Eurographics Symposium on Rendering 2012
Fredo Durand and Diego Gutierrez
(Guest Editors)

*Volume 31 (2012), Number 4*

# Geometry Presorting for Implicit Object Space Partitioning

M. Eisemann[1] and P. Bauszat[1] and S. Guthe[1] and M. Magnor[1]

[1]TU Braunschweig, Germany

**Abstract**

*We present a new data structure for object space partitioning that can be represented completely implicitly. The bounds of each node in the tree structure are recreated at run-time from the scene objects contained therein. By applying a presorting procedure to the geometry, only a known fraction of the geometry is needed to locate the bounding planes of any node. We evaluate the impact of the implicit bounding plane representation and compare our algorithm to a classic bounding volume hierarchy. Though the representation is completely implicit, we still achieve interactive frame rates on commodity hardware.*
***This is the author version of the paper. The definitive version is available at*** `diglib.eg.org`.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

## 1. Introduction

Driven by consumer demands, progress in general purpose processors (CPU) and graphics processing hardware (GPU) aims at ever-increasing rendering realism and scene complexity. To ensure interactivity suitable acceleration data structures (ADS) are needed. Common applications of ADS include ray tracing, culling, nearest neighbor searches, and collision detection.

With increased scene detail and complexity, available on-board memory resources can become a bottleneck. If a model and its ADS do not fit into main memory, slow disk I/O performance dominates rendering time [WDS05, YLM06]. Unfortunately, ADS are often a major factor in overall memory consumption, requiring typically between ten to twenty percent additional memory, with even higher values reported [LYTM08]. Savings can spare the need for out-of-core rendering and make memory available for more geometry, textures, etc.

The main contribution of our paper is an implicit representation of a complete object space partitioning (OSP) that requires no memory at all. The core idea is to presort the geometry, access the portion that spans each node directly and reconstruct the bounding planes on the fly. Going all the way, we can remove any memory requirement by representing the hierarchy as a heap, resulting in an OSP that requires no memory at all: it is represented completely implicit by triangle order. It is easy to parallelize and well suited for many-core processors. A resorting is only necessary if the geometry changes, no rebuilt is necessary if only the viewpoint or lights are moved. We additionally present a parallel construction technique, demonstrating that our approach is applicable to fully dynamic scenes rendered at interactive frame rates.

The paper is structured as follows: We review previous work (Section 2) before describing our implicit bounding plane representation (Section 3). We go on to present the completely implicit representation of the ADS and how to remove remaining memory requirements (Section 4). Finally, we present a statistical evaluation of our approach for several test scenes (Section 5) and conclude with a critical discussion of our approach (Section 6).

## 2. Previous Work

Ray tracing is widely used in different variations for high-quality rendering due to its physically-motivated light-transport simulation [PH10]. The computational demands can be alleviated through acceleration data structures which exclude most scene objects from intersection testing. A good survey can be found in [WMG*09]. These ADS have a non-negligible memory requirement. Generally, OSP schemes are less memory demanding than spatial subdivision [WK06, GPSS07], as each primitive is referenced only once.

)

Here, we will concentrate on further memory reduction techniques for OSP. Our approach is inspired by multidimensional nearest neighbor search structures [Sam05], where primitive references in inner nodes can be used for early pruning of subtrees.

**Hybrid BVHs** Many OSP schemes are derivations of the classic bounding slab hierarchy by Kay and Kajiya [KK86]. The most common derivation is a standard Bounding Volume Hierarchy (BVH) with axis-aligned bounding boxes (AABB) consisting of six bounding planes per node perpendicular to the world coordinate axes. Such a BVH is a common acceleration data structure for rendering [KK86, WMG*09]. When an intersection query between a ray and the scene is started, the hierarchy is traversed in a top-down fashion. If one of the nodes is missed, the whole subtree can be skipped. One common way to reduce the number of nodes is to use a higher branching factor [DHK08, EG08], but this often comes at the cost of reduced performance.

Memory efficiency of hybrid approaches is achieved by storing only a subset of the bounding planes. Several authors proposed to remove half of the bounding planes due to the observation that the twelve planes of the children of a node always share six sides with their parent [Kar07, FD09, EW11]. By saving the active ray interval, a hit or miss can be conservatively estimated with even less planes. This hybrid BVH was developed independently by several researchers [Zac02, WK06, ZU06, WMS06, HHHPS06]. Zachmann *et al.* [Zac02] proposed a single bounding plane approach for collision detection with oriented bounding boxes. A similar representation but with axis-aligned bounding planes and a fast global construction heuristic was used by Wächter and Keller [WK06]. Woop *et al.* [WMS06] showed a hardware implementation of a similar structure which uses two opposing bounding planes per node, called B-KD tree. The DE-Tree by Zuniga and Uhlmann [ZU06] shows similarities with the B-KD tree but uses wide object isolation to keep larger objects higher in the hierarchy plus a higher branching factor. Havran *et al.* [HHHPS06] adapted a version of the SKD-tree by Ooi *et al.* [OSDM87] and extended them to incorporate different node types in order to improve efficiency. Our approach shows similarities to these hybrid techniques in that we can also use only a subset of the usual six bounding planes. In contrast, we derive the position of the bounding planes directly from the contained geometry of each node instead of saving it explicitly.

**Memory Reduction Techniques** Mahovsky and Wyvill [MW06] investigated a hierarchical encoding scheme for BVHs reducing the storage requirements by 63%–75% at the cost of decreased performance. A similar approach was taken by Cline *et al.* [CSE06] compressing a node to 12 bytes combined with a higher branching factor. Segovia and Ernst [SE10] follow Mahovsky's approach, but additionally save the BVH in clusters to reduce the byte count of child node references. Bauszat *et al.* [BEM10] reduced the memory requirements down to one single bit per node, but performance drops naturally. Additionally, both use a two-level BVH using uncompressed nodes for the top levels. The idea of a two-level BVH was previously presented by Lauterbach *et al.* [LYTM08]. At the lower levels, triangle strips of up to 256 triangles are encoded in an implicit SKD-tree where the vertices encode the bounds. Unfortunately, duplication of vertices may be necessary to create a valid SKD-tree. Kim *et al.* [KBK*10] build upon this two-level approach and additionally introduce tree templates to reduce the number of necessary child pointers. Wächter and Keller [WK07] proposed a new termination criterion for spatial subdivision schemes and use a fixed memory footprint, but rendering efficiency quickly deteriorates if less than five bytes per scene primitive are used.

Recently, Keller and Wächter [KW09] filed a patent for an algorithm using a completely implicit OSP and a spatial subdivision scheme. The idea is based on a divide-and-conquer approach. In each traversal step they first compute the bounding box for the current primitives of the object. In the next step, the active rays that intersect the box are computed. The primitives are partitioned into two sets according to a chosen splitting plane. The algorithm is then recursively called for the active rays and the new partition. If the number of primitives is below a certain threshold the active rays are directly tested for intersection. Unfortunately, no performance statistics are available for this approach. A very similar technique based on a spatial subdivision scheme mentioned in [KW09] was proposed by Mora [Mor11]. Instead of computing bounding boxes, the space of the current node is subdivided and all active rays and active primitives are tested against it. If only primary rays are traced these algorithms have an almost perfect time to image as only those parts of the hierarchy are created which are actually traversed. Occluded parts are left unpartitioned. The implicit reconstruction has to be repeated for each ray batch. According to Mora [Mor11] an efficient GPU implementation poses difficulties and has not yet been further investigated. Our approach nicely benefits from the parallelism and computational power of current GPUs both in construction as well as rendering.

## 3. Implicit Bounding Plane Representation

We seek to obtain interactive rendering performance without the usage of any additional memory. The first step we take is to introduce an implicit bounding plane representation. Our main observation here is that each bounding plane of a node in a BVH is defined by at least one scene primitive. In cases of polygons the plane is defined by a polygon vertex; for B-splines, by a control point; or by a bounding volume if instancing is used. For simplicity of explanation, we will concentrate on scenes solely composed of triangles.

Instead of saving each bounding plane of a node $n_i$ ex-

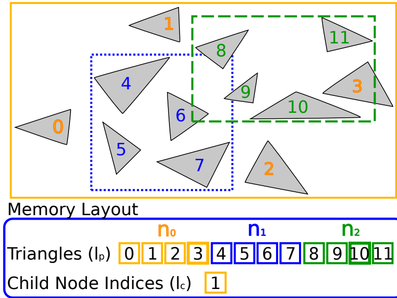



Figure 1: 2D example partitioning: The bounding boxes of the hierarchy are solely described by the vertices of the triangles. Only an array of triangles ($l_p$) and an array of offsets for the left child node is saved ($l_c$).

plicitly, we save the scene primitives spanning $n_i$ in the inner nodes. Using min/max operations on the bounds of the contained primitives, the AABB of each node can be recreated during traversal from only the six bounding triangles. As we keep primitives in inner and leaf nodes, the bounds of the child nodes do not necessarily share a common bounding plane, but the enclosing property of BVHs is still guaranteed, see Figure 1 for a 2D example. All six primitives are contiguously mapped to memory and each such chunk is presorted so that the ordering in the triangle array corresponds to the ordering of the nodes in the BVH. In cases where a single triangle spans more than one bounding plane of a node, e.g. triangle 4 and 7 in Figure 1, less than six triangles are required to represent the bounds. To keep the memory layout consistent we pad the node with the second closest triangle to the respective bounding plane of the same subtree.

Using a structure of arrays representation, we have two arrays, one containing the geometric information of the primitives, $l_p$, and one containing the child node indices $l_c$. Assuming the root node to have index 0, the index of the first bounding triangle $p_i$ is derived from the child node index $n_i$ by $p_i = n_i \cdot b$, where $b$ is the number of bounding triangles per node. The original node index is used for storing the child pointer in $l_c$.

Hybrid BVHs conservatively estimate the AABB of a node by saving less than the standard six bounding planes and using an active ray interval. Following these approaches, we can choose an arbitrary number of bounding planes to represent a node. For current standard (graphics) processors two opposing bounding planes per node seem to be the best choice in our setting, similar to [WMS06, ZU06], see evaluation in Section 5. In the following, our description refers to the two triangle version. The single 4 byte child node index then encodes the following information: The lower two bits indicate the bounding axis that is spanned by the triangles (00: x, 01: y, 10: z) or whether it is a leaf node (case 11). We always use opposing bounding planes, therefore, the bounding axis is the axis perpendicular to the bounding planes. As we map the left and right child next to each other in memory, the remaining 30 bits are used as the offset for the left child node only. For the leaf nodes, we use three bits to encode the number of triangles additionally contained in the node. During construction we ensure that only an even number of additional triangles is available in each leaf node as this allows us to encode up to fourteen additional primitives. Note that the count can be zero. The residual 27 bits encode the according offset into the triangle array which resides in memory right after all bounding triangles of the hierarchy. By sorting the children of each node according to their extent along the bounding axis we can incorporate ordered traversal [WBS07] based on the ray direction.

### 3.1. Ray Intersection

Intersecting a ray with our implicit bounding plane representation is equivalent to a hybrid BVH traversal with an additional reconstruction and triangle intersection step. In each traversal step, we first compute the offset $p_i$ of the first bounding triangle which is derived from the current traversal index $n_i$ and we reconstruct the bounding planes. For this, we load only the data required for the current bounding axis, i.e. one float for each vertex of the two bounding triangles. After reconstructing and testing the minimum bounding plane, we test the maximum plane only if we found a valid intersection. If the intersection of a ray with the bounding planes is outside the active ray interval the subtree is skipped. Otherwise, the active ray interval is updated and the two bounding triangles are tested for intersection. Finally, we fetch the leaf node bits to test whether we reached a leaf node. Traversal either continues with the child nodes or in case of a leaf node the additional triangles are tested.

The chance of a hit with the bounding triangles in the first levels of the hierarchy is usually very low. It seems therefore beneficial to first test against the AABB of each triangle before testing it directly. Therefore, we could first reconstruct and test against the remaining bounding planes of each triangle along the current bounding axis. The vertex data of each further axis would only be loaded one axis at a time if the triangle was not already rejected beforehand. Only if a valid intersection for the complete AABB of the triangle is found the triangle itself would be tested. Even though this reduces the theoretical bandwidth requirements and the number of triangle intersections drops by approximately 50%, no real speed-up was experienced with the processor architectures we tested due to higher register pressure. However, this can be beneficial for future processor generations. We therefore test the triangles directly if the node is hit.

### 3.2. Construction

Most top-down BVH construction schemes can be directly applied to our representation. The only difference is an additional search step to find the bounding triangles span-

ning each node. These are excluded from further partitioning steps. Finding the bounding triangles requires a single scan over the active partition per node. The overall complexity is then $O(n \log n)$, with $n$ being the number of primitives. During construction and evaluation of the surface area heuristic (SAH) [MB90] it is important to keep in mind that a two-plane representation reduces the bounding volume only along a single dimension in each subdivision step. The bounding triangles are always chosen based on the partitioning axis of the parent node, as we can expect the largest surface reduction along this axis. We call the partitioning axis the axis along which the triangles are subdivided into two new partitions and passed on to the child nodes.

## 4. Complete Implicit Representation

In the following we present changes that remove any explicit memory storage for the ADS. We remove the necessity for the bounding axis bits by using round-robin for choosing the axis, i.e. $xyzxyz\ldots$, depending on the depth in the tree. In order to be able to compute the children for any node, we enforce the hierarchy to be a complete, left-balanced tree arranged in breadth-first order. This allows us to index it like a heap without explicitly saving any pointers or indices [CSE06]. For any implicit node $n_i$ its children are indexed with $kn_i + m$ where $k$ is the branching factor. In our case $k = 2$, and $m \in \{1, \ldots, k\}$ denotes the first child node, the second child and so on. By enforcing the hierarchy to be a complete tree, the leaf node property can be directly derived from the index, i.e. if the child index is larger than the number of implicit nodes in the scene a leaf has been reached. The last non-leaf node might have only one child instead of two, Figure 2, as we only require the number of triangles in the scene to be even. In case of an odd number of triangles, the last one is replicated. We do not save any additional triangles in the leaf nodes, instead each primitive is a bounding primitive in some node of the hierarchy. Unfortunately, the compulsion of a complete tree forces us to use an object median split technique during construction.

### 4.1. Ray Intersection

Intersecting a ray with the completely implicit representation is similar to the approach in Section 3.1 with few exceptions. Instead of testing for a leaf node, the current node traversal index is compared to the total number of nodes in the scene. Traversal is terminated if the index is larger. Otherwise, the child indices are computed and traversal continues.

### 4.2. Construction

The restrictions on the completely implicit BVH open up the possibility for a parallel construction technique suitable for multiprocessor architectures. The hierarchy is built top-down and all nodes of one level are processed in parallel. In contrast to other approaches, parallelism in the upper nodes

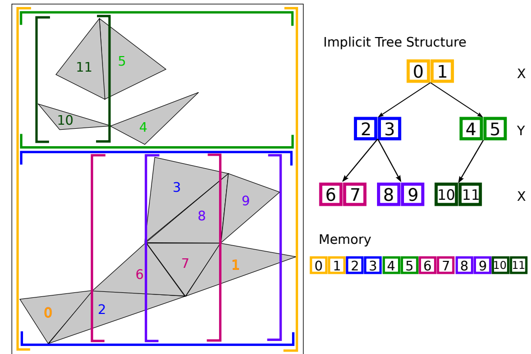



Figure 2: 2D Example of the complete implicit object partitioning with three levels: The triangle arrangement implicitly describes a hierarchy. The bounds of each node are spanned by exactly two triangles. Left: Representation of the resulting bounding planes. The first and third level are bounded along the x-axis, the second level along the y-axis due to the round-robin scheme employed. The triangle index is colored according to the bounds the triangle represents. Top right: The scene triangles implicitly represent a complete binary tree of bounding planes. Bottom right: Representation in memory. Note that beyond the triangles, no additional memory is used.

of the hierarchy is not enforced on a per node basis but threads operate across node boundaries, as will be described in the following.

As the bounding and partitioning axes are chosen in a round-robin fashion, see Section 4, all nodes of the same level in the hierarchy need to partition their enclosed primitives along the same axis. The partitioning axis is always equal to the bounding axis of the next hierarchy level. The structure of the hierarchy is already known due to the required left-balanced tree. We make use of an additional node index array $I$ saving the currently active node a triangle might belong to and a split list $S$ in which the starting index and the size of each active partition are saved.

The basic algorithm, as shown in Figure 3, consists of four main steps for each level of the hierarchy. In the first step, all triangles and their according node indices are sorted along the current bounding axis. Then a stable sort using the node indices as keys is applied. While the first step sorts the triangles according to their spatial position, the second sort partitions the triangles according to their current node index without changing their respective order. This puts the minimum bounding triangles at the correct positions and allows for a direct partitioning of the active nodes for further subdivions. We then search for the maximum bounding triangle in each split and swap it to the second position in the split. We update the node indices for the next iteration (details are given below), remove the old splits, as they are already processed, and emit new splits for each node of the next level.

```
axis = 0;
S = { (0,n) }; // Split list
I = {0,0,0,...,0}; // Index list
parallel_construction(triangles,S,I,axis);

void parallel_construction(triangles,splits,
     I, axis){
  for all levels of the hierarchy{
    lexicographicalSort(triangles, S[0][0],
        I, axis);
    if(lastLevelreached){ return; }
    maximumTriangleSearch(triangles,S,axis);
    updateIndices(I, S);
    createNewSplitsFromOld(S);
    axis = (axis + 1) % 3;
  }}
```

Figure 3: Pseudo-Code of the parallel construction scheme for the completely implicit representation.

The process is repeated until no split contains more than two triangles. We do not sort the triangles and indices directly but rather utilize a permutation array for efficiency. The memory requirement for the parallel construction is $O(n)$ as we need one integer per triangle plus the split list, which is of the size $n/4$ at most.

The following describes the construction process in more detail for reimplementation. We start with a single split at index 0 with a size $n$ equal to the number of scene primitives and initialize $I$ to zeros. The algorithm then loops over all $\lfloor \log_2(n/2) \rfloor + 1$ levels of the hierarchy. In each loop we first apply the lexicographical sort to all triangles, i.e. sort them according to their spatial position and then a stable sort on the node indices is applied. Already finalized triangles in front of the first split are excluded. The sort swaps already finished nodes to the front and sorts all triangles of the same node along the bounding axis. The remaining triangles in each node are split into its two child nodes where each child already has the minimum triangle at the correct position. Next, we search for the bounding triangle of the maximum bound in the remainder of the child triangles and swap it with the second position in each split. We can use a simple swap operation instead of shifting the triangles over to the end of the split since we only required the triangles to be sorted for the actual split operation. As long as the average over all splits of a given level holds more than 4 triangles, we assign $\mathit{averageTrianglesPerSplit}\,/\,4$ threads to each split. Since for the maximum search each thread will find its own maximum, we use atomic compare and swap (atomicCAS) functions in case a new maximum was found to ensure the overall maximum is found. The threads are assigned in reverse order per split to minimize the warp serialization due to the atomic operations. As soon as the average number of triangles per split falls down to 4 triangles, we only use a single thread per split and can therefore switch to a kernel without atomic operations.

For each split the algorithm now updates the node index values. The first two indices of each open split $S_i$ are assigned a value of $\mathit{idx} = \mathit{finalized} + i$ where $\mathit{finalized} = 2^{lvl} - 1$ is the number of already correctly created nodes. $i$ is the index of the split in the split list and $lvl$ the current level of the hierarchy. The value of the other triangles in a split are set to $2\,\mathit{idx} + 1$ for their respective splits which is the index of the left child. We use the same thread distribution for each split as described in the last paragraph for parallelism.

We remove the active splits and insert new splits for the left and right child nodes into the queue if they contain two or more triangles. Let $\mathit{numSplits}$ be the number of the old open splits, $pos_i$ the starting position of the $i^{\text{th}}$ split and $num_i$ its size. The size $num_L$ and $num_R$ for the new splits is chosen in a way to guarantee a left-balanced, complete tree.

$$
\begin{aligned}
\mathit{half}_i &= \frac{num_i}{2} \\
H &= \lfloor \log 2\,(\mathit{half}_i) \rfloor \\
num_R &= 2\left((2^{H-1}-1) + \max(0, \mathit{half}_i - 3 \cdot 2^{H-1} - 3)\right) \\
num_L &= num_i - 2 - num_R
\end{aligned} \quad (1)
$$

where $H$ is the depth of the tree.

The positions of the new left and right splits resulting from $pos_i$ are computed by

$$
\begin{aligned}
pos_L &= pos_i + 2(\mathit{numSplits} - i) \\
pos_R &= pos_L + num_L
\end{aligned} \quad (2)
$$

The computed positions of the splits are already at the positions that are needed *after* the next lexicographical sort. Finally, the bounding axis is incremented to the next level.

This procedure automatically builds a breadth-first tree. An example for the first three levels of a scene with twelve triangles is given in Figure 4.

## 5. Results

We evaluate our presented algorithm on several scenes with varying complexity, including ones with high triangle count (THAI STATUE), teapot-in-a-stadium problems (FAIRY), largely differing scene primitives (CRYTEK SPONZA), animation (FAIRY, BREAKING LION) and problematic scenes for object median cut (VENICE and CRYTEK SPONZA), or combinations of these scene attributes. To evaluate the influence of the implicit bounding plane representation we show results for both the Implicit Object Space Partitioning with 4 bytes (IOSP-4) and the complete implicit representation (IOSP-0). We also implemented a hybrid version that saves the top-levels as uncompressed BVH nodes using a SAH builder where each leaf points towards a separate IOSP-0

)

| | Splits | | S0 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Triangles | | 7 | 5 | 0 | 8 | 2 | 1 | 3 | 4 | 11 | 6 | 9 | 10 |
| | Node index | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Lvl0 | Splits | sort + max search | S0 | | | | | | | | | | | |
| | Triangles | | 0 | 1 | 2 | 11 | 6 | 5 | 4 | 3 | 8 | 7 | 9 | 10 |
| | Node index | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Splits | update node indices | S0 | | | | | | | | | | | |
| | Triangles | | 0 | 1 | 2 | 11 | 6 | 5 | 4 | 3 | 8 | 7 | 9 | 10 |
| | Node index | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | Splits | new splits | | | S0 | | | | | | S1 | | | |
| | Triangles | | 0 | 1 | 2 | 11 | 6 | 5 | 4 | 3 | 8 | 7 | 9 | 10 |
| | Node index | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Lvl1 | Splits | sort + max search | | | S0 | | | | | | S1 | | | |
| | Triangles | | 0 | 1 | 2 | 3 | 7 | 6 | 8 | 9 | 4 | 5 | 10 | 11 |
| | Node index | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | Splits | update node indices | | | S0 | | | | | | S1 | | | |
| | Triangles | | 0 | 1 | 2 | 3 | 7 | 6 | 8 | 9 | 4 | 5 | 10 | 11 |
| | Node index | | 0 | 0 | 1 | 1 | 3 | 3 | 3 | 3 | 2 | 2 | 5 | 5 |
| | Splits | new splits | | | | | | | S0 | | S1 | | S2 | |
| | Triangles | | 0 | 1 | 2 | 3 | 7 | 6 | 8 | 9 | 4 | 5 | 10 | 11 |
| | Node index | | 0 | 0 | 1 | 1 | 3 | 3 | 3 | 3 | 2 | 2 | 5 | 5 |
| Lvl2 | Splits | sort + max search | | | | | | | S0 | | S1 | | S2 | |
| | Triangles | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | Node index | | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 5 | 5 |
| ... | | | | | | | | | | | | | | |

Figure 4: Example of the first three levels in the parallel hierarchy creation process for the completely implicit representation. The spatial arrangement of the triangles according to the construction is shown in Figure 2.

(2-Lvl IOSP). We analyze and discuss our optimizations, bandwidth considerations, incoherent rays as encountered in global illumination simulations, construction performance, as well as the two-level approach for increased performance.

We have produced both a CPU variant and a GPU implementation using NVIDIA CUDA. All statistics were measured on a system with an Intel Core i7-2600 with 3.4 GHz, 16GB RAM, and an NVIDIA GeForce GTX 580 with 3GB of memory, running on a 64-bit Windows system. All results are produced at a resolution of $1024 \times 768$ pixels if not stated otherwise.

For a general comparison, if appropriate, we make use of a BVH implementation using the surface area heuristic - BVH(SAH) - and using an object median split - BVH(OMS). In accordance with [Wal07] we use a binning approach with ten bins during construction for evaluation of the SAH. We impose a minimum triangle count of four triangles per leaf node. The same strategy was used for our IOSP-4 and 2-Lvl IOSP. The associated statistics are given in Table 5.

**Number of Bounding Triangles** We first verified our choice of using only two boundary triangles by comparing performance for different numbers of bounding triangles for the IOSP-0. For one bounding triangle we follow the approach of [EWM08] where the single bound encodes the half-space in which the geometry resides. We extend the round-robin scheme so that for the first three levels the maximum bounds are saved for the left child nodes (respectively the minimum bounds for the right children) and the minimum bounds for the next three levels (respectively the maximum bounds for the right children). For the six triangles case, a complete AABB is reconstructed in each traversal step. For current standard (graphics) processors, choosing two bounding triangles per node resulted in the best performance in our test scenes, Table 1. This may change in future hardware with larger cache lines or higher costs per memory access compared to the computational power. Using less bounding triangles per node would require empty nodes for efficiency [WK06], while using more causes a too high computational load on current processors. In the following experiments, we always used the version with two bounding triangles.

| Scene | 1 (CPU) | 2 (CPU) | 6 (CPU) | 1 (GPU) | 2 (GPU) | 6 (GPU) |
|---|---|---|---|---|---|---|
| Breaking Lion | 0.629s | 0.489s | 0.639s | 0.094s | 0.077s | 0.060s |
| Crytek Sponza | 1.972s | 1.614s | 7.383s | 0.144s | 0.085s | 0.268s |
| Fairy | 1.710s | 1.077s | 1.996s | 0.131s | 0.060s | 0.104s |
| Robot Girl | 0.969s | 0.718s | 0.922s | 0.083s | 0.045s | 0.050s |
| Thai Statue | 0.819s | 0.383s | 0.933s | 0.153s | 0.050s | 0.103s |
| Venice | 3.068s | 2.102s | 3.694s | 0.248s | 0.164s | 0.176s |

Table 1: Evaluation of the influence of bounding triangles per node for primary rays. CPU and GPU traversal time in seconds are given.

**Bandwidth Considerations** We measured the bandwidth requirements assuming a perfect memory access and tracing one ray after the other, i.e. no caching is assumed, each tested bounding box of a BVH is assumed to be 32 bytes in size, each tested triangle is counted as 36 bytes (nine float values for the three vertices). Additional data like texture coordinates, normals etc. are not included, as these are accessed only in the shading step, which is the same for all tested approaches. Statistics are given in Table 5.

Compared to the BVH(SAH) the theoretical bandwidth increases by a factor of 2.77 to 8.35 with 5.03 on average for the IOSP-0, 1.49 to 2.04 with 1.78 on average for the IOSP-4, and a factor of 1.09 to 3.05 with 1.69 on average for a two-level IOSP-0 with 15 uncompressed top-levels. In practice the values will vary depending on the hardware capabilities, like cacheline size and traversal technique used.

**Incoherent Rays** One of the main advantages of ray tracing is that it can employ secondary rays to compute effects such as global illumination, soft shadows, reflection or refraction. The incoherency of these rays, especially in Monte-Carlo simulations, poses problems on the efficiency of ray tracers due to incoherent memory access and diverging traversal paths, especially on a highly parallel processor as the GPU. Table 2 shows the results of our test scenes rendered with up to three light bounces. One ray path per pixel is created using pure random sampling over the pixel domain and the hemisphere domain (to increase incoherency) and one shadow ray is traced for each light source at each path vertex.

)

| Scene | (#B) | BVH(SAH) | IOSP-4 | IOSP-0 |
|---|---|---|---|---|
| Breaking Lion | 1 | 69.391 | 39.322 | 19.181 |
| Crytek Sponza | 1 | 32.319 | 12.788 | 2.844 |
| Fairy | 1 | 61.280 | 29.127 | 5.761 |
| Robot Girl | 1 | 85.020 | 49.152 | 19.784 |
| Thai Statue | 1 | 73.156 | 21.845 | 11.523 |
| Venice | 1 | 41.391 | 18.614 | 3.456 |
| Breaking Lion | 3 | 93.437 | 56.510 | 26.963 |
| Crytek Sponza | 3 | 25.233 | 9.180 | 1.405 |
| Fairy | 3 | 62.915 | 29.677 | 4.575 |
| Robot Girl | 3 | 106.63 | 63.550 | 28.468 |
| Thai Statue | 3 | 88.612 | 29.263 | 15.271 |
| Venice | 3 | 41.665 | 18.559 | 3.299 |

Table 2: Influence of the number of bounces (#B) in a path tracing simulation according to the number of bounces on the GPU. Numbers are given in million rays per second. Computations include ray generation, traversal, shading and texturing. The first three scenes contain 2 light sources each, while the latter three contain 1 light source.

| Scene | 1 Thread/Tri | 1 Thread/Split | Adaptive |
|---|---|---|---|
| Fairy | 0.603s | 0.284s | 0.159s |
| Breaking Lion | >10s | 1.533s | 0.226s |

Table 3: Comparison of the construction times using one thread per triangle for all levels of the hierarchy (1 Thread-/Tri), one thread per split (1 Thread/Split) for all levels and our adaptive approach that uses multiple threads per split.

**Animations** For animated and dynamic scenes not only traversal performance but also construction times are of importance. Here, we analyze our construction technique for the IOSP-0 from Section 4.2. All experiments were conducted directly on the GPU. In a straightforward implementation we would simply assign a single thread to each split at all levels. However, this does not create enough parallelism at the top levels of the hierarchy. An alternative is to assign one thread to each triangle and search for the split that this triangle falls into. Obviously, this will cause issues at the lower levels of the hierarchy as the number of splits to search for doubles for each level. In Figure 3, we show a comparison of construction timings between creating the hierarchy using one thread per triangle, one thread per split and our adaptive approach. In the adaptive approach we assign multiple threads to each split, so that the average number of triangles per thread is limited to the same value at all levels of the hierarchy. Naturally, this will revert to one thread per split at the bottommost levels of the hierarchy. Our adaptive approach reduces construction time for the animated scenes up to 85% compared against the straightforward implementation, Table 3.

Figure 5 illustrates the construction performance in detail for each level. Our algorithm shows a virtually constant construction time per level. About 95% of the construction time is used by the lexicographical sort for which we used the CUDA Thrust library.

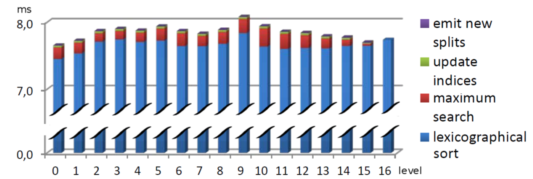
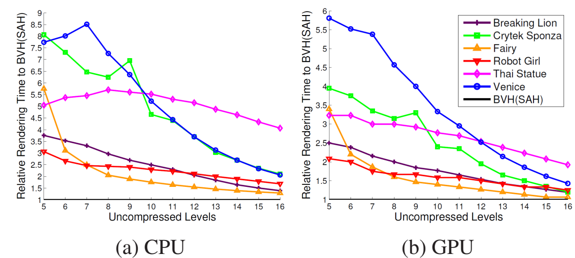



Figure 5: In order to evaluate our parallel construction algorithm, we show the time taken by the different steps of our construction per level for the animated scene FAIRY consisting of 174k triangles. The timings include the lexicographical sort (blue), updating the node indices (green), creating the new splits (violet) and the search for the maximum triangle (red).




(a) CPU (b) GPU

Figure 6: Comparison of render times using different numbers of levels for the uncompressed BVH.

**Two-level approach** Representing the important top-levels of the hierarchy in an uncompressed BVH format and using the compressed representation for the lower levels is an established technique to provide a convenient trade-off between performance and memory requirements for several compression schemes [LYTM08, LGS*09, SE10, BEM10, PL10, GPM11]. We analyzed the influence of the ratio between uncompressed levels and compressed levels in terms of ray tracing performance in Figure 6. For 15 uncompressed levels the memory requirements are only up to 1MB for our 2-Lvl IOSP while performance is between 23-74% for the CPU and 48-93% for the GPU compared to the BVH(SAH), depending on scene complexity, Table 5. The two-level approach works best for non-uniform triangle distributions. For the THAI STATUE a median cut in the upper levels is of similar quality compared to a cut based on the SAH, therefore only a relatively small speed-up is achieved.

**Comparison to other memory reduction techniques** In the following we compare our technique to more sophisticated techniques than a standard BVH. Reusing shared bounding planes [Kar07, FD09, EW11] reduces memory and

| Scene | Ours (CPU) Single | Ours (CPU) Packets | Ours (GPU) Single | [Mor11] (CPU) Packets |
|---|---|---|---|---|
| Fairy | 0.60 | 14.28 | 10.4 | 6.8 |
| Thai Statue | 2.35 | 3.37 | 12.05 | 1.28 |

Table 4: Comparison to [Mor11]. Resolution is $1024 \times 1024$, only primary rays are traced and simple eye shading used. Frames per second are reported. Single = Single raytracing. Packets = Packet tracing.

bandwidth requirements in a BVH by 43–50% and 35–38%, respectively, without a negative influence on the rendering times [FD09]. The bounding interval hierarchy [WK06] performs up to par to an optimized BVH or kd-tree but the memory requirements are only ten bytes on average with a careful implementation (69% reduction). If memory reductions of 50% to 70% are sufficient, these techniques allow for performance similar to a BVH.

In table 4 we compare our technique with the divide-and-conquer (DAC) approach by Mora *et al.* [Mor11] which also does not require to save any acceleration data structure. The resolution was set to $1024 \times 1024$ pixels and only primary rays were traced. We chose the FAIRY and THAI STATUE scene as they provide the best insights into the strength and weaknesses of both approaches. Please note that the comparison has to be done carefully as the processor architectures differ. In [Mor11] an Intel-core 2 duo E6850 with 3 GHz was used while we use a Core i7-2600 with 3.4 GHz. Also note that the DAC uses conic packets for primary rays resulting in an additional speed-up of factor 1.3 – 3.8 depending on the scene (speed-up taken from Figure 8 in [Mor11]). As expected DAC achieves better performance on the CPU for smaller scenes, probably due to the spatial median split employed which provides drastically better clipping quality than the object median split required by our technique. For larger scenes the overhead due to the triangle streaming in the DAC approach becomes more apparent. Our technique can be easily ported to the GPU where we achieve speed-ups between a factor of 1.5 and 9.4 compared to DAC. No packet tracing was used in our GPU timings which would further increase performance, especially since packets are more robust to non-optimal subdivision schemes in terms of performance.

## 6. Discussion and Conclusion

In this paper we have presented a complete implicit object space partitioning scheme which is easy to parallelize and therefore well suited for many-core processors. We have shown that the bounding planes of a hierarchical acceleration data structure can efficiently be represented and accessed by geometry presorting. Our IOSP-0 approach is statically represented by the underlying geometry and must be created only once per timestep of an animation, independent of the viewpoint or lighting condition which is an important difference to previous implicit ADS approaches. If memory is the limiting factor, our approach can be a useful alternative to classic acceleration data structures.

Implicit acceleration data structures have only recently gained a higher attention in the rendering community. Therefore, several limitations still exist and its applicability on current hardware may be limited but we see good prospects for further research. The object median cut partitioning scheme proposed for the complete implicit representation is known to be inferior to other tree structures [Wäc08], but is currently a necessity for the implicit child index computation and the main reason for the reduced performance in the CRYTEK SPONZA and VENICE scene. Finding a solution for an implicit representation with an arbitrary splitting scheme is an open problem. For comparable performance to state-of-the-art techniques, integration of spatial splits [SFD09] would be a necessity but the requirement for multiple object references seems problematic for an implicit representation. Another fruitful direction might be to investigate if spatial partitioning schemes can be implicitly represented without a lazy evaluation scheme. We plan to delve further in this direction. As most object space partitioning schemes our approach suffers from the same drawbacks when encountering a mixture of small and large primitives in a scene. Larger primitives are kept higher in the hierarchy in our approaches which is generally beneficial for some scenes [ZU06, IH11] but incorporating early split clipping [EG07] is an important challenge for future work and improved performance. Currently, we investigated only triangles as the basic primitive though other scene representations are possible as well in theory. Our focus is on ray intersections with the scene, but collision detection is another possible application of OSP. However, AABB are usually not the bounding volume of choice for this task and a direct application of our approach is difficult due to the triangles in the inner nodes of the hierarchy. Our approach should also benefit from a fast mesh compression technique possibly decreasing the overall bandwidth requirements [RKB06]. A dedicated hardware implementation of our IOSP-0 is also a promising direction, as the main ingredients are a sorting procedure and triangle intersections. Both can be efficiently implemented in hardware [KW05, WMS06].

## Acknowledgments

We would like to thank Yoji for the Robot Girl model provided at BlendSwap.com, Crytek for the improved Sponza scene, UNC for the Breaking Lion, the University of Utah for Fairy, Stanford for the Thai Statue and Stefan John for the Venice model, courtesy of Intel Visual Computing Institute `http://www.ivci.de/`. This project was partly funded by DFG project MA 2555/1-3.

| Method | $N_T$ | $N_I$ | $R$ | BW/frame | Mem |
|---|---|---|---|---|---|
| Scene - Breaking Lion - 1,604,054 triangles, 96.331 MB of memory used for geometry | | | | | |
| BVH(SAH) | 26,577k | 5,678k | 35.747 Mrays/s | 0.983 GB | 33.524 MB |
| BVH(OMS) | 58,942k | 10,384k | 18.289 Mrays/s | 2.105 GB | 33.554 MB |
| **IOSP-4** | 29,503k | 32,975k | 20.696 Mrays/s | 1.469 GB | 2.694 MB |
| **IOSP-0** | 59,435k | 62,471k | 10.213 Mrays/s | 2.725 GB | 0 MB |
| **2-Lvl IOSP (15)** | 28,624k | 13,087k | 23.831 Mrays/s | 1.067 GB | 1.001 MB |
| Scene - Crytek Sponza - 279,163 triangles, 19.587 MB of memory used for geometry | | | | | |
| BVH(SAH) | 80,626k | 6,814k | 41.391 Mrays/s | 2.631 GB | 5.608 MB |
| BVH(OMS) | 289,334k | 37,531k | 11.398 Mrays/s | 9.881 GB | 5.283 MB |
| **IOSP-4** | 77,406k | 85,146k | 26.214 Mrays/s | 3.819 GB | 0.471 MB |
| **IOSP-0** | 260,178k | 278,989k | 9.252 Mrays/s | 12.051 GB | 0 MB |
| **2-Lvl IOSP (15)** | 128,160k | 49,780k | 27.778 Mrays/s | 4.643 GB | 0.535 MB |
| Scene - Fairy - 174,117 triangles, 12.365 MB of memory used for geometry | | | | | |
| BVH(SAH) | 47,680k | 5,349k | 56.174 Mrays/s | 1.600 GB | 3.577 MB |
| BVH(OMS) | 191,754k | 22,226k | 20.165 Mrays/s | 6.460 GB | 4.194 MB |
| **IOSP-4** | 58,461k | 66,259k | 32.768 Mrays/s | 2.936 GB | 0.294 MB |
| **IOSP-0** | 186,332k | 194,406k | 13.107 Mrays/s | 8.510 GB | 0 MB |
| **2-Lvl IOSP (15)** | 52,533k | 9,973k | 46.875 Mrays/s | 1.738 GB | 0.582 MB |
| Scene - Robot Girl - 1,010,054 triangles, 60.653 MB of memory used for geometry | | | | | |
| BVH(SAH) | 25,693k | 2,832k | 71.494 Mrays/s | 0.861 GB | 21.625 MB |
| BVH(OMS) | 143,301k | 27,639k | 18.289 Mrays/s | 5.197 GB | 16.777 MB |
| **IOSP-4** | 39,033k | 42,496k | 39.322 Mrays/s | 1.910 GB | 1.710 MB |
| **IOSP-0** | 114,559k | 119,955k | 17.476 Mrays/s | 5.242 GB | 0 MB |
| **2-Lvl IOSP (15)** | 32,457k | 14,857k | 46.875 Mrays/s | 1.206 GB | 0.513 MB |
| Scene - Thai Statue - 10.000.002 triangles, 640.002 MB of memory used for geometry | | | | | |
| BVH(SAH) | 21,031k | 3,708k | 65.536 Mrays/s | 0.751 GB | 212.332 MB |
| BVH(OMS) | 57,481k | 5,607k | 37.449 Mrays/s | 1.901 GB | 237.348 MB |
| **IOSP-4** | 29,663k | 35,532k | 26.214 Mrays/s | 1.536 GB | 16.792 MB |
| **IOSP-0** | 51,519k | 52,454k | 15.729 Mrays/s | 2.324 GB | 0 MB |
| **2-Lvl IOSP (15)** | 57,723k | 36,169k | 27.778 Mrays/s | 2.293 GB | 0.520 MB |
| Scene - Venice - 2.447.208 triangles, 192.161 MB of memory used for geometry | | | | | |
| BVH(SAH) | 46,617k | 5,477k | 39.332 Mrays/s | 1.573 GB | 49.473 MB |
| BVH(OMS) | 314,809k | 39,094k | 8.278 Mrays/s | 10.693 GB | 55.957 MB |
| **IOSP-4** | 66,098k | 71,579k | 21.845 Mrays/s | 3.229 GB | 4.130 MB |
| **IOSP-0** | 285,213k | 302,466k | 4.795 Mrays/s | 13.136 GB | 0 MB |
| **2-Lvl IOSP (15)** | 73,191k | 34,957k | 22.059 Mrays/s | 2.754 GB | 0.938 MB |

Table 5: Comparison of our proposed techniques (IOSP-4, IOSP-0 and 2-lvl IOSP (with 15 uncompressed top-levels) with a SAH-BVH (BVH(SAH)) and an object median Split BVH (BVH(OMS)). Measurements have been made on an Intel Core i7-2600 with 3.4 GHz, 16GB RAM, and an NVIDIA GeForce GTX 580. Resolution is $1024 \times 768$ pixels. $N_T$ is the number of tested nodes in total, $N_I$ is the number of ray-object intersections in total, $R$ is the number of traversed rays in millions per second on the GPU excluding the construction step, including ray generation, traversal, simple shading and texturing. Only primary rays are considered. BW/frame is the minimal necessary data throughput (bandwidth) based on the number of node and triangle intersections. Mem is the memory usage of only the acceleration data structures in megabytes.

)

## References


[BEM10] BAUSZAT P., EISEMANN M., MAGNOR M.: The Minimal Bounding Volume Hierarchy. In *Proc. of Vision, Modeling, and Visualization* (11 2010), pp. 227–234. 2, 7

[CSE06] CLINE D., STEELE K., EGBERT P.: Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphic Tools 11*, 4 (2006), 61–71. 2, 4

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1234. 2

[EG07] ERNST M., GREINER G.: Early split clipping for bounding volume hierarchies. In *Proc. of the IEEE Symposium on Interactive Ray Tracing* (2007), pp. 73–78. 8

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proc. of the IEEE Symposium on Interactive Ray Tracing* (2008), pp. 35 –40. 2

[EW11] ERNST M., WOOP S.: Ray tracing with shared-plane bounding volume hierarchies. *Journal of Graphics, GPU, and Game Tools 15*, 3 (2011), 141–151. 2, 7

[EWM08] EISEMANN M., WOIZISCHKE C., MAGNOR M.: Ray Tracing with the Single-Slab Hierarchy. In *Proc. of Vision, Modeling, and Visualization (VMV'08)* (2008), pp. 373–381. 6

[FD09] FABIANOWSKI B., DINGLIANA J.: Compact BVH storage for ray tracing and photon mapping. In *Proc. of Eurographics Ireland Workshop* (2009), pp. 1–8. 2, 7, 8

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster hlbvh with work queues. In *Proc. of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), pp. 59–64. 7

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proc. of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (2007), pp. 113–118. 1

[HHHPS06] HAVRAN V., HERZOG R., H.-P-SEIDEL: On Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proc. of IEEE/Eurographics Symposium on Interactive Ray Tracing 2006* (2006), pp. 1–10. 2

[IH11] IZE T., HANSEN C.: RTSAH traversal order for occlusion rays. *Computer Graphics Forum 30*, 2 (2011), 297–305. 8

[Kar07] KARRENBERG R.: *Memory Aware Realtime Ray Tracing: The Bounding Plane Hierarchy*, 2007. BA thesis, Universität des Saarlandes. 2, 7

[KBK*10] KIM T.-J., BYUN Y., KIM Y., MOON B., LEE S., YOON S.-E.: HCCMeshes: Hierarchical-culling oriented compact meshes. *Computer Graphics Forum (Eurographics) 29*, 2 (2010), 299–308. 2

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. *SIGGRAPH Computer Graphics 20* (1986), 269–278. 2

[KW05] KIPFER P., WESTERMANN R.: Improved GPU sorting. In *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Addison-Wesley, pp. 733–746. 8

[KW09] KELLER A., WÄCHTER C.: Efficient ray tracing without acceleration data structure. *U.S. Patent Applications Publication No. US 2009/0225081 A1* (2009). 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 7

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and memory efficient ray tracing of large models. *Computer. Graphics Forum 27*, 4 (2008), 1313–1321. 1, 2, 7

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 3 (1990), 153–166. 4

[Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Transactions on Graphics 30* (2011), 117:1–117:12. 2, 8

[MW06] MAHOVSKY J., WYVILL B.: Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum 25*, 2 (2006), 173–182. 2

[OSDM87] OOI B., SACKS-DAVID R., MCDONNEL K.: Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE International Computer Software and Applications Conference* (Tokio, Japan, October 1987), pp. 433–438. 2

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers Inc., 2010. 1

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. of the Conference on High Performance Graphics* (2010), pp. 87–95. 7

[RKB06] RATANAWORABHAN P., KE J., BURTSCHER M.: Fast lossless compression of scientific floating-point data. In *Proc. of the Data Compression Conference* (2006), pp. 133–142. 8

[Sam05] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005. 2

[SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proc. of Graphics Interface* (2010), pp. 153–160. 2, 7

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proc. of High Performance Graphics* (2009), pp. 7–13. 8

[Wäc08] WÄCHTER C.: *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Universität Ulm, 2008. 8

[Wal07] WALD I.: On fast Construction of SAH based Bounding Volume Hierarchies. *Proc. of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007), 33–40. 6

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–28. 3

[WDS05] WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses* (2005). 1

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Proc. of Eurographics Symposium on Rendering* (2006), pp. 139–149. 1, 2, 6, 8

[WK07] WÄCHTER C., KELLER A.: Terminating spatial hierarchies by a priori bounding memory. In *Proc. of IEEE Symposium on Interactive Ray Tracing* (2007), pp. 41 –46. 2

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 1, 2

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proc. of Graphics Hardware* (2006), pp. 67–77. 2, 3, 8


)

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-LODs: fast LOD-based ray tracing of massive models. *The Visual Computer 22*, 9-11 (2006), 772–784. 1

[Zac02] ZACHMANN G.: Minimal hierarchical collision detection. In *Proc. of the ACM symposium on Virtual reality software and technology* (2002), pp. 121–128. 2

[ZU06] ZUNIGA M., UHLMANN J.: Ray queries with wide object isolation and the de-tree. *Journal of Graphics Tools 11*, 3 (2006), 27–45. 2, 3, 8

)