Vision, Modeling, and Visualization (2018)
F. Beck, C. Dachsbacher, and F. Sadlo (Eds.)

Fast and Dynamic Construction of Bounding Volume Hierarchies based on Loose Octrees

Feng Gu Johannes Jendersie Thorsten Grosch

TU Clausthal, Germany

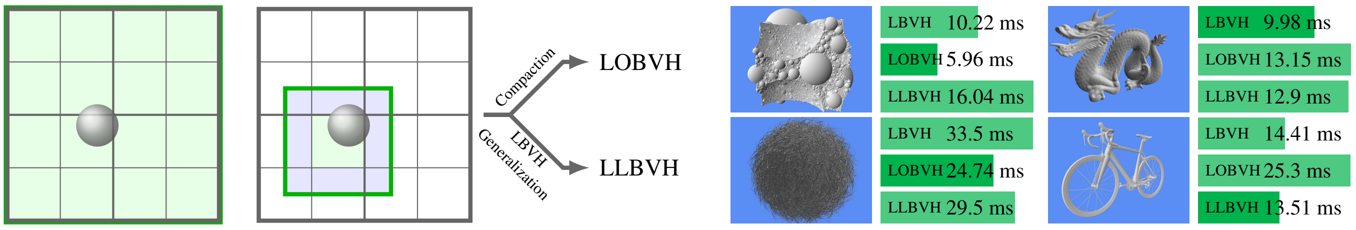


Figure 1: *We propose two new methods, LOBVH and LLBVH, based on inserting an object into a loose octree. For scenarios with high occlusion this results in a faster build+trace time (right) than previous methods.*

Abstract

Many fast methods for constructing BVHs on the GPU only use the centroids of primitive bounding boxes and ignore the actual spatial extent of each primitive. We present a fast new way and a memory-efficient implementation to build a BVH from a loose octree for real-time ray tracing on fully dynamic scenes. Our memory-efficient implementation is an in-place method and generalizes the state-of-the-art parallel construction for LBVH to build the BVH from nodes of different levels.

1. Introduction

With increasing hardware capabilities ray-tracing becomes more and more attractive for different simulation and visualization purposes. One of the key points for a fast ray-tracing engine is the acceleration data structure which is used to find ray-intersections. Bounding Volume Hierarchies (BVH) are among the most effective ones when used on a GPU [VHB14]. They use a hierarchy of simple primitives, e.g. axis aligned bounding boxes, to exclude large parts of a scene in a ray intersection test.

Usually, the build time for those structures is considerably high and increases with the target quality. Therefore, acceleration structures are only built once (preprocessing) and used multiple times. However, highly dynamic scenes need to rebuild the acceleration structure more frequently, especially for simulations in which many moving objects may invalidate the entire topology in each iteration. Thus, a static structure will degenerate and lose its acceleration properties after a short period of time.

An example, for which we try to solve the problem, is the simulation of heterogeneous particle mixtures like concrete. Often such simulations are done with periodic boundaries, causing large changes in the BVH when objects wrap around.

We propose two fast BVH build methods which are based on a loose octree. A loose octree is an octree where each node has an enlarged boundary [Ulr00]. This leads to an overlap of nodes, like in a BVH, but has the advantage of a deterministic indexing scheme. This allows to add geometry very fast, but shows a bad tracing performance if used as octree directly. We propose two different build algorithms to produce a higher quality BVH from an initial loose octree. Our first method (Loose Octree Bounding Volume Hierarchies (LOBVH)) starts with a dense octree where all octree nodes are allocated within memory and results in an 8-ary tree. This process leads to the best results for our application of particle mixtures with completely dynamic geometry. However, it requires huge amounts of temporary memory for deeper hierarchies and performs bad on triangle scenes. We then develop a memory efficient algorithm (Loose Octree Linear Bounding Volume Hierarchies (LLBVH)) which is based on the same initial positioning of primitives regarding the loose octree. It can be seen as a generalization of Linear Bounding Volume Hierarchies (LBVH) from Karras [Kar12], where we allow storing geometry on each level.

Finally, we compare our methods to the LBVH from Karras [Kar12] with respect to simulation output (spherical primitives) and triangle scenes. For our application the LOBVH shows the shortest iteration times, where one iteration consists of building the Bounding Volume Hierarchies (BVH) and tracing. In a parallel CUDA im-

DOI: 10.2312/vmv.20181257

plementation we achieve a full rebuild of the hierarchy in 4.5 ms for 1 million primitives on a GTX 1080. For triangle scenes LOBVH is much slower in most cases. With LLBVH we achieve a similar tracing performance as LBVH [Kar12], but have a slightly higher construction time with the exception of a few scenes.

2. Related Work

In the acceleration of ray-tracing operations, many kinds of search data structures are used. The most used ones are kD-trees [Kap85] and BVHs. On GPU a BVH is typically faster (especially for primary rays), has lower memory footprint, as shown by Vinkler et al. [VHB14], and works well with packet tracing techniques [WK07].

The core of each BVH is the *Surface Area Heuristic (SAH)* which was first described by Goldsmith and Salomon [GS87] and formalized later by MacDonald et al. [MB90]. It measures the expected intersection cost for an object hierarchy and thus defines a target function to optimize a BVH.

One of the best building algorithms is the Split Bounding Volume Hierarchies (SBVH) [SFD09] which splits references to large triangles. This overcomes an issue where the quality of the hierarchy decreases due to triangles which are far larger than most others. Its build time spans several seconds, which makes it a good choice for static scenes.

Another group of algorithms targets the fast parallel construction of the hierarchy. Lauterbach et al. [LGS*09] introduced the LBVH. It starts by sorting the centroids of the geometry along a space-filling Morton-curve, followed by a hierarchical clustering process. For the hierarchy construction they proposed to split either by using the bits from the Morton-code, or by applying an SAH build strategy. This already parallel build method was further improved by Karras [Kar12] by constructing the entire hierarchy in a single pass – instead per level. Apetrei [Ape14] also improved LBVH by performing both the tree construction and the bounding box calculation in a single bottom- up traversal.

A different modification, called Hierarchical Linear Bounding Volume Hierarchies (HLBVH), also optimized multiple levels at the same time by building parts of the hierarchy using the LBVH-middle-split approach and applying an SAH construction on top of the clusters [PL10]. Garanzha et al. [GPM11] improved the HLBVH algorithm build time and reduced its memory consumption. In a similar fashion Treelet Bounding Volume Hierarchies (TRBVH) [KA13] optimizes parts of the tree (the treelets) with respect to SAH as a post-process. The idea was then adopted and improved by Domingues and Pedrini [DP15] using bottom-up agglomerative clustering inside the treelets instead of the full search for an optimal solution. All above methods are Morton-order based and can be further improved by extended Morton codes introduced by [VBH17], which increases the coherency for BVHs by encoding the size of objects with one bit in every 4 or 7 bits and using adaptive axis order and count until the subdivided volume becomes close to a cube. The enhancement in [VBH17] by using adaptive axis order and count requires precomputed global axis-aligned bounding box (AABB), which restrains its applications in fully dynamic scenes. Our methods inherently implies encoding size information every 3 bits in Morton code, which does not consume extra

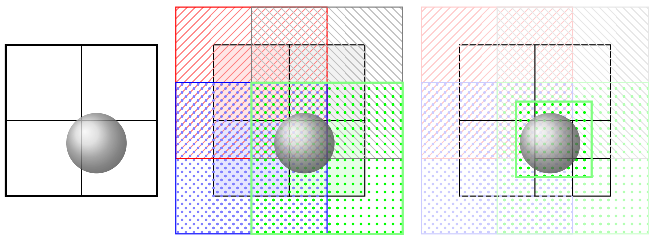

(a) Regular quadtree
(b) Enlarged bounds shown for level 1
(c) Possible placement on level 2

Figure 2: *Example of a loose quadtree with $a = 2$ compared to a regular quadtree. The sphere would be added to the root in (a) and to the green node in (b) and (c). While the computed level for the sphere is 1 (b) it would also fit into a level 2 node as shown in (c).*

bits in 32/64 bits Morton code. With a few extra lines of code in the initialization, our LLBVH algorithm can also encode the size information every two levels (6 bits) up to a specific level. However, this did not improve the total build + trace time and the memory footprint for our test scenes on average.

In comparison to the above implementations our first algorithm has two major differences: First, we add data to different levels of the initial tree, instead of inserting to the leaf level only. Second, our final tree is still an octree instead of a binary tree, flattening the hierarchy. We show that for certain applications this results in a better total iteration time than the aforementioned methods. Our second algorithm keeps the order formed by the loose octree and reduces the memory footprint.

A different category for fast-to-build acceleration structures is based on grids. Kalojanov and Slusallek [KS09] filled a uniform grid with a parallel algorithm leading to a very fast construction. On the other hand, uniform grids require a lot of space and are often slower than HLBVHs [KS09]. Irregular grids [PGKS17] solve both problems and even outperform SBVH with respect to tracing performance. However, their construction algorithm is still slower than the LBVH derivatives and has a slower total iteration time.

3. Building BVHs with a Loose Octree

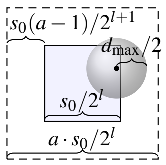
Our algorithms are based on the concept of loose octrees which is introduced in this section, first. Then, we go into the details of our build algorithm and show how to implement it in parallel.

3.1. The Loose Octree

In a D-dimensional octree each node is a box which contains 2^D equally sized children – one split in each dimension. The 2D variant is also known as a quadtree (shown in Figure 2 left). In a dense octree, having the same tree depth in any branch, a point can be localized distinctly in a single cell and for that cell the index can be computed in a closed form.

However, inserting a shape like a sphere or a triangle, leads to an ambiguity. On a fine level it may overlap multiple nodes. Alternatively, the shape could be added on a coarse level where it does not overlap any edge. With this strategy, a very small shape might be added to the root node because it overlaps an edge in its center, leading to bad acceleration properties.

A loose octree scales the bounding box of each node by a factor of a. If the edge length of the root node on level $l = 0$ is s_0 then $a \cdot s_0/2^l$ is the edge length of the enlarged bounding volumes for all nodes at level l.



For a loose octree it is possible to define an insertion position for a primitive o which is only based on its AABB. An AABB whose centroid is within a cell of the octree cannot leave the cell's boundary, if $d_{\max} \leq s_0 \cdot (a-1)/2^l$, where $d_{\max}$ is the AABB's longest side. Thus, given the size measure d for a primitive, its level can be computed as

$$l(d) = \left\lfloor \log_2 \frac{s_0 \cdot (a-1)}{d} \right\rfloor. \quad (1)$$

With $d = d_{\max}$ it is possible that the primitive also fits into one of the extended child nodes (see Figure 2 (c)). However, with $a \geq 2$ it is never possible to have a deviation of more than one level using Eq. (1). To find the smallest possible cell for insertion, it is only necessary to test if the primitive fits into the cell at level $l(d)+1$. For level $l(d)$ we have a guarantee that the primitive fits without further testing.

While using the largest side of the AABB, $d = d_{\max}$, gives a guarantee that the primitive fits into the node, we used its minimum side $d_{\min}$ instead. This moves triangles further down in the hierarchy and provides better tracing performance. We have also tried to use the average side length and the cubic root of the AABB's volume as the value of d for triangles, which all provide inferior performance. For spheres d is simply their diameter.

3.2. Dense Build Algorithm (LOBVH)

In this section, we describe how to build a BVH from a densly stored loose octree with maximum depth L from a given set of primitives.

The octree data structure consists of four arrays to depict the properties of each node: one for the bounding box, one for the first-child pointer, one for the next pointer and one for the pair of pointers to the first and last primitive of the current node.

The algorithm steps are:

Initialization For each primitive its level and its Morton code are computed and all elements are sorted according to that code.
Pointer Setup References of a densely stored octree to the data are established (visualized in Figure 3 (a)).
BVH Pointer Setup The pointer arrays first-child and next are filled (Figure 3 (b)).
Compaction All vacant nodes are removed (Figure 3 (c)).

3.2.1. Initialization

First of all, levels for each primitive are calculated with equation (1). The corresponding key k for a primitive i in level l is then computed as

$$k_i = \Delta(l) + m(c_i, l) \quad (2)$$

where c_i is the centroid of the AABB, $m(c_i, l)$ is the Morton code of c_i with l bits used for each dimension and $\Delta(l)$ defined as

$$\Delta(l) = \frac{8^l - 1}{7}$$

counts the number of octree nodes from level 0 to level $l-1$. The key k_i specifies the octree node that owns primitive i. We use k to denote indices of nodes itself. Also, we set $m(k) = m(c_i, l)$ as abbreviation for the node's Morton code.

Using Eq. (2) we can sort all N primitives. After that, all primitives which fall into the same node are adjacent to each other.

3.2.2. Pointer Setup

In this step, the first and the last primitive for each non-empty octree node will be initialized in an array of size $\Delta(L+1)$. By comparing each pair of adjacent keys, the first and the last element of a node can be identified.

Whenever the key between two primitives i and $i+1$ changes, the last-pointer $\mathsf{E}(k_i)$ points to i and the first-pointer $\mathsf{S}(k_{i+1})$ points to $i+1$ respectively.

3.2.3. BVH Pointer Setup

Denote the first-child pointer of node k on level l as $\mathsf{F(k)}$ and its next pointer as $\mathsf{N(k)}$. $\mathsf{N(k)}$ points to the next node of k in a breadth-first search (BFS) of the octree, thus $\mathsf{N(k)} = k+1$ if k is not the last node among its siblings $((m(k)\&7) = 7)$, otherwise we repeat $m(k) = m(k) \gg 3$ until $(m(k)\&7) \neq 7$. The value of $\mathsf{F(k)}$ can be simply initialized as

$$\mathsf{F(k)} = \Delta(l+1) + (m(k) \ll 3).$$

3.2.4. Compaction

For compaction we need the following steps:

1. Adjust $\mathsf{F(k)}$.
Since usually many of the octree nodes are empty, the first-child pointer of node k should be adjusted so that $\mathsf{F(k)}$ points to the first non-vacant of this child-nodes. Here we define a node as *vacant* if it has no primitive and all of its descendants are empty. Thus we can mark all vacant nodes with a bottom-up approach and adjust $\mathsf{F(k)}$ for non-vacant nodes so that it points to the first non-vacant child of the current node, if such a node exists. Due to this bottom-up nature, the bounding box for each node is calculated at this phase.
2. Adjust $\mathsf{N(k)}$.
After marking all vacant nodes, $\mathsf{N(k)}$ can be adjusted as repeating $\mathsf{N(k)} = \mathsf{N(N(k))}$ as long as node $\mathsf{N(k)}$ is vacant.
3. Compaction.
To improve the data locality for better ray tracing performance, all vacant octree nodes should be omitted through a scan operation, since they are all marked in the preceding step.

It may appear that the pointer setup could be solved on the primitive data array only, using binary searches. This is not the case, because the internal nodes without geometry would be missed by this process. The blue boxes in Figure 3 (c) would be missing if not using the full process. The important difference is that an empty

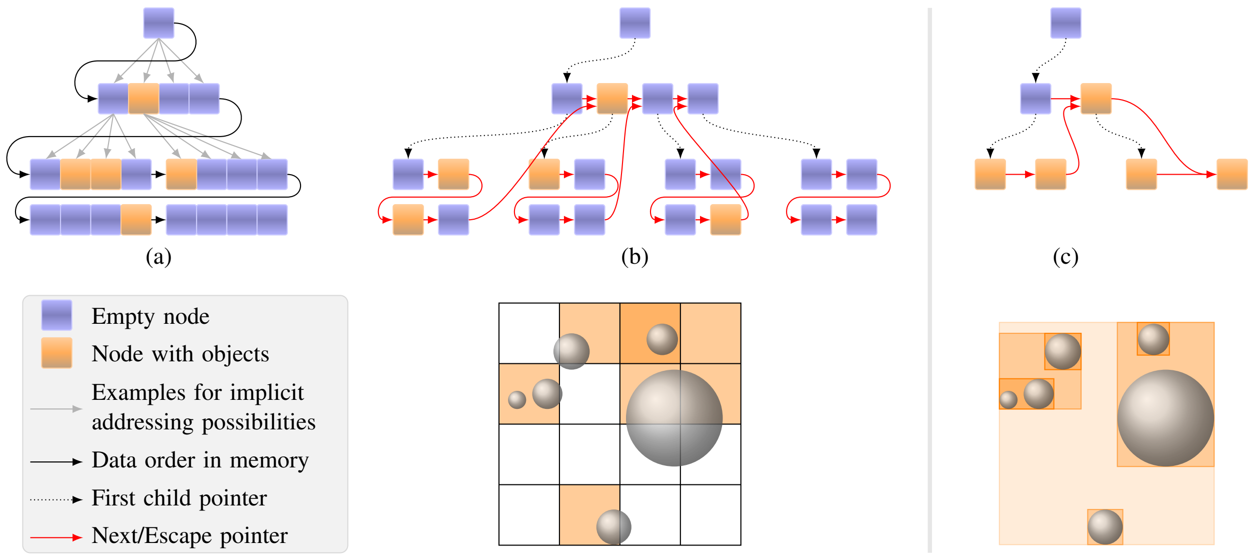


Figure 3: *LOBVH: The upper row shows the tree structure after insertion into loose quadtree (a), after explicit pointer setup (b) and after compaction (c). Below, the scene and the geometrical structure of the respective trees are shown. The spheres overlap the associated (quadtree) nodes' boundaries due to the loose property (center). After compaction we have a BVH with tightly fitting bounding boxes (right).*

node is not vacant if any of its descendants is non-empty. Artificially inserting the missing internal nodes can be implemented, but results in a slow multi-pass solution. The pointer setup for stackless traversal of the BVH is now completed.

3.2.5. Parallel Realization of the Algorithm

The process described so far can easily be parallelized. In the steps from Section 3.2.1 and 3.2.2 each of the N primitives is associated with one thread. For the steps in Section 3.2.3 and 3.2.4, each node is assigned to a thread. Note that the adjustment for $\mathsf{N}(\mathsf{k})$ does not need to be changed for parallelization, since the side-effects of other threads adjusting their $\mathsf{N}(\mathsf{k})$ only accelerate the whole adjustment process. For scan and radix sort processes, we apply the CUB library 1.8.0 [Nvi18].

3.3. Memory Efficient Implementation (LLBVH)

The approach in the last section requires temporary memory proportional to the total number of nodes of an octree with maximum depth L, which is $\Delta(L+1)$. For example with $L = 10$ (30 bits for the lowest level), $\Delta(L+1) \approx 1.227 \times 10^9$. Such a huge memory requirement also increases the build time for large L. Thus in this section, we present a memory efficient implementation to build a specific BVH based on the octree.

3.3.1. From Loose Octree to Binary Tree

To maximize the parallelism for constructing the ray tracing acceleration structure, we first transform the loose octree into a binary tree. This process is straightforward because each octree node with level l matches a binary tree node with level $3l$ and between each adjacent levels in the octree, two extra levels are inserted to ensure that each internal node has two children, as shown in Figure 4 (b). Denoting L' as the maximum depth of the normal binary tree, we have $L' = 3L$.

3.3.2. Extending the Binary Tree

As shown in Figure 4, the normal binary tree built from the octree can have internal nodes containing their own primitives. Such a structure has two disadvantages for constructing an accelerating structure based on a loose octree:

1. To minimize the memory requirement, it is important to know the number of non-vacant nodes a priori. However, it is not easy to get such a number quickly for real time applications. Besides, even knowing this number, after removing all vacant nodes, redundant nodes such as a chain of non-vacant empty nodes still need to be considered, if we want to minimize the memory requirement.
2. Since each node has only one bounding box, as long as a ray-bounding box test succeeds, all primitives (if they exist) of an internal node need to be tested for possible intersections, even if it turns out that the ray only intersects with the primitives from its descendant nodes.

Therefore, we transform the current binary tree to an extended binary tree by inserting two empty nodes for each non-empty internal node in the original binary tree: the original node holds its own primitives as a leaf node with a separate AABB and one empty node holds its descendant nodes as its sibling, as shown in Figure 4 (c). Now, all non-empty nodes of the original octree/binary tree are leaf nodes and the number of required internal nodes is the number of all leaf nodes minus one. Besides, all primitives belonging to an internal node have their own bounding box and we are free to decide whether to test the descendant nodes or the primitives of the new leaf node first.

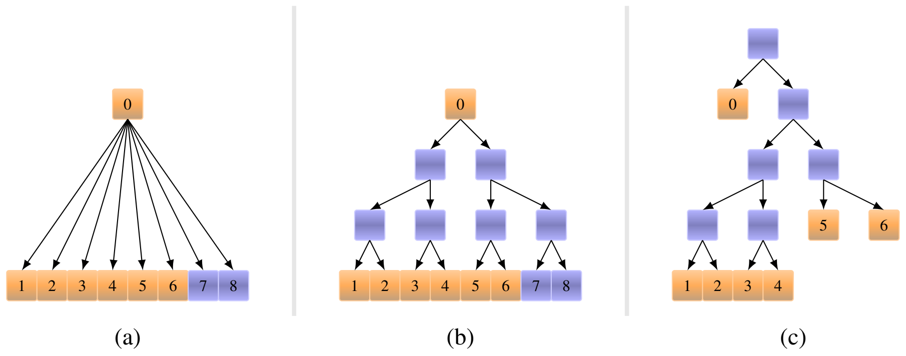


Figure 4: *LLBVH: (a) shows the octree, (b) the implicit binary tree and (c) the extended binary tree in which internal nodes with data are split into one data and two internal nodes. Note that in (c), the two empty leaf nodes 7 and 8 are omitted.*

3.3.3. Building the Extended Binary Tree in Parallel

Basically, we use the build method of Karras [Kar12], but need to do some modifications such that the algorithm works with primitives in internal nodes. First, we introduce Karras' method before explaining the necessary modifications.

Suppose we have n nodes and each has its unique Morton code. Then the common prefix is used to group the nodes. After sorting all nodes based on their Morton codes in ascending order, we can construct $n-1$ internal nodes to build the binary tree with each internal node covering a range of nodes in $[0, n-1]$ by utilizing the following fact. For simplicity, we will use I_i to refer to internal nodes explicitly and E_i for explicitly denoting leave nodes. For arbitrary nodes we keep using its index i only.

Let $\delta(i, j)$[†] denote the length of the common prefix between the Morton codes m_i and m_j, and $[a, b]$ is an arbitrary interval of Morton codes. Then all nodes in a sorted range $[a, b]$ share a common prefix $\delta(a, b)$ corresponding to the common ancestor node. Therefore, $\delta(a', b') \geq \delta(a, b)$ holds for any $a', b' \in [a, b]$. In each internal node the children are partitioned by the first differing bit, i.e. the bit following after the prefix $\delta(a, b)$. The split position $\gamma \in [a, b-1]$ satisfies $\delta(\gamma, \gamma+1) = \delta(a, b)$. This also implies that $\delta(\gamma, \gamma-1) > \delta(a, b)$ if $\gamma - 1 \in [a, b]$ and $\delta(\gamma+1, \gamma+2) > \delta(a, b)$, if $\gamma+2 \in [a, b]$. Thus, the left child of $I_{[a,b]}$ covers $[a, \gamma]$ and the right child covers $[\gamma+1, b]$.

Therefore, for each $i \in [0, n-2]$, if $\delta(i, i+1) < \delta(i, i-1)$, we take $i = \gamma$ and search on the left side of i for the other end a of $I_{[a,b]}$. Similarly, if $\delta(i, i+1) > \delta(i, i-1)$, we take $i = \gamma+1$ and search for b on the right side. The search direction can be encoded as $d = -1$ for the left, $d = 1$ for the right direction and is calculated with

$$d = \operatorname{sign}(\delta(i, i+1) - \delta(i, i-1)). \tag{3}$$

It will not happen that $d = 0$ due to the unique Morton code requirement.

Take the search for a as example, we must have $\delta(i, a) > \delta(i, i+1)$ since $I_{[a,i]}$ is covered by $I_{[a,b]}$ and $\delta(i, a-1) \leq \delta(i, i+1)$, otherwise $[a-1, b]$ shall be covered by $I_{[a,b]}$. With the help of this constraint, we can use a binary search to find a or b. Once we have $[a, i]$ or $[i, b]$, we can use another binary search to find the current node's split position. Note that with this approach, we cannot find an internal node twice, since only one side of each range is used as split position for its parent, and we have $n-1$ nodes in $[0, n-2]$ and the number of internal nodes is also $n-1$ —- all internal nodes will be found.

Now, let us extend this concept for nodes from different levels. To simplify the comparison of nodes, we use the common prefix based on adjusted Morton codes $m'(i) = m(i) \ll (L' - l_i)$, i.e. the original Morton codes with 0s appended, so that all Morton codes from different levels start from the same bit. For each adjusted Morton code of level l, only the first l bits are valid, which implies that the common prefix $\eta(i, j)$ of nodes i, j should not be larger than $\min\{l_i, l_j\}$. Hence, it can be computed by $\eta(i, j) = \min\{l_i, l_j, \delta(i, j)\}$.

Initialization

We start with the computation of the level, the adjusted Morton code and the depth-first search (DFS) key k_{DFS} for each primitive in parallel. The DFS key identifies the order of the primitive's nodes when performing a DFS from the root of the normal binary tree. For a primitive from node i of level l in the normal binary tree, its DFS key can be calculated as

$$k_{\text{DFS, i}} = 2m'(i) - \operatorname{popc}(m'(i)) - l \tag{4}$$

where $\operatorname{popc}(m'(i))$ counts the number of 1-bits in $m'(i)$. [‡] For the derivation of equation (4), see the supplement.

The resulting arrays for the level and the adjusted Morton code are sorted with respect to k_{DFS}. After the sorting, all m's are in ascending order, too. Since all primitives which fall into the same node have the same Morton codes, they will be adjacent after sorting. With adjusted Morton codes it is possible that a parent and a

[†] $\delta(i, j)$ can be efficiently computed by counting the leading zeros of $m_i \oplus m_j$ using the CUDA __clz() intrinsic.

[‡] CUDA provides the functionality with the *__popc*() compiler intrinsic.

child share the same code. Only by sorting with respect to k_{DFS} the correct order between parents and children can be guaranteed.

For the new array, we can apply a compaction, removing redundant entries with equal k_{DFS}. The final array contains m' and l for all non-empty nodes in the normal binary tree, which are equivalent to the leaf nodes in the extended binary tree.

Construction

The construction algorithm summarized in Algorithm 1 takes the new array from the initialization as input. Since we used the DFS order in the sorting process we can simplify the computation of the common prefix to $\theta(i,j) = \min\{l_j, \delta(i,j)\}$ instead of $\eta(i,j)$ as shown in the supplement.

Algorithm 1: Construction of a LLBVH binary tree. We define $\theta(i,j) = -1$ if $j < 0$ and $\theta(i,j) = -2$ if $j > n-1$, for simplicity. See section 3.3.3 for details.

```
1  for each internal node $i \in [0, n-2]$ in parallel do
       // Determine direction of the range
2      $d \leftarrow \begin{cases} +1 & \theta(i,i+1) \geq \theta(i,i-1) \\ -1 & \text{otherwise} \end{cases}$
       // Compute upper bound for the length of the range
3      $\theta_{\min} \leftarrow \theta(i,i-d) - \max(d,0)$
4      $\sigma_{\max} \leftarrow 2$
5      while $\theta(i, i+\sigma_{\max} \cdot d) > \theta_{\min}$ do
6          $\sigma_{\max} \leftarrow \sigma_{\max} \cdot 2$
       // Find the other end using binary search
7      $\sigma \leftarrow 0$
8      for $t \leftarrow \sigma_{\max}/2, \sigma_{\max}/4, \ldots, 1$ do
9          if $\theta(i, i+(\sigma+t)\cdot d) > \theta_{\min}$ then
10             $\sigma \leftarrow \sigma + t$
11     $j \leftarrow i + \sigma \cdot d$
       // Find the split position using binary search
12     $\theta_{\text{node}} \leftarrow \theta(i,j)$
13     if $l_{\min(i,j)} = \theta_{\text{node}}$ then
14         $\gamma \leftarrow \min(i,j)$
15     else
16         $s \leftarrow 0$
17         for $t \leftarrow \lceil l/2 \rceil, \lceil l/4 \rceil, \ldots, 1$ do
18             if $\theta(i, i+(s+t)\cdot d) > \theta_{\text{node}}$ then
19                 $s \leftarrow s+t$ ;
20         $\gamma \leftarrow i + s \cdot d + \min(d,0)$
       // Output child pointers
21     $left \leftarrow$ if $\min(i,j) = \gamma$ then $E_\gamma$ else $I_\gamma$
22     $right \leftarrow$ if $\max(i,j) = \gamma+1$ then $E_{\gamma+1}$ else $I_{\gamma+1}$
23     Output:
24     $I_i \leftarrow (left, right)$
```

As stated before, we insert two empty internal nodes if one non-empty node a has descendants. One is the former internal node with all descendants of a and one holds a as left leaf child and the first as right child. Let $a = \min\{i,j\}$ and $b = \max\{i,j\}$. Each time the range $[a,b]$ is found, we check if a is the ancestor for all nodes in $[a+1,b]$, i.e. level l_a of node a is equal to $\theta(a,b)$ (line 13), and set the split position to a. If the level is not equal to the prefix length, we search the split position $\gamma \in [a,b]$ as described before (lines 16–20). In figure 4 only node 0 fulfills the equality (leaf nodes on a higher level). The handling in line 14 and lines 21–22 then produces the extended tree node, shown as root in Figure 4 (c).

Now the task for $a+1$ (processed by another thread) is first to find the other end b and then its own split position, which requires $d = 1$. This can be calculated with equation (3) except that if $a+1$ has no descendants itself, d will become zero in which case we define d as one as required (line 2). The case where a is an internal non-empty node and $a+1$ has no descendants is the only case for which d becomes zero.

Another problem to address here, is that $a+1$ needs to find the position of the last descendant of a, which is b. This cannot be simply done by requiring $\theta(a+1,b) > \theta_{\min}$ as in line 5 or line 9, since $\theta_{\min} = \theta(a+1,a)$ and if b is not a descendant of $a+1$, we have $\theta(a+1,b) = \theta_{\min}$. A solution is to change the condition from *if*$(\theta(a+1,b) > \theta_{\min})$ to *if*$[(\theta(a+1,b) > \theta_{\min})$ or $(d = 1$ and $\theta(a+1,b) = \theta_{min})]$ so that b can be found by $a+1$ as the other end of its range. However, this imposes an extra comparison for each iteration. Therefore, we adjust the value of $\theta_{\min}$ in line 3 for $d = 1$, instead. Note that this change will not affect other nodes to find their ends since $\theta(i,i-1) = \theta(i,j)$ with $j > i$ only happens when i, j are descendants of $i-1$ but j is not the descendant of i, which is exactly the case we discussed here. A proof can be found in the supplement.

After the BVH based on the loose octree has been constructed, the bounding boxes can be calculated as described in [Kar12] where the paths from leaf nodes to the root are processed in parallel by using atomic counters.

3.4. Optimizing the Trees for the View Direction

In Morton code-based algorithms, we can improve the order of nodes by a simple modification of the code. The goal is to test nodes, which are closer to the ray origin, first, to speed up first-hit intersection tests. Since we rebuild the tree in each frame, we can presort the geometry according to the global view direction. Therefore, we modify the Morton codes before sorting in two ways:

1. Each bit is negated, dependent on the sign of the related coordinate in the view direction. Usually, the Morton code enumerates the primitives in ascending order in each dimension. If the view direction is negative, toggling the related bits produces a descending order, which means that primitives are always ordered in the view direction afterwards.
2. In the 3D Morton code a triple of bits is associated with the dimensions x,y and z (in that order). We change the order according to the absolute values of the view direction to test the dimension with the largest changes first. For example if y is the fastest changing dimension, y will be set as the first bit and consequently the split in y direction will occur first in the binary tree.

This optimization improves the performance of our stack free traversal algorithm of the octree resulting from the LOBVH from Section 3.2 by an order of magnitude. For our memory efficient algorithm LLBVH from Section 3.3 and LBVH we use a stack-based traversal of the binary trees. Since this algorithm already expands the closer node first, the effect of the above modification is almost non-existent.

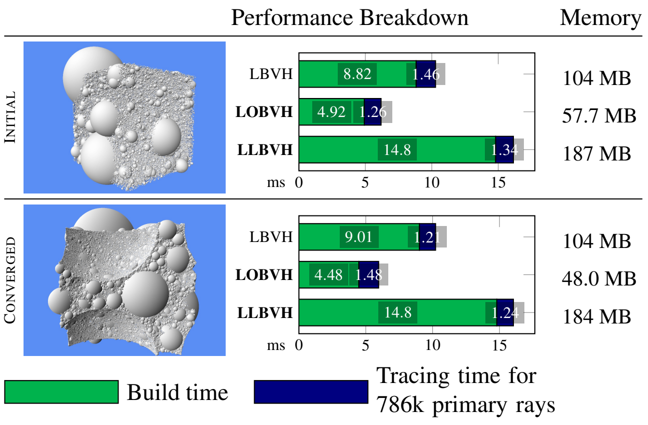


Figure 5: *Performance comparison for particle mixture scenes of 1‘000‘000 spheres.*

4. Evaluation

We proposed two different build strategies based on the concept of loose octrees. The first (LOBVH described in section 3.2) assigns geometry directly to a large array, followed by a compaction step. In LOBVH the fast growing array-size for larger octrees limits the number of possible levels. Our second method (LLBVH described in section 3.3) builds a binary tree with a modified version of the LBVH algorithm [Kar12]. The difference is that we still add data to different levels of the tree based on the loose octree idea. The two binary tree builders LBVH and LLBVH are both limited by the Morton code length which we set to 30 bit in all benchmarks. This produces binary trees with at most 30 levels which would be equivalent to a 10 level octree. For the octrees of the LOBVH builder we use a different number of levels for each scene and always enable the Morton code modification as described in Section 3.4.

We report test results on a GeForce 1080. In order to measure the performance of BVHs, we use Aila’s et al. ([AL09], [ALK12]) ray tracing framework, compare performance by tracing the exact same set of rays and use the same intersector with each ray tracing implementation. We adopt the implementation of LBVH from [DP15] with extra operations omitted. For scan and sort operations we apply the CUB library [Nvi18].

4.1. Particle Mixtures

Our first test scenarios are the results of a particle mixture simulation. Figure 5 shows two distinct situations in such a simulation. The first is the initial state in which spheres are uniformly distributed and may overlap a lot, the second situation is the result after several iterations where the overlap of spheres is minimized and their distribution is not uniform anymore.

For LOBVH we set the maximum octree level to 6. With maximum octree level set to 7, the construction time is increased by about 70%, while keeping up similar tracing times. The memory requirements for the other two methods is higher because of the bigger number of internal nodes in the binary tree. If we allowed 7 levels in the octree, LOBVH would already require eight times the memory ($\approx$ 400 MB); 10 levels would even require over 200 GB. This shows why we call LLBVH memory efficient.

Comparing LBVH with LLBVH, it does not pay off to add the larger spheres further up in the tree. In any case does the construction time increase more than the tracing time could be reduced. However, in the INITIAL scenario our method produces the slightly better hierarchy because LBVH only uses the centroids of spheres and therefore puts overlapping large and small spheres into the same leaf nodes resulting in bad bounding volumes.

4.2. Triangles

Since, in most applications, acceleration structures are used for triangle scenes, we compare our algorithm for a number of such scenes in Figure 6. For LLBVH we expected a performance gain similar to SBVH [SFD09], because we add larger triangles at the top of the hierarchy. Often our algorithm has the slower build time compared to LBVH. The gain in the tracing time is, if at all, minimal and sometimes even worse. The only scene for which LLBVH is faster by about 4% is also the HAIR BALL. This scene is similar to the simulation output as it consists of many very uniformly distributed primitives with high occlusion. It is also the only scene for which LOBVH is not significantly slower. Indeed, LOBVH is even faster than both other methods for this specific scene.

With respect to memory requirements our LLBVH is often better than LBVH whereas LOBVH can be better or worse depending on the chosen level. We tested 6,7 and 8 levels for the dense octree and report the results with the best round-trip-times in the figure. While tracing time often improves for 8 levels, the build time increases by a similar factor like the memory.

Other methods like HLBVH are expected to have lower timings, because they all add optimization operations to the LBVH method. Since construction time already dominates the benchmarks in Figure 6, the gain in tracing performance of this methods will not suffice for better round-trip-times.

5. Conclusions

We introduced a new fast way, based on a loose octree, to take the spatial extent of primitives into consideration for building a BVH. The implementation based on the dense structure performs well for highly complex/occluded scenes such as the particle mixing and the hair ball scene. We also proposed a memory-efficient implementation, to reduce the memory requirement when BVHs with finer levels are required. The memory-efficient algorithm can be seen as a generalization of the state-of-the-art parallel construction for LBVH with the size of primitives explicitly being considered. Our methods show good performance for real-time ray tracing targeting fully dynamic scenes, especially for primitives with random positions in space. For traditional triangle scenes, our memory-efficient algorithm yields comparable results with one of the fastest methods (LBVH) being used for real-time ray tracing.

For future work, we intend to investigate on how to improve the tree quality by applying SAH but without introducing too much extra overhead. Possibly the memory requirement for our LLBVH can be reduced, if either k_{DFS} or m' can be omitted. Therefore, both sorting and the later comparisons need to use the same key.

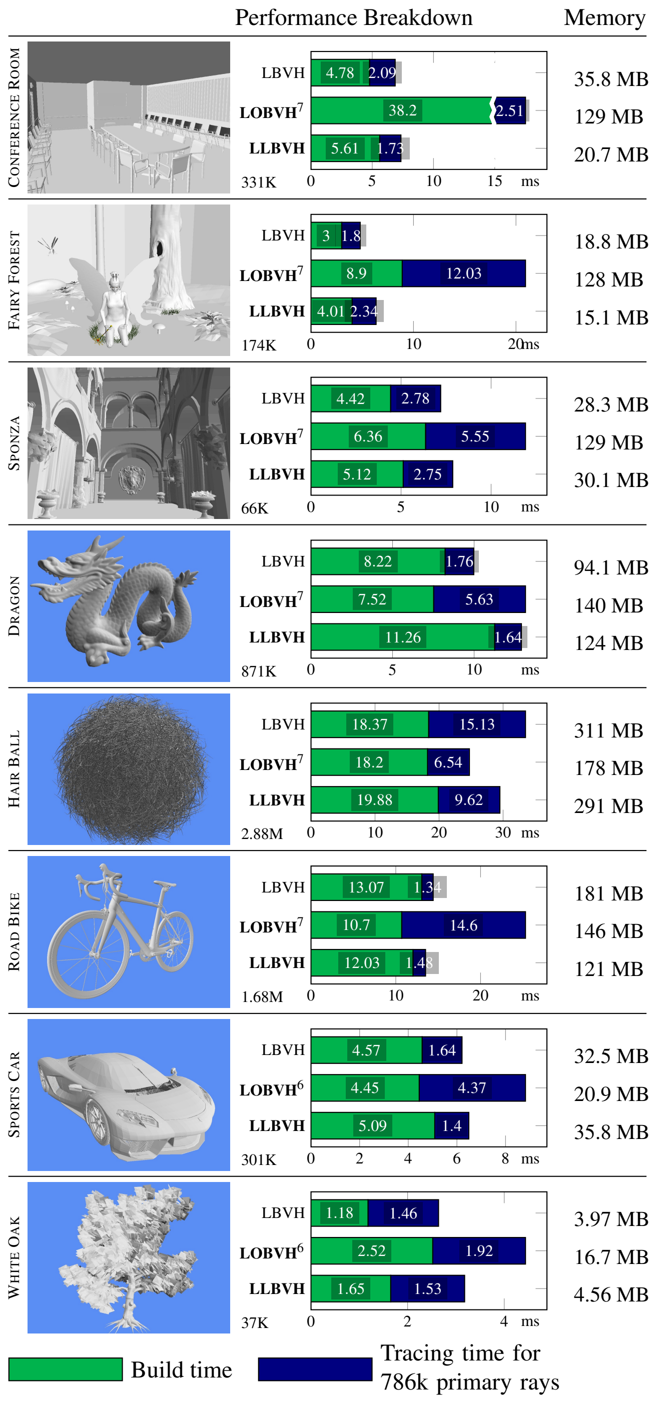


Figure 6: *Performance comparison for different triangle scenes. LLBVH often uses less memory, but takes more time than LBVH. In LOBVH the memory requirement depends on the maximum octree level which is 6 or 7 as noted next to the method label. The number of triangles is shown on the side of each scene.*

References

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. of High Performance Graphics (TOG)* (2009), HPG, ACM, pp. 145–149. 7

[ALK12] AILA T., LAINE S., KARRAS T.: Understanding the Efficiency of Ray Traversal on GPUs–Kepler and Fermi Addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02* (2012). 7

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. *Computer Graphics and Visual Computing (CGVC)* (2014). 2

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In *Proc. of High-Performance Graphics (HPG)* (Aug. 2015), ACM, pp. 13–20. 2, 7

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and Faster HLBVH with Work Queues. In *Proc. of High Performance Graphics (HPG)* (2011), ACM, pp. 59–64. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20. 2

[KA13] KARRAS T., AILA T.: Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *Proc. of High-Performance Graphics (HPG)* (July 2013), ACM, pp. 89–99. 2

[Kap85] KAPLAN M.: Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH Tutorial* (1985), Addison-Wesley, pp. 149–158. 2

[Kar12] KARRAS T.: Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proc. of High Performance Graphics (HPG)* (June 2012), Eurographics Association, pp. 33–37. 1, 2, 5, 6, 7

[KS09] KALOJANOV J., SLUSALLEK P.: A Parallel Algorithm for Construction of Uniform Grids. In *Proc. of High-Performance Graphics (HPG)* (2009), ACM, pp. 23–28. 2

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

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing Using Space Subdivision. *The Visual Computer 6*, 3 (May 1990), 153–166. 2

[Nvi18] NVIDIA: NVIDIA CUB, 2018. URL: `https://nvlabs.github.io/cub/`. 4, 7

[PGKS17] PÉRARD-GAYOT A., KALOJANOV J., SLUSALLEK P.: GPU Ray Tracing Using Irregular Grids. *Computer Graphics Forum (CGF) 36*, 2 (May 2017), 477–486. 2

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *Proc. of High-Performance Graphics (HPG)* (June 2010), Eurographics Association, pp. 87–95. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial Splits in Bounding Volume Hierarchies. In *Proc. of High Performance Graphics (HPG)* (2009), HPG, ACM, pp. 7–13. 2, 7

[Ulr00] ULRICH T.: Loose Octrees. In *Game Programming Gems*, vol. 1. Charles River Media, 2000, pp. 444–453. Mark DeLoura, Editor. 1

[VBH17] VINKLER M., BITTNER J., HAVRAN V.: Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction. In *Proceedings of High Performance Graphics* (2017), ACM, p. 9. 2

[VHB14] VINKLER M., HAVRAN V., BITTNER J.: Bounding Volume Hierarchies Versus Kd-trees on Contemporary Many-core Architectures. In *Proc. of Spring Conference on Computer Graphics (SCCG)* (May 2014), ACM, pp. 29–36. 1, 2

[WK07] WÄCHTER C., KELLER A.: Terminating Spatial Hierarchies by A Priori Bounding Memory. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on* (2007), IEEE, pp. 41–46. 2