Vision, Modeling, and Visualization (2010)

The Minimal Bounding Volume Hierarchy

Pablo Bauszat[1] and Martin Eisemann[1] and Marcus Magnor[1]

[1]Computer Graphics Lab, TU Braunschweig, Germany

Abstract

Bounding volume hierarchies (BVH) are a commonly used method for speeding up ray tracing. Even though the memory footprint of a BVH is relatively low compared to other acceleration data structures, they still can consume a large amount of memory for complex scenes and exceed the memory bounds of the host system. This can lead to a tremendous performance decrease on the order of several magnitudes. In this paper we present a novel scheme for construction and storage of BVHs that can reduce the memory consumption to less than 1% of a standard BVH. We show that our representation, which uses only 2 bits per node, is the smallest possible representation on a per node basis that does not produce empty space deadlocks*. Our data structure, called the* Minimal Bounding Volume Hierarchy *(MVH) reduces the memory requirements in two important ways: using implicit indexing and preset* surface reduction factors*. Obviously, this scheme has a non-negligible computational overhead, but this overhead can be compensated to a large degree by shooting larger ray bundles instead of single rays, using a simpler intersection scheme and a two-level representation of the hierarchy. These measure enable interactive ray tracing performance without the necessity to rely on out-of-core techniques that would be inevitable for a standard BVH.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object Hierarchies I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Ray tracing is one of the fundamental methods for image synthesis in computer graphics. Due to the algorithmic advances in this field and more processing power, interactive ray tracing is nowadays possible [PMS*99, WBWS01]. The biggest gain in efficiency is achieved by using acceleration data structures, like kd-trees or BVHs.

The idea behind a BVH [RW80, KK86] is to subdivide the primitives of a scene into possibly overlapping sets. For each of these sets a bounding volume (BV) is computed and these are arranged in a tree structure. 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 primitives. Ray traversal then starts at the root node and if a ray misses a BV in this hierarchy, the whole subtree can be skipped.

Unfortunately, all common acceleration data structure can use a significant amount of memory, especially if complex models are to be rendered. Compared to other approaches BVHs usually have the lowest memory requirements and a precomputable memory footprint as shown in [WK06]. Several approaches exist which try to minimize the memory usage by using one or more of the following methods:

1. Reduction of the information that is stored for each BV
2. Reduction of the precision of the data the BV is stored with
3. Removal of child and primitive pointers by implicit indexing
4. Increasing the branching factor, i.e., the number of children per node, to reduce the total number of nodes in a BVH
5. Compression of the hierarchy data

Hybrid techniques have been thoroughly investigated during the last years which try to combine the benefits of kd-trees and BVHs. Most of them are a variation of the Bounding Slab Hierarchy by Kay and Kajiya [KK86]. Originally they used at least six bounding slabs to form a closed

DOI: 10.2312/PE/VMV/VMV10/227-234

hull around an object. If these slabs are perpendicular to the world coordinate axis, the BV is called an axis-aligned bounding box (AABB). By saving the active ray interval, one is able to estimate a hit or miss of a ray with a BV even if it uses less than six sides. Common approaches use one [Zac02, WK06, EWM08], two [WMS06, ZU06], or a mix of different bounding slabs [HHHPS06]. This sparse bounding slab representation may reduce the memory requirements up to a third, compared to a standard BVH. But not all of these approaches are able to prevent empty space deadlocks, which we describe in Sect. 2.1, or need at least some special treatment for them.

In standard BVHs the bounds of an AABB are stored using six floating point values.Mahovsky and Wyvill [MW06] investigated a hierarchical scheme for encoding BVs in a BVH relative to their parent node that reduces the storage requirements by 63%-75%. They also noted that the computational overhead can be alleviated by tracing bundles of rays, as whole frustra can be easily culled. Cline *et al.* [CSE06] take a similar approach. They also use a hierarchical encoding scheme, compressing a node to 12 bytes, plus a high branching factor of four and implicit encoding of the child pointer due to a heap like data structure. Segovia and Ernst [SE10] follow Mahovsky's approach and use 4 bytes per BVH node, but extend it in two important ways. Despite compressing the triangle data as well, they store the BVH nodes in clusters to reduce the size of all references to child nodes and they use a two-level BVH which uses both uncompressed BVH nodes for the top levels and compressed nodes for the rest of the hierarchy. This idea of a two-level hierarchy for compression was first presented in [LYTM08]. We make use of some of their techniques, allowing us to reduce the memory requirements to two bits per node.

Kim *et al.* [KMKY10] apply a compression algorithm to reduce the memory footprint of a BVH. The algorithm introduces a decompression cost during the traversal which should result in a performance decrease. For large scenes (especially scenes which cannot be rendered without out-of-core techniques) this performance loss is canceled by the possibility to keep the whole BVH in the main memory and better cache usage. Their approach achieves a compression ratio of 12:1, while we propose a technique that is able to achieve a compression ratio of up to 101:1.

Though other factors like geometry, textures, etc. influence the memory requirements as well, we will concentrate in this paper only on the acceleration data structure and present the smallest possible representation on a per node basis for a BVH. The other factors are left for further research.

2. The Minimal Bounding Volume Hierarchy

We propose a minimal representation for a BVH which offers the lowest memory consumption possible on a per node basis, without surface reduction deadlocks. In this paper we present the Minimal Bounding Volume Hierarchy (MVH) as a k-ary object partitioning scheme. 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. 2.3). Standard AABBs used in BVHs usually encode the following information in 32 byte structures: minimal and maximal bounds, reference to the child nodes, is it a leaf or inner node, number of contained triangles in case of a leaf node, axis used for ordered traversal [WBS07] and the first node that is to be traversed along this axis. We will show how to remove or implicitly save all this information using only 2 bits per node plus very few global parameters. There has been a lot of work over the years in compression of BVHs and we would like to be able to leverage that work as much as possible, adopting the implicit indexing from [CSE06] and the 2-level BVH from [SE10].

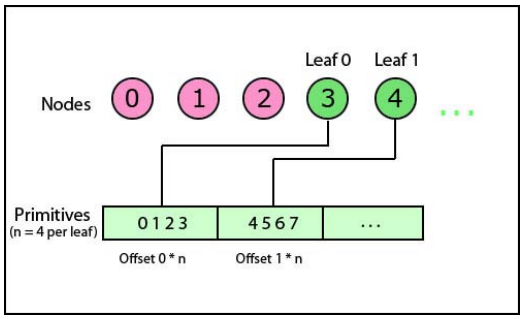


Figure 1: *Memory layout of the MVH. The nodes array contains the inner nodes first and then the leaf nodes. Each node itself is represented by only 2 bits. All other information is reconstructed on the fly.*

2.1. Data Representation

A MVH is essentially a complete k-ary tree that is stored in an array, and indexed like a heap. Every node is either a leaf node or has k children. Node zero is the root node and for any other inner node with index i its children are indexed with $ik+1$ to $ik+k$, we adopted this indexing scheme from [CSE06]. In our examples we used $k = 2$.

We set a fixed number of primitives n per leaf node. This serves several purposes. Firstly, no triangle count in the leaf nodes is needed. Secondly, for each leaf node we can compute the offset into the primitive array from its index i by

$$primID = (i - l)n \ ,$$

where l is the index of the first leaf node in the array. This implicitly declares all nodes with an index smaller than l as inner nodes, which is important for the traversal in Sect. 2.3. This is visualized in Fig. 1.

During the construction of the MVH, described in

Sect. 2.2, every primitive in the left child is assured to contain smaller or equal coordinate values than the primitives in the right child with respect to the splitting axis. This way we can get rid of the traversal axis and the index of the first child node to test along that axis. Please note, that this is an approximation of the real traversal axis. In a standard BVH the traversal and splitting axis do not necessarily need to be the same, but in most cases they are. Sect. 2.2 describes our choice of the splitting axis.

After getting rid of all the child and primitive pointers the next step is to minimize the information that is stored for the bounding boxes of the nodes. In the common BVH all 6 slabs of the AABB are stored. For each of the min/max slabs the exact position is given. The AABB of a node and the AABBs of the child nodes share at least six out of twelve slabs. Testing these slabs would be redundant as already stated in [EWM08, FD09] and can be omitted by saving the active ray interval, i.e. entry and exit parameter of a ray $r = \vec{o} + t \cdot \vec{d}$.

The AABB for a node which exactly fits the primitives encapsulated by the node is furthermore called the *real* AABB of a node. Hybrid BVH techniques, as e.g [WK06, HHHPS06, WMS06, ZU06, EWM08] reduce the memory footprint for the AABB by only storing bounding informations for one dimension or even only one slab. The traversal starts with the roots real bounding box which covers all the primitives in the scene. Then for each traversal step the ray is only tested against the new slabs. One can think of this as the process of constantly reducing the size of the previous bounding box using the slabs information stored in the traversed node. The deeper the traversal steps into the tree the more empty space is *carved* away and the smaller is the volume of the node's bounding box. Notice that former hybrid techniques stored the exact slabs positions as floating point values.

For the MVH node we also store only one dimension of the AABB, but we do not store the min and max slabs directly. Instead we use a fixed reduction factor $\zeta \in (0,1)$, i.e. in each subdivision step the size of the AABB along the splitting axis is either reduced by a constant factor or stays the same if the reduction is not possible due to the underlying geometry. For a given dimension we can reduce the parent's AABB volume by increasing the minimum slab, decreasing the maximum slab or both. The four resulting bounding box variants are:

1. No-Cut: if no surface reduction is possible for this node
2. Left-Cut: if the minimum slab is increased
3. Right-Cut: if the maximum slab is decreased
4. Both-Cut: if the Left-Cut and Right-Cut is used

For each node of the MVH we therefore only store the above information requiring 2 bits. The 2 bits cover the four possible cases, see Figure 2, and we set the bits accordingly:

1. Bit 00: if none of the reduced AABB would include the real bounding box

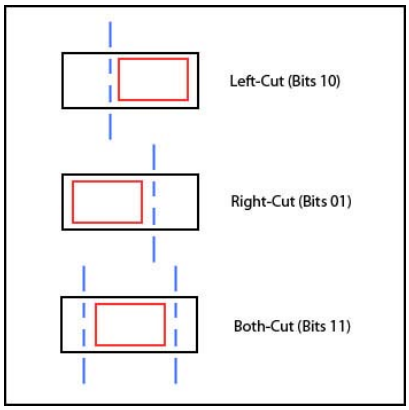


Figure 2: *The image shows the cuts that are performed depending on the node bits for the 2-dimensional case. The black box is the previous bounding box, while the red box represents the real bounding box (i.e. the bounding box that encapsulates all the primitives of the node). The blue dotted lines define where the previous box is cut and the min and/or max slabs are replaced. Notice that the real bounding box might be smaller than the bounding box used in the MVH node. However the resulting virtual bounding box always includes the real bounding box and accordingly always encapsulates all the primitives for the node.*

2. Bit 10: The Left-Cut AABB includes the real bounding box
3. Bit 01: The Right-Cut AABB includes the real bounding box
4. Bit 11: The Both-Cut AABB includes the real bounding box

One could even go one step further and reduce the information stored to 1 bit by removing the Both-Cut and reducing the slab according to the splitting axis, i.e. the left child of a node has either the same bounds as its parent or its maximum slab is decreased and the right child has either the same bounds or its minimum slab is increased. A similar approach was used in [WK06], though the slabs were saved with full precision. Unfortunately this can lead to what we call an *empty space deadlock*. In such a deadlock empty space cannot be removed without introducing specialized or empty bounding boxes. Figure 3 gives an example. This deadlock can appear whenever objects are diagonally arranged. When the objects are divided into subsets and only the maximum slab of the left child and the minimum slab of the right child is adjusted the resulting boxes will contain a large amount of empty space. This empty space cannot be removed with further subdivision, only split. It will therefore always stay part of at least one child node down to the leaf nodes. Havran *et al.* [HHHPS06] therefore made use of additional six-sided AABB to solve this problem. It is possible to remove deadlocks with even a 1bit representation, by cycling through the

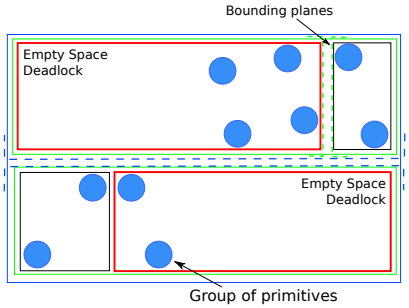


Figure 3: *An empty space deadlock example. If the left child node is restricted to only adjust the maximum slab and the right child node to only adjust the minimum slab, empty space cannot be removed in certain arrangements by further subdivisions. The empty space in the red boxes will always be part of at least one of the child nodes or subdivided among them, but cannot be removed. The bounding boxes have been slightly enlarged for better readability.*

splitting axes depending on the level in the hierarchy, but we refrained from doing so due to the tremendous performance loss incurred. Therefore the additional bit is essential to be able to carve away the empty space if necessary, as it allows to carve away empty space on both sides of a splitting axis.

2.2. Construction

To ensure the applicability of all the memory reduction techniques proposed in Sect. 2.1 we use an object median like split strategy inspired by [CSE06] to build our MVH:

1. The total number of primitives P (or its indices) is extended by duplicating the last primitive until the number of primitives is a multiple of the given fixed primitives per leaf count n. This ensures that each leaf will contain exactly the same amount of primitives.
2. The total number of nodes is computed. For a binary BVH this is simply $2(P/n) - 1$.
3. For each node the number of primitives in the subtree is computed. This is done in a two-step algorithm. First the fixed primitive per leaf count is assigned to all leaf nodes. Then the count for all inner nodes is computed by summing up the count of its children in a bottom-up manner.
4. Corresponding to the total number of nodes N, an array of at least $2N$ bits is allocated, we use an array of 32-bit integers.
5. The MVH is then built using an object median split where the partitioning process uses the precomputed counts for dividing the object list in two parts. For each node the parent's bounding box is cut using the three variants (Left-Cut, Right-Cut and Both-Cut) and compared to the real bounding box for the primitive list. If one of them is applicable the result is stored using the corresponding two bits in the nodes array. When a leaf node is reached, the primitives or their indices are put at the appropriate offset position in the primitive array. In each subdivision step the longest extent of the approximated AABB is chosen as the new splitting axis. The reduction factor ζ plus the roots bounding box are saved as global parameters.

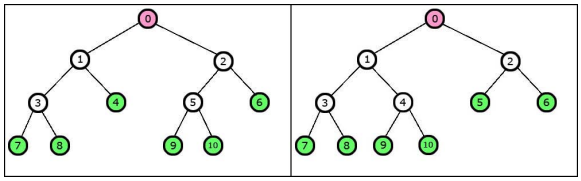


Figure 4: *The left image shows a tree for a primitive count of six, $k = 2$ and $n = 1$ built from a usual left-balancing object median split. The right image shows the tree built with the modified object median split. The tree is constructed in a way that all inner nodes are stored first in the memory, followed by the leaf nodes, and still is left-balanced.*

A comparison of our construction technique to a standard object median split is given in Fig. 4.

A study on the influence of ζ is given in Sec. 3. If preprocessing time is not crucial, ζ could be optimized using the expected execution time for a BVH as described in [GS87].

2.3. Traversal

The traversal of the MVH resembles the traversal of a hybrid BVH [EWM08] with an additional bounding plane reconstruction step.In order to init the traversal, the active ray interval is computed by clipping the ray against the root's AABB. For each traversed node we first reconstruct the bounds of that node based on the corresponding bit representation and the approximated parent's AABB. If the node is a leaf node, the according primitives are tested. A leaf node is reached when the index of the left child for a node is greater or equal to the number of nodes in the tree. If any other node is hit, we first reconstruct the approximated bounding plane and adjust the ray parameters accordingly. To reconstruct the splitting plane we only need to search for the axis of the longest extent of the parent node. A simplified pseudocode for the intersection test is given in Fig. 5, the rest of the traversal is similar to a standard BVH traversal scheme.

2.4. Two-Level MVH

Compressing a BVH node always leads to a significant overhead and performance drop, compared to an uncompressed BVH representation [Mah05]. There are two findings which allow to improve the performance while maintaining a small memory footprint:

1. In a balanced BVH half of the memory requirements are

```
float minTab[4] = { 0.00f, zeta, 0.00f, zeta };
float maxTab[4] = { 0.00f, 0.00f, zeta, zeta };
bool intersect(Ray& r, float& tHit,
  int* mvhArray, int node, AABB& parent,
  float tNear, float tFar){

  // reconstruct AABB //
  vec3 size  = parent.max - parent.min;
  char axis  = GetAxisOfMaximumExtent(size);
  char bitID = GetNodeCut(node);
  if(bitID == 0){ return true;}
  parent.min[axis] += size[axis]*minTab[bitID];
  parent.max[axis] -= size[axis]*maxTab[bitID];

  // intersection //
  float tmin, tmax;
  parent.intersect(r, tmin, tmax, axis);
  float tNear = max(min(tmin, tmax), tNear);
  float tFar = min(max(tmin, tmax), tFar);
  return ((tNear<=tFar) && (tNear<=tHit));
};

char GetNodeCut(int node, int* mvhArray)
{
  int iIndex = node » 4;
  int iShift = (node & 15) « 1;
  return ((mvhArray[iIndex] » iShift) & 0x3);
}
```

Figure 5: *Intersection scheme for a single node of the MVH. For the rest of the traversal (not shown here) standard techniques with implicit indexing can be used.*

used by the last level, as the number of nodes doubles per level

2. Most rays will have to traverse the top nodes of the hierarchy, therefore a good partitioning here is crucial for a good performance.

We therefore create a two-level MVH. Such a two level approach was also successfully applied to BVH compression in [SE10]. The top of the tree is is stored in an uncompressed BVH format and we use the very efficient surface area heuristic for subdivision [MB90, Wal07]. The leafs of this BVH do not point to triangles directly but to a separate MVH instead. We also save the offset into the primitive array in this leaf node as well as the number of triangles per leaf. Each MVH therefore only needs to save one additional integer for its number of nodes despite the compressed nodes, keeping the additional memory overhead very low. This way the user is able to trade of performance vs. compression ratio.

3. Results and Discussion

In this section we compare the common BVH with our implementation of the MVH for rendering speed and memory usage. We implemented the MVH in our own interactive ray-tracing system which supports tracing single rays as well as packets.

The statistics are taken on an AMD 5600+ 2.8 GHz Dual core system with 2 GB Ram. The images are rendered completely on the CPU using both cores of the system. All rendered images are of size 1024×768 pixels. We tested our MVH on a variety of different test scenes and compared it to an optimized BVH using the surface area heuristic for construction [MB90, Wal07]. Some of them are shown in Fig. 6. The maximum number of primitives per leaf node was set to four.

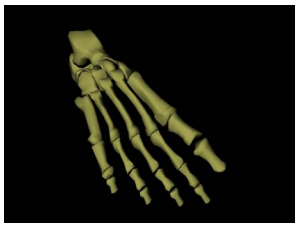
Memory Footprint: At first we compare the memory consumption of the common BVH and the MVH, whose size reduction is our main goal. In Table 1 an overview of our test scenes and the memory usage for the BVH and MVH is given. While for the MVH the number of primitives per leaf node is fixed due to our specialized construction routine, the BVH is built using the sophisticated surface area heuristic (SAH) [MB90, Wal07]. For one primitive per leaf this would result in a constant factor of 128:1, i.e., for all scenes a BVH would consume 128 times more memory than our complete MVH. If we let the SAH decide on the subdivision and when to create a leaf node the ratio might de- or increase by a small amount, due to the fact that more or less than four primitives might end up in one leaf node.

Using the two-level representation of Section 2.4 results in a small memory overhead but at tremendously increased performance, see Table1. The influence of the ratio between uncompressed levels and compressed levels in terms of memory requirements and ray tracing performance is given exemplarily for the Fairy scene in Figure 7. Note that even at a level of 14 the memory requirements for our two-level MVH is below 250kB compared to 1.91MB for an uncompressed BVH (not all leafs of the two-level MVH are at level 14 due to the SAH, therefore the memory requirements are below the 512kB for a complete tree).

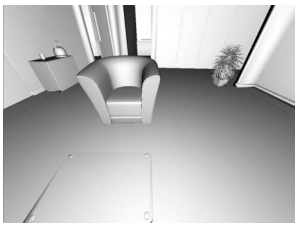
For all of our tests we set the number of uncompressed BVH levels to ten, which results in a memory overhead of at most 35kB (32 kilobytes for the BVH nodes and one integer for the global MVH information for each BVH leaf node).

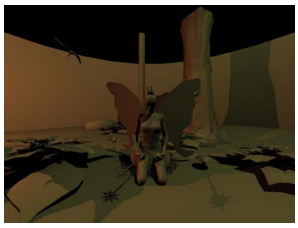
Reduction factor: We investigated the influence of different reduction factors for our test scenes. We choose 4 primitives per leaf for all scenes and used packet ray tracing render times with a complete MVH (without two-level acceleration) for comparison. Fig. 8 shows the render times in seconds for the different reduction factors from 0.1 (10% per cut) up to 0.5 (50% per cut). The best factor was usually achieved with a value of between 0.35 and 0.30.

Different cuts on the higher levels of the MVH can influence the overall quality. This is the case with any greedy construction scheme. Therefore, it may happen, that the function is non-convex on the reduction factor, see e.g. the Sponza graph in Fig. 8. However this is seldomly the case.

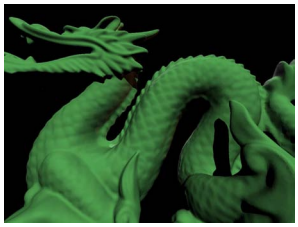

Bones


Sponza


Office


Fairy


Cars


Dragon


Buddha

Figure 6: *Scenes used to test the MVH performance.*

Scene	# Tris	BVH	MVH	Ratio BVH:MVH	2-level MVH	Ratio BVH:2-level MVH
Bones	4,204	70.75KB	0.51KB	100:1	26.59KB	3:1
Sponza	67,462	797.94KB	8.23KB	97:1	41.02KB	20:1
Office	385,376	1,636.50KB	47.04KB	35:1	72.71KB	23:1
Fairy	174,117	1,957KB	21.25KB	92:1	50.41KB	39:1
Cars	549,662	6,051.63KB	67.09KB	90:1	170.20KB	36:1
Dragon	871,306	10.44MB	0.11MB	101:1	142.08KB	75:1
Buddha	1,087,716	13.39MB	0.13MB	101:1	168.50KB	81:1

Table 1: *Comparison of the memory consumption between a BVH, a complete MVH and a 2-level MVH with ten uncompressed hierarchy levels.*

Scene	BVH			MVH					2-level MVH				
	ts	tt	rt	ts	tt	rt	ζ	Loss	ts	tt	rt	ζ	Loss
Bones	2.5	1.3	0.10s	67.7	84.3	2.70s	0.3	1:27	3.2	2.7	0.16s	0.3	1:1.6
Sponza	39.2	16.7	0.98s	3,828	415	160.7s	0.3	1:164	159	196.6	7.35s	0.3	1:7.5
Office	26.1	63.3	1.42s	1,992	262	328.2s	0.4	1:231	110	153.6	5.11s	0.4	1:3.6
Fairy	21.0	10.5	0.56s	12,690	3,740	519.0s	0.35	1:926	72.4	83.9	3.46s	0.35	1:6.1
Cars	44.5	15.4	1.06s	4,461	1,943	221.0s	0.4	1:208	148.9	186.4	7.07s	0.4	1:6.5
Dragon	19.3	6.6	0.51s	2,081	1,676	66.8s	0.35	1:131	154.1	188.0	8.04s	0.35	1:15.7
Buddha	2.5	0.76	0.11s	311	262	10.37s	0.35	1:94	23.3	28.4	1.27s	0.35	1:11.5

Table 2: *Comparison of the single ray traversal between a BVH, our MVH and our 2-level MVH. As expected the performance strongly decreases for single rays. ts = traversal steps, tt = triangle tests, rt = render time. Number of intersection is given in millions.*

Scene	BVH			MVH					2-level MVH				
	ts	tt	rt	ts	tt	rt	ζ	Loss	ts	tt	rt	ζ	Loss
Bones	19.1	219	0.02s	316.4	369.9	0.06s	0.3	1:3	24.6	256.6	0.02s	0.3	1:1.09
Sponza	190.8	22	0.08s	1,542	3,393	1.87s	0.3	1:25	784.6	2,102	0.13s	0.3	1:1.71
Office	179.2	2,961	0.10s	25,002	7,895	3.26s	0.4	1:33	702.5	6,464	0.17s	0.4	1:1.74
Fairy	185.6	2,337	0.08s	49,250	8,114	3.83s	0.35	1:46	683.9	5,527	0.17s	0.35	1:1.98
Cars	332.9	2,671	0.12s	20,458	14,350	2.36s	0.4	1:20	1,067	6,996	0.23s	0.4	1:1.96
Dragon	391.0	4,399	0.15s	11,244	19,142	1.55s	0.35	1:11	1,777	15,468	0.40s	0.35	1:2.71
Buddha	281.3	1,191	0.10s	3,042	20,939	0.58S	0.35	1:6	882.6	11,155	0.25s	0.35	1:2.38

Table 3: *Comparison of the packet traversal between a BVH and our MVH techniques. When using* 16×16 *ray bundles the overhead can be compensated to a large degree. Please keep in mind the drastic memory reduction of the MVH techniques. ts = traversal steps, tt = triangle tests, rt = render time. Number of intersection is given in thousands.*

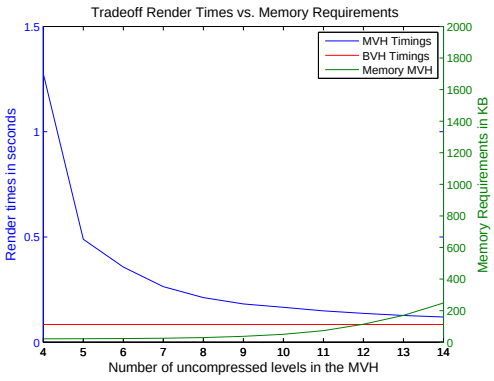


Figure 7: *Comparison between render times and memory requirements for the Fairy scene using different numbers of levels for the uncompressed BVH. Each leaf in the BVH points towards a separate MVH. The memory requirements for a complete BVH would be 2MB for this scene.*

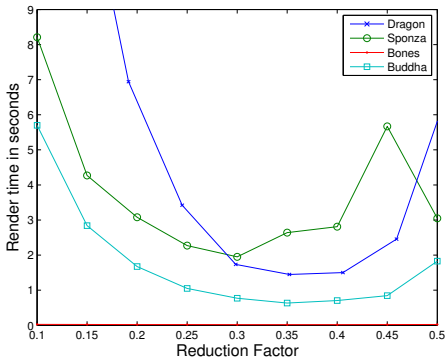


Figure 8: *Influence of the reduction factor* ζ *on the render times.*

Performance: We compared the ray shooting performance for both single and packet-based ray tracing using a simple eyelight shader. As expected, the approximated AABBs of the MVH lead to significant performance losses for single ray traversal, see Table 2. However, when shooting ray bundles the introduced overhead can be compensated to a much larger degree, see Table 3. We use 16×16 ray bundles. Frustum culling is used to skip nodes that are not hit by the packet at all. We use the ranged packet traversal technique as proposed in [WBS07]. The performance ratio of the MVH compared to a BVH is obviously better with packet traversal than for single ray traversal as expected. The rendering times for a complete MVH are $3-46$ times slower than for a BVH. This shows the drawback of the object median split, which has a tough time to deal with large triangles and non-uniform distributions. Using the two-level MVH, the rendering times increase only by a factor between 1.09 and 2.71 to an optimized BVH, but only a few kilobytes of memory are used. We could even go on and increase the number of uncompressed BVH nodes to improve the rendering times at the cost of a worse compression, which is still better than any published BVH compression published so far, to the best of our knowledge. Note that for the more complex scenes, like the dragon or Buddha more than 99% of the nodes are in compressed format. This two-level representation is in fact a very convenient way to tradeoff memory reduction for performance.

We also tested our approach for more realistic shadings, multiple light sources and shadows, as well as effects such as reflections and refractions, to reduce to amount of coherency in the ray directions. Some results are shown in Table 4.

Please note that all the given examples fitted into main memory, even for the standard BVH. We expect the performance ratio to improve strongly as soon as a standard BVH including all primitves and additional information does not fit into main memory anymore.

4. Conclusion

In this paper we proposed a scheme for a BVH-like acceleration data structure which trades off rendering performance for smaller memory consumption. The MVH has the lowest

Scene	BVH			MVH					2-level MVH				
	ts	tt	rt	ts	tt	rt	ζ	Loss	ts	tt	rt	ζ	Loss
Fairy	536.3K	8,250K	0.29s	104.6M	153.3M	11.8s	0.35	1:41	2,135K	35.4M	0.91s	0.35	1:3.11
Cars	1,221K	10,560K	0.56s	90.7M	921.6M	25.3s	0.4	1:45	5,612K	79.0M	2.29s	0.4	1:4.1

Table 4: *Comparison of the packet traversal between a BVH and our MVH techniques with advanced shading effects. For both scenes two point light sources were placed in the scene and whitted-style ray tracing is performed, including texturing. We restrict the secondary rays to two bounces at the maximum. ts = traversal steps, tt = triangle tests, rt = render time.*

per-node memory footprint ever proposed for an acceleration structure that can still achieve interactive performance on many standard scenes. We showed a specialized construction algorithm which is based on the object median split and how to modify the common BVH single ray traversal routines to handle MVH traversals. While a complete MVH is probably more of a theoretical than practical interest, the two-level MVH approach gives us a convenient way to trade-off memory consumption for performance. The MVH is best used for huge, static scenes with millions of primitives and for systems with low memory resources, as it allows the rendering of much more complex models without using out-of-core techniques.

For future work we would like to investigate the impact of using more than two children per node, as this might not only reduce the memory consumption even further, but might be beneficial for a better subdivision, as well as tracing incoherent rays [DHK08]. Also an adaptive reduction factor per level of the hierarchy might increase the rendering performance. The additional memory requirements would be only $\log_k(N)$ floating point values for N nodes.

References

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

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

[EWM08] EISEMANN M., WOIZISCHKE C., MAGNOR M.: Ray Tracing with the Single-Slab Hierarchy. In *Proceedings of Vision, Modeling, and Visualization (VMV'08)* (10 2008), pp. 373–381.

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

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

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

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *Proceedings of SIGGRAPH '86* (1986), pp. 269–278.

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2010), 273–286.

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: Reducem: Interactive and memory efficient ray tracing of large models. *Comput. Graph. Forum 27*, 4 (2008), 1313–1321.

[Mah05] MAHOVSKY J.: *Ray Tracing With Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, 2005.

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

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

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Symposium on Interactive 3D Graphics* (1999), pp. 119–126.

[RW80] RUBIN S., WHITTED T.: 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* (New York, NY, USA, 1980), ACM Press, pp. 110–116.

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

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

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 1–18.

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 (Eurographics Symposium on Rendering)* (2006), pp. 139–149.

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

[Zac02] ZACHMANN G.: Minimal hierarchical collision detection. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2002), ACM, pp. 121–128.

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