T_C is the total time needed for construction of the hierarchy, T_R is the total time needed for traversal, $s(T_R)$ is the speedup achieved by the SSH with respect to the BVH, Mem is the memory usage of the acceleration data structures in megabytes, excluding the triangle data.

```

void createSSH(TriangleList tris ,
              AABB parent ,
              SSHNode& node){
AABB bounds(tris); // bounding box
float bestSurface = HUGE_VAL;
int bestSide = 0;
for_all(sides of the AABB){
  AABB temp = parent;
  temp.side = bounds.side;
  if (surface(temp) < bestSurface){
    bestSurface = surface(temp);
    bestSide = side;
  } }
node.boundingPlane = bounds.bestSide;
if (tris.size() < n){
  createLeafNode(); return; }
TriangleList leftTris , rightTris;
subdivide(tris , leftTris , rightTris);
// subdivide using a common scheme,
// like SAH or median-cut
parent.bestSide = bounds.bestSide;
createSSH(leftTris , parent ,
          node.firstChildNodeID);
createSSH(rightTris , parent ,
          node.firstChildNodeID+1);
}

```

Figure 5: Creation scheme for the single slab hierarchy.

Looking at the statistics in Table 1, the current bottleneck is the slightly more incoherent memory access when compared to standard BVHs, as more nodes need to be accessed. We strongly believe that this problem will disappear with newer hardware architectures.

For future work we intend to implement more sophisticated subdivision schemes, like the SAH [16] as well as acceleration schemes [22, 28] to find the first intersections and a GPU version of our ray tracer [9] to improve rendering performance even further. We would also like to try discretization schemes [17] or 4-ary BVH [4] to save even more memory. As the single slab hierarchy is relatively similar to BVHs in its structure, current techniques for ray tracing of dynamic scenes [30] could be adapted as well.

6 Acknowledgements

Our thanks go to the University of North Carolina at Chapel Hill, the Stanford University and the University of Utah for providing us with the 3D models.

References

- [1] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 206–208. Academic Press, 1989.
- [2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.
- [3] D. Cline, K. Steele, and P. Egbert. Lightweight Bounding Volumes for Ray Tracing. *Journal of graphic tools*, 11(4):61–71, 2006.
- [4] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Computer Graphics Forum (Proceedings of EGSR 2008)*, 27(4), 2008.
- [5] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.
- [6] D. S. Fussell and K. R. Subramanian. Fast ray tracing using K-d trees. Technical Report CS-TR-88-07, Austin, TX, USA, 1 1988.
- [7] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.
- [8] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, first edition, 1989.
- [9] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.
- [10] V. Havran, R. Herzog, and H.-P.-Seidel. On Fast Construction of Spatial Hierarchies for Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2006*, 2008.
- [11] T. Ize, I. Wald, and S.G. Parker. Asynchronous bvh construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*, pages 101–108, 2007.
- [12] M. R. Kaplan. Space tracing a constant time ray tracer. In *State of the Art in Image Synthesis (Course Notes on ACM SIGGRAPH '85*,

- volume 11, pages 149–158, July 1985.
- [13] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM Press.
- [14] A. Knoll, I. Wald, S.G. Parker, and C.D. Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006.
- [15] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–46, Salt Lake City, Utah, 2006.
- [16] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166, 1990.
- [17] J. Mahovsky. *Ray Tracing With Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, 2005.
- [18] B.C. Ooi, R. Sacks-David, and K.J. McDonnell. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 433–438, Tokio, Japan, October 1987.
- [19] S. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, April 1999.
- [20] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [21] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).
- [22] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics*, 24(3):1176–1185, 2005.
- [23] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM Press.
- [24] P. Shirley and R.K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, second edition, 2003.
- [25] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006.
- [26] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Universität des Saarlandes, January 2004.
- [27] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, 20(3):153–164, 2001.
- [28] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.
- [29] I. Wald, T. Ize, A. Kensler, A. Knoll, and S.G. Parker. Ray tracing animated scenes using coherent grid traversal. In *Proceedings of SIGGRAPH '06*, pages 485–493, New York, NY, USA, 2006. ACM.
- [30] I. Wald, W.R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S.G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In D. Schmalstieg and J. Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007.
- [31] I. Wald, T.J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.
- [32] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, pages 67–77, 2006.
- [33] M.R. Zuniga and J.K. Uhlmann. Ray queries with wide object isolation and the dectree. *Journal of Graphics Tools*, 11(3):27–45, 2006.