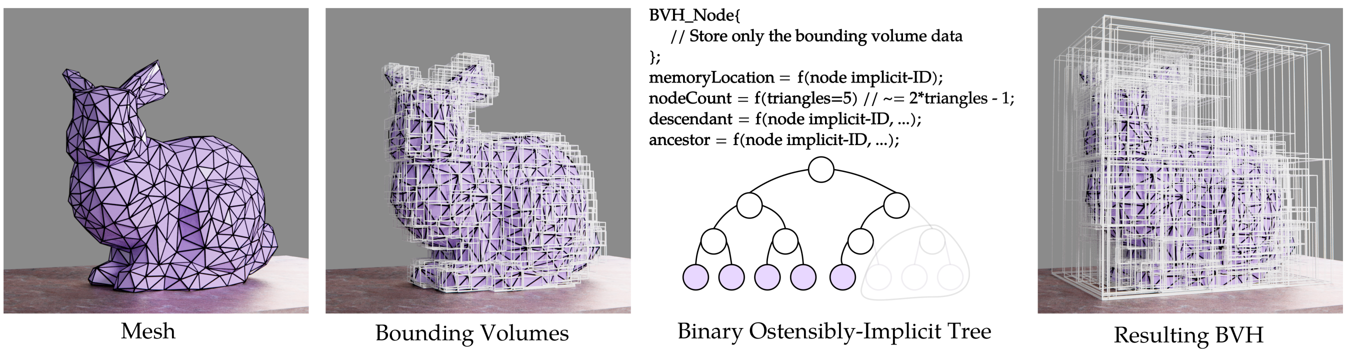
DOI: 10.1111/cgf.13948
EUROGRAPHICS 2020 / U. Assarsson and D. Panozzo
(Guest Editors)

Volume 39 (2020), Number 2

Binary Ostensibly-Implicit Trees for Fast Collision Detection

Floyd M. Chitalu Christophe Dubach Taku Komura

University of Edinburgh



```
BVH_Node{
    // Store only the bounding volume data
};
memoryLocation = f(node implicit-ID);
nodeCount = f(triangles=5) // ~= 2*triangles - 1;
descendant = f(node implicit-ID, ...);
ancestor = f(node implicit-ID, ...);
```

Figure 1: *We present a fast encoding of bounding volume hierarchies (BVH) for broad-phase collision detection with low-memory usage. Our method can generate trees supporting canonical indexing of implicit trees without the need for padding memory. We achieve this by observing that for a given number of objects, an almost-perfect binary tree can be completely determined, and the nodes missing for it to be "perfect" can be characterized through simple bit-manipulations. The figure shows a sequence where a BVH is constructed over a mesh using our novel representation. A minimal number of nodes are stored which can be indexed like the heap data structure.*

Abstract

We present a simple, efficient and low-memory technique, targeting fast construction of bounding volume hierarchies (BVH) for broad-phase collision detection. To achieve this, we devise a novel representation of BVH trees in memory. We develop a mapping of the implicit index representation to compact memory locations, based on simple bit-shifts, to then construct and evaluate bounding volume test trees (BVTT) during collision detection with real-time performance. We model the topology of the BVH tree implicitly as binary encodings which allows us to determine the nodes missing from a complete binary tree using the binary representation of the number of missing nodes. The simplicity of our technique allows for fast hierarchy construction achieving over 6× speedup over the state-of-the-art. Making use of these characteristics, we show that not only it is feasible to rebuild the BVH at every frame, but that using our technique, it is actually faster than refitting and more memory efficient.

CCS Concepts

• ***Computing methodologies*** → *Collision detection;*

1. Introduction

Computer graphics researchers have developed diverse methods for accelerating GPU-based broad-phase collision detection by constructing bounding volume hierarchies (BVHs) and evaluating their intersections by expanding bounding volume test trees (BVTT) [GS87; Eri04]. Since BVH construction and BVTT expansion are expensive operations, techniques such as BVH refitting and BVTT front tracking are widely adopted to reduce the runtime cost.

Refitting is an operation to build the BVH once or at regular intervals and then resize bounding volume extents or perform local restructuring. Notably, refitting has inherent limitations because the spatial agglomerative structure of the objects which are enclosed within the BVHs is likely to change (potentially drastically) as commonly seen with deformable objects such as cloth and volumetric simulations. Failure to sufficiently capture this spatial structure can degrade performance and worsen runtime storage costs due to an increase in the number of overlapping bounding volumes.

BVTT front tracking, which is an approach to cache the BVTT [KHM*98] between frames, can be detrimental for GPU processing because it has a high memory cost and will complicate traversal

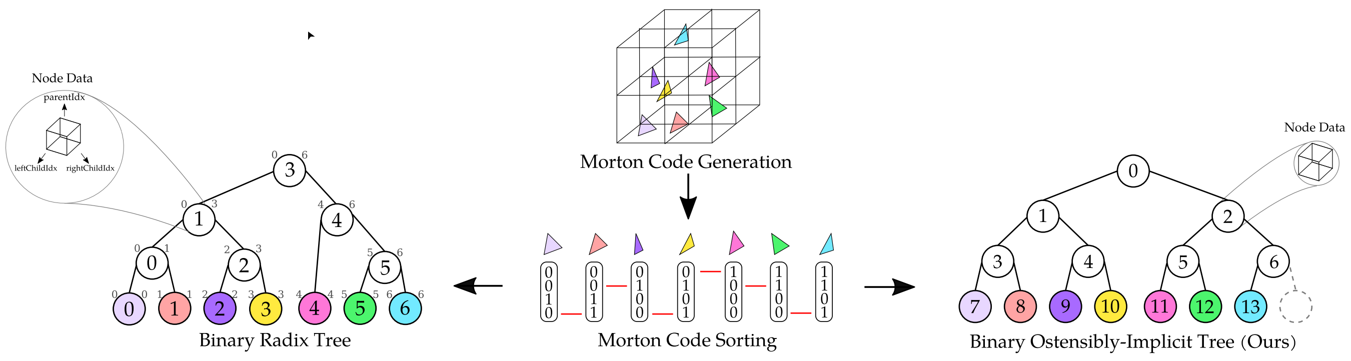


Figure 2: *Illustrative comparison with the binary radix tree (left) [Ape14]. Each leaf node is associated with a 4-bit Morton code which is in lexicographical order, and initially generated from the position of an object (e.g. triangle). Our technique (right) offers an implicit structure which is derived from the total number of objects, supporting fast indexing, and without memory padding. The explicit structure of the radix tree is encoded in the linear range of keys covered by an internal node which affects storage costs and limits construction performance.*

logic. Also, front tracking assumes that the BVH structures will remain unchanged across simulation frames - otherwise the cached fronts are invalidated by structural changes to the BVH. This assumption does not hold well for scenes involving deformable objects.

One possible way to circumvent these issues is to construct the BVHs and BVTT from scratch at every frame, without refitting or front tracking. However, the bottleneck then becomes the representations that are commonly used for BVH data structures. Most BVH-based methods on GPUs [Ape14; Kar12; WTMT18] explicitly compute and store the connectivity between nodes, which introduces indirection (see Fig. 2), and will affect the construction time due to added overheads. Aside from the fact that nodes must store this connectivity, traversing these BVH trees from one node to a descendant several levels deep requires using loop constructs and memory lookups which can significantly drop GPU performance. Alternatively, existing implicit structures, which do not require storing the connectivity, may either waste a lot of memory due to padding [CDK18], or they may not suit GPU architectures for the construction [CSE06].

We present a fast, memory-friendly, parallel broad-phase collision detection approach to construct and traverse large-scale hierarchies - which is characterised by using an implicit binary tree for final topology, and a novel way to encode this (logical) tree layout. Our method is supported by the notion of an *ostensibly-implicit* tree data structure as illustrated in Fig. 1 (middle) and Fig. 2 (right), which is a novel implicit binary tree structure specially designed to achieve fast construction and traversal on GPU architectures. In this structure, the BVHs are represented by series of implicit binary trees. The relationship between nodes can be computed by closed-form descriptions which can be implemented efficiently in hardware using fast bit-shifting operations. We also provide formulae to associate node indices with respective memory locations, which results in compact memory storage and fast access for construction and traversal. It supports fast bottom-up construction based on Morton codes, which is more suitable for modern parallel architectures compared to heap-based top-down constructions. Our method achieves a construction rate of over 4.7 billion nodes per second and is over 6× faster than the state-of-the-art solution [Ape14].

Our evaluation with the UNC dynamics benchmarking suite [CGK*09] shows that our collision detection pipeline is 1.3× faster than the state-of-the-art [WTMT18] while using 5× less memory and re-building BVHs every frame. This is achieved by sidelining the use of monolithic BVHs for the entire scene in favor of BVH-BVH tests where traversal workloads scale according to the proximity between meshes. These savings are also due to our simplified setup in which explicit BVTT front tracking is avoided to mitigate inhibitive memory costs.

1.1. Contributions

The contributions of this paper are summarised as follows:

- *Compact Implicit Tree* – We represent the BVH as a novel layout called the *ostensibly-implicit tree*, decoupling storage costs from the implicit structure and enabling fast construction (Section 3).
- *Construction* – We offer a fast $O(n)$ algorithm which maps well to GPU architectures and without complex tracking of radix key-ranges (Section 4).
- *Lightweight Collision Detection Pipeline* – We present a simple and fast broad-phase collision detection pipeline where we construct the BVH and BVTT from scratch at every frame (Section 5).

The rest of the paper proceeds as follows: After reviewing related work in Section 2, we introduce our ostensibly-implicit tree structure and how we map it to the memory in Section 3. Next, we explain how we construct a BVH based on ostensibly-implicit trees in Section 4, and then how we use them for collision detecion in Section 5. We present our experimental results in Section 6 and conclude the paper in Section 7.

2. Related Work

In this section, we first review methods for constructing BVHs in parallel. Next, we review methods based on implicit tree structures to optimise search problems in related areas, and simpler tree updates. Finally, we review GPU-based approaches for handling collision detection.

BVH Construction: Fast BVH construction is a common problem for collision detection and ray tracing [LAM06; Wal07; Ken08]. Our work shares much in common with recent efforts which focus on a multitude of acceleration strategies and trade-offs between construction time versus BVH quality. Lauterbach *et al.* [LGS*09] introduce the Linear BVH (LBVH) sorting objects along the Z-curve to facilitate partitioning and significantly improve construction time. Since its introduction LBVH has been extended numerous times and has inspired the construction algorithm presented in our work (see also [PL10; GPM11; Kar12]).

In general, fast construction is achieved with a loss in BVH quality. The BVH quality of these solutions will fall short of the gold standard making them useful especially when the number of queries is relatively small as in collision detection. Karras [Kar12] has presented a technique for depth-first ordered binary radix trees and building the entire tree in $O(n)$ time. The algorithm maps well to GPUs by addressing the shortcomings of prior methods (see e.g. [GPM11]) which generated the hierarchy sequentially for individual tree levels. Conversely, Karras [Kar12] required separate kernels to generate the hierarchy and fit bounding volumes. A bottom-up strategy is proposed by Apetrei [Ape14] which is known to be the fastest, requiring one GPU kernel to build the hierarchy and calculate bounding volume extents. The method is relatively efficient but complex, requiring an analysis of the split positions of internal nodes for establishing a connection between their indices and the ranges of Morton codes that they cover etc. Our reliance on a topologically implicit structure means that we surpass requirements to establish explicit node-connectivity which is in contrast to the approach of Karras [Kar12] and Apetrei [Ape14].

Implicit representations: Several implicit BVH representations have been proposed in literature which are related to the data structure layout that we describe. Eisemann *et al.* [EBGM12] present an implicit representation for partitioning object space to reduce storage costs similar to the bounding interval tree (BIH) [WK06]. Their BVH is implicit in the sense that node bounding volumes are inferred at runtime from a set of bounding triangles and only storing a few indices. The minimal bounding volume hierarchy (MBVH) [BEM10] is another implicit structure in the form of a full and complete binary tree for BVH compression. However, while storage per node is reduced, the total number of elements is a constant maximum $2N-1$ nodes. Conversely, we store the minimal number of BVH nodes for a given set of objects to reduce memory costs while retaining the benefits of implicitly-indexed trees.

Cline *et al.* [CSE06] present the well-related lightweight implicit BVH which is indexed like a heap. Their non-parallel solution for generating an implicit tree is done in a top-down manner by recursively splitting the leaf nodes into half - such an operation is not well suited to GPU architectures [LGS*09]. The generated tree is also less flexible since leaf nodes may not reside on the same level. In particular, the number of objects enclosed by each node must be known before the lightweight-BVH can be initialized requiring at least two 'passes' for construction - object partitioning requires that the number of objects in each internal node is known by summing the number of nodes in its children. Their approach will also calculate the total number of BVH nodes using the amortized cost of leaf nodes resulting in additional bookkeeping which will degrade the construction performance. Conversely, we only need to know the number of objects to infer the implicit tree structure. Further, our approach offers an exact closed-form solution to calculate the number of nodes given the number of objects - which can be done using trivial bit-manipulations as described in Section 3.

Simpler Tree Updates: In collision detection problems, simpler BVH update strategies such as refitting and selective restructuring are common [LAM06; LMM10; KIS*12]. This choice is motivated by speed, and in-part by the fact that these strategies are well suited for generalised front-tracking [KHM*98] which would otherwise require significant bookkeeping when BVHs are rebuilt from scratch (see e.g. Wang *et al.* [WTMT18]). However, a degradation of BVH quality is also inevitable when accumulated deformations within dynamic scenes cause significant increases in the overlap among child bounding volumes. Worse yet, in the case of breakable objects, refitting and selective restructuring are insufficient and a full reconstruction is needed.

Intermediate solutions such as Kopta *et al.* [KIS*12] use hybrid methods which heuristically track sub-trees to rebuild (see also [Ken08; Gar08]). Kopta *et al.* [KIS*12] propose a well-related incremental update scheme by combining refitting with local restructuring to modify sub-trees via rotations, and node splitting. However, they still advocate for a full rebuild when extreme degenerations occur, which has seen recent application within GPU-based collision detection [WTMT18].

Parallel Collision Detection on GPUs: Since the work of Lauterbach *et al.* [LMM10] on BVH-based broad-phase collision detection on GPUs, research has taken a number of approaches to accelerate this notoriously difficult task. Within parallel graphics, these methods range from those accelerating collision tests with the BVH, to spatial hashing schemes formulated to obviate the bounding volume test tree (BVTT) in favour of a lower memory footprint and a guaranteed worst-case number of intersecting polygon pairs (see works by [TLTM18; TWL*18; WLZ14; WDZ17]). However, these latter approaches can be limited in several ways: the grid size is an important factor in the overall performance, pipelines may need to be coupled with normal-cone culling to sustain performance (e.g. [TLTM18]), and there are restricted opportunities to exploit frame-to-frame coherence for which our (BVH) approach can be readily extended.

Though spatial hashing methods are widely explored, BVH based methods still comprise much of collision detection approaches. Spatial coherence based methods were among the first and performed broad-phase collision detection as a caching scheme with collision-fronts [LMM10; PM10]. However, these methods are also limited: GPU parallelism can only be exploited with a sufficiently large collision-front, and traversal logic relies on thread-level private work-stacks which constrain performance due to divergence between threads (see e.g. [LMM09; TMLT11]). Most no-

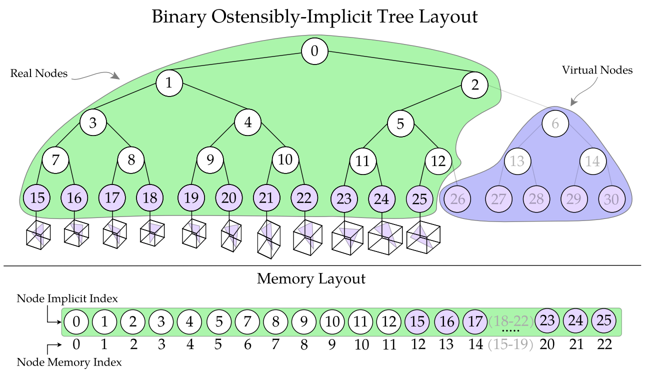


Figure 3: *Our tree representation is defined by so-called real and virtual nodes. The real nodes are the actual data (i.e. bounding volumes), and virtual nodes are simply non-existent placeholders. The layout is a* left-leaning *implicit structure, where the real nodes occupy the left-most slots on each level and are implicitly assigned a memory index.*

tably, the reliance on a collision-front for performance is memory intensive and assumes that the underlying BVH structure will remain fixed between frames, otherwise fronts are invalidated.

A number of methods offer in-part successful solutions to solving the memory problem of caching collision fronts. Tang *et al.* [TMLT11; TWT*16] have addressed the memory problem by deferring collision fronts to fit BVH node pairs in memory and propagating the BVTT in localised sets of nodes, respectively. However, these approaches curb the memory problem heuristically, they are complex, and their success-rate is not thoroughly investigated. A simpler approach is discussed in [WTMT18] which accounts for the order between BVH node pairs to cull up-to 25% of redundant self-collision tests, and is well suited for the implicit tree setting. Wang *et al.* [WTMT18] present one of the fastest BVH-based methods by ordering and restructuring the collision front and BVH, while using stackless depth-first search (DFS) traversal. Their method proposes to use a histogram sort and auxiliary data structures to reduce random data access patterns arising from front updates, while providing a quality metric to mitigate BVH degradation. Unfortunately, the benefits of restructuring are offset by its cost in scenes with large deformation and their selective restructuring of BVHs yields a complex pipeline since changes must be reflected within the collision fronts. To simplify traversal, Chitalu *et al.* [CDK18] present a non-cached front approach using implicit trees to traverse BVHs from pre-specified levels to the leaves, thereby circumventing the drawbacks of explicit collision-front tracking. Notably, their method is promising but requires implicit structures that are prohibitively expensive due to padding which we overcome.

3. The Binary Ostensibly-Implicit Tree

In this section, we introduce and describe our novel ostensibly-implicit tree layout for reducing the memory costs of implicit structures without need for post-processing to compact data (Fig. 3, Section 3.1). We also describe a mapping between the perfect implicit tree layout and ours, linking implicit index labels to actual data in memory (Section 3.2). For convenience, we assume a binary tree layout (e.g. Fig. 4), but the concept is extendable to arbitrary *arity* (see Appendix C).

3.1. Tree Layout

With a perfect binary tree layout that is full, one can completely remove all pointers and store the pointerless nodes in an array. This layout is determined by a parameter t, which could be the number of objects such as triangles. However, when t is a non-power-of-two, space still has to be allocated by introducing *virtual nodes* which accommodate for unused elements.

The heap data structure (e.g. [CSE06]) can eliminate virtual nodes but requires post-processing which will affect construction performance, and nodes may have to store additional reference data.

We resolve this problem using an implicit layout which is free from post-processing (Fig. 3) to eliminate all virtual nodes *and* explicit pointers. The idea is to produce a perfect implicit binary tree layout where the virtual nodes are brought to the right-hand-side (see Fig. 3, blue nodes), and are then encoded as a series of smaller perfect trees. With this representation, we provide an analytical form to map the remaining real nodes sequentially into the memory, and thus can minimize the memory usage to the size of the real nodes since virtual nodes are not materialised in memory.

Power-Sum Decomposition: We now describe how to decompose the number of objects t into a tree of the real nodes and a series of implicit binary trees of the virtual nodes. This decomposition is used to map implicit indices to compacted memory locations.

To intuitively illustrate the representation of our layout, observe that the residual number of leaves in an implicit binary tree which are virtual nodes is

$$L_v = 2^{\lceil \log_2 t \rceil} - t, \tag{1}$$

giving a total count of $L_c = t + L_v = 2^{\lceil \log_2 t \rceil}$ leaves, such that $\log_2(L_c) - \lceil \log_2(L_c) \rceil = 0$. Thus, the total number of nodes in the perfect binary tree will be

$$N_c = 2L_c - 1. \tag{2}$$

With N_c, we then seek to find the total number of *real* nodes

$$N_r = N_c - N_v, \tag{3}$$

where N_v is the total number of virtual-nodes (refer to Fig. 3).

We compute N_v following the observation that L_v may be expressed as a sum of powers-of-two. This observation gives a decomposition of L_v which yields a set $\mathcal{X}(L_v) = \{x \mid x = 2^y\}$, where $y \in \mathbb{N}$ and $y \leq \lfloor \log_2(L_v) \rfloor$. Specifically, we define this set by

$$\mathcal{X}(L_v) = \left\{2^{y_1}, 2^{y_2}, \ldots, 2^{y_N}\right\}, y_i \in \mathcal{Y}(L_v), \tag{4}$$

where $\mathcal{Y}(L_v) = \{y_1, y_2, \ldots, y_N\}$, such that

$$y_1 = \lfloor \log_2(L_v) \rfloor,$$
$$y_2 = \left\lfloor \log_2\left(L_v - 2^{y_1}\right) \right\rfloor,$$
$$\ldots$$
$$y_N = \left\lfloor \log_2\left(L_v - \sum_{i=1}^{N-1} 2^{y_i}\right) \right\rfloor.$$

The set $\mathcal{X}(L_v)$ is optimal in the sense that it is defined using the largest powers-of-two summing to L_v. Thus, the general analytical form for N_v given $\mathcal{X}(L_v)$ is then evaluated by

$$N_v = \sum_{k=1}^{N} 2x_k - 1, \quad x_k \in \mathcal{X}(L_v), \tag{5}$$
$$= 2\left(\sum_{k=1}^{N} x_k\right) - N$$

which will evaluate N_v as a finite sum of perfect implicit-tree sizes containing only virtual nodes as shown in Fig. 3. $N = |\mathcal{X}(L_v)|$ is the cardinality of the set $\mathcal{X}(L_v)$, representing the total number of powers of two which sum to L_v.

Binary Encoding: Our approach so far offers a general solution requiring several steps in order to evaluate the total number of real nodes N_r by first determining the number of virtual nodes N_v. We now describe a practical implementation utilizing bit-wise operations to refactor these formulas as simple and fast one-line calculations. In particular, we extensively rely on a function `count_set_bits` to count the number of non-zero bits in a given integer's binary representation.

Thus, in practice we evaluate Eq. (5) by

$$N_v = 2L_v - \texttt{count\_set\_bits}(L_v), \tag{6}$$

following a key observation that the i-th non-zero bit, $0 \leq i$, in the binary representation of L_v uniquely identifies a corresponding sub-tree of virtual nodes with 2^i leaves. This sub-tree will have $2 \times 2^i - 1$ nodes. Consequently, by summing over all set bits we arrive at the solution. Also, from Eq. (2), (3) and (6), the exact total number of real node nodes in the tree is

$$N_r = 2t - 1 + \texttt{count\_set\_bits}(L_v). \tag{7}$$

We use these solutions to map the implicit index of each node to a unique memory location as described next (Section 3.2).

3.2. Mapping Implicit Indices to Memory Locations

We now describe a method to compute a mapping between the implicit index of a real node and the location in memory where it is stored - providing a complete solution for generalised pointer-less traversal with *zero indirection*. We use the term "implicit index" to refer to the numerical label given to each node in the perfect tree in breadth first search (BFS) order as shown in Fig. 4.

For a given real node, its location in memory is determined by its implicit index, depth level, and the number of virtual leaves in its tree as described in Section 3.1. To define our memory mapping, let i be the implicit index of a real node which is at level

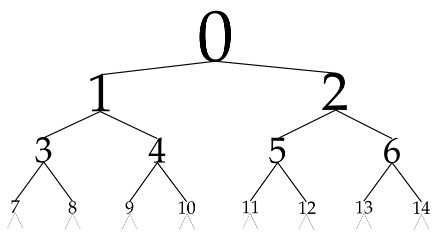


Figure 4: *A perfect binary tree which is full and complete with implicit-indices (labels) following a pre-order traversal pattern i.e. breadth-first search labelling. Our algorithm assumes this layout where leaf nodes occupy the deepest level.*

$l_i = \lfloor \log_2(i+1) \rfloor$, $(0 \leq l_i \leq \bar{l})$, where $\bar{l} = \lceil \log_2 t \rceil$ is the leaf level. Further, let

$$L_{vl} = \left\lfloor \frac{L_v}{2^{\bar{l}-l}} \right\rfloor \equiv L_v \gg (\bar{l} - l) \tag{8}$$

be the number of virtual nodes at level l due to the consecutive and approximate halving of the number of virtual nodes at each level when moving up tree from $\bar{l}$ by $\bar{l} - l$ levels, where $\gg$ is the bitwise right-shift operator. Thus, the memory location of i is computed by

$$i_m = i - N_{vl}, \tag{9}$$

where

$$N_{vl} = 2L_{vl} - \texttt{count\_set\_bits}(L_{vl}), \tag{10}$$

which is similar to Eq. (6), but with L_{vl} computed as in Eq. (8) using $l = l_i - 1$.

Intuitively, our goal in Eq. (9) is to account for the number of virtual nodes above l_i from which a memory location can be determined given i (thanks to BFS labelling). Eq. (9) provides a seamless solution for bridging between the perfect implicit tree (Fig. 4) and our layout Fig. 3. The solution is simple and fast (due to bitwise encoding) offering an indirection-free description of data layout in memory.

With these properties, our layout is particularly attractive since it is compact by eliminating the nuances of explicit and/or padded tree structures. The node data (i.e. the 'payload') is smaller compared to the case of including child (and parent) pointers, or when the tree really is fully padded (or perhaps just a few nodes off on the short side from being full). Also, a level l, $0 \leq l$ is only completely filled with real nodes iff $2^l < \left\lfloor 2(t-1) / 2^{\lfloor \log_2 t \rfloor} \right\rfloor$. This is in contrast to data structures such as the heap in which all levels, except possibly the last, are filled.

4. BVH construction

We now describe a method to construct a BVH using the ostensibly-implicit tree layout. The basic idea of our approach is to utilize Morton order [Mor66] and a specific node layout (which is implicit in our case) to establish a mapping between GPU threads and BVH nodes. Here, we lay emphasis on a GPU implementation since our target application is parallel collision detection, but the method is easily extendable to other implementations (e.g. single or multi-threaded CPU). In this approach, we simplify and extend Apetrei's method [Ape14], but mapping *entire* GPU thread-groups to sub-trees and without explicit tracking of radix-key ranges. Section 4.1 first provides a high-level perspective on how the layout is derived from the number of objects. We then describe the algorithm and two implementations in Section 4.2.

4.1. Hierarchy construction

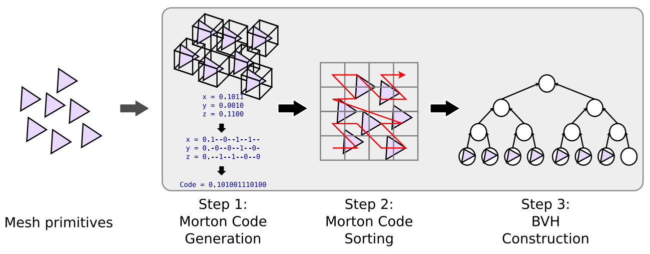


Figure 5: *BVH construction pipeline.*

The summary of the construction process of the hierarchy is shown in Fig. 5. The objects (triangles in our case) are assigned to the leaf nodes, their bounding volumes are computed then and Morton codes are computed based on their center's 3D positions such that spatially adjacent nodes are given closer codes on the Z-curve. Leaf nodes are then sorted using the corresponding Morton codes as in [LGS*09]. Next, we walk up the tree one level at a time processing internal nodes until reaching the root. We have used the parallelisation strategy by Karras [Kar12] but extended to further maximise parallelism and guarantee optimal usage of local shared memory while ensuring $O(n)$ complexity (thanks to the implicit layout). We now describe this process.

4.2. GPU Kernel Implementation

Given the sequence of objects sorted according to their Morton code, we construct the BVH while saving only bounding volumes to memory.

Algorithm Steps: Algorithm 1 provides a general outline to construct an ostensibly-implicit BVH (multi-kernel version illustrated in Fig. 6). We assume that internal-nodes and leaf-nodes are stored separately since leaf bounding boxes are already computed during Morton code evaluation. Threads start from a unique node on the *construction entry-level*, which is the first level processed when the kernel starts - executing as many threads as there are real nodes on this entry-level. Each thread then walks up the tree computing the parent node, and memory location as described in Eq. (9) (see also: lines 21 and 22). Additional indexing parameters, such as relative positions, sub-tree level etc., are inferred directly using implicit indices and thread IDs.

A group of threads is mapped to a sub-tree which is processed independently from the rest (line 1). A subtree is assigned to a group based on the given size and ID of the group thanks to the implicit layout (see Fig. 6). A group is defined by mapping threads to nodes

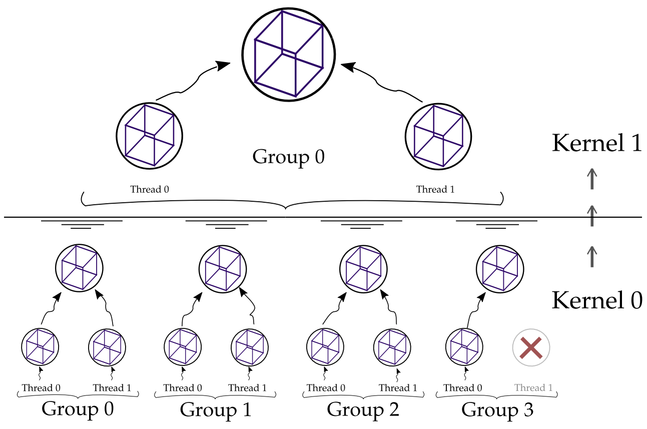


Figure 6: *Parallel tree construction where groups of threads are mapped to independent sub-trees.*

ALGORITHM 1: High-level ostensibly-implicit BVH construction

```
Input  : tIntArr - internal-node bounding-box array
Input  : tEntryLev - level from which to begin aggregation
Input  : meshFaceCount - number of faces in input mesh
Input  : tLeafArr - mesh-order real-leaf bounding-box array
Output: tIntArr
[1]  parallelfor foreach group do
[2]      tLevPos = global_id // $\in [0,t)$
[3]      tNode = $(2^{\mathrm{tEntryLev}} - 1)$ + tLevPos
[4]      if tEntryLev == tLeafLev - 1 then
[5]          lBB = get_leftchild_bbox(tLeafArr, tNode, ...)
[6]          if rightChildIsReal then
[7]              rBB = get_rightchild_bbox(tLeafArr, tNode, ...)
[8]          end
[9]      else
[10]         lBB = get_leftchild_bbox(tIntArr, tNode, ...)
[11]         if rightChildIsReal then
[12]             rBB = get_rightchild_bbox(tIntArr, tNode, ...)
[13]         end
[14]     end
[15]     tNodeBB = merge(lBB, rBB)
[16]     write_bounding_box(tIntArr, tNodeBB, ...)
[17]     // sub-tree root level
[18]     tLevMin = tEntryLev $-\log_2$(group_size)
[19]     tLev = tEntryLev
[20]     while tLev $\geq$ tLevMin do
[21]         tLevPos = global_id/$2^{\mathrm{tEntryLev\text{-}tLev}}$
[22]         tNode = $(2^{\mathrm{tLev}} - 1)$ + tLevPos
[23]         if rightChildReal AND firstThreadToReach(tNode) then
[24]             terminate()
[25]         end
[26]         lBB = get_left_child_bv(tArr, tNode, ...)
[27]         if rightChildReal then
[28]             rBB = get_right_child_bv(tIntArr, tNode, ...)
[29]         end
[30]         tNodeBB = merge(lBB, rBB)
[31]         write_bounding_box(tIntArr, tNodeBB, ...)
[32]         tLev = tLev - 1
[33]     end
[34] endparallelfor
```

of a subtree on the entry-level (lines 4 to 14) before proceeding to iteratively compute bounding volumes at higher levels (lines 20 to 33). When the entry-level is the second-last level of the tree, a thread will process its node by reading the array of leaf bounding volumes using the sorted triangle-IDs at relative positions determined by the thread global-ID. Otherwise, the thread will access the bounding volumes of the left and right child (which are internal nodes) in order to process current node. For operations that are localised to a group of threads, we also utilise local shared memory to effectively cache the computed bounding volumes - permitting fast access when processing the next level, which is guaranteed until the subtree root node is processed.

Each internal node is processed by exactly one thread by using atomic operations (line 23) to synchronise bounding volume updates. Threads are terminated if they are first to reach a node which is not on the entry level and has a right child - otherwise they remain active. The active thread will proceed to evaluate this node and continue until termination or reaching the root of the subtree.

Implementation: We propose two implementations for our construction algorithm distinguished by how they synchronise threads using GPU global memory. The first is the multi-kernel implementation following a bulk-synchronous parallel (BSP) approach [MGG12] to synchronise threads using *only* local atomic operations when processing nodes. Global barriers (e.g. multiple kernel launches) synchronize thread-groups by unifying communication and storage after the sub-tree root is processed as shown in Fig. 6. This approach favours building large trees (e.g. more than 2^{17} triangles as shown in Fig. 8). The second implementation is single-kernel construction (similar to Karras [Kar12]) which also uses one group-thread to update the sub-tree root node. However, nodes above the sub-tree are processed using global atomic operations to synchronise threads from different groups to build the entire tree in one kernel. Single-kernel construction is to be most useful with relatively smaller meshes where the overhead of global atomics is negligible due to having less demanding parallel workloads in terms of global memory accesses.

GPU Scheduling: By knowing the maximum size of a group $g_{\texttt{user}}$, $(2 \leq g_{\texttt{user}} \leq 2^{\lceil \log_2 t \rceil})$, the height of the subtree in each kernel $\log_2(g_{\texttt{user}})$ can be determined, and thus we can compute the total number of kernels to schedule, as well as the configured groups of threads for each kernel, including the total number of GPU threads, thread-group size, and the number of real nodes at the construction entry-level. These parameters can be computed as soon as the number of objects t is known (e.g. during the initialization time on the CPU). The readers are referred to Appendix B for the details of the implementation.

4.3. Summary

As seen in this section, the GPU thread will require only the implicit index of the node to determine a path to the root thanks to the implicit layout. Further, our approach guarantees all synchronisation between threads in a group to be done using only local atomics which will reduce overhead. In particular, all the memory locations are directly determined from the implicit representation. In contrast, state-of-the-art methods [LGS*09; PL10; GPM11; Kar12; Ape14] require tracking radix-key ranges as a part of the bottom-up reduction and using them to deduce the index of parent nodes. This requires additional memory accesses which inevitably leads to lower performance as our experimental results will show.

5. BVH Traversal for Collision Detection

As a target application, we describe how our data structure can assist parallel collision detection. In contrast to *refitting* approaches, where one must forego full BVH maintenance to enable collision-front tracking, our approach allows one to maintain up-to-date BVHs of a given scene, knowing that the underlying polygons will be sufficiently captured and at minimal cost. *Broad-phase* collision detection is particularly important since it serves to cull the search space of costly polygon intersection tests.

Chitalu et al. [CDK18] describe a simple but fast method for simultaneously traversing multiple BVHs on GPUs. In their approach BVH data is accessed like the heap but extended to allow arbitrary jumps to descendants for maximising GPU workloads. Thus, we adopt their algorithm and extend it using the ostensibly-implicit tree layout with further improvement to BVH traversal. Traversal is accelerated with the BSP model [MGG12], decomposing the task into a series of iterative *level-synchronous* kernels (see Fig. 7).

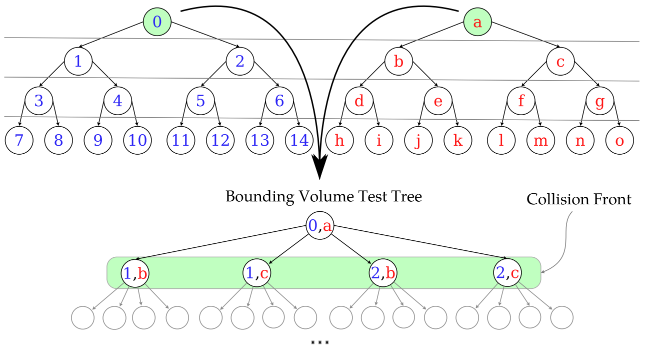


Figure 7: *Simultaneous BVH tree traversal. Pairwise collision detection is performed with multiple BVH-BVH tests at the same time producing one collision-tree.*

Quick Access to Ancestors and Descendants Quick access to ancestors and descendants are essential operations for BVH traversal. The implicit tree structure provides analytical solutions for accessing the ancestors, descendants and siblings in $O(1)$ time. See Appendix A for the details.

Expanding the BVTT We perform explicit BVH-BVH tests and corresponding BVH levels (and henceforth, the BVTT) are explored before the next. In our representation, a BVTT node is a pair of integers which encode a BVH ID and an implicit index to form a node descriptor, so that the node data can be quickly accessed and tested for further intersections. Each kernel will map threads to unexplored parts of the resulting BVTT in an input queue.

The BVTT is managed within GPU global memory and used as the input for next kernel, which simplifies the operation as threads can be mapped to a small fixed number of work elements which are evaluated to produce new BVH node pairs that will be processed by the next kernel. Compared to front tracking [LMM10; TMLT11; WTMT18], less data is marshalled in and out of global memory (Fig. 10) because the average BVTT sprouting size for each tested pair of BVH nodes is less than a factor of 2^{2n}, where n is the depth-step (jumping) parameter (Appendix A). The resulting BVTT computation can be done very efficiently even when starting from the root, and performs better than BVTT front tracking as shown in our experimenal results (Section 6).

6. Experiments and Results

In this section we present the results of our methods which are implemented using OpenCL with platform version "OpenCL 1.2 CUDA 9.1.84". Experiments are performed on a system with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and an NVIDIA GeForce GTX 1080 @ 1733MHz equipped with 8GB of GDDR5X VRAM. We first evaluate the performance of our approach for fast BVH construction in Section 6.1. We then evaluate our method in collision detection scenarios and compare against the state-of-the-art in Section 6.2.

6.1. BVH Construction Performance and Comparison

We evaluate the performance of our construction algorithm where triangles are assumed to be already sorted.

Table 1 provides a breakdown of BVH construction time and compares against a well-known fast-construction method by Apetrei [Ape14] which is implemented in CUDA. Our construction algorithms are faster than Apetrei [Ape14]. At best, we achieve over 6.5× speedup over this state-of-the-art method, and averaging 5× across the evaluated datasets. At worst, we achieve 4.17× speedup on the Happy Buddha mesh dataset, which is significant given that this is our lowest score. Thus, our proposed method maps well to GPUs, offering a simpler and faster alternative for categorically fast BVH construction.

A comparison is also made against the naïve perfect implicit binary-tree BVH which has padded nodes. Although the performance is similar, our new layout saves up-to 48.1% of memory on the evaluated datasets which is significant because real-world meshes can require large BVHs where the size is just a few nodes off on the short side from being full. We conceivably occur some overhead during node index translation but it is a reasonable assumption that the impact is negligible: Calculation requires only a few arithmetic instructions (plus e.g. popcount), and without additional reads from a table. Thus, our approach is efficient by storing an optimal number of nodes and with minimal overhead.

Performance Scaling: The overall scaling of our BVH construction performance in terms of BVH nodes constructed per second is shown in Fig. 8. We analysed scalability by evaluating construction time using a gradually refined mesh, from 2^{10} to 2^{22} triangles. Our construction algorithms yield high throughputs, reaching a rate of at-least 4.7 billion BVH nodes per second. This experiment also reveals that ostensibly-implicit tree construction scales optimally and retains faster execution time than Apetrei [Ape14] with 3.2–6.2× speedup (averaging 4.5×). Our speedup is highest with large meshes, where the number of threads is sufficient to saturate the hardware.

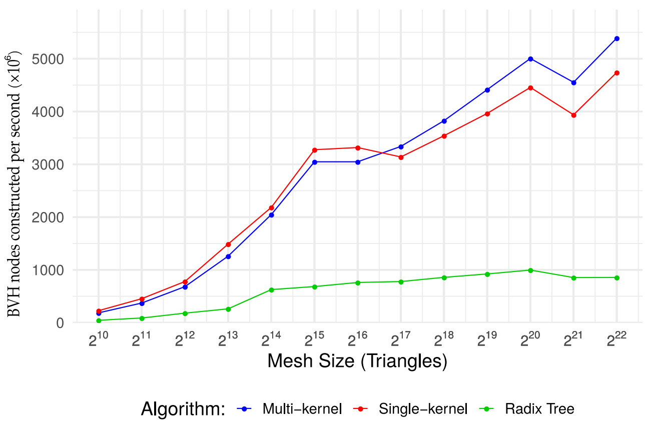


Figure 8: *BVH construction performance.*

In relative terms, our construction algorithms yield competitive performance where the multi-kernel version is approximately 7% faster than the single-kernel implementation. Fig. 8 reveals that a crossing-point in throughput between the two algorithms is reached when using a mesh with approximately 2^{17} triangles. Our single-kernel algorithm is 1.15× faster below this threshold, but saturates the hardware with lower throughput due to the reliance on global atomics which increase with the number of triangles. The multi-kernel version is 1.22× faster above the threshold since local atomics amortize the cost of global barrier synchronisation.

6.2. Collision Detection Performance Comparison

In this section, we compare our method to two other techniques for handling collision detection - including broad-phase and narrow-phase. We show that our approach is faster across a number of benchmarks. Comparisons on worst-case runtime memory usage are also discussed. In all results shown, our BVHs are re-built from scratch at every frame.

Comparison against Wang *et al.* [WTMT18]: Table 2 summarizes our collision detection performance using datasets from the UNC dynamics benchmarking suite [CGK*09], and compares with Wang *et al.* [WTMT18] (see also Fig. 9).

Speed: We compare against their speed using BVH refitting and front tracking. Our method is up to 30% faster which is significant since we reconstruct our BVHs from scratch at every frame (Note: our speedup is 4× when they re-build BVHs every frame). At best the state-of-the-art BVH based techniques build the BVH once and simply refit at every frame, which degrades the overall efficiency over time. With our approach however, not only it is feasible to rebuild the BVH at every frame, but our technique is actually faster than refitting.

	Bunny			Erato			Dragon			Happy Buddha			Hairball		
Builder	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)	**Nodes**	**Padding** (%)	**Time** (ms)
Perfect Implicit BVH	524287	45.05	0.14	1048575	21.2	0.24	2097151	16.9	0.42	4194303	48.1	0.9	8388607	31.3	1.6
Apetrei [Ape14]	288091	0	0.40	825337	0	1.10	1742611	0	2.29	2174947	0	2.92	5759999	0	8.50
Oi-BVH (Single-kernel)	288100	3.1e-3	0.074	825344	8.4e-4	0.25	1742621	5.7e-4	0.55	2174957	4.5e-4	0.83	5760004	8.6e-5	1.37
Oi-BVH (Multi-kernel)	288100	3.1e-3	0.077	825344	8.4e-4	0.23	1742621	5.7e-4	0.5	2174957	4.5e-4	0.7	5760004	8.6e-5	1.3
Speedup: Oi-BVH vs. Apetrei	-	-	5.4×	-	-	4.78×	-	-	4.58×	-	-	4.17×	-	-	6.53×

Table 1: *Results for BVH construction (time is in milliseconds), with a comparison between our two proposed implementations (single-kernel and multi-kernel) and the radix-tree BVH by Apetrei [Ape14]. Speedup is calculated using our fastest algorithm over Apetrei [Ape14]'s total time to compute the hierarchy topology and fit bounding boxes. Models are sourced from the McGuire Computer Graphics Archive [McG17].*

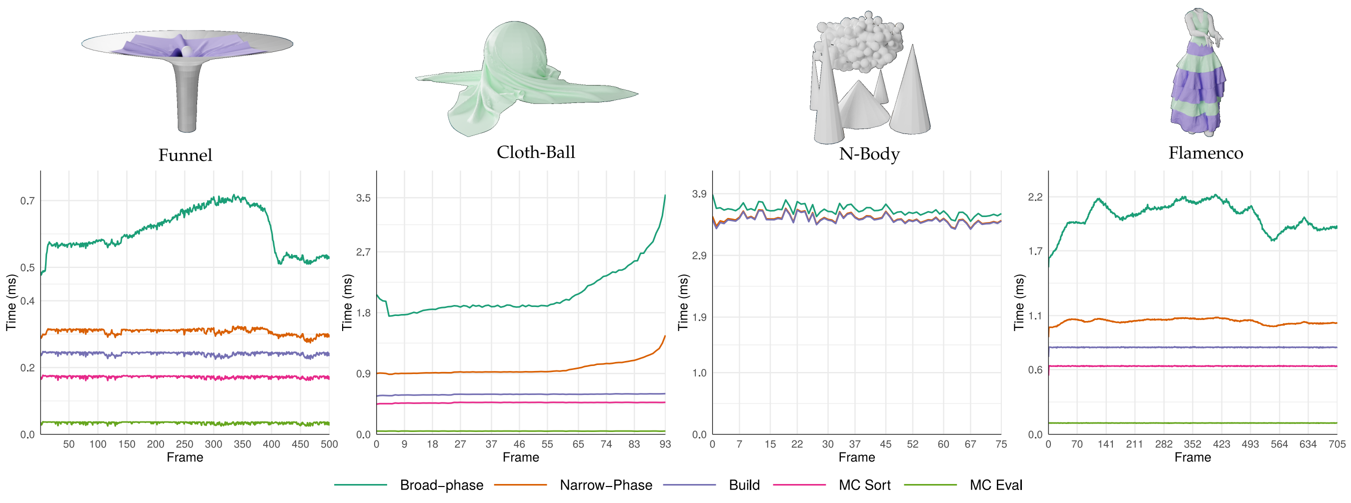


Figure 9: *Collision detection time using datasets from the UNC dynamics benchmarking suite [CGK*09]. Full BVH construction occurs every frame for the datasets with self-collisions (Funnel, Cloth-Ball and Flamenco) by including Morton-code evaluation (MC Eval) and sorting (MC Sort). We perform only refitting (Build) with N-body for all 305 objects because it is rigid-body dataset. We have used the same implementation as gProximity [LMM10] for narrow-phase collision detection which is also adopted by Wang* et al. *[WTMT18]. The Broad-phase component highlights our simultaneous BVH traversal execution time to find the set of potentially colliding pairs which are forwarded onto the narrow-phase.*

				Wang *et al.* [WTMT18]		Ours		Comparison	
Benchmark	**Triangles**	**Objects**	**Frames**	**Time** (stdev)	**Runtime Memory** (mb)	**Time** (stdev)	**Runtime Memory** (mb)	**Performance Speedup**	**Memory Usage** (%)
Cloth-Ball	92k	2	94	2.3 (1.33)	145.46	1.9 (0.34)	85	1.2×	58.4
Funnel	19k	4	500	0.6 (0.16)	58.6	0.6 (0.06)	8.53	1×	14.5
N-Body	142k	305	75	4.3 (0.24)	296.85	3.62 (0.07)	15.1	1.18×	5
Flamenco	49k	10	705	2.59 (0.52)	130.37	2 (0.1)	24.2	1.29×	18.57

Table 2: *Execution time comparison (milliseconds) of our collision detection (broad + narrow phase) with Wang* et al. *[WTMT18].*

.

Memory: When performing broad-phase collision detection, front-tracking will explicitly cache BVH node pairs where traversal stops - managing such a scheme consumes a lot of memory, especially when interaction between the objects are intense. Fig. 10 shows a comparison of average BVTT size against Wang *et al.* [WTMT18] for the same benchmarks used in Table 2. Our approach can reduce the BVTT size by up-to 97.7% (see: N-Body benchmark), making our approach simpler, faster and more memory efficient.

Pipeline analysis: In general, BVH construction and traversal, which are the focus of this paper, occupy most of the execution pipeline. Fig. 9 shows a breakdown of total collision detection time for the results presented in Table 2. We execute the full BVH construction pipeline on datasets with self-collisions (MC Eval, MC Sort and Build) which takes 28-40% of the total time. With the N-body dataset, BVH construction accounts for over 95% of the total execution time surmounting to 3.4ms. While the individual rigid-bodies are small (approximately 1024 triangles per mesh), the quantity leads to an overall degradation in performance because BVHs are constructed sequentially. Nonetheless, the total execution time for N-Body is below 4ms which is faster than Wang *et al.* [WTMT18]. BVH Traversal (broad-phase) accounts for approximately 48-52% of the execution time on the self-collision datasets leading to large workloads arising from the tested BVH node-pairs. For N-Body, traversal accounts for approximately 5% of the total time because it is a rigid body simulation (no self-collisions) and mesh sizes are small. In general, traversal is largely affected by mesh configurations in each frame relative to the density of the dataset under consideration. On the other hand, the cost of construction is largely dependent on the number of meshes.

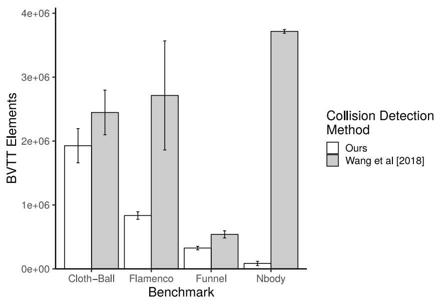
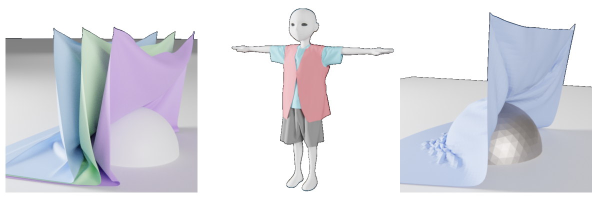


Figure 10: *Average BVTT size across frames.*

Comparison against I-Cloth [TWL*18]: Table 3 presents the results of our method applied to larger datasets from the I-Cloth benchmark suit by Tang *et al.* [TWL*18] (see Fig. 11). Our method is on average 59.8× faster than I-Cloth, achieving up to 97.7× on the largest dataset which is Bridson-3, with 198k triangles. We compare against their publicly available CUDA source code with measured average timings per-frame. These measurements are obtained with `nvprof` and `nvidia-smi` tools, where we compare specifically against their collision detection kernels and without including the execution time for resolving collisions. Table 3 shows a comparison of runtime memory costs with I-Cloth. We compared against memory figures which are a 25% proportion of the values reported by the `nvidia-smi` tool, making our measurements estimations due to source code access restrictions. Our method performs collision detection using less than 10% memory relative to I-Cloth (actual GPU RAM used). Note that the I-Cloth source-code simply provides a `C++` interface to pre-compiled libraries.



(a) *Bridson-3* **(b)** *Andy* **(c)** *Bridson*

Figure 11: *I-Cloth benchmarks (source: [TWL*18])*

7. Discussion

The implicit tree is a technique used in computer graphics which is usually ideal when a BVH is perfectly balanced or the node payload size is relatively small. We have presented an adaptation of BVH representation in memory which is characterised by using the implicit binary tree for final topology, and a novel way to encode the (logical) layout. Thus, our approach improves the generality, efficiency and scalability of classical implicit trees through a novel encoding of their structure using simple bitwise manipulations. With this adaptation, BVH traversal and construction are performed while assuming an implicitly defined structure, but we store nodes compactly in memory, and without post-processing (where the storage cost scales linearly with the number of objects). We demonstrated the advantages by comparing against the state-of-the-art for GPU-based collision detection. We also presented a fast construction algorithm which when combined with our simple collision detection pipeline enabled a decoupling of performance from BVTT front tracking and BVH refitting. Consequently, our pipeline is able to perform collision detection in a reasonably short time - with BVH construction occurring every frame.

Our tree layout requires a small number of redundant nodes to retain the benefits of implicit trees. The layout requires 'support' nodes with only one child (e.g. node 12, Fig. 3) to maintain the implicit structure, which is a side-effect of moving virtual nodes to the right, and placing all leaves at the lowest level. However, in the worst-case ($t - 2^{\lfloor \log_2 t \rfloor} = 1$), storage costs are only approximately $N_r = N_c \times (.5 + \varepsilon)$, where

$$\varepsilon = \frac{\log_2(t)}{N_c} \equiv \frac{\texttt{count\_set\_bits}(L_v)}{N_c} \tag{11}$$

accounts for the number of support nodes. Thus, our ostensibly-implicit layout is most effecient when t is just a few increments off on the short side from creating a full tree, reducing memory costs by up-to approximately 50%.

Limitations & Future Work: The tree layout, as currently defined, is constrained to labelling nodes using breadth-first search

				I-Cloth [TWL*18]		Ours		Comparison	
Benchmark	**Triangles**	**Objects**	**Frames**	**Time**	**Runtime Memory** (mb)	**Time** (stdev)	**Runtime Memory** (mb)	**Perf' Speedup**	**Mem' Reduction** (%)
Andy	127k	4	80	121.58	498.5	2.65 (0.02)	41.16	45.87×	91.7
Bridson	18k	2	867	17.93	456.25	0.5 (0.02)	6.62	35.86 ×	98.54
Bridson-3	198k	4	842	392.97	534.75	4.02 (0.17)	85.4	97.7×	84.02

Table 3: *I-Cloth [TWL*18] benchmark execution time in milliseconds (see also Fig. 11).*

.

(BFS), thus alternative depth-first search (DFS) optimizations cannot be applied during traversal. DFS is particularly useful when optimizing for cache coherent data accesses as cache lines will be potentially filled with nodes on the traversal path to the leaves. In choosing BFS, we favour traversal patterns that span the search across neighbouring branches to maximise workloads. In future work, we wish to bridge these layouts (and others, like the van Emde Boas layout [EB75]) for the benefit of cache efficiency since BFS may not fit all applications.

As a categorically "fast-construction" approach, our technique emphasizes build performance and simplicity which can limit BVH quality by a considerable amount in some cases. We achieve performance by simply pairing nodes at each level and without looking at the actual radix bit values to construct the hierarchy. This is indeed fast for tree construction, but fails to account for adjacent pairs of triangles in a sorted list that are spatially far apart. The alternative LBVH [Ape14] produces trees which consider this spatial adjacency, and thus the quality is better as presented in Table 4. We also conduct an experiment where we gradually move one triangle away from the original mesh (see Fig. 12) to examine how the quality of the BVHs by our method and those by LBVH varies with respect to the distance of the separate triangle (from 0 to 10 times the model diameter). The ratio of the SAH by the two methods are plotted in Fig. 13. As expected, the SAH cost of our method grows much faster compared to the LBVH, which reveals the weakness of our approach. Therefore, we trade BVH quality for simplicity and construction performance, which is fast but requires further consideration for building good quality trees.

Scene (#tris)	LBVH ([Ape14])	Oi-BVH (ours)	SAH Cost Increase
Happy Buddha (1087k)	86	229.16	-2.66×
Hairball (2880k)	669.8	1122.34	-1.67×
Dragon (873k)	77.43	201.54	-2.6×

Table 4: *Comparison of surface area heuristic (SAH) with the LBVH [Ape14]. We compute SAH using Eq. 1 in [AKL13].*

Acknowledgements

We thank all the anonymous reviewers for their constructive feedback. This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

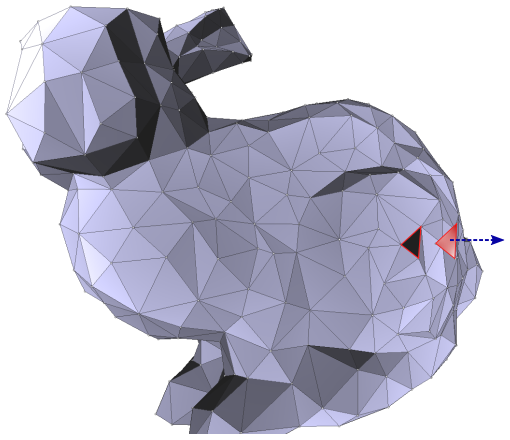


Figure 12: *Fig. 13 experiment setup: We move a single triangle away from the rest of a given mesh by a multiple of the bounding box diagonal in the normal direction.*

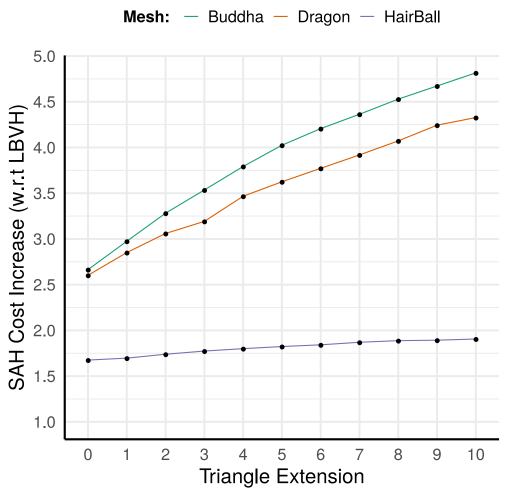


Figure 13: *A plot of the SAH cost increase for our technique which is calculated relative to the LBVH [Ape14] (see also Table 4). Our setup is illustrated in Fig. 12, where the x-axis represents a triangle's offset multiple as it is moved away from the rest of the mesh.*

References

[AKL13] AILA, TIMO, KARRAS, TERO, and LAINE, SAMULI. "On Quality Metrics of Bounding Volume Hierarchies". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, 101–107. ISBN: 978-1-4503-2135-8 11.

[Ape14] APETREI, CIPRIAN. "Fast and Simple Agglomerative LBVH Construction". *Computer Graphics and Visual Computing (CGVC)*. Ed. by BORGO, RITA and TANG, WEN. The Eurographics Association, 2014. ISBN: 978-3-905674-70-5 2, 3, 6–9, 11.

[BEM10] BAUSZAT, PABLO, EISEMANN, MARTIN, and MAGNOR, MARCUS. "The Minimal Bounding Volume Hierarchy". *Vision, Modeling, and Visualization (2010)*. 2010. ISBN: 978-3-905673-79-1 3.

[CDK18] CHITALU, FLOYD M., DUBACH, CHRISTOPHE, and KOMURA, TAKU. "Bulk-synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs". *Proc of I3D*. 2018, 4:1–4:9. ISBN: 978-1-4503-5705-0 2, 4, 7.

[CGK*09] CURTIS, SEAN, GOVINDARAJU, NAGA, KABUL, ILKNUR, et al. *UNC Dynamic Scene Benchmarks*. 2009. URL: http://gamma.cs.unc.edu/DYNAMICB/ 2, 8, 9.

[CLRS09] CORMEN, THOMAS H., LEISERSON, CHARLES E., RIVEST, RONALD L., and STEIN, CLIFFORD. *Introduction to Algorithms, Third Edition*. 3rd. 2009. ISBN: 0262033844, 9780262033848 13.

[CSE06] CLINE, DAVID, STEELE, KEVIN, and EGBERT, PARRIS. "Lightweight Bounding Volumes for Ray Tracing". *Journal of Graphics Tools* 11.4 (2006), 61–71 2–4.

[EB75] Van EMDE BOAS, P. "Preserving Order in a Forest in Less Than Logarithmic Time". *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*. SFCS '75. Washington, DC, USA: IEEE Computer Society, 1975, 75–84 11.

[EBGM12] EISEMANN, M., BAUSZAT, P., GUTHE, S., and MAGNOR, M. "Geometry Presorting for Implicit Object Space Partitioning". *Comput. Graph. Forum* 31.4 (June 2012), 1445–1454. ISSN: 0167-7055 3.

[Eri04] ERICSON, CHRISTER. *Real-Time Collision Detection*. 2004. ISBN: 1558607323, 9781558607323 1.

[Gar08] GARANZHA, K. "Efficient clustered BVH update algorithm for highly-dynamic models". *2008 IEEE Symp on Interactive Ray Tracing*. 2008, 123–130 3.

[GPM11] GARANZHA, KIRILL, PANTALEONI, JACOPO, and MCALLISTER, DAVID. "Simpler and Faster HLBVH with Work Queues". *Proc of HPG*. Vancouver, British Columbia, Canada, 2011, 59–64. ISBN: 978-1-4503-0896-0 3, 7.

[GS87] GOLDSMITH, J. and SALMON, J. "Automatic Creation of Object Hierarchies for Ray Tracing". *IEEE Computer Graphics and Applications* 7.5 (1987), 14–20. ISSN: 0272-1716 1.

[Kar12] KARRAS, TERO. "Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees". *Proc of HPG*. Paris, France, 2012, 33–37. ISBN: 978-3-905674-41-5 2, 3, 6, 7.

[Ken08] KENSLER, A. "Tree rotations for improving bounding volume hierarchies". *2008 IEEE Symp on Interactive Ray Tracing*. 2008, 73–76 3.

[KHM*98] KLOSOWSKI, JAMES T., HELD, MARTIN, MITCHELL, JOSEPH S. B., et al. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs". *IEEE Trans on Vis and Comp Graph* 4.1 (1998), 21–36. ISSN: 1077-2626 1, 3.

[KIS*12] KOPTA, DANIEL, IZE, THIAGO, SPJUT, JOSEF, et al. "Fast, Effective BVH Updates for Animated Scenes". *Proc of I3D*. Costa Mesa, California, 2012, 197–204. ISBN: 978-1-4503-1194-6 3.

[LAM06] LARSSON, THOMAS and AKENINE-MÖLLER, TOMAS. "A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection". *Comput. Graph.* 30.3 (June 2006), 450–459. ISSN: 0097-8493 3.

[LGS*09] LAUTERBACH, C., GARLAND, M., SENGUPTA, S., et al. "Fast BVH Construction on GPUs". *Computer Graphics Forum* 28.2 (2009), 375–384 3, 6, 7.

[LMM09] LAUTERBACH, CHRISTIAN, MO, QI, and MANOCHA, DINESH. "Work distribution methods on GPUs". 2009 3.

[LMM10] LAUTERBACH, C., MO, Q., and MANOCHA, D. "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries". *Computer Graphics Forum* 29.2 (2010), 419–428 3, 8, 9.

[McG17] MCGUIRE, MORGAN. *Computer Graphics Archive*. 2017. URL: https://casual-effects.com/data 9.

[MGG12] MERRILL, DUANE, GARLAND, MICHAEL, and GRIMSHAW, ANDREW. "Scalable GPU Graph Traversal". *SIGPLAN Not.* 47.8 (Feb. 2012), 117–128. ISSN: 0362-1340 7.

[Mor66] MORTON, GEORGE. "A computer oriented geodetic data base and a new technique in file sequencing". 1966 5.

[PL10] PANTALEONI, J. and LUEBKE, D. "HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry". *Proc of HPG*. Saarbrucken, Germany, 2010, 87–95 3, 7.

[PM10] PAN, JIA and MANOCHA, DINESH. "GPU-Based Parallel Collision Detection for Real-Time Motion Planning". en. *Algorithmic Foundations of Robotics IX*. Springer Tracts in Advanced Robotics 68. 2010, 211–228. ISBN: 978-3-642-17451-3 978-3-642-17452-0. (Visited on 12/15/2015) 3.

[TLTM18] TANG, MIN, LIU, ZHONGYUAN, TONG, RUOFENG, and MANOCHA, DINESH. "PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs". *Proc. ACM Comput. Graph. Interact. Tech.* 1.1 (July 2018), 18:1–18:18. ISSN: 2577-6193 3.

[TMLT11] TANG, MIN, MANOCHA, DINESH, LIN, JIANG, and TONG, RUOFENG. "Collision-Streams: Fast GPU-based collision detection for deformable models". *Proc of I3D*. San Fransisco, CA, 2011, 63–70 3, 4, 8.

[TWL*18] TANG, MIN, WANG, TONGTONG, LIU, ZHONGYUAN, et al. "I-cloth: Incremental Collision Handling for GPU-based Interactive Cloth Simulation". *ACM Trans. Graph.* 37.6 (2018), 204:1–204:10. ISSN: 0730-0301 3, 10, 11.

[TWT*16] TANG, MIN, WANG, HUAMIN, TANG, LE, et al. "CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation". *Computer Graphics Forum* 35.2 (2016), 511–521 4.

[Wal07] WALD, INGO. "On Fast Construction of SAH-based Bounding Volume Hierarchies". *Proc of the 2007 IEEE Symp on Interactive Ray Tracing*. RT '07. 2007, 33–40. ISBN: 978-1-4244-1629-5 3.

[WDZ17] WELLER, RENÉ, DEBOWSKI, NICOLE, and ZACHMANN, GABRIEL. "kDet: Parallel Constant Time Collision Detection for Polygonal Objects". *Comput. Graph. Forum* 36.2 (May 2017), 131–141. ISSN: 0167-7055 3.

[WK06] WÄCHTER, CARSTEN and KELLER, ALEXANDER. "Instant Ray Tracing: The Bounding Interval Hierarchy". *Proc of EGSR*. Nicosia, Cyprus, 2006, 139–149. ISBN: 3-905673-35-5 3.

[WLZ14] WONG, TSZ HO, LEACH, GEOFF, and ZAMBETTA, FABIO. "An Adaptive Octree Grid for GPU-based Collision Detection of Deformable Objects". *Vis. Comput.* 30.6-8 (June 2014), 729–738. ISSN: 0178-2789 3.

[WTMT18] WANG, XINLEI, TANG, MIN, MANOCHA, DINESH, and TONG, RUOFENG. "Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring". *Computer Graphics Forum (Proc of Eurographics 2018)* 37.2 (2018) 2–4, 8–10.

Appendix A: Bi-directional Implicit Binary Tree Exploration

Here we describe a scheme to access ancestor and descendant nodes within a pointer-less implicit tree in $O(1)$ time, which are operations used during BVH construction and traversal. We assume that nodes are labelled in BFS order (see Fig. 4).

Given an implicit tree that is full, the immediate relatives of a node with an implicit index i, $(0 \leq i)$ are computed by $parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$, $child(i,j) = 2i + j, (1 \leq j \leq 2)$. To allow for arbitrary indexing, we compute an n-th generation descendant j of a node i by

$$j = 2^n i + 2^n - 1 + k, \quad 0 \leq k < 2^n, \tag{12}$$

where k is the relative position of j w.r.t the leftmost descendant. Eq. (12) shall reduce to i as $n \to 0$ thereby satisfying the *min-heap property* [CLRS09], where $parent(i) \leq i$ for every node i other than the root since it has the lowest index. The inverse relation to determine the node i as the n-th generation ancestor of j is obtained from Eq. (12) by

$$i = \frac{1}{2^n}(j - k + 1) - 1, \tag{13}$$

which shall reduce to j as $n \to 0$.

Appendix B: BVH Construction Scheduling Parameters

Given the total number of real leaf nodes and the preferred number of threads in a group, we show that the remaining parameters needed to run multi-kernel construction can be pre-computed for seamless batch scheduling. These *scheduling parameters* are computed by

$$r_{k+1} = \left\lceil \frac{r_k}{g_k} \right\rceil \tag{14}$$

$$t_{k+1} = \frac{t_k}{g_k} \tag{15}$$

$$g_{k+1} = \begin{cases} g_k & \text{if} \quad g_k \leq t_{k+1} \\ 2^{\lfloor \log_2(r_{k+1}) \rfloor} & \text{otherwise.} \end{cases} \tag{16}$$

For each kernel k: r_k is the total number of real nodes at the corresponding entry level; t_k is the total number of threads; and g_k is the number of threads per group.

Initial parameter values are set either by the user or depending on configurations. As we assume that each leaf node stores one triangle, r_1 is the number of triangles in the BVH being constructed. Next, $g_1 = \min(g_{\texttt{user}}, L_c)$ is set by the user, and with the condition that $g_{\texttt{user}}$ is a power of two. Our condition on g_1 ensures that we can calculate $t_1 = g_1 \left\lceil \frac{r_1}{g_1} \right\rceil$ to allow seamless mapping of thread-groups to subtrees which have leaves whose total is a power of two.

Appendix C: κ-ary Trees

An extension to κ-ary ostensibly implicit trees (κ = 2, 3, 4, etc.) is briefly described in this section, which is similar to descriptions given in Section 3.1 and Section 3.2. We start by defining a set $\mathcal{X}_\kappa(L_v) = \{\kappa^{y_1}, \kappa^{y_2} \ldots, \kappa^{y_N}\}, y_i \in \mathcal{Y}_\kappa(L_v)$, where $\mathcal{Y}_\kappa(L_v) = \{y_1, y_2, \ldots, y_N\}$, such that

$$\begin{aligned} y_1 &= \lfloor \log_\kappa (L_v) \rfloor \\ y_2 &= \lfloor \log_\kappa (L_v - \kappa^{y_1}) \rfloor \\ &\ldots \\ y_N &= \left\lfloor \log_\kappa \left(L_v - \sum_{i=1}^{N-1} \kappa^{y_i} \right) \right\rfloor . \end{aligned}$$

The total number of virtual nodes in the tree will then be (cf. Eq. (5))

$$N_v = \sum_{i=1}^{N} \left(\frac{\kappa x_i - 1}{\kappa - 1} \right), \quad x_i \in \mathcal{X}_\kappa(L_v), \tag{17}$$

and memory locations are computed as in Eq. (9) but with

$$L_{vl} = \left\lfloor \frac{L_v}{\kappa^{\bar{l} - l}} \right\rfloor, \tag{18}$$

as the general form of Eq. (8) for an implicit tree with κ children per node.