The Visual Computer
https://doi.org/10.1007/s00371-020-01886-6

ORIGINAL ARTICLE

OLBVH: octree linear bounding volume hierarchy for volumetric meshes

Daniel Ströter[1] · Johannes S. Mueller-Roemer[2] · André Stork[2] · Dieter W. Fellner[2,3]

Abstract
We present a novel bounding volume hierarchy for GPU-accelerated direct volume rendering (DVR) as well as volumetric mesh slicing and inside-outside intersection testing. Our novel octree-based data structure is laid out linearly in memory using space filling Morton curves. As our new data structure results in tightly fitting bounding volumes, boundary markers can be associated with nodes in the hierarchy. These markers can be used to speed up all three use cases that we examine. In addition, our data structure is memory-efficient, reducing memory consumption by up to 75%. Tree depth and memory consumption can be controlled using a parameterized heuristic during construction. This allows for significantly shorter construction times compared to the state of the art. For GPU-accelerated DVR, we achieve performance gain of 8.4×–13×. For 3D printing, we present an efficient conservative slicing method that results in a 3×–25× speedup when using our data structure. Furthermore, we improve volumetric mesh intersection testing speed by 5×–52×.

Keywords Bounding volume hierarchy · GPGPU · Volumetric meshes · Direct volume rendering · Intersection detection · Slicing

1 Introduction

Bounding volume hierarchies (BVHs) and spatial data structures in general are indispensable tools in computer graphics. They are used to accelerate a multitude of algorithms, including collision detection, frustum culling, and ray tracing. Due to the low cost and high performance of massively parallel manycore graphics processing units (GPUs), BVHs that can be efficiently constructed and traversed on the GPU are of particular interest in GPU-accelerated physically based animation and ray tracing.

However, current research focuses on GPU BVHs for triangular surface meshes or point clouds (see Sect. 2). BVHs for volumetric meshes, e.g., tetrahedral meshes used in finite element simulations, must have different characteristics to achieve high performance. For example, volumetric meshes generally fill a space more densely than surface meshes and therefore lead to deeper hierarchies. Volumetric meshes are widely used in physically based animation, computational physics, and scientific visualization of simulation results (see Fig. 1). Furthermore, volumetric meshes are advantageous for the description of 3D printed models, as they enable the description of material gradients (see, e.g., Altenhofen et al. [2]).

In this paper, we examine how construction and traversal performance as well as memory use can be improved by adapting previous GPU-optimized BVHs for volumetric meshes. We introduce a novel octree-based linear BVH (OLBVH) based on the linear BVH (LBVH) by Lauterbach et al. [12]. We demonstrate the performance and memory benefits of using an octree-based BVH instead of a binary tree-based BVH for Direct volume rendering (DVR), plane intersection for cross-section computation or slicing, and inside-outside intersection testing. Furthermore, OLBVH creates a hierarchy of tightly fitting bounding boxes, unlike the original LBVH and most other LBVH variants (see Sect. 2). This allows for the introduction of boundary flags (see Sect. 3.1) that enable efficient space skipping in DVR, conservative slicing, and inside-outside intersection testing between volumetric meshes (see Sect. 4).

✉ Daniel Ströter
daniel.stroeter@gris.tu-darmstadt.de

1 Technische Universität Darmstadt, 64277 Darmstadt, Germany

2 Fraunhofer IGD and Technische Universität Darmstadt, Darmstadt, Germany

3 Technische Universität Graz, Graz, Austria

Published online: 06 July 2020

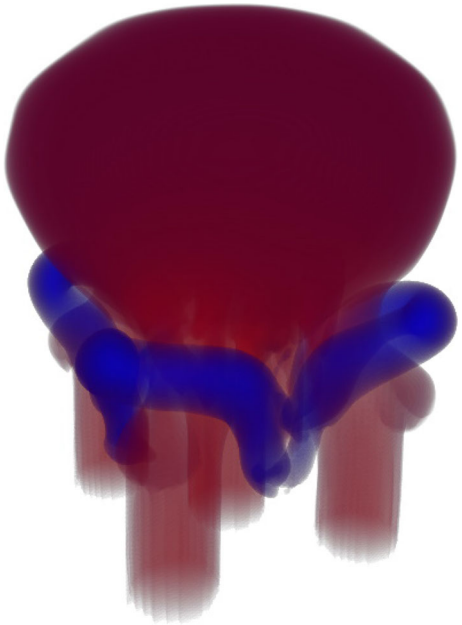

Fig. 1 The octree linear bounding volume hierarchy OLBVH enables efficient direct volume rendering of scientific data on unstructured volumetric meshes. The mesh shown has 12 million tetrahedra

In the following, Sect. 2 discusses related work in the field of GPU-optimized BVHs. Section 3 describes the concepts behind our novel BVH and relevant implementation details. Section 4 discusses the performance and memory characteristics of our data structure and compares them with current state-of-the-art GPU BVHs. Finally, Sect. 5 summarizes the paper, discusses limitations, and provides avenues for further research.

2 Related work

In this section, we discuss related work in the field of boundary volume hierarchies with GPU-accelerated construction and traversal.

The linear bounding volume hierarchy (LBVH) introduced by Lauterbach et al. [12] forms the basis of many of the newer approaches. They use Morton codes [15] (see Fig. 2) to approximately spatially sort discretized element centroids in parallel. Split positions for the hierarchy are determined in parallel according to differing bits of neighboring codes. While construction is very fast, it results in many singleton nodes, i.e., nodes with only one child, and nodes overlap significantly due to the approximate nature of Morton order sorting. They also introduce a surface area heuristic (SAH) hierarchy that ameliorates these issues, but significantly increases construction cost.

A number of authors have since improved SAH-LBVH construction performance. Pantaleoni and Luebke [20] introduced a two-level hierarchical LBVH (HLBVH) Morton sorting method with better performance for dynamic meshes. This method was further improved by Garanzha et al. [8] who introduced a task-based approach to HLBVH construction that enables construction in a single kernel call. Karras [11] developed an approach that computes all hierarchy levels in parallel leading to better scaling, while also generalizing to k-D trees and octrees. This approach was further improved by Apetrei [3] using atomic operations to avoid binary searches to improve construction speed. However, surface area heuristics do not map well to volumetric meshes.

y \ x	0	1	2	3
	00	01	10	11
0 00	0000	0001	0100	0101
1 01	0010	0011	0110	0111
2 10	1000	1001	1100	1101
3 11	1010	1011	1110	1111

Fig. 2 Morton codes for a 4×4 grid. The bits of the individual integer coordinates are interleaved, leading to a space-filling curve with a fractal Z-like shape

A potential alternative to the SAH are the extended Morton codes (EMCs) introduced by Vinkler et al. [25]. EMCs encode the discretized bounding box size into an arbitrary number of bits of the code, while the number of bits per coordinate can be varied as well. However, the position of these bits must be chosen carefully for good performance. Furthermore, high-quality volumetric meshes for simulation tend to vary smoothly in element size (see, e.g., Alliez et al. [1]), reducing the potential benefit of EMCs.

For time-varying data, refitting, as used in an early CPU-based work by Wald et al. [26], is an interesting alternative. However, the resulting BVHs are of lower quality and would lose the beneficial properties of tightly fitting nodes.

Besides efficient construction, traversal requires special attention on the GPU as it is typically performed recursively and the stack resides in local memory (a thread-private area of GPU RAM) and the compute-to-bandwidth ratio is very large. Murguia et al. [18], García et al. [9], as well as Binder and Keller [5], use bit trails for stackless traversal. Bit trails store the path through a tree as individual bits in an integer. However, these approaches are designed for binary trees and incur additional memory overheads such as allocation of empty nodes or hash maps. Vaidyanathan et al. [22] recently introduced a short stack approach which avoids these over-

heads and supports higher tree arities, albeit at the cost of requiring some stack space.

While Zellmann et al. [28] use the LBVH data structure for volumetric data, they do so for volumetric data on a sparse regular grid, avoiding looseness a priori. General unstructured volumetric meshes require a different approach. In particular, octrees are beneficial as volumetric data fill space more densely and octrees help avoid excessive hierarchy depth. For the use case of surface reconstruction from point clouds, Zhou et al. [29] construct an octree using a bottom-up approach based on Morton order sorting. While large point clouds can be stored efficiently, the memory overhead of the hierarchy itself is large and limits tree depth. Furthermore, hierarchy levels are allocated separately, leading to increased allocation and synchronization overhead. Gu et al. [10] introduce the octree-based LOBVH and a binary tree-based variant LLBVH based on the work of Karras et al. [11]. However, their approach requires full allocation of the finest octree level followed by a compaction step. This severely limits the maximum octree depth and results in a large memory overhead.

With the advent of specialized ray tracing cores in consumer graphics cards (see, e.g., Stich [21]), another option is to use the built-in hardware-accelerated BVH. Wald et al. [27] trace very short rays to perform point location in tetrahedral meshes. They achieve significant speedups compared to a pure general purpose computing on graphics processing units (GPGPU) approach, despite performing ray casting instead of direct point location. However, availability of GPUs with ray tracing hardware is currently limited to only the newest generation from a single vendor. Furthermore, efficient hardware BVH construction is itself an open research topic (see, e.g., Doyle et al. [7] or Viitanen et al. [23,24]).

3 Concept and implementation

In this section, we describe the basic concept of the OLBVH data structure and detail the implementation of the data structure as well as construction and traversal algorithms.

Like previous LBVH variants (see Sect. 2), our data structure relies on approximate spatial sorting of elements ordered by Morton code [15]. We quantize the input coordinates x, y, and z as l-bit integers $\hat{x} = (x_{l-1}, x_{l-2}, \ldots, x_0)$ and interleave their bits as shown in Fig. 2:

$$m(\hat{x}, \hat{y}, \hat{z}) = (x_{l-1}, y_{l-1}, z_{l-1}, \ldots, x_0, y_0, z_0) \tag{1}$$

For quantization purposes, we use the enclosing axis-aligned bounding box (AABB) $H(\mathbb{M})$ of the volumetric mesh $\mathbb{M}$:

$$H(\mathbb{M}) = \left(\mathbf{x}^{\mathbb{M}}_{\min}, \mathbf{x}^{\mathbb{M}}_{\max}\right), \tag{2}$$

where $\mathbf{x}^{\mathbb{M}}_{\min}$ and $\mathbf{x}^{\mathbb{M}}_{\max}$ are the componentwise minimum and maximum positions within the mesh.

A key difference to previous LBVH variants is that we do not encode the centroids of AABBs. Instead, we use the fact that Morton codes inherently span an equidistant grid. The resolution of this grid depends on the number of bits l used for quantization. The cell size of the Morton grid can be calculated as:

$$\mathbf{s}^l = (s^l_x, s^l_y, s^l_z)^T := \frac{\mathbf{x}^{\mathbb{M}}_{\max} - \mathbf{x}^{\mathbb{M}}_{\min}}{2^l - 1} \tag{3}$$

Using this size, quantization and reconstruction follow the equations below, respectively:

$$\begin{aligned} q(x) &= \left\lfloor \frac{x - x^{\mathbb{M}}_{\min}}{s^l_x} + \frac{1}{2} \right\rfloor = \hat{x} \\ q^{-1}(\hat{x}) &= x^{\mathbb{M}}_{\min} + \hat{x} \cdot s^l_x \end{aligned} \tag{4}$$

The quantized AABB $\hat{H}(\mathbb{P})$ of a primitive $\mathbb{P} \in \mathbb{M}$ is defined by two quantized points:

$$\hat{H}(\mathbb{P}) := \left(\hat{\mathbf{x}}^{\mathbb{P}}_{\min}, \hat{\mathbf{x}}^{\mathbb{P}}_{\max}\right) \tag{5}$$

For each primitive $\mathbb{P} \in \mathbb{M}$, we generate the Morton codes enclosed by the primitive's quantized AABB $\hat{H}(\mathbb{P})$. The set of generated Morton codes for a primitive $\mathbb{P} \in \mathbb{M}$ is given by:

$$M(\hat{H}(\mathbb{P})) := \left\{ m(\hat{x}', \hat{y}', \hat{z}') \mid \bigwedge_{c}^{\{x,y,z\}} \hat{c}^{\mathbb{P}}_{\min} \leq \hat{c}' \leq \hat{c}^{\mathbb{P}}_{\max} \right\} \tag{6}$$

Combined with a heuristic to determine the tree depth $L = l + 1$ (the root level is present, even for 0-bit Morton codes), we use $M(\hat{H}(\mathbb{P}))$ to split primitives a priori and eliminate looseness, i.e., BVH sibling cells never overlap. Additionally, we store boundary flags at every node of the tree, allowing for early termination at nodes containing only interior cells when only determining if a point or bounding box is inside or outside the mesh $\mathbb{M}$. As interior primitives

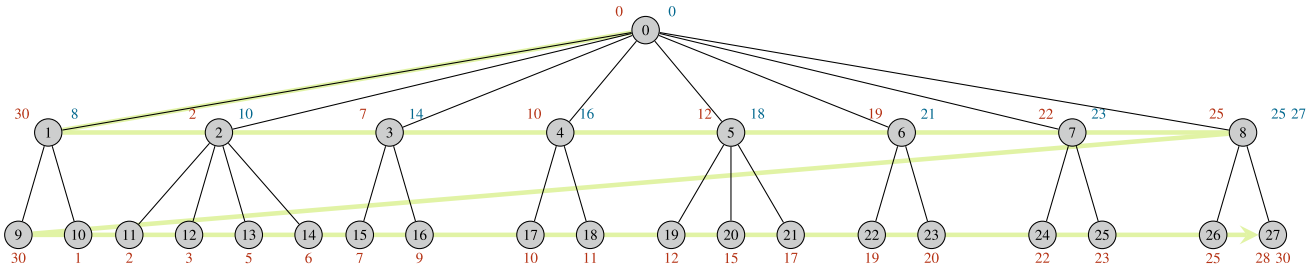

Fig. 3 This figure shows the data layout of the CO and PO arrays for a sample tree. The gray circles represent the tree nodes. The blue numbers above the tree nodes represent entries of the CO array, and the red numbers represent entries of the PO array. The green arrow in the background indicates the in-memory order

near the boundary must share points with boundary primitives and the sets $M(\hat{H}(\mathbb{P}))$ are inclusive, cells that contain empty space must always contain boundary primitives provided that no primitive has a negative signed volume.

3.1 Data structure

The OLBVH data structure consists of six arrays containing:

1. primitive indices $\texttt{P}[n_m]$ in $[0, n_p)$ sorted by Morton code,
2. tree node bounding volumes $\texttt{BV}[n_n]$,
3. child node offsets $\texttt{CO}[n_{n_i}+1]$ in $[0, n_n)$,
4. primitive index offsets $\texttt{PO}[n_n+1]$ in $[0, n_m]$,
5. boolean boundary flags $\texttt{BF}[n_n]$,
6. per-level node offsets $\texttt{NO}[L]$ in $[1, n_n]$,

where n_p is the number of primitives in $\mathbb{M}$, n_m is sum of the numbers of Morton codes per primitive, n_n is the number of tree nodes, and n_{n_i} is the number of internal tree nodes excluding leaves.

As the bounding volume of a primitive may intersect with several spatial cells of the Morton grid, P may contain duplicate indices. The bounding volumes of tree nodes BV are laid out linearly in memory following a breadth-first traversal order. In order to allow for top-down traversal, CO stores child offsets for the tree nodes. Every tree node is associated with a maximum of eight children. Due to the contiguous levelwise order in memory, the CO array enables retrieval of the child node indices C_i of a given tree node i:

$$C_i := \{j \mid \texttt{CO}[i] < j \leq \texttt{CO}[i+1]\}. \tag{7}$$

In order to manage memory efficiently, the CO array only contains entries for internal, i.e., non-leaf, nodes. As a result of the chosen data structure, the node index of an internal node's rightmost child is equal to the child offset of its succeeding node in memory, i.e.,

$$\text{Node } i \text{ is } j\text{'s rightmost child} \implies \texttt{CO}[j+1] = i \tag{8}$$

This property is useful while constructing the internal hierarchy levels starting from the leaves (see Sect. 3.2).

The OLBVH also incorporates a primitive offsets array PO to infer the primitive indices in P are associated with node i. As nodes reside in memory in breadth-first order, it is necessary to handle adjacent nodes of different hierarchical levels:

$$P_i := \begin{cases} \{\texttt{P}[j] \mid 0 \leq j < \texttt{PO}[i+1]\}, & \text{if } \texttt{PO}[i] > \texttt{PO}[i+1] \\ \{\texttt{P}[j] \mid \texttt{PO}[i] \leq j < \texttt{PO}[i+1]\}, & \text{otherwise} \end{cases} \tag{9}$$

As any rightmost child has the same upper offset as its parent node, it holds that:

$$\text{Node } i \text{ is } j\text{'s rightmost child} \implies \texttt{PO}[i+1] = \texttt{PO}[j+1] \tag{10}$$

The memory layout of the CO and PO arrays is illustrated for a sample tree in Fig. 3.

The boundary flag array BF stores a boolean boundary flag for each tree node i. The final array NO stores the node index offsets for each level. It is used to determine the hierarchical level of a node given its index.

3.2 Construction

The OLBVH construction procedure receives any volumetric mesh $\mathbb{M}$ with marked boundary primitives as input. We focus on tetrahedral meshes in our evaluation. However, OLBVH is equally applicable to general polyhedral meshes. We use the GPU-optimized tetrahedral mesh data structure by Mueller-Roemer et al. [16,17] to efficiently store and process tetrahedral meshes on the GPU, and use it to determine which primitives lie on the boundary, i.e., have one or more faces with only one neighbor. As this is part of pre-processing, not BVH construction, and covered in previous publications, we do not evaluate the computation of initial boundary primitive marking. If $H(\mathbb{M})$ is not known beforehand, a parallel reduction on the mesh's vertices calculates $\mathbf{x}^{\mathbb{M}}_{\min}$ and $\mathbf{x}^{\mathbb{M}}_{\max}$. Parallel primitives, such as reductions, parallel prefix sums, and sorting, are efficiently implemented in libraries such as

CUB [13] or Thrust [4]. We use Thrust in our implementation. Construction is performed in four stages:

1. Determine the tree depth heuristically
2. Calculate and sort Morton codes of primitive AABBs
3. Record at which levels the sorted Morton codes split
4. Bottom-up construction based on split positions.

In the initial stage, we determine the number $l \leq l_{\max}$ of bits to use for quantization, and therefore the depth of the tree, according to element size. The maximum possible number of quantization bits is $l_{\max} = 10$ in our implementation, as we use 32-bit integers to store Morton codes. In parallel over primitives, we estimate the tree level by first computing the binary logarithm of the largest tetrahedron AABB axis in relation to the maximum grid resolution:

$$\alpha_{\mathbb{P}} = \max\left(\left\lfloor \mathrm{ld}\left\lfloor \frac{\mathbf{x}^{\mathbb{P}}_{\max} - \mathbf{x}^{\mathbb{P}}_{\min}}{\mathbf{s}^{l_{\max}}}\right\rfloor_{\max}\right\rfloor, 0\right) \tag{11}$$

where the division is performed per component. The binary logarithm can be efficiently implemented using a count leading zeros instruction (the `__clz` intrinsic in CUDA). Subsequently, the construction procedure chooses

$$l = \mathrm{clamp}\left(\lfloor 10.5 - (\mathrm{avg}\,(\alpha_{\mathbb{P}}) + \alpha)\rfloor, 0, 10\right) \tag{12}$$

by performing a parallel reduction on $\alpha_{\mathbb{P}}$, where $\alpha \in [0, 10]$ is a tuning coefficient. While it is possible to choose a negative α, it is not advised, as the resulting Morton grid would be finer than most primitive AABBs. This would lead to oversampling. Therefore, we define $\alpha \in [0, 10]$. Increasing α results in a coarser Morton grid and thereby faster construction and reduced use of memory.

Since the maximum level is heuristically determined, we continue with calculating the Morton codes in the second stage. The size of the Morton grid cells $\mathbf{s}^l$ can be computed using Eq. 3. On the basis of $\mathbf{s}^l$, we generate a temporary array of Morton codes `MC`. As the overall number of Morton codes n_m is not known a priori, we perform two parallel passes over primitives to determine n_m and allocate and fill `MC`. Each thread calculates $\left|M(\hat{H}(\mathbb{P}))\right|$ for one primitive. A parallel exclusive prefix sum determines the array offsets the Morton codes of each primitive are written to. By performing the prefix sum for $n_p + 1$ elements, the final offset corresponds to n_m and additional branching is avoided. A second pass over all primitives generates the sets $M(\hat{H}(\mathbb{P}))$ and writes the Morton codes into `MC` at the computed offsets, while also writing each primitive's index to `P` at the corresponding offset.

With a parallel sort, the primitive indices `P` are sorted using the Morton codes in `MC` as keys. As a result, the primitive indices are ordered along the Z-curve. Primitive indices are potentially duplicated, as $M(\hat{H}(\mathbb{P}))$ may include several Morton codes. Additionally, the root node of the OLBVH is constructed at this point. Because the root node encloses all primitives $\mathbb{P} \in \mathbb{M}$, its AABB is set to $H(\mathbb{M})$. Moreover, the two initial entries of `PO` are set to 0 and the overall number of Morton codes n_m, respectively. Straightforwardly, the first entry of `CO` is 0, the first entry of `NO` is 1, and the last is n_n. Additionally, the root boundary flag `BF`[0] is set to *true*.

In the third stage, we determine the indices at which the Morton codes in `MC` indicate splits between BVH nodes. As the OLBVH is an octree, we compare Morton codes advancing in 3-bit steps beginning from the most significant bit. Thus, each split is associated with a level $l_c < L$ and an index referring to a Morton code in the `MC` array. As the split positions refer to the Morton codes of primitives, the split generation procedure writes these positions to the `PO` array. As in the case of Morton code generation, we first determine the number of splits between neighboring codes. A parallel scan is then used to determine the size of and offsets into the corresponding arrays. As no split is recorded for the last Morton code, a split is appended sequentially by the CPU for each level. This can be done concurrently while filling the remainder of the `PO`. A parallel stable sort by key procedure sorts `PO` by split level resulting in the intended primitive offset order. As the splits are sorted by level, the remaining entries of the `NO` array can be calculated by exploiting the per-level contiguous order. In parallel over splits, we search for adjacent splits of different level. If two such splits are found at positions i and $i + 1$, a node level offset was found and thus $\mathtt{NO}[splits[i].l_c] = i + 1$.

The final stage relies on `NO` and `PO` to construct the OLBVH in a bottom-up manner, i.e., starting at the leaf level. As the data layout does not allow for immediate inference of the parent node for a given node, we cannot construct the hierarchy in a single kernel launch using atomic flags as in Apetrei's [3] agglomerative LBVH builder.

We first calculate the leaf node AABBs and boundary flags `BF` using Algorithm 1. As all Morton codes associated with a leaf node are equal, the minimum AABB coordinates are obtained by calculating $q^{-1}(m^{-1}(m_{x,y,z}))$ for an arbitrary $m_{x,y,z}$ associated with the leaf node. Furthermore, the maximum AABB coordinates are calculated by adding the grid size $\mathbf{s}^l$ to the minimum coordinates. As a result, leaf node AABBs can be calculated in constant time. The boundary flag `BF`$[i]$ of a leaf node is set if any primitive in P_i is tagged as a boundary element. Therefore, boundary flag calculation for a leaf node is $\mathcal{O}(|P_i|)$.

Algorithm 2 outlines internal node generation. We allocate an auxiliary array `CI` of the same size as the overall number of generated Morton codes n_m. The hierarchy construction procedure writes node indices to the primitive offset positions of `CI` for each level. On construction of the parent

Algorithm 1 Leaf node bounding volume and boundary flag determination.

```
1: procedure GENERATELEAFNODES(l)
2:     for all i ∈ [NO[l − 1], NO[l]) do                    ▷ In parallel
3:         p ← P[PO[i + 1] − 1]                             ▷ Pick any entry in P_i
4:         m_{x,y,z} ← MC[p]                                ▷ Calculate AABB
5:         min ← q^{-1}(m^{-1}(m_{x,y,z}))
6:         max ← min + s^l
7:         BV[i] ← (min, max)
8:         for all ℙ ∈ P_i do                               ▷ Calculate boundary flag
9:             if ℙ is marked as boundary primitive then
10:                BF[i] ← true
11:                return
12:            end if
13:        end for
14:        BF[i] ← false
15:    end for
16: end procedure
```

hierarchical level, we exploit Eq. 10 to lookup the rightmost child indices for internal nodes at the upper primitive offset positions of the CI array. To reduce the number of kernel launches, the computation of CI can be merged with GENERATELEAFNODES for the first iteration. Between internal levels, a separate kernel must be used to prevent conflicting reads and writes. CI and PO are used to determine the child offsets CO according to Eq. 8. Finally, internal node AABBs and boundary flags are determined from their children.

Algorithm 2 Construction of internal nodes of level l_i.

```
1: procedure GENERATEINTERNALNODES(l_i)
2:     for all i ∈ [NO[l_i], NO[l_i + 1]) do                ▷ In parallel
3:         CI[PO[i + 1]] ← i
4:     end for
5:     CI[0] ← NO[l_i] − 1
6:     for all i ∈ [NO[l_i − 1], NO[l_i]) do                ▷ In parallel
7:         (begin, end) ← (PO[i], PO[i + 1])
8:         if begin > end then                              ▷ Calculate primitive range
9:             begin ← 0
10:        end if
11:        CO[i] ← CI[begin]
12:        c_begin ← CO[i] + 1
13:        c_end ← CI[end]
14:        if i = NO[l_i] − 1 then                          ▷ Set last child offset
15:            CO[i + 1] ← c_end
16:        end if
17:        BV[i] ← BV[c_begin] ∪ ··· ∪ BV[c_end]            ▷ Merge AABBs
18:        BF[i] ← false
19:        for j ← c_begin, ..., c_end do                   ▷ j ∈ C_i
20:            if BF[j] then                                ▷ Calculate boundary flag
21:                BF[i] ← true
22:            end if
23:        end for
24:    end for
25: end procedure
```

3.3 Traversal

Algorithm 3 outlines our OLBVH traversal approach. Figure 4 presents a sample traversal execution. Traversing the OLBVH for input geometric predicates F_N for nodes and F_P for primitives proceeds by pushing child nodes to a traversal stack. If a full traversal stack is used, its size is bounded by $7 \cdot l + 8$ node indices. This typically exceeds the amount of available fast on-chip shared memory, resulting in having to use slower cache-backed local memory. Consequently, OLBVH traversal uses a short stack [22] to reduce the memory requirement for the stack. As a result of the shallow hierarchy, a 32-bit integer is sufficient to encode the trail. Leaf nodes can be determined efficiently by comparison with the lower level offset of the last level. Since siblings are on the same level, it suffices to check the first child node. The short stack approach uses a queue Q to record the nodes in C_i (cf. Eq. 7) for which the predicate F_N evaluates as true. On the evaluated GPUs, the use of a short stack results in a minor approx. 5% speedup, which is not evaluated in detail, on older GPUs we observed larger speedups.

Algorithm 3 OLBVH traversal using a short stack.

```
1: procedure TRAVERSEOLBVH(F_N, F_P)
2:     trail ← 0x0
3:     l_cur ← 0
4:     S_short ← ∅
5:     i ← 0                                                ▷ Initiate at root
6:     result ← 0                                           ▷ Alternatively, N ← ∅
7:     found ← false
8:     nodeBound ← NO[l − 1]
9:     exit ← false
10:    do
11:        Q ← ∅
12:        k ← (trail ≫ 3l_cur) & 0x7
13:        leafLevel ← CO[i + 1] > nodeBound
14:        for j ∈ C_i do
15:            if F_N(j) then
16:                if leafLevel then
17:                    exit ← true
18:                    found ← true
19:                    result ← j                           ▷ Alternatively, push j onto N
20:                else
21:                    Q ← Q ∪ {j}
22:                end if
23:            end if
24:        end for
25:        MANAGESHORTSTACK(S_short, trail, l_cur, i, Q, k, exit)
26:    while ¬exit
27:    if found then ▷ Alternatively, test intersection for every j ∈ N
28:        if F_P(ℙ_𝕄) for any ℙ_𝕄 ∈ P_result then          ▷ cf. Eq. 9
29:            treat incident
30:            return
31:        end if
32:    end if
33: end procedure
```

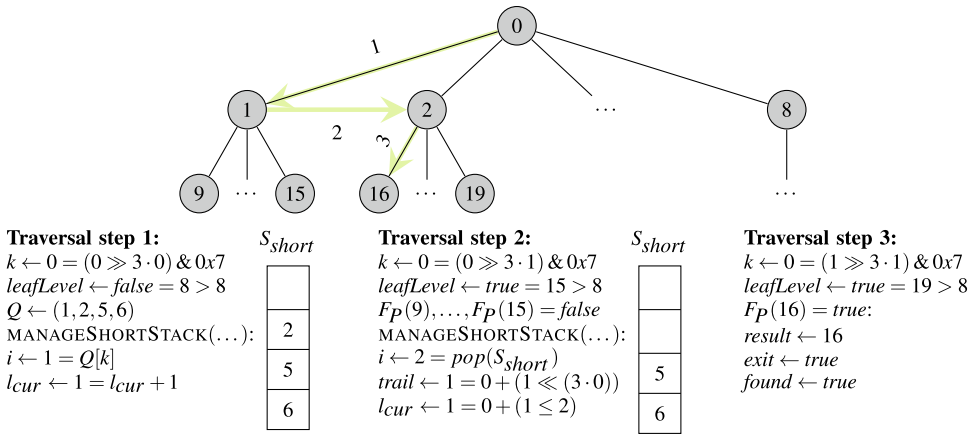

Fig. 4 Traversal proceeds on the compacted tree until a leaf node satisfies F_N. The green arrows indicate the traversal path. Variable assignments and short stack states appear below the tree. Initially, traversal concludes that F_N holds for nodes 1, 2, 5 and 6. Thus, the next node is 1, while nodes 2, 5 and 6 remain on the short stack. In the second step, none of 1's children satisfy F_N. Thus, traversal pops 2 from the short stack, determines the corresponding parent level, increments the trail bits of that level, and finds the current level using Eq. 13. Finally, F_N holds for 16 and traversal terminates

Unlike Vaidyanathan et al.'s [22] short stack implementation, we do not tag parent nodes, avoiding unnecessary branching. Additionally, our implementation does not treat leaf nodes satisfying F_N throughout traversal. This is due to the fact that depending on α leaf nodes contain several primitives and a high primitive count may lead to significant thread divergence. After pushing intersecting child nodes to Q, the MANAGESHORTSTACK(...) procedure pushes tree nodes from Q to the short stack and identifies the tree node at which traversal continues. This procedure corresponds to lines 10–27 of Vaidyanathan et al.'s [22] BVH$-N$ traversal method with $N = 8$. The only two adaptations are the bit-trail handling and comparison of NO[*parentLevel*] to determine the level of a node on short stack pop to avoid unnecessary branching (lines 16–20 in Stack Pop):

$$l_{cur} = parentLevel + (\texttt{NO}[parentLevel] \leq n) \tag{13}$$

In many applications, it is sufficient to traverse the tree evaluating F_N until a leaf node is reached, as sibling cells never overlap. Primitives associated with the resulting leaf node are then checked until F_P evaluates to true. However, for some applications such as intersection detection it is required to keep track of intersecting leaf nodes and to test primitives for intersection. In such scenarios, a node set N is used to maintain leaf nodes satisfying F_N. This approach is inherently prone to overflow. Thus, additional treatment is required in order to prevent exceeding the size of N on node push. One solution is to evaluate F_P for the primitives of all nodes in N if a node push would result in an overflow. After traversal, we evaluate F_P for the primitives associated with the resulting leaf node or the set of leaf nodes N.

4 Results

In this section, we evaluate OLBVH construction time and memory consumption. In addition, we investigate our acceleration structure's benefits in three application areas: direct volume rendering of unstructured meshes for scientific visualization, conservative slicing for 3D-printing, and intersection detection. In order to avoid inaccuracies in runtime measurements, we calculate the median of 50 repetitions. All algorithms were implemented in C++ and CUDA using Visual Studio 2015 and CUDA 10.1. Our implementation of Wald et al.'s [27] method additionally uses OptiX 7.0 [19]. All measurements were performed on one machine with an RTX 2080 Ti and another with a Quadro GP100. The latter GPU does not support RTX in hardware. We consistently use a short stack size of 8. The meshes used throughout the evaluation are shown in Fig. 5. While good simulation meshes have smoothly varying element sizes and tetrahedra with high aspect ratios, the meshes were chosen to include both good meshes and meshes with very large variances in element size, as shown in Table 1. Additionally, while most of the meshes fill the majority of their bounding box, we included two shell-like meshes, "Part" and "Pot."

4.1 Construction

We compare OLBVH construction with Apetrei's [3] improved LBVH construction and OptiX' acceleration structure construction with subsequent compaction. The details of OptiX' acceleration structure are not documented and are considered a black box. The resulting construction times are shown

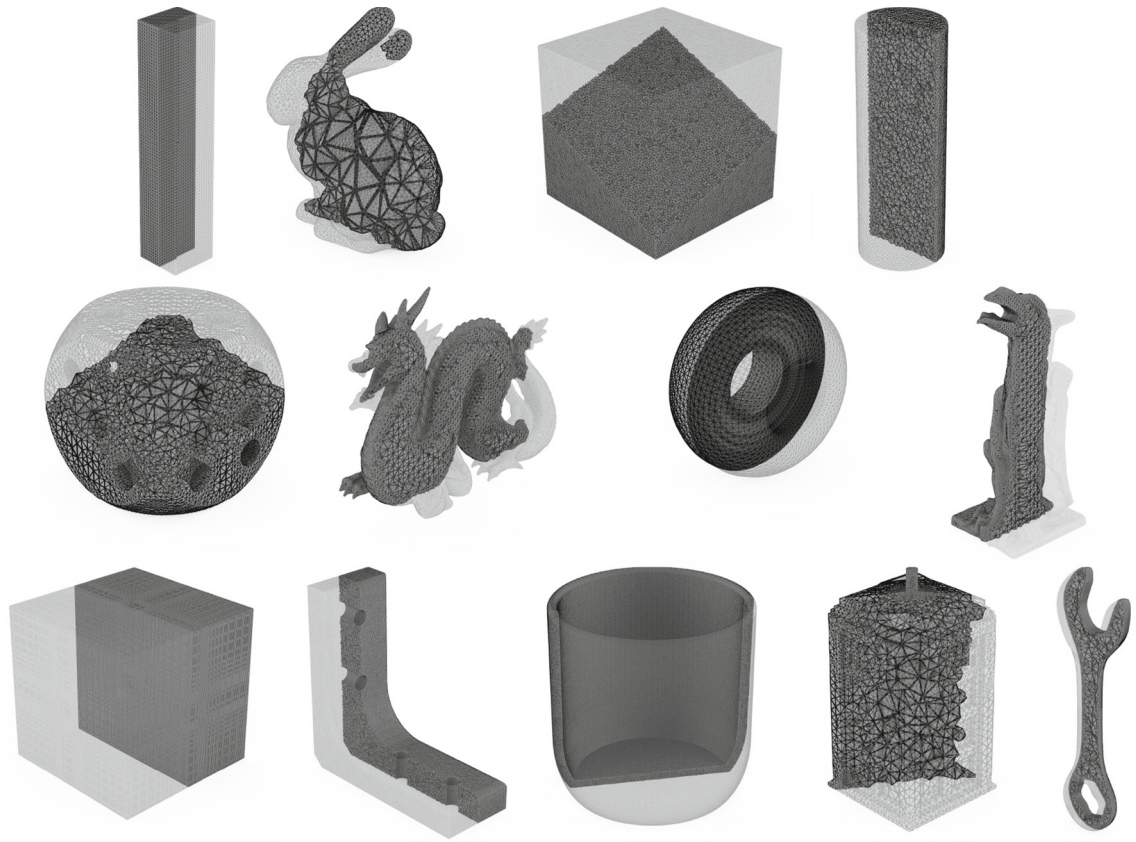

Fig. 5 Meshes used in the evaluation. Surface overlaid over a cross section shown to visualize both exterior and interior resolution. From top left to bottom right: Bar, Bunny, Cube, Cylinder, Die, Dragon, Fusion, Gargoyle, Jets, Part, Pot, Tardis, and Wrench. The models include both meshes with highly regular primitive size (e.g., Part) and others with large variance in primitive size (e.g., Tardis)

Table 1 Mesh element size and quality information for the meshes shown in Fig. 5

Mesh	Volume Ratio	Aspect Ratio		
		Minimum	Median	Maximum
Bar	1	0.233	0.233	0.233
Bunny	$5.5 \cdot 10^5$	0.005	0.126	0.331
Cube	183	0.043	0.268	0.333
Cylinder	693	0.059	0.259	0.333
Die	$2.5 \cdot 10^4$	0.002	0.256	0.333
Dragon	$2.6 \cdot 10^{11}$	0.011	0.258	0.333
Fusion	128	0.007	0.054	0.196
Gargoyle	$5.2 \cdot 10^{11}$	0.000	0.195	0.333
Jets	1	0.239	0.239	0.239
Part	$4.4 \cdot 10^3$	0.003	0.266	0.333
Pot	586	0.022	0.259	0.333
Tardis	$1.2 \cdot 10^8$	0.011	0.258	0.333
Wrench	$2.5 \cdot 10^6$	0.000	0.197	0.333

Besides the ratio of the largest absolute element volume to the smallest, we list the minimum, median and maximum element aspect ratios (inradius over circumradius) [6] rounded to the third decimal

in Fig. 6. As tree depth is affected by the choice of α, we present construction times for $\alpha \in \{0, 1, 2\}$. For $\alpha = 0$ our construction approach is slower than Apetrei's LBVH builder. For $\alpha \in \{1, 2\}$ our approach outperforms or matches the construction times of the LBVH builder in many cases. If no hardware acceleration is used, OLBVH construction is faster than OptiX on all but the smallest meshes (cf. Table 2), even for $\alpha = 0$. When hardware acceleration is present, our construction approach is slightly slower than OptiX' BVH construction for $\alpha = 0$, but achieves significantly lower runtimes when $\alpha \in \{1, 2\}$.

Besides construction time, another beneficial aspect of our data structure is its memory consumption. Table 2 compares the memory consumption of our OLBVH to LBVH and OptiX' acceleration structure. OLBVH consumes significantly less memory than LBVH or OptiX without specialized RTX hardware. For the jets and dragon meshes, OptiX consumes slightly less memory than OLBVH with $\alpha = 0$ when RTX hardware is used. Increasing α results in a significant reduction of memory consumption.

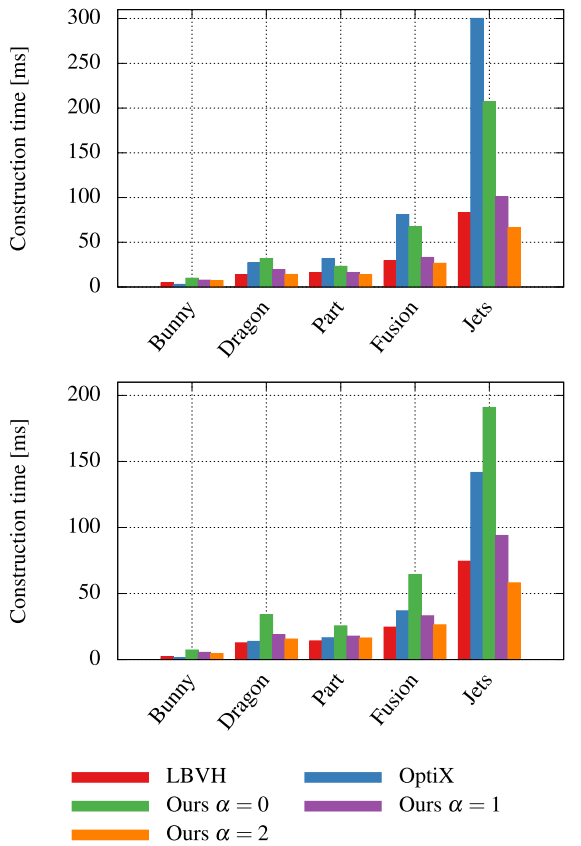

Fig. 6 Comparison of construction times between LBVH using Apetrei's [3] fast agglomerative approach, OptiX, and OLBVH with $\alpha \in \{0, 1, 2\}$ on a Quadro GP 100 (upper) and an RTX 2080 Ti (lower)

Table 2 Mesh sizes (in number of tetrahedra) and comparison of memory consumption between LBVH, OptiX' compacted BVH, and OLBVH for $\alpha \in \{0, 1, 2\}$

Mesh	Tets	LBVH	OptiX		Ours $\alpha =$		
			non-RTX	RTX	0	1	2
Bunny	32.0 k	2.07	10.4	1.94	1.83	0.57	0.27
Dragon	824.8 k	53.5	250	44.1	52.9	15.9	7.41
Part	1.1 M	71.3	313	54.9	28.4	12.3	7.22
Fusion	3.0 M	195	840	158	116	45.2	24.6
Jets	12.3 M	797	3427	399	433	164	91.2

All memory consumptions are given in MiB. Even for $\alpha = 0$, our approach results in significantly lower memory consumption for most meshes due to the flatter octree hierarchy and compact offset-based encoding

4.2 Direct volume rendering

Direct volume rendering of unstructured meshes requires repeated point location along each view ray. A potential performance issue is that a large number of samples may not hit the geometry. We use the boundary marking to efficiently compute the relevant ray interval in which the ray intersects the geometry. Prior to ray marching, we traverse only the boundary nodes of the OLBVH for each ray and determine the intervals for which each ray intersects leaf node AABBs. Subsequently, we perform efficient ray marching by traversing the OLBVH for each sample until a leaf node containing the sample is found.

Table 3 Mesh sizes (in number of tetrahedra) for the DVR evaluation meshes. We also give the ray interval computation times in milliseconds on the Quadro GP 100

Mesh	Tets	Ray Interval $\alpha =$		
		0	1	2
Wrench	390.3 k	2.5	1.4	0.8
Part	1.1 M	3.1	1.6	0.8
Fusion	3.0 M	5.7	3.3	1.9
Pot	4.0 M	4.8	2.5	1.4
Jets	12.3 M	5.3	3.0	2.1

We compare our approach with the LBVH as well as Wald et al.'s [27] approach with and without RTX hardware acceleration. In order to provide a good basis for comparisons, we evaluate all approaches using the same camera settings. The sizes of the evaluated meshes are given in Table 3. In every measurement, a 1024^2 image was rendered. The runtimes for the ray interval computation appear in Table 3. Because the traversal for determining the relevant ray interval safely skips the majority of nodes, its runtime imposes a negligible overhead. Unlike Moriccal et al.'s [14] space skipping extension to Wald et al.'s approach, we do not require a secondary acceleration structure.

Figure 7 shows the sampling rates per second for the different ray marching variants. If no specialized RTX hardware is used and $\alpha = 0$, our approach outperforms the ray marching technique of Wald et al. [27] by up to 8.4× and the LBVH variant by 2×–13×. The Jets mesh benefits the least, which is due to the fact that it is geometrically a cube (compare Fig. 1 and 5). Thus, our ray marching approach cannot benefit from the prior space skipping. Nonetheless, we observe a significant speedup. For the fusion and part meshes, it is even possible to choose $\alpha = 1$ and match the performance of Wald et al.'s method [27]. When the Pot mesh is used, we are able to match the sampling rate of Wald et al.'s method [27] with $\alpha = 2$. Therefore, an application could construct the OLBVH more quickly and use less memory while achieving the same performance. The LBVH results in lower sampling rates than the other methods. Traversing the binary radix tree consumes significantly more time than using our method with $\alpha \in \{0, 1\}$. However, RTX hardware accelerated ray marching is faster than our method. As only a minority of GPUs provides RTX hardware, we expect our method to be superior on the majority of GPUs.

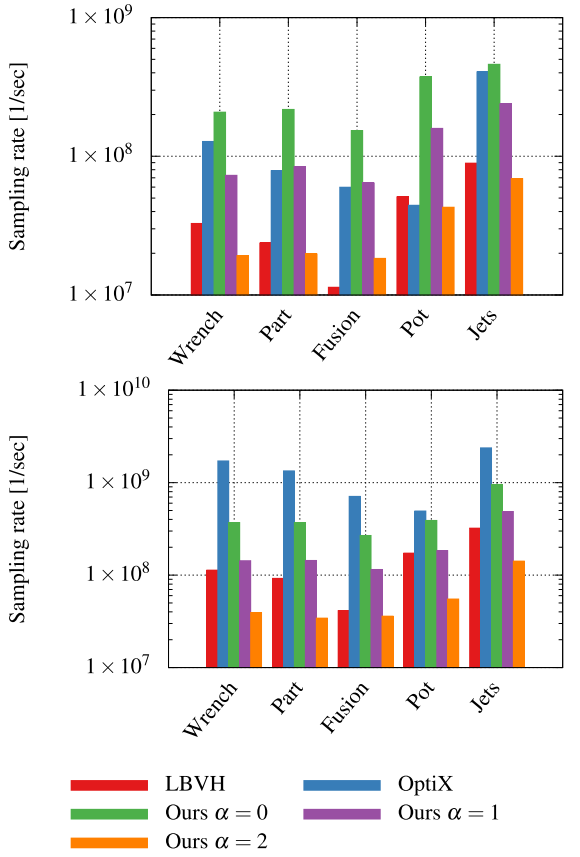

Fig. 7 Comparison of sampling rates using LBVH, Wald et al.'s method, and our approach for $\alpha \in \{0, 1, 2\}$ on a Quadro GP 100 (upper, no RTX) and an RTX 2080 Ti (lower, with RTX). We consider only samples inside the geometry

4.3 Conservative slicing

Like direct volume rendering, slicing of volumetric meshes with volumetrically varying materials (see, e.g., Altenhofen et al. [2]) is a point location problem. However, as post-processing of 3D printed models using conventional CAM or by polishing can only remove material, slicing typically has to be performed conservatively, i.e., material must be deposited in a voxel of the printing volume even if only the AABB of the voxel intersects the geometry and not the centroid itself. Therefore, if a tetrahedron intersects the AABB, the sampling point is valid. We interpolate material/scalar parameters for a sampling point inside the input mesh using the containing tetrahedron. In case of a sampling point outside the mesh, we extrapolate using the closest tetrahedron intersecting the conservative AABB. We evaluate on a 999^2 grid using planes and slice thicknesses normal to the slicing plane as they appear in Table 4.

In the following, we describe an efficient conservative slicing algorithm using OLBVH. We initially traverse the OLBVH for a sampling point, attempting to find a containing tetrahedron. Due to tightly fitting AABBs, it is sufficient to find the single leaf node that contains the sampling point. If no such tetrahedron can be found, the sampling point is potentially near the boundary. In that case, we reinitialize the short stack and traverse the OLBVH for the corresponding AABB while only considering boundary nodes. This decreases the number of relevant tree nodes during traversal. In the second traversal phase, we maintain a set N of intersecting leaf nodes. The competing LBVH variant traverses the tree for the voxel AABB finding the closest point in the mesh to the sampling point. As Wald et al.'s [27] approach only applies to point sampling, we do not consider it in this use case.

Table 4 Mesh sizes (in number of tetrahedra) and slicing setups for conservative slicing where the origin is the mesh AABB's midpoint. The slice thickness is the voxel size normal to the slicing plane

Mesh	Tets	AABB	Slice	
			Plane	Thickn.
Bunny	32.0 k	1×0.99×0.77	xy	0.05
Cylinder	172.4 k	0.52×1.4×0.52	xz	0.025
Cube	1.5 M	2×2×2	xy	0.05
Fusion	3.0 M	4.9×4.9×2.9	xy	0.1
Jets	12.3 M	127×127×127	xy	1

Figure 8 presents the runtimes for slicing using our approach and LBVH. Our approach outperforms LBVH for $\alpha \in \{0, 1\}$ with speedups between 3× and 25×. Therefore, the OLBVH allows for acceleration of conservative slicing, while applications can benefit from fast, memory-efficient construction on GPUs.

4.4 Mesh intersection

The third use case is intersection/overlap detection between two volumetric meshes. This has use cases in interactive mesh editing and Lagrangian (moving mesh) simulations. Initially, we construct an OLBVH for both input meshes. Subsequently, traversing the OLBVH of one mesh for each tetrahedron of the other yields a set of intersecting leaf nodes N for each tetrahedron. By testing the tetrahedra contained by the nodes in N for intersection with the traversal tetrahedron, we identify the tetrahedra which intersect the other mesh. In order to avoid the vast majority of intersection tests, we exploit the boundary marking to infer intersection for interior nodes. When a tetrahedron intersects the AABB of a non-boundary node, that tetrahedron must intersect with the other mesh. In addition, we perform both traversals ($A \subset B$ and $B \subset A$) simultaneously.

We compare our OLBVH-accelerated mesh intersection method to LBVH traversal using the test cases in Table 5, and the benchmark results appear in Fig. 10. For example, Fig. 9

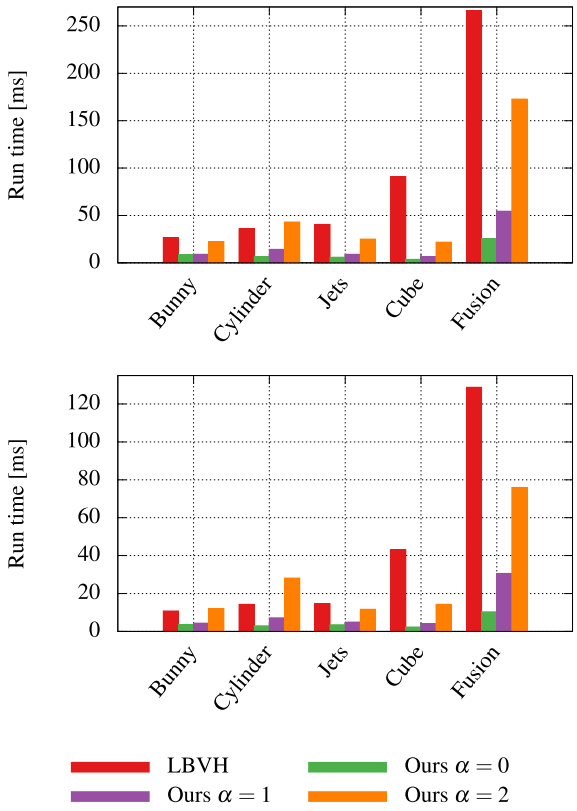

Fig. 8 Comparison of slicing times for a single slice using LBVH and our approach for $\alpha \in \{0, 1, 2\}$ on a Quadro GP 100 (upper) and an RTX 2080 Ti (lower)

Table 5 Mesh sizes (in number of Tetrahedra) for the intersection test cases

No.	Meshes		Tets	
1	Die	Tardis	232.8 k	150.8 k
2	Bunny	Bar	32 k	491.5 k
3	Cube	Cube	1.5 M	1.5 M
4	Cube	Gargoyle	1.5 M	792.1 k

illustrates the first intersection case. OLBVH-accelerated intersection detection results in a speedup of 5×–52× for $\alpha = 0$. This is due to the fact that the overwhelming majority of intersections can be inferred from the boundary flags, while the LBVH-accelerated intersection detection method has to perform narrow phase intersection checks for each tetrahedron due to overlapping AABBs. Furthermore, a significant benefit can be found for all $\alpha \in \{0, 1, 2\}$. Mesh intersection using OLBVH is fast and applications can benefit from quicker OLBVH construction and significantly reduced memory consumption by choosing a higher α.

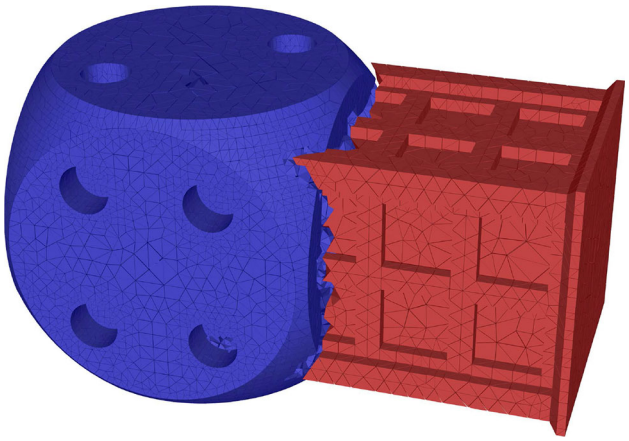

Fig. 9 Intersection case 1 between Tardis (red) and Die (blue) meshes. Intersecting tetrahedra have been removed. As the Tardis mesh extends far into the interior of the Die mesh, the vast majority of intersecting tetrahedra can be detected early throughout traversal. Narrow phase intersection detection is only required where mesh boundaries are close to each other

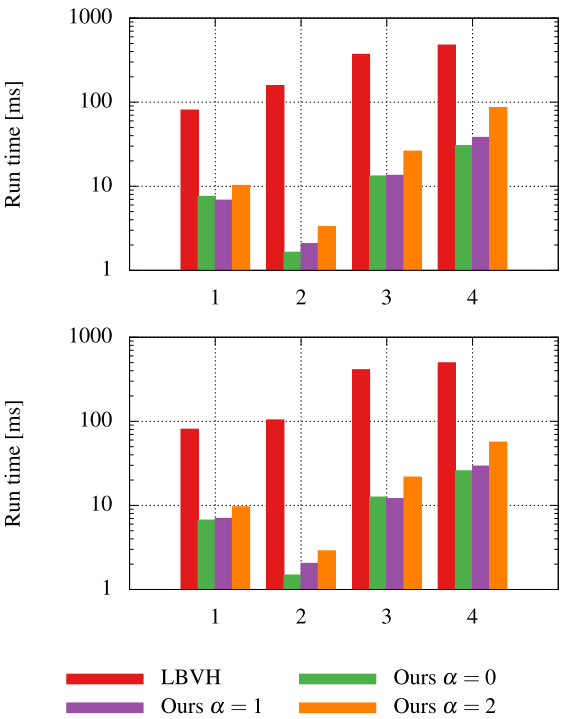

Fig. 10 Comparison of calculation times for computing the intersections using LBVH and our method for $\alpha \in \{0, 1, 2\}$ on a Quadro GP 100 (upper) and an RTX 2080 Ti (lower). The numbers underneath the bars correspond to the test cases in Table 5

5 Conclusion

In summary, we have introduced the OLBVH, a novel Morton curve based linear BVH for volumetric meshes. We have proposed a novel linear BVH construction approach for vol-

umetric meshes, which produces memory efficient trees in a time efficient manner. The heuristic used during construction is parameter controlled allowing for faster construction and reduced memory footprint by generating shallower trees. In addition, the resulting BVH is of high quality, as sibling cells are non-overlapping. We have demonstrated that the OLBVH is a versatile acceleration structure for volumetric meshes by examining three use cases: direct volume rendering for scientific visualization, conservative slicing for 3D printing, and inside-outside intersection detection.

The introduction of boundary marking, which requires a tightly fitting BVH, has enabled the development of efficient algorithms for all three use cases. We have achieved a significant increase of sampling rates in DVR, 8.4×–13× compared to state-of-the-art methods without specialized hardware for BVH traversal. OLBVH has allowed us to implement an efficient conservative slicing algorithm that results in a 3×-25× speedup. Additionally, we have accelerated intersection detection by 5×–52×. As a result, our data structure has opened the door for efficient GPU-accelerated volumetric mesh lookups.

5.1 Limitations

While memory consumption and construction time can be significantly reduced by increasing α, a more shallow tree may lead to lower performance, as leaf nodes contain more primitives. Furthermore, tetrahedra are assigned to nodes based on their quantized AABBs. Therefore, some nodes contain tetrahedra that do not overlap the node's AABB. For very large meshes, such as the 62M-tetrahedron jpn-qk dataset used by Wald et al. [27], GPU memory on the systems used in our evaluation is insufficient. Wald et al. partially work around this issue by generating separate BVHs for groups of 1M tetrahedra in the mesh.

5.2 Future work

A promising approach to improve the OLBVH is to generate Morton codes for each tetrahedron based on the tetrahedron itself instead of its AABB. Since the relevant ray intervals can be computed efficiently, it may be worthwhile to sort rays by their workload, in order to benefit from load balancing and efficient memory access using the short stack. To overcome the memory limitation, the workaround used by Wald et al. could be implemented using OLBVH. Alternatively, out-of-core variants could be examined. While current RTX hardware allows for custom intersection shaders, it does not allow for custom BVHs. If it should become possible to combine the built-in triangle intersection with custom BVHs, it may be interesting to revisit tetrahedral DVR using the same approach as Wald et al. but with OLBVH. Another interesting avenue for future research is exploring the performance benefits of a hardware or FPGA implementation of OLBVH. Such an implementation would benefit from the short stack, due to the small number of registers during traversal.

Acknowledgements Open Access funding provided by Projekt DEAL. This work was supported by the Qu4lity EU project which is co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement No. 825030. The Fusion and Jets meshes are courtesy of the university of Utah.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Alliez, P., Cohen-Steiner, D., Yvinec, M., Desbrun, M.: Variational tetrahedral meshing. ACM Trans. Graph. **24**(3), 617 (2005). https://doi.org/10.1145/1073204.1073238
2. Altenhofen, C., Luu, T.H., Grasser, T., Dennstädt, M., Mueller-Roemer, J.S., Weber, D., Stork, A.: Continuous property gradation for multi-material 3D-printed objects. In: Proceedings of the 29th Annual International Solid Freeform Fabrication Symposium—An Additive Manufacturing Conference, SFF '18, pp. 1675–1685 (2018)
3. Apetrei, C.: Fast and simple agglomerative LBVH construction. In: Computer Graphics and Visual Computing (CGVC), pp. 41–44 (2014). https://doi.org/10.2312/cgvc.20141206
4. Bell, N., Hoberock, J.: Thrust 1.8.1 (2015). https://thrust.github.io/
5. Binder, N., Keller, A.: Efficient stackless hierarchy traversal on GPUs with backtracking in constant time. In: Assarsson, U., Hunt, W. (eds.) Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics. The Eurographics Association, Aire-la-Ville (2016). https://doi.org/10.2312/hpg.20161191
6. Cheng, S.W., Dey, T.K., Shewchuk, J.: Delaunay Mesh Generation. CRC Press, Boca Raton (2012)
7. Doyle, M.J., Tuohy, C., Manzke, M.: Evaluation of a BVH construction accelerator architecture for high-quality visualization. IEEE Trans. Multi Scale Comput. Syst. **4**(1), 83–94 (2017). https://doi.org/10.1109/tmscs.2017.2695338
8. Garanzha, K., Pantaleoni, J., McAllister, D.: Simpler and faster HLBVH with work queues. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, pp. 59–64 (2011). https://doi.org/10.1145/2018323.2018333
9. García, A., Murguia, S., Olivares, U., Ramos, F.F.: Fast parallel construction of stack-less complete LBVH trees with efficient bit-

trail traversal for ray tracing. In: Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry, pp. 151–158. ACM (2014). https://doi.org/10.1145/2670473.2670488
10. Gu, F., Jendersie, J., Grosch, T.: Fast and dynamic construction of bounding volume hierarchies based on loose octrees. In: Vision, Modeling and Visualization (2018). https://doi.org/10.2312/vmv.20181257
11. Karras, T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In: Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics, HPG '12, pp. 33–37 (2012). https://doi.org/10.2312/EGGH/HPG12/033-037
12. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH construction on GPUs. Comput. Graph. Forum **28**(2), 375–384 (2009). https://doi.org/10.1111/j.1467-8659.2009.01377.x
13. Merill, D.: CUB 1.8.0 (2018). https://nvlabs.github.io/cub/
14. Morrical, N., Usher, W., Wald, I., Pascucci, V.: Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In: IEEE Visualization Conference (VIS) (2019). https://doi.org/10.1109/visual.2019.8933539
15. Morton, G.M.: A computer oriented geodetic data base; and a new technique in file sequencing. IBM Corporation (1966)
16. Mueller-Roemer, J.S., Altenhofen, C., Stork, A.: Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. Comput. Graph. Forum **36**(5), 59–69 (2017). https://doi.org/10.1111/cgf.13245
17. Mueller-Roemer, J.S., Stork, A.: GPU-based polynomial finite element matrix assembly for simplex meshes. Comput. Graph. Forum **37**(7), 443–454 (2018). https://doi.org/10.1111/cgf.13581
18. Murguia, S., Avila, F., Reyes, L., Garcia, A.: GPU Pro 4: Advanced Rendering Techniques, chap. Bit-trail traversal for stackless LBVH on DirectCompute, pp. 319–336. CRC Press (2013)
19. NVIDIA: NVIDIA OptiX Ray Tracing Engine (2020). https://developer.nvidia.com/optix
20. Pantaleoni, J., Luebke, D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In: Proceedings of the Conference on High Performance Graphics, HPG '10, pp. 87–95 (2010)
21. Stich, M.: Introduction to NVIDIA RTX and DirectX ray tracing (2018). https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/
22. Vaidyanathan, K., Woop, S., Benthin, C.: Wide BVH traversal with a short stack. In: Proceedings of the Conference on High-Performance Graphics (2019). https://doi.org/10.2312/hpg.20191190
23. Viitanen, T., Koskela, M., Jääskeläinen, P., Kultala, H., Takala, J.: MergeTree: a fast hardware HLBVH constructor for animated ray tracing. ACM Trans. Graph. **36**(5), 169 (2017). https://doi.org/10.1145/3132702
24. Viitanen, T., Koskela, M., Jääskeläinen, P., Tervo, A., Takala, J.: PLOCTree. Proc. ACM Comput. Graph. Interact. Tech. **1**(2), 1–19 (2018). https://doi.org/10.1145/3233309
25. Vinkler, M., Bittner, J., Havran, V.: Extended Morton codes for high performance bounding volume hierarchy construction. In: Proceedings of High Performance Graphics, HPG '17, pp. 9:1–9:8 (2017). https://doi.org/10.1145/3105762.3105782
26. Wald, I., Friedrich, H., Knoll, A., Hansen, C.D.: Interactive isosurface ray tracing of time-varying tetrahedral volumes. IEEE Trans. Vis. Comput. Graph. **13**(6), 1727–1734 (2007). https://doi.org/10.1109/tvcg.2007.70566
27. Wald, I., Usher, W., Morrical, N., Lediaev, L., Pascucci, V.: RTX beyond ray tracing: exploring the use of hardware ray tracing cores for tet-mesh point location. High Perform. Graph. Short Papers (2019). https://doi.org/10.2312/hpg.20191189
28. Zellmannn, S., Hellmann, M., Lang, U.: A linear time BVH construction algorithm for sparse volumes. In: 2019 IEEE Pacific Visualization Symposium (PacificVis), pp. 222–226 (2019). https://doi.org/10.1109/pacificvis.2019.00033
29. Zhou, K., Gong, M., Huang, X., Guo, B.: Data-parallel octrees for surface reconstruction. IEEE Trans. Vis. Comput. Graph. **17**(5), 669–681 (2011). https://doi.org/10.1109/TVCG.2010.75

Daniel Ströter received his M.Sc. in Computer Science at TU Darmstadt in 2019. He received the best thesis award of Fraunhofer Institute for Computer Graphics Research (IGD) for his master thesis in GPGPU accelerated tetrahedral mesh processing. As a PhD candidate at the Interactive Graphics Systems Group of TU Darmstadt, he continues his research efforts.

Johannes S. Mueller-Roemer joined the Interactive Engineering Technologies department of the Fraunhofer Institute for Computer Graphics Research (IGD) in 2011 after receiving his M.Sc. in Information and Media Technology (with honors and best thesis award) from BTU Cottbus. He received the doctorate in computer science (summa cum laude) from TU Darmstadt in 2019. His research interests include massively parallel GPU-accelerated geometry processing, visualization, and simulation.

André Stork is Head of the Interactive Engineering Technologies group at Fraunhofer Institute for Computer Graphics Research (IGD) and Honorary Professor of Computer Science at TU Darmstadt. His research interests are geometry processing, interactive techniques, simulation, and visualization. He has authored and co-authored more than 200 papers. He has been member of the program committee or acted as reviewer in many international conferences, workshops, and journals. He lectured "Computer Graphics III" and has lectured "Geometric Methods in CAD/CAE" at TU Darmstadt. He is a member of IEEE, Eurographics, ACM, Gesellschaft für Informatik (GI), and VDI.

Dieter W. Fellner is professor of computer science at TU Darmstadt, Germany and Director of the Fraunhofer Institute of Computer Graphics (IGD) at the same location. He is also professor at TU Graz, Austria, and he currently serves as Chairman of the Fraunhofer Information and Communication Technology Group. His research activities over the last years covered efficient rendering and visualization algorithms, generative and reconstructive modeling, virtual and augmented reality, graphical aspects of internet-based multimedia information systems, cultural heritage and digital libraries as well as visual healthcare technologies. He is a member of the Academia Europaea and a Fellow of the EUROGRAPHICS Association.