Dr.-Ing. Martin Eisemann
eisemann@cg.tu-bs.de
Computer Graphics Lab, TU Braunschweig

Pablo Bauszat
bauszat@cg.tu-bs.de
Computer Graphics Lab, TU Braunschweig

Prof. Dr. Ing. Marcus Magnor
magnor@cg.tu-bs.de
Computer Graphics Lab, TU Braunschweig

Implicit Object Space Partitioning: The No-Memory BVH

Technical Report 2011-12-16

December 23, 2011

Computer Graphics Lab, TU Braunschweig

Contents

Abstract

We present a new ray tracing algorithm that requires no explicit acceleration data structure and therefore no memory. It is represented in a completely implicit way by triangle reordering. This new implicit data structure is simple to build, efficient to traverse and has a fast total time to image. The implicit acceleration data structure must be constructed only once and can be reused for arbitrary numbers of rays or ray batches without the need to rebuild the hierarchy. Due to the fast build times it is very well suitable for dynamic and animated scenes. We compare it to classic acceleration data structures, like a Bounding Volume Hierarchy, and analyze its effciency.

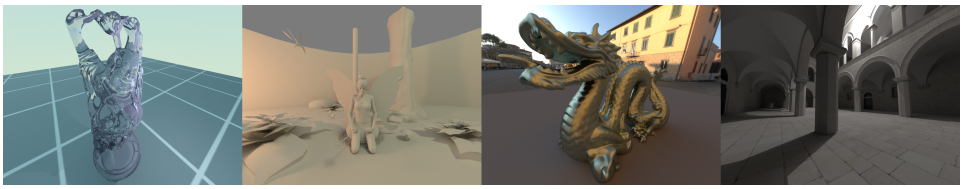

Figure 1: Images from our test scenes generated by our new ray tracing algorithm.

1 Introduction

Ray tracing (RT) is one of the fundamental algorithms for image synthesis and is favoured in many offline renderers for its flexibility and physically-based methodology [PH10, Sol]. Unlike in rasterization, interoperability between different rendering effects and materials is of no concern. However, to synthesize an image, millions of rays have to be traced and their intersection with scene geometry needs to be computed. As the complexity of the naive algorithm is $O(primitives \times rays)$, additional acceleration data structures are commonly used, such as kd-trees or Bounding Volume Hierarchies (BVH). Their shortcomings are well-known: additional memory overhead, often inconsistent memory footprints, update or recreation complexity when it comes to animated scenes and applicability for RT hardware implementations is complicated [SWW+04, WMS06]. Therefore, an algorithm that computes the necessary intersections without any additional data structure is of great interest.

The main idea underlying this paper is the following: since object-partitioning schemes are derived from the geometric properties, why not use the geometric properties directly to implicitly represent the acceleration data structure? Each bounding plane in a BVH node is also represented by a vertex in the scene geometry (or in the case of analytical surfaces usually by a control point). If one knows during traversal which scene point is responsible for spanning the current node, the acceleration data structure itself becomes redundant. In this paper we propose a way to provide this information without any additional memory usage besides the geometric data by pre-sorting the primitives before tracing rays. This has several benefits compared to classic explicit acceleration data structures. Memory consumption is reduced to a minimum. Only the geometry and ray information need to be saved. The acceleration data structure is solely represented by the primitives itself, turning the construction into a simple sorting procedure. This allows to ray-trace fully dynamic scenes and reduces the total time to image. As long as the scene does not change, the implicit hierarchy stays valid even if the lighting or viewpoint changes.

It should be noted that beside the acceleration data structure other components like vertex and normal data, textures and texture coordinates

largely affect the memory requirements, often even more than the acceleration data structure but these are different research topics and it is up to the developer who needs the application whether saving this memory is necessary.

The rest of the paper is organized as follows. In Section 2 we will discuss some previous work before we introduce our implicit object-partitioning scheme in Section 3. Section 4 evaluates the performance of our algorithm, and we conclude and give an outlook on promising future work in Section 5.

2 Related Work

Most acceleration data structures for RT can be classified as being either a *space partitioning* or *object partitioning scheme*. Good surveys on the classic structures can be found in [WPS+03, WMG+07]. Our goal is to represent the data structure implicitly by rearranging the geometry itself. Therefore, space-partitioning schemes are less well suited for our task as a single primitive might need to be associated with more than one node which renders the task impossible without frequent rebuilds of the data structure. Thus, we will concentrate on object partitioning schemes here.

Object Partitioning: Object partitioning schemes subdivide the scene into potentially overlapping sets of primitives. For each of these sets a Bounding Volume (BV) is usually computed and a Bounding Volume Hierarchy (BVH) is constructed [RW80, KK86]. If a ray misses a node the whole subtree can be skipped.

Most object partitioning schemes are derived from the bounding slabs hierarchy of Kay and Kajiya [KK86]. The most common derivation is an axis-aligned bounding box representation where each node is bound by six planes orthogonal to the world coordinate axes as it provides a good trade-off between efficiency and effectiveness.

Hybrid approaches overcome the limitation that the BV must form a closed hull by saving the active ray interval and employ this to estimate a hit of a ray with a node conservatively. This kind of hierarchy was independently introduced to computer graphics by several researchers [HHHPS06, WK06, WMS06, ZU06]. The DE-Tree by Zuniga *et al.* [ZU06] uses two opposing slabs to bound each node and tries to keep larger objects higher in the hierarchy. Woop *et al.* [WMS06] showed a hardware implementation of a similar structure called the b-kd tree. In Wächter *et al.*'s Bounding Interval Hierarchy (BIH) [WK06] a single slab for each child represents the node. A more flexible alternative to the BIH was later presented in [EWM08] where the bounding plane was no longer fixed by the splitting plane of the node allowing for arbitrary subdivision schemes. A similar observation

was made by Havran *et al.* [HHHPS06] which incorporate additional six-sided bounding volumes to overcome the problem. Our hierarchy shows similarities to the b-kd tree by Woop *et al.* [WMS06] but does not save any of the information explicitly.

Construction: Especially for complex and animated scenes, not only the rendering time but also the construction time is an essential part of the total time to image. A survey of current techniques for animated RT can be found in [WMG+07]. Once a BVH is constructed simple refitting is arguably the fastest update [vdB97, WBS07, LYTM06] but for arbitrary polygon soups the hierarchy may quickly degrade in quality. Therefore, the choice is to rebuild the hierarchy as fast as possible or find other ways to update. Wächter *et al.* [WK06] proposed a grid-guided construction scheme for BVHs which can greatly decrease build time compared to standard methods like spatial splits or the surface area heuristic (SAH). In cases where only parts of the scene change a selective update as proposed by Garanzha *et al.* [Gar08] may be more suitable. Lauterbach *et al.* [LGS+09] introduced the Linear Bounding Volume Hierarchy which is based on sorting the primitives along a space filling Morton curve which implicitly partitions the objects and speeds-up BVH construction. This approach was later extended with a more efficient hierarchical algorithm [PL10] and adjusted for current graphics cards in [GPM11] resulting in the fastest construction up-to-date.

Our construction allows for real-time updates of medium sized scenes containing hundreds of thousands of triangles, where additional memory is only needed temporarily during the construction.

Memory Reduced Representations: Though the benefits, as mentioned earlier, are various, very few currently existing techniques try to implicitly represent the acceleration data structure. This is inherent to the problem as it is a very challenging task to represent an acceleration structure without any memory. What has been done so far is to make use of a lazy evaluation scheme by an efficient divide-and-conquer approach.

Starting with a given set of rays and primitives as well as the space covered by these primitives, Mora [Mor11] proposes to subdivide the current space along one axis and then test all active rays and active objects against this space. If an overlap is detected the procedure is recursively continued with the subdived space and the new active rays and objects. As soon as a termination criterion is reached, e.g. the number of objects is below a certain threshold the algorithm switches to a naive ray tracing by testing all active rays left against all objects left. As objects and rays can intersect both subdivided spaces, the partitioning has to be performed twice per traversal step.

Keller *et al.* [KW09, KW11] proposed an implicit Bounding Volume Hier-

archy that builds on a similar divid-and-conquer technique. In each traversal step they first compute the Bounding Box for the current objects. In the next step the active rays are computed, i.e. those who intersect the box. The objects are partitioned into two sets according to a chosen splitting plane. The algorithm is then recursively called for the active rays and the new partitions.

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 by the rays. Occluded parts are left unpartitioned. A common drawback of these approaches is that the partitioning has to be repeated for each ray batch and therefore also for each new image, even if the scene is static. Our implicit hierarchy is created once in a preprocessing step and can be reused for arbitrary numbers of rays as many times as necessary as long as the scene geometry does not change.

3 Implicit Object Space Partitioning

We propose a simple ray tracing algorithm which offers interactive rendering times for both static and dynamic scenes. Our acceleration data structure for RT exploits scene geometry directly to represent an implicit hierarchical acceleration data structure without requiring any memory.

3.1 Overview

For our purposes we will suppose that the primitives are always triangles though it should be noted that the same approach works for any type of object which provide their extends along the world coordinate axes. Therefore instancing could be directly employed as well.

In the classic BVH traversal scheme the BVH is tested in a recursive top-down manner skipping nodes missed by the query ray. Our No-Memory Hierarchy (NMH) works in very much the same way, only that the bounds of the BVH nodes are derived directly from the primitive information itself. Our main observation here is that each side of the AABB is defined by at least one triangle vertex. Accordingly it is sufficient to find the six vertices and their according triangles that span the AABB to recreate the bounds. These bounding triangles can be computed in advance. An example is given in Figure 2. Therefore, a node in our NMH is essentially only a small set of triangles contiguously mapped in memory. In cases where less than six triangles span the AABB arbitrary scene triangles can be used for padding, see the green and blue box in Figure 2. The nodes, i.e. the scene triangles, are sorted so that the child nodes can be directly derived from the the parent nodes index, see Section 3.2 for details. Note that the child boxes do not necessarily share a common bounding plane with their parents, as it is the case in standard BVHs. During traversal the AABB is reconstructed

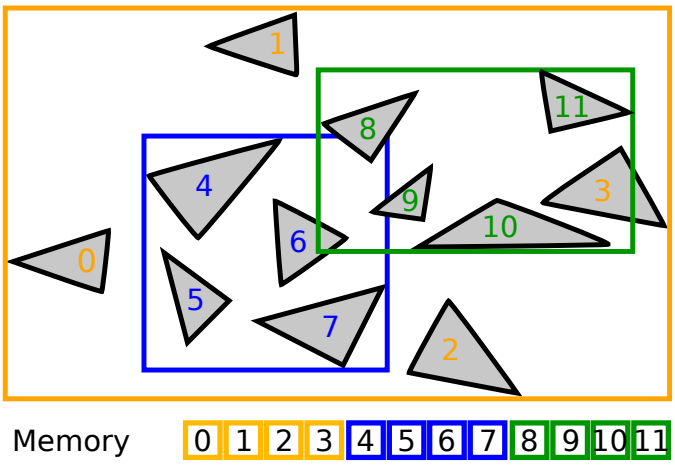

Figure 2: Example partitioning: The bounding boxes of the No-Memory Hierarchy are solely described by the vertices of the triangles.

from these bounding triangles only. If the ray intersects with the box these triangles are tested before the traversal continues with the child nodes. The very basic algorithm is described in Algorithm 1 and will be improved in the next sections.

3.2 Representation

As testing six triangles in each traversal step requires a non-negligible computational overhead we can approximate the AABB with less than six triangles and saving the active ray interval. We only save the two triangles spanning the AABB along one of the coordinate axis. The axis is chosen in a round-robin fashion, i.e. $xyzxyz\ldots$, depending on the depth in the tree. This way we can easily recompute the axis during traversal. Another option would be to choose the longest axis in each node as this can be recomputed from the bounding triangles as well but we did not experience any speed up from this. Such a scheme has problems with elongated triangles but could be more efficient for elongated scenes. In our tests, we experimented with one, two, and six triangles per node but using two has proven to be the most efficient for our scenes.

In order to compute the children for any node, we arrange them in a way that the resulting tree is complete, left-balanced, and in breadth-first order. This allows us to index it like a heap without explicitly saving any pointer or indices. Index zero is the first triangle in the root node. For any other node whose first triangle index is i its children, i.e. the first triangle in the respective nodes, are indexed with $ik + 2m$ 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

```
void traverse(int index, Ray ray,
              float& tHit,
              Triangle* tris,
              int trisPerNode){
  // test for valid node
  if(index >= totalNumberOfTriangles)
    return;

  // compute the AABB from the six triangles
  AABB current box =
    getBox(index, trisPerNode, tris);
  if( !intersect(ray, box) )
    return;

  // if the AABB was hit, test triangles
  for( i=index; i<index+trisPerNode; ++i )
    intersect(ray, tris[i], tHit);

  // continue traversal with child nodes
  int lChild = indexForLeftChild(index);
  int rChild = indexForRightChild(index);
  traverse(lChild, ray, tHit, tris, trisPerNode);
  traverse(rChild, ray, tHit, tris, trisPerNode);
}
```

Algorithm 1: Basic single ray traversal algorithm

node property can be directly derived from the index, i.e. if the child index is larger than the number of triangles in the scene a leaf is reached, this implicit leaf node detection was also used by Cline *et al.* [CSE06] and is a common property of complete trees. As the number of triangles in the scene is not restricted the last non-leaf node might have only one child instead of two, see Figure 3.

Definition: *The NMH is a complete binary tree where each node recursively subdivides the geometry of the scene into two disjoint subsets represented by its two children plus two bounding triangles that span the node. Each node is inherently associated with the index of a coordinate axis defined by its depth in the tree. The bounding triangles represent conservative bounds on the geometric extent of its two children along this axis often also referred to as slabs, Figure 3.*

Incorporating these changes results in the advanced traversal algorithm given in Algorithm 2.

3.3 Hierarchy Creation

The NMH can be created in different ways. We will first describe the simple recursive technique and then introduce a parallel implementation which we use for our GPU version of the NMH. We assume that we have an even number of triangles, otherwise we duplicate the last triangle. The algorithm can be adopted to handle this case as well but it complicates the building procedure and traversal.

3.3.1 Recursive Build

Despite the few kilobytes used by the recursion stack, the algorithm needs nothing more but the list of triangles contained in the scene. Therefore the memory requirement during construction is in the order of $O(\log n)$. Starting with the index 0 and the complete triangle array the algorithm proceeds recursively. In each step the triangles are partitioned in the following way: First, the bounding triangles are detected along the bounding axis starting with the x-axis and are swapped to the beginning of the active triangle array. Next, the bounding axis in incremented and the midpoints of the remaining triangles are sorted along this axis.

To create a complete tree we need to compute the amount of triangles for the left and right partition using Algorithm 3 which results in a left-balanced object median split. The algorithm is then called recursively on the resulting partitions. It should be worth noting that no full sorting is necessary to find the pivot element at which to divide the triangle sets. Instead Hoare's selection algorithm can be used to do the partitioning in expected linear time [Hoa61]. Therefore the theoretical complexity is equal to a spatial median split.

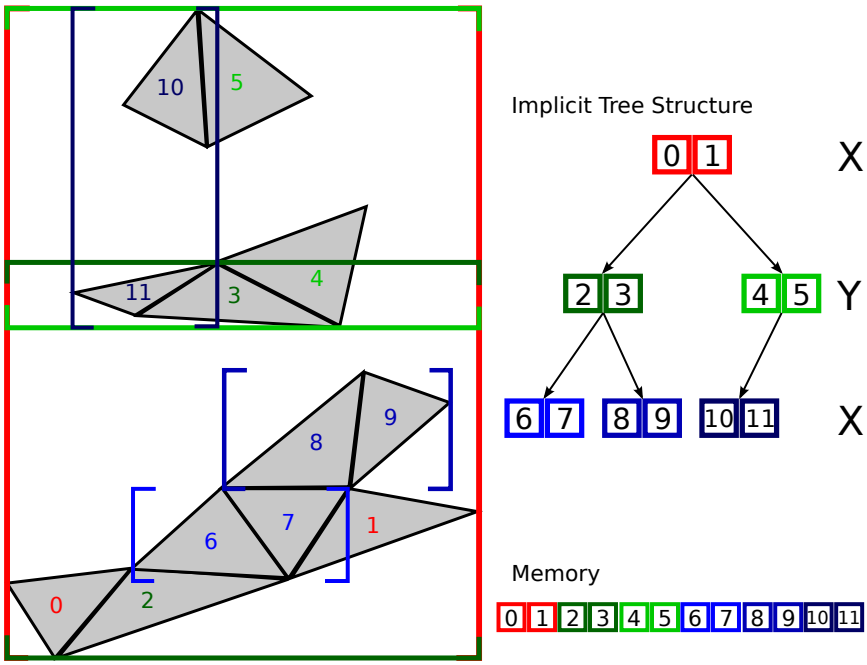

Figure 3: 2D Example of the NMH with three levels: The triangle arrangement implicitly describes a hierarchy. The bounds of each node are spanned by exactly two triangles. Left: Representation of the resulting bounding slabs. The first and third level are bounded along the x-axis, the second level along the y-axis due to the round-robin scheme employed. The triangle index is colored according to the slab the triangle represents. Top right: The scene triangles implicitly represent a complete binary tree of bounding slabs. Bottom right: Representation of the NMH in memory. Note that beyond the triangles, no additional memory is used.

```
void traverse(int index, Ray& ray,
              float tNear, float tFar,
              float& tHit,
              Triangle* tris, char axis){
  // test if node is out of range
  if(index >= totalNumberOfTriangles)
    return;

  // compute the bounding slabs
  // for the current axis
  // from the two bounding triangles
  float minSlab = min(tris[index],
                      tris[index+1],
                      axis);
  float maxSlab = max(tris[index],
                      tris[index+1],
                      axis);

  // intersect ray with slabs
  float tmin, tmax;
  intersect(ray, minSlab, maxSlab,
            tmin, tmax, axis);
  tNear = max(min(tmin, tmax), tNear);
  tFar  = min(max(tmin, tmax), tFar);
  if( !((tNear<=tFar) && (tNear<=tHit))
    return;

  // if the AABB was hit, test triangles
  intersect(ray, tris[index+0], tHit);
  intersect(ray, tris[index+1], tHit);

  // continue traversal with child nodes
  axis = (axis + 1) % 3;
  int lChild = 2*index+2;
  int rChild = lChild+2;
  if( ray.dir[axis] < 0 )
    swap(lChild, rChild);
  traverse(lChild, ray, tNear, tFar, tHit,
           tris, axis);
  traverse(rChild, ray, tNear, tFar, tHit,
           tris, axis);
}
```

Algorithm 2: Advanced single ray traversal algorithm.

The algorithm stops as soon as no more triangles are left to partition. Calling the recursion on the left child first results in a depth-first order which is then converted to a breadth-first order as this simplifies the later traversal.

```
void partitionsize(int& L, int& R,
                   int partitionSize)
{
  nodeCount = partitionSize/2;
  H = floor( log2(nodeCount) );
  s = pow(2,H-1) - 1;
  S = pow(2,H) - 1;
  O = max(0, (nodeCount - 1) - s - S);
  R = 2(s + O);
  L = 2(nodeCount - 1) - R;
}
```

Algorithm 3: Computations for the left-balanced object median split. L and R are the number of necessary triangles for the left and right partition respectively, partitionSize is the number of triangles in the currently investigated partition. H represents the height of the resulting subtree from the current node on, whereas s and S are the minimum and maximum number of possible nodes in the resulting subtrees. As we know that the height of the resulting subtrees may vary at most by one, there must be at least s nodes in the right subtree, plus a potential residual O not fitting into a potentially full left subtree. The partition size L for the left subtree is then simply the number of nodes minus the current node and the number of triangles in the right partition.

3.3.2 Parallel Build

To fully exploit multiprocessor architectures like the GPU we propose a fast parallel construction. The parallel construction is mainly build around the fact that all nodes of one hierarchy level need to partition their triangles along the same axis. Therefore the main ingredients for a fast parallel construction are finding the bounding triangles for each partition and partitioning them along the bounding axis, both of them can be elegantly parallelized.

To determine which partitions are currently active we carry along a split list in which the starting index and the size of each active partition are saved. Each open split S is processed in parallel and writes out new splits. Additionally, we need an index array I which saves the current node index in the hierarchy for each triangle. The memory requirements for the parallel construction is then $O(n)$ as we need one integer per triangle plus the split

list which is of the size $n/4$ at most.

We initially set the first split to index 0 and its size is the number of triangles. The index array I is initialized to 0's and the bounding axis is set to x. The algorithm then loops over all $\lfloor \log_2(n/2) \rfloor + 1$ levels of the hierarchy where n is the total number of triangles. In each loop the triangles are first sorted by a lexicographical sort, i.e. they are first sorted by their vertex positions along the bounding axis and then a stable sort is applied according to the index array I. This is the key element to parallelizing the construction as all nodes are sorted in parallel. After the first sort the triangles are sorted according to their spatial position. The stable sort on the index value arranges triangles of the same node together without changing their relative position. This automatically builds us the desired tree in breadth-first order. Note that I is affected by the first sort as well. The probably fastest way on the GPU is to use sort by key functions, a permutation array, and gather functions to sort the arrays when needed instead of sorting them directly. Also note that for efficiency reasons we do not sort the triangle array but rather an index array pointing to the triangles. The stable sort can be skipped for the first level. If we are at the lowest level of the hierarchy the algorithm finishes at this point.

In the next step, we search in each active split for the bounding triangle of the maximum slab and swap it with the second position in each split. The first is already the correct minimum triangle due to the sorting. To achieve this efficiently, we assign a single thread to each element. The thread checks its split index and uses an atomic compare and assign function to test the triangle at the position $numTriangles - threadID - 1$ with the current maximum triangle of the split. If a better triangle is found the new maximum is saved. As the triangles are already sorted at this stage the proposed reverse assignment of the triangle index leads to less warp serialization conflicts.

For each split the algorithm now updates the index values. The first two triangles of each open split are assigned a value of $idx = assigned + i$ where $assigned$ is the number of already correctly created nodes, so $assigned = 0$ in the first loop. i is the index of the split in the split list. The value of the other triangles in a split are set to $2idx + 1$ for their respective splits, i.e. the index of the left child.

The node is now finished. Therefore the split is removed and two new splits for the child nodes are inserted into the queue if it contains more than two triangles. Let $numSplits$ be the number of open splits, p_i be the starting position of the i'th split and num_i its size. The size $numL_i$ and $numR_i$ for the new splits is computed from num_i as described in Algorithm 3. The position of the new splits resulting from p_i is computed by $pL_i = p_i + 2(numSplits - i)$ for the first split and $pR_i = pL_i + numL_i$ for the second split. The computed positions of the splits are already the positions that are needed after the next lexicographical sort. For efficiency we use

two split lists, one as input and one as output and exchange them in each iteration of the loop.

In the last step of the loop we increment the bounding axis by $axis = (axis + 1)\%3$. After $\lfloor \log_2(n/2) \rfloor + 1$ iterations the algorithm is finished. Please see the additional material for some more details on the parallel construction.

3.4 Two-level NMH

The topology of the resulting implicit tree resembles an object median cut which is known to be inferior to state-of-the-art techniques like a SAH [Hav00] due to the possibly larger overlap and larger bounding boxes. For the basic approach in this work it was important for us to focus on the implicit representation, but by spending some amount of memory the sub-optimal traversal can be redeemed.

The solution is to use a two-level NMH. The top n levels are created using a variation of the SAH and our creation scheme from Section 3.3. In each subdivision step we partition the active triangle array not at the left-balanced object median but at the optimal position according to the SAH. To ensure that enough triangles are left to create a perfect tree, we enforce a minimum of m triangles for the left and right subtree where m is the necessary number of triangles for a perfect subtree of height n minus the current depth. While this constraints the subdivision to some degree it has been shown that a relatively coarse binning during the SAH evaluation does not substantially decrease the overall quality as well which is comparable to our restriction [WMG$^+$07].

With a perfect tree it suffices to save a single integer per leaf node which point towards separate NMH hierarchies in the triangles array, Figure 4. This way we have a near optimal subdivision for the most important top levels while using only a fraction of the memory compared to a full BVH with 32 Byte nodes (2KB compared to 33KB of a classic BVH with $n = 10$ for example).

Such a two-level separation has also been used for BVHs by several researchers to increase rendering performance [LGS$^+$09, SE10, BEM10, PL10, GPM11] but with our two-level representation (NMH-NMH) we only need to save one integer per leaf node of the top-level tree. If we were to save one integer per node we could as well create arbitrary trees, i.e. apply an optimal subdivision in each step. The full evaluation of such an approach is left for future work.

4 Results

We implemented and measured the traversal and construction of our implicit hierarchy for the CPU and the GPU. Additionally, we evaluated the perfor-

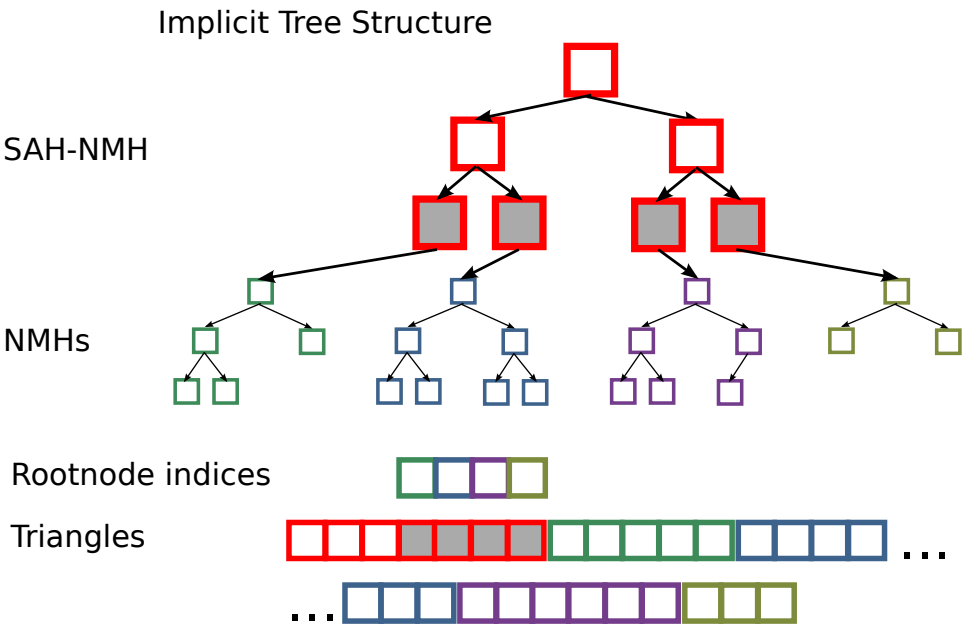

Figure 4: Two-Level NMH (NMH-NMH): During construction the top n levels are created using a variation of the SAH enforcing a perfect tree. For each leaf node an index is saved partitioning the triangles array. Each partition of the array is a NMH by itself.

mance for packet ray tracing using ranged traversal [WBS07]. All statistics were measured on a system with an Intel Core i7-2600 with 3.40 GHz per core, 16 GB Ram and an NVIDIA GeForce GTX 580. Our CPU traversal and construction is currently implemented in a single-threaded manner. As a comparison we implemented an object median split BVH (BVH(OMS)) and a BVH relying on the surface area heuristic and using binning during construction [WBS07] (BVH(SAH)). A resolution of 1024×768 pixels was used for all scenes. In the following we point out interesting properties of our approach.

Implicit representation: Since our hierarchy is implicitly represented by the triangles in the scene we have an optimal, deterministic memory usage. Even with a NMH-NMH or BVH-NMH the requirements are in the range of few kilobytes. Figure 5 gives a theoretical comparison of the memory usage.

Rendering performance: Using no memory at all inevitably comes at the cost of reduced performance during rendering. In our case performance is reduced by roughly a factor of four to eight compared to a highly optimized BVH ray tracer on the CPU, Figure 6. However, this is primarily due to the scene structure not due to the size of the scene. Adopting our Two-level

NMH (NMH-NMH) from Section 3.4, we are able to alleviate this drawback, resulting in a measurable speed-up with just a few kilobytes of additional memory. 0.5kb of memory is needed for eight top-levels, 2kb for ten and 8kb for twelve levels. Using more levels usually degrades performance again as our scheme requires a perfect top-level tree. Therefore, the constraints on the subdivision procedure reduce the overall quality of the tree again after ten to twelve levels in our test scenes.

We also tested another approach by representing the top levels of the tree as a classic BVH created with the SAH and 32 Bytes per node while the lower levels are still created using our NMH (BVH-NMH). Please note that even with twelve top levels more than 99.1% of the nodes in a scene containing roughly one million triangles are still implicitly represented by our NMH, e.g. in the Buddha scene. The complete performance comparisons for our CPU single ray implementation is given in Figure 6. Only primary rays are sent in this test. As expected there is a clear correlation between performance and memory requirements. However, we can also see a drastic improvement if we use the proposed alternative representations for the first eight to twelve levels.

Ray Packets: If a set of coherent rays are given, we can easily incorporate frustum or packet tracing [RSH05, WBS07]. In this case, our data structure benefits even more as the volume elements appear generally larger due to the restrictions on the partitioning and two-plane representations for each node. We only need to track the corresponding bounding box on the stack during traversal. An evaluation of the impact on rendering time is given in Figure 7.

Construction: Our data structure is well suited for dynamic and animated scenes as fast sorting algorithms can be used for construction. Timings and comparisons are given in Figure 8. On the CPU we used the simple recursive creation scheme while the GPU implementation is based on the highly parallel algorithm presented in Section 3.3.2. The fast parallel construction allows to render even complex animated scenes at interactive frame rates. As the hierarchy is created from scratch in each frame the quality does not deteriorate, as it would be possible with simple but fast refitting schemes often used in conjunction with BVHs. The slight performance drop for the GPU construction of the Sponza scene is probably due to the varying triangle size which for some yet unknown reason increases the memory copies.

Hardware Considerations: We implemented our approach on the GPU using CUDA which resulted in a traversal speed-up of approximately factor ten to sixty compared to our CPU single-core implementation, Figure 6 and

9. An evaluation of the GPU performance is given in Figure 9.

We deem our approach also suitable for a dedicated hardware implementation due to several aspects. Firstly, no additional memory is necessary, simplifying the hardware architecture as it does not need to deal with any additional acceleration data structure internally. And secondly, even a complete rebuild is possible in hardware, like an FPGA, as we only need sorting operations which have been shown to work well on dedicated hardware [KW05]. The higher bandwidth required by our current implementation needs further investigation.

Global Illumination: Our NMH is created only once per time step or static scene. As it does not rely on a lazy evaluation scheme the coherency of the rays does not influence the construction time. This might become an important factor in divide-and-conquer schemes. Though a complete NMH is slower than an optimized BVH by a factor of four to eleven, it is only slower by a factor of 1.5 to 2.3 if we use our two-level BVH-NMH, Figure 10. Furthermore, it is actually faster if we compare the total time including construction as well though faster construction schemes for BVHs exist.

Large Objects: Most classical object partitioning and spatial partitioning schemes suffer when encountering a mixture of small and large primitives in a scene, because the overlap of the bounding boxes increases [EG07]. The NMH is also affected but since larger triangles will be kept higher in the hierarchy, as they sooner or later become bounding triangles, they do not affect the quality of the hierarchy to such a large degree [ZU06]. To some extend, they might even speed up the traversal as testing inner triangles allows for earlier exits when tracing shadow rays. Only recently has it been shown that optimizing a BVH for such early shadow exits is beneficial [IH11].

5 Conclusion and Discussion

We proposed a technique for interactive ray tracing without an explicit acceleration data structure. By reordering of the geometry we are able to implicitly represent a hierarchical acceleration data structure by the geometry extends itself. The hierarchy is statically represented by the underlying geometry and needs to be created only once per frame of an animation independend of the viewpoint or lighting conditions. We have also shown that our technique is well suited for GPU implementation.

Limitations and Future Work: We strongly believe that the field of implicit acceleration data structures is a prospering and important area for further research. Due to its inferiority, research on optimizing object median splits for ray tracing has been largely neglected, even though median

Scene	# Tris	BVH	NMH	NMH-NMH	BVH-NMH
Sponza	67,462	4,216	0	2	33
Fairy	174,117	10,882	0	2	33
Dragon	871,306	54,457	0	2	33
Buddha	1,087,716	67,982	0	2	33
Breaking Lion	1,604,054	100,253	0	2	33
Power Plant	12.7M	793,749	0	2	33
St. Matthew	372M	23,250,000	0	2	33

Figure 5: **Memory requirements:** While the memory requirements for classic acceleration data structures like a BVH increases for complex scenes it is constant or even zero with our NMH. For the BVH we assume one triangle per leaf node and 32 Byte of memory per node. In this example we use ten levels for the top-level hierarchy of the NMH-NMH and the BVH-NMH. Numbers are given in kilobytes.

Scene	BVH (OMS)	BVH (SAH)
Sponza	2.461s	0.413s
Fairy	0.645s	0.246s
Dragon	0.295s	0.237s
Buddha	0.150s	0.093s

Scene	CPU NMH	CPU NMH-NMH	CPU BVH-NMH
Sponza	2.510s (×6.1)	1.123s (×2.7 / 10)	0.562s (×1.4 / 12)
Fairy	2.085s (×8.5)	0.894s (×3.6 / 8)	0.354s (×1.4 / 12)
Dragon	1.018s (×4.3)	0.790s (×3.3 / 10)	0.454s (×1.9 / 12)
Buddha	0.329s (×3.5)	0.270s (×2.9 / 12)	0.165s (×1.8 / 12)

Figure 6: **Single-ray CPU rendering performance:** The rendering times are given in seconds, the number in the parentheses are the relative timings factor compared to the BVH built with the SAH and the second number is the number of top-levels used. The BVH-NMH uses complete BVH nodes for its top-levels while the NMH-NMH stores only one offset per top-level leaf node and uses the implicit storage also for the inner nodes of the top-levels.

Scene	BVH (OMS)	BVH (SAH)
Sponza	0.031s	0.019s
Fairy	0.023s	0.020s
Dragon	0.032s	0.028s
Buddha	0.032s	0.028s

Scene	CPU NMH	CPU NMH-NMH	CPU BVH-NMH
Sponza	0.034s (×1.8)	0.029s (×1.5 / 10)	0.026s (×1.4 / 12)
Fairy	0.046s (×2.3)	0.033s (×1.7 / 8)	0.031s (×1.6 / 12)
Dragon	0.068s (×2.4)	0.055s (×2.0 / 12)	0.053s (×1.9 / 12)
Buddha	0.073s (×2.6)	0.069s (×2.5 / 12)	0.065s (×2.3 / 12)

Figure 7: **Packet ray tracing CPU rendering performance:** Timings are given in seconds. The first number in brackets is the relative rendering time compared to the optimized BVH. The second number is the number of top-levels used. We used the packet tracing algorithm described in [WBS07].

Scene	BVH (OMS)	BVH (SAH)
Sponza	0.056s	0.508s
Buddha	1.505s	8.893s
Fairy	0.189s	1.372s
Breaking lion	2.386s	14.003s

Scene	CPU NMH	CPU NMH-NMH	GPU NMH
Sponza	0.061s (×0.120)	0.211s (×0.415 / 10)	0.095s (×0.187)
Buddha	1.798s (×0.202)	4.963s (×0.558 / 10)	0.196s (×0.022)
Fairy	0.229s (×0.167)	0.681s (×0.496 / 10)	0.063s (×0.046)
Breaking lion	2.228s (×0.159)	7.673s (×0.548 / 10)	0.258s (×0.018)

Figure 8: **Construction:** Construction time comparison for building the hierarchy on the CPU and the GPU. The CPU construction is single-threaded. Timings are given in seconds. The first number in brackets is the relative construction time compared to the BVH (SAH). The second number is the number of top-levels used.

Scene	BVH (OMS)	BVH (SAH)
Sponza	0.039s	0.017s
Fairy	0.045s	0.011s
Dragon	0.023s	0.014s
Buddha	0.013s	0.008s

Scene	NMH	NMH-NMH	BVH-NMH
Sponza	0.042s (×2.5)	0.029s (×1.7 / 10)	0.019s (×1.1 / 12)
Fairy	0.052s (×4.7)	0.032s (×2.9 / 8)	0.016s (×1.5 / 12)
Dragon	0.047s (×3.4)	0.045s (×3.2 / 8)	0.027s (×1.9 / 12)
Buddha	0.027s (×3.4)	0.025s (×3.1 / 12)	0.015s (×1.8 / 12)

Figure 9: **Single-ray GPU rendering performance:** Only primary rays are sent. Timings are given in seconds. The first number in brackets is the relative rendering time compared to the optimized BVH. The second number is the number of top-levels used.

splits are still common in related areas such as nearest-neighbor searches [AMN+94]. Using shrinking boxes or sorting along the principal component axis might considerably increase performance of object median splits. Hopefully, we can find ways in the future to implicitly represent other creation schemes, like the SAH, as well.

Another fruitful direction might also be to investigate how to represent spatial data structures implicitly without any lazy evaluation scheme. Our first experiments have shown that it is indeed possible and we are planning to investigate this direction further.

Our algorithm does not necessarily need triangles to represent the slabs. Bounding boxes of separate objects, as they are used for instancing can be used as well without changing the basic algorithm. This way even out-of-core implementations are possible. It would be interesting to see whether this concept can be further generalized to handle free-form and procedural surfaces.

A dedicated hardware implementation of our algorithm is also a promising direction for future research.

Scene	# Bounces	BVH (SAH)	NMH
Sponza (2)	1	0.415s	2.743s (×6.6)
Fairy (2)	1	0.299s	3.091s (×10.3)
Dragon (3)	1	0.282s	1.312s (×4.7)
Buddha (1)	1	0.083s	0.356s (×4.3)
Sponza (2)	2	0.774s	5.721s (×7.4)
Fairy (2)	2	0.455s	5.007s (×11.0)
Dragon (3)	2	0.387s	1.842s (×4.6)
Buddha (1)	2	0.122s	0.515s (×4.2)
Sponza (2)	3	1.116s	8.635s (×7.7)
Fairy (2)	3	0.581s	6.340s (×10.9)
Dragon (3)	3	0.456s	2.130s (×4.7)
Buddha (1)	3	0.155s	0.638s (×4.1)

Scene	# Bounces	NMH-NMH	BVH-NMH
Sponza (2)	1	1.397s (×3.4 / 10)	0.605s (×1.5 / 12)
Fairy (2)	1	1.495s (×5.0 / 8)	0.507s (×1.7 / 12)
Dragon (3)	1	1.063s (×3.7 / 8)	0.635s (×2.3 / 12)
Buddha (1)	1	0.334s (×4.0 / 12)	0.186s (×2.2 / 12)
Sponza (2)	2	2.849s (×3.7 / 10)	1.200s (×1.6 / 12)
Fairy (2)	2	2.227s (×4.9 / 8)	0.800s (×1.8 / 12)
Dragon (3)	2	1.449s (×3.7 / 8)	0.880s (×2.3 / 12)
Buddha (1)	2	0.477s (×3.9 / 12)	0.277s (×2.3 / 12)
Sponza (2)	3	4.237s (×3.8 / 10)	1.776s (×1.6 / 12)
Fairy (2)	3	2.828s (×4.9 / 8)	1.026s (×1.8 / 12)
Dragon (3)	3	1.660s (×3.6 / 8)	1.032s (×2.3 / 12)
Buddha (1)	3	0.585s (×3.8 / 12)	0.355s (×2.3 / 12)

Figure 10: **Global Illumination:** The table shows a rendering performance comparison for path tracing on the GPU using four samples per pixel and multiple number of bounces. The number in parentheses in the scene row is the number of light sources in the scene.

References

[AMN+94] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[BEM10] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. The Minimal Bounding Volume Hierarchy. In *Proc. of Vision, Modeling, and Visualization (VMV'10)*, pages 227–234, 11 2010.

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

[EG07] Manfred Ernst and Gunther Greiner. Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 73–78, 2007.

[EWM08] Martin Eisemann, Christian Woizischke, and Marcus Magnor. Ray Tracing with the Single-Slab Hierarchy. In *Proceedings of Vision, Modeling, and Visualization (VMV'08)*, pages 373–381, 2008.

[Gar08] K. Garanzha. Efficient clustered bvh update algorithm for highly-dynamic models. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 123 – 130, 2008.

[GPM11] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and Faster HLBVH with Work Queues. In *Proceedings of High Performance Graphics*, 2011.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HHHPS06] 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*, pages 1–10, 2006.

[Hoa61] C.A.R. Hoare. Find (algorithm 65). *Communications of ACM*, 4:321–322, 1961.

[IH11] T. Ize and C.D. Hansen. RTSAH Traversal Order for Occlusion Rays. In *Computer Graphics Forum (Proceedings of Eurographics 2011)*, volume 30, 2011.

[KK86] 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, 1986.

[KW05] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[KW09] A. Keller and C. Wächter. Efficient ray tracing without acceleration data structure. *U.S. Patent Applications Publication No. US 2009/0225081 A1*, 2009.

[KW11] A. Keller and C. Wächter. Efficient ray tracing without acceleration data structure. *Poster at High Performance Graphics Conference*, 2011.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[LYTM06] 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, 2006.

[Mor11] B. Mora. Naive ray tracing: A divide-and-conquer approach. *ACM Transactions on Graphics*, 2011. Accepted for publication.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., 2nd edition, 2010.

[PL10] J. Pantaleoni and D. Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95, 2010.

[RSH05] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transactions on Graphics*, 24(3):1176–1185, 2005.

[RW80] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Pro-*

ceedings of the 7th annual conference on Computer graphics and interactive techniques, pages 110–116, 1980.

[SE10] Benjamin Segovia and Manfred Ernst. Memory efficient ray tracing with hierarchical mesh quantization. In *Graphics Interface 2010*, pages 153–160, 2010.

[Sol] Solid Angle. Arnold renderer. http://solidangle.com.

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 95–106, 2004.

[vdB97] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphic Tools*, 2(4):1–13, 1997.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–28, 2007.

[WK06] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Eurographics Symposium on Rendering*, pages 139–149, 2006.

[WMG+07] 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 *Eurographics State of the Art Reports*, pages 89–116, 2007.

[WMS06] 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.

[WPS+03] 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.

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