DOI: 10.1111/cgf.13356

EUROGRAPHICS 2018 / D. Gutierrez and A. Sheffer
(Guest Editors)

Volume 37 (2018), Number 2

Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring

Xinlei Wang [1] Min Tang [1,3] Dinesh Manocha [2] and Ruofeng Tong [1]

[1]Zhejiang University, China
[2]University of North Carolina at Chapel Hill, America
[3]Alibaba-Zhejiang University Joint Institute of Frontier Technologies

Abstract

Bounding volume hierarchy (BVH) has been widely adopted as the acceleration structure in broad-phase collision detection. Previous state-of-the-art BVH-based collision detection approaches exploited the spatio-temporal coherence of simulations by maintaining a bounding volume test tree (BVTT) front. A major drawback of these algorithms is that large deformations in the scenes decrease culling efficiency and slow down collision queries. Moreover, for front-based methods, the inefficient caching on GPU caused by the arbitrary layout of BVH and BVTT front nodes becomes a critical performance issue. We present a fast and robust BVH-based collision detection scheme on GPU that addresses the above problems by ordering and restructuring BVHs and BVTT fronts. Our techniques are based on the use of histogram sort and an auxiliary structure BVTT front log, through which we analyze the dynamic status of BVTT front and BVH quality. Our approach efficiently handles inter- and intra-object collisions and performs especially well in simulations where there is considerable spatio-temporal coherence. The benchmark results demonstrate that our approach is significantly faster than the previous BVH-based method, and also outperforms other state-of-the-art spatial subdivision schemes in terms of speed.

CCS Concepts

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

1. Introduction

Collision detection (CD) has usually been a major performance bottleneck in physically-based computer simulations. The simulator needs to check for not only potential primitive overlaps between pairs of objects but also numerous self-collisions within each model. This challenge is also encountered in other fields like haptics, robotics, and manufacturing.

Generally, a CD pipeline consists of several levels of filtering processes, typically a broad-phase step followed by a narrow-phase step. The broad-phase step coarsely excludes large parts of primitives that cannot collide according to their bounding volumes (BVs) and generates a potential collision set [Wel13] to be checked for exact intersection in the narrow-phase step. In this paper, we only focus on the broad-phase CD.

There are two main approaches used in broad-phase CD: spatial subdivision and object partition. Spatial subdivision algorithms subdivide the space into cells. Many auxiliary data structures have been researched, including binary space partitioning (BSP) trees (including k-d trees), grids, hierarchical spatial hashing tables, etc [Wel13]. Bounding volume hierarchy (BVH) is the primary choice in terms of object partition approaches. It is essentially a tree-like structure where each node is associated with a subset of primitives of objects enclosed by a specific type of bounding volume.

Many parallel algorithms have been proposed, which exploit the benefits of GPU architectures for better performance. Several spatial subdivision methods have been developed to accelerate CD. [WLZ14] proposed an adaptive octree grid method *OTG*, in which a two-stage scheme is used to improve octree subdivision, removing a considerable number of broad-phase tests, and thus adequately addresses the two main issues in previous spatial subdivision methods: uneven triangle sizes and uneven triangle spatial distributions. [WDZ17] designed a novel CD algorithm *kDet* with a linear complexity based on a geometric predicate that revealed not only the topology of the mesh but also its volumetric configuration. They used hierarchical spatial hashing for GPU implementation. To the best of our knowledge, both of the above methods reported the fastest performance results on their respective GPUs.

Among BVH-based techniques, the primary issue is the maintenance of BVHs. In the context of CD, the construction speed is generally preferred over tree quality, therefore linear bounding volume hierarchy (LBVH) [LGS*09] is a more suitable option than other high-quality BVHs like surface area heuristic (SAH) BVHs [GS87, MB90]. With the rise of general-purpose GPU-computing,

it has become possible for data structures like BVHs, octrees, and k-d trees to be built upon millions of primitives in real-time [Kar12]. Nevertheless, parallel construction of BVHs is still at least one order of magnitude slower than just refitting especially in complex scenes. In fact, refitting is the optimal operation for BVH maintenance as long as the BVH retains its culling efficiency.

However, refitting is not enough for most dynamic scenes consisting of deformable objects. Therefore, many selective restructuring approaches, which trade tree quality for maintenance speed, have been researched and developed. [LAM06] selected degenerated subtrees based on the ratio of a parent's bounding box volume to the sum of its children's, re-splitting them later in the CD query phase. [OCSG07] used AVL trees for the implementation of BVHs and realized restructuring operations through tree edge rotations and grandchildren permutations. [YCM07] used a cost/benefit evaluation of the culling efficiency of ray intersection tests to restructure pairs of nodes, while [Gar08] searched for nodes whose children undergo large motion. These two approaches use multiple phases to identify candidates for restructuring before reconstruction. [HSK*10] developed a restructuring method in their CD algorithm for large-scale fracturing models based on a culling efficiency metric that measured the expected number of overlap tests of a BVH. Unfortunately, none of the above restructuring algorithms has been proven portable or efficient on GPU. [KIS*12] added a single rotation iteration, performing local restructuring to each frame's refit phase with only a small increase in processing overhead, and thus improved the quality of the trees. They implemented the algorithm on GPUs in a bottom-up fashion, with the drawback being that the parallelism only exists within nodes at the same level. Recent methods [KA13, DP15] restructured treelets recursively as an optimization of BVH quality for better ray tracing performance, but the overhead is still relatively high for CD.

Another practical issue is the memory layout of BVH nodes. [YM06] assessed the runtime access patterns of BVHs based on localities using a probabilistic model and computed an optimized layout that reduced cache misses. It showed that cache locality can be substantially improved by reordering nodes in the BVH according to a certain access pattern. [NPK*10] later introduced an ordered depth-first layout that suits ray tracing systems well. We demonstrate that a depth-first layout (see Figure 2.3) can be calculated quickly and is efficient in collision detection.

There is usually a certain degree of spatio-temporal coherence between successive frames in physically-based simulated environments. The traversal path of non-overlapping nodes of a bounding volume test tree (BVTT), i.e. BVTT front [TTSD06], is also present. Keeping track of the BVTT fronts enables the collision detection to skip highly unnecessary bounding volume overlap tests. [LC98] first came up with an incremental scheme that takes advantage of spatial coherence to accelerate CD, which is also known as generalized front tracking [EL01]. It was later used in a GPU-based streaming algorithm by [TWT*16, TMLT11, DZPW15], which to our knowledge demonstrates the best CD performance among BVH-based methods so far.

There are other optimization techniques for specific types of deformation, specialized models like hair [Sob05], or necklaces [GNRZ02], or with specific presentations like reduced models

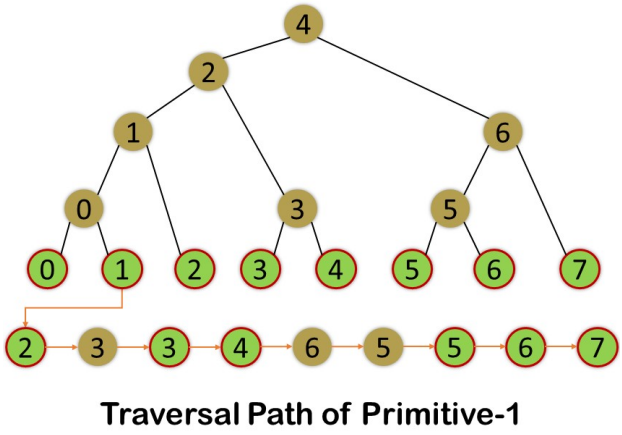


Figure 1: *Traversal path of stackless self-collision detection of primitive-1 without BVTT front, assuming that the primitive-1 overlaps with all other primitives in this BVH.*

[JP04]. Those peculiar restrictions limit the usefulness to a much smaller subset of applications, and may not work well in terms of dealing with more general models and deformations. Much effort has also been put into reducing redundant self-collision tests [SPO10, WLT*17]. These algorithms worked well for flat materials like cloth and can be integrated into our scheme as well.

Motivated by the above issues and techniques, we investigate the optimization of memory layouts of BVHs and BVTT fronts, and present a novel, BVH-based broad-phase collision detection scheme on GPU. We make the following contributions:

- **An efficient computation method for ordering**
  Instead of performing an expensive comparison sort on BVH and BVTT front, we build our ordering scheme on top of histogram sort and make adjustments to compute a cache-friendly layout for these nodes in parallel. As a result, our CD traversal gains significant speedup. Additionally, this technique is integrated with our maintenance operations (including restructuring) and results in considerable speedup. This procedure is at least two orders of magnitude faster than other common GPU sorting algorithms (including radix sort) and the overhead is very small.

- **An adaptive front-based collision detection scheme**
  Conventional front-based collision detection algorithms lack sensitivity to BVH quality degenerations. We develop a framework that can periodically detect such variations through *BVTT front log*, which contains the updated statistics of the BVTT front, and can also perform efficient restructuring. Our quality detection method is more correlated with front-based CD performance than inspecting the status of BVHs due to our novel quality metric, and therefore achieves efficiency and robustness.

2. Related Work

In this section, we give an overview of previous work most related to our approach, including fast construction of LBVHs, stack-less BVH traversal and generalized BVTT front tracking.

Linear BVH Construction: BVHs are now widely used in various computer graphics applications because of their low memory footprint and flexibility in being adapted to model deformation. Unlike ray tracing, collision detection tends to trade tree quality for faster construction. [LGS*09] first came up with a parallel algorithm for rapidly constructing linear BVH on many-core GPUs by first sorting the primitives along a space-filling curve using Morton codes. [Kar12] further improved this technique by eliminating the serialization caused by constructing BVH level by level, and this method scales well with number of GPU cores. [Ape14] continued optimization by implementing LBVH construction in a single kernel that performs construction and refitting. In this paper, we adopt [Ape14] for fast construction of LBVHs and design our restructuring algorithm in a similar manner.

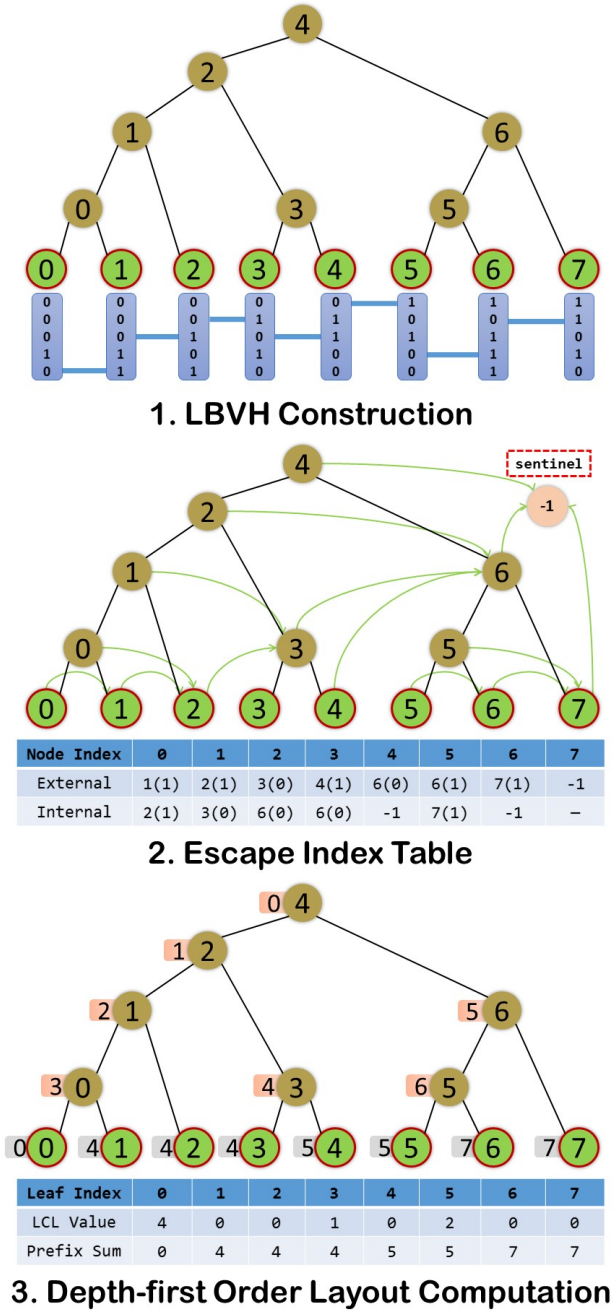
Stack-less Hierarchy Traversal: Traditionally, a stack (depth-first order) or a queue (breadth-first order) is needed for CD depending on a certain traversal rule. In GPU-based implementations, the storage of these structures is usually declared as arrays inside kernels that reside in local memory, and eventually end up in the DRAMs, resulting in a bandwidth-limited application.

Another problem in self-collision detection is that there exists one corresponding duplicate for each collision pair that is the result of switching its two components if no order between the two components of a pair is specified. To eliminate redundant pairwise BV tests, the BV of every primitive should be checked against BVs only enclosing primitives with greater (or smaller) indices.

The stack-less hierarchy traversal technique handles the above problems well. The first stack-less hierarchy traversal algorithm was proposed by [NT05] in the context of ray tracing, and used a forward pointer called *escape index* (see Figure 2.2). [Dam07] later extended it to handle collision detection between two BVHs. In their experiments, several predictable algorithms with corresponding stack-less versions were evaluated in scenarios with different setups in terms of performance, scalability and robustness. Though the dynamic stack algorithm performs best in most cases due to the heuristic descending strategy (BV with larger volume traversed first), its traversal rule is unpredictable so the stack cannot be omitted. Furthermore, the lack of parallelism makes it hard to be ported to GPU. In our observation, the leaf algorithm lends itself best to the GPU parallel architecture, and we use this method to generate the BVTT front and gather collision pairs.

They also introduced the concept of *left child levels value* (LCL value) indicating how many straight left children there are above the node (see Figure 2.3), which is essential to our implementation of BVH ordering as discussed in Section 3. The resulting memory layout of our BVH is similar to its pre-indexed tree (depth-first order) except that the storage of external nodes (i.e. leaves) is separated from internal nodes.

BVTT Front-based Traversal: There is extensive research on exploiting spatio-temporal coherence for faster collision detection. We refer the reader to the literature [TTSD06, TMT10]. The maintenance of BVTT fronts is composed of two operations: *sprouting* (BVTT front node replaced by its descendants) and *pruning* (a set of BVTT front nodes replaced by their common ancestor covering no extra BVTT nodes). See Algorithm 1 for more details.



Node Index	0	1	2	3	4	5	6	7
External	1(1)	2(1)	3(0)	4(1)	6(0)	6(1)	7(1)	-1
Internal	2(1)	3(0)	6(0)	6(0)	-1	7(1)	-1	–

Leaf Index	0	1	2	3	4	5	6	7
LCL Value	4	0	0	1	0	2	0	0
Prefix Sum	0	4	4	4	5	5	7	7

Figure 2: *BVH Computation*
1. LBVH built upon the ordered primitives by [Ape14]. The index of each internal node is that of the external node which splits its range into two halves, i.e. the rightmost leaf index of its left children.
2. The escape indices of all BVH nodes used for stackless traversal. The integer within the braces denotes whether it is the index of an external node, 1 means true, 0 otherwise. All escape indices falling out of the BVH are marked as -1.
3. Sort all internal nodes in a depth-first traversal order (specified in the pink boxes). Note that prefix sum computation uses the exclusive scan.

Algorithm 1: Conventional BVTT front-based collision detection

```
1  Function maintain(PrevFront, NextFront, CPs):
2      for front(i,j) ∈ PrevFront do
           // acquire BV of primitive i
           // primitive i maps to BVH
              external (leaf) node i
3          PrimBV ← BoundingVolume(i)
           // acquire BV of BVH node j
4          BvhBV ← BoundingVolume(j)
5          if Overlap(PrimBV, BvhBV) then   // sprout
6              sprout(i, j, NextFront, CPs)
7          else                             // prune
8              prune(i, j, NextFront)
9          end
10     end
11 return
12 Function sprout(PrimIndex, BvhNodeIndex, Front,
     CPs):
13     if IsLeaf(BvhNodeIndex) then
14         CPs.PushPair(PrimIndex, BvhNodeIndex)
15     end
16     PrimBV ← BoundingVolume(PrimIndex)
17     SearchIndex ← LeftChild(BvhNodeIndex)
       // stack-less depth-first CD
          traversal
18     while SearchIndex ∈ SubBVH(BvhNodeIndex) do
19         BvhBV ← BoundingVolume(SearchIndex)
20         while !IsLeaf(SearchIndex) and
21         Overlap(PrimBV, BvhBV) do
22             SearchIndex ← LeftChild(SearchIndex)
23         end
           // front nodes are classified into
              external and internal front
              according to their second
              indices
24         Front.PushNode(PrimIndex, SearchIndex)
25         if IsLeaf(SearchIndex) and
            Overlap(PrimBV, BvhBV) then
26             CPs.PushPair(PrimIndex, SearchIndex)
27         end
28         SearchIndex ← EscapeIndex(SearchIndex)
29     end
30 return
31 Function prune(PrimIndex, BvhNodeIndex, Front):
32     PrimBV ← BoundingVolume(PrimIndex)
33     Child ← BvhNodeIndex
34     Father ← Parent(Child)
35     while IsLeftChild(Father) and
36     Overlap(PrimBV, BoundingVolume(Father))
        do
37         Child ← Father
38         Father ← Parent(Child)
39     end
40     if IsLeftChild(Father) or
41     Overlap(PrimBV, BoundingVolume(Father))
        then
42         Front.PushNode(PrimIndex, Child)
43     end
44 return
```

A BVH-based CD pipeline using the incremental scheme was proposed by [TMLT11, TWT*16]. In their GPU-streams, the collision pipeline involves a one-time preprocessing stage and a run-time stage for collision queries. The preprocessing stage is decomposed into the construction of BVHs (one for clothes and one for obstacles) and the initialization of BVTT fronts (one for cloth/obstacle CD and one for cloth self CD), each containing one BVTT node composed of two root nodes of corresponding BVHs. During the run-time stage, BVHs are refitted and CD is performed through a front-based traversal. Note that their application disabled the *pruning* operator in most benchmarks for faster front-based CD and periodically performed total reconstruction, in case of degenerate scenarios.

In this paper we extend the work of [TMLT11]. A stack-less depth-first order traversal algorithm like the leaf algorithm [Dam07] is used as a substitute for the original stack-based algorithm [TMLT11] in both inter- and intra-object collision queries. Also, instead of pushing just one BVTT node into the front and then expanding it to the full extent in a single thread, our front is directly generated from checking all primitive BVs against the whole BVH in parallel. Our method also supports a *pruning* operator in our regular maintenance cycle so that the front retains CD efficiency.

3. Ordered BVH-based Collision Detection

In this section we examine the issues that limit the performance of conventional BVH-based collision detection pipelines. We observe that low memory bandwidth efficiency and divergent parallel executions are priority problems within existing parallel CD algorithms, and accordingly propose a lightweight ordering scheme that sorts all the nodes in BVHs and BVTT fronts in a cache friendly layout for CD traversal.

3.1. A Simple Workaround

In a BVH-based CD pipeline, all input primitives of each object are sorted according to their Morton codes, then an internal hierarchy is built upon these primitives [Ape14] before collision queries are performed. The resulting node layout of each BVH is displayed in Figure 2.1. Afterwards, all collisions associated with this BVH (including self collision detections) are tested by checking for intersections between primitive BVs and the BVH, recursively. A typical traversal path of a self collision BV test is marked in Figure 1. It is obvious that a lot of cache misses happen during the depth-first order BVH CD traversal, thus greatly reducing memory bandwidth efficiency.

However, reorganizing the BVH nodes in a depth-first layout can substantially increase the cache hit rate as no related BVH nodes will be fetched from DRAMs into the cache more than once for each individual primitive. For many-core architectures like GPUs, consecutive threads checking intersections between spatially adjacent (due to primitive sortings) primitive BVs and the same BVH tend to share a similar traversal path, letting DRAMs work close to the peak global memory bandwidth. In fact, matching the BVH layout to its access pattern is an optimization practice of the *Coalesced Global Memory Access* technique, because global memory is accessed in chunks of aligned 32, 64, or 128 bytes and cached

in L1/L2. Furthermore, the *structure of array* (SoA) layout is enforced on BVH nodes to avoid bandwidth waste. This improvement not only significantly speeds up BVH-based CD, but also lays the foundation of our efficient front-based CD. Check all BVH related queries in lines 3, 4, 16, 17, 18, 19, 20, 22, 25, 28, 32, 34, 35, 36, 38, 40, 41 in Algorithm 1.

Previous GPU implementations of BVTT front maintenance generally used atomic add instruction to insert BVTT nodes into front (see lines 24, 42 in Algorithm 1) during *sprouting* and *pruning*. Consequently, all newly built nodes are stored randomly in a compacted array. As line 2 in Algorithm 1 suggests, all these nodes are actually independent and each corresponds to a fine-grained task which is to be scheduled on GPU. More specifically, each front node, i.e. *front(i, j)*, indicates that its thread should traverse the BVH from node *j*. Because the count of front nodes is numerous, cache miss is also a serious issue for front-based CD.

Fortunately, layout optimization can be used to improve the performance. Assume adjacent GPU threads start searching from the same BVH node or consecutive BVH nodes and that BVH nodes are also in depth-first layout, then the same segment of BVH nodes residing within a small region of global memory will usually be concurrently accessed, no matter what operator is executed (*sprouting* or *pruning*).

To achieve the above cache friendly layout, a naive yet common method is to sort all the front nodes according to their second components in ascending order by comparison. However, computation overhead of this method is rather high for a real-time application. Instead, we propose an auxiliary structure *order log*, composed of a *count log* and an *offset log*, to realize a histogram sort on both BVH nodes and front nodes. The central idea of our ordering scheme is as follows:

- **1.** Produce the elements and count the number of elements inside each segment at the same time. Counts are stored in *count log*.
- **2.** Compute prefix sums of these counts in *offset log*. They work as the starting positions of each segment.
- **3.** Group all the elements into their corresponding segments by looking up their keys.

Since the counting in Step 1 is lightweight and Step 3 is inevitable for any sorting algorithm, the performance gap between this method and others exists in Step 2, specifically the prefix sum computation, whose complexity is linear with the number of segments. Fortunately, its parallel implementation is fast enough that its overhead is almost negligible.

3.2. Cache-Friendly BVH Layout Computation

We notice that during depth-first traversal, the left boundaries of the traversed internal nodes, i.e. the indices of external nodes, are strictly in non-descending order. Therefore, each external node is treated as a segment and the index of each internal node's left boundary is used as a reference to its corresponding segment. Moreover, the positions of internal nodes within each segment from top to bottom should be in strict ascending order so that the order matches the exact depth-first order.

Our BVH layout computation starts with counting LCL values of external nodes (see Figure 2.3) in *count log*. Note that the LCL values in our scheme are off-by-one compared to those in [Dam07], because external nodes and internal nodes of the BVH are stored separately. This is primarily due to the LBVH construction algorithm [Ape14] which handles these nodes in two dependent phases. Thus, only internal nodes need ordering. Another motivation is related to BVTT front classification as detailed in Subsection 3.3.

After being computed within construction kernel, LCL values are exclusively-scanned into the *offset log*. Before going into Step 3, the mapping between original indices and ordered indices of internal nodes (the numbers in the pink boxes shown in Figure 2.3) are calculated by each individual segment (leaf node) from top to bottom using *offset log*. Finally, all internal nodes are scattered in the ordered array with their tree topology links (parent handle, left/right child handle) updated. The resulting layout suits our stackless depth-first collision queries very well.

3.3. Cache-Friendly and Less Divergent BVTT Front

Conventionally, the BVTT front stores all front nodes in a single mega array, and all collision queries initiated from the front begin with accessing random BVH nodes. This obviously results in a huge number of cache misses. As described in Subsection 3.1, ordering *front(i, j)s* by their second components largely addresses this issue and reduces the memory latency (see pipeline **a** in Figure 4). However, this simple workaround does not take *Thread Divergence*, more precisely *branch divergence* and *loop divergence*, into account. Since the hardware serializes different execution paths (due to branch divergence) inside each warp (a group of threads running concurrently on a multiprocessor), and idles the whole warp until the thread with the highest iteration count (due to loop divergence) finishes, there can be an unexpected performance drag as a result of the animation and collision status.

In our collision detection scheme, BVTT front is only enabled when there exists enough temporal coherence between successive frames; hence the front in the previous frame resembles the one in the current frame. As for each individual front node, the workload in *sprout* or *prune* correlated with coherent motion is therefore comparatively small. Even if a certain amount of BVH degeneration happens and causes adaptive front update, the depth-first layout optimization of front nodes can partially counteract this negative impact, because all the front nodes that need considerable maintenance share the same BVH node index *j* and are therefore aggregated. On the whole, loop divergence is within our tolerance.

Branch divergence is a more critical issue. As shown in Algorithm 1, branch divergence happens whenever consecutive threads adopt different maintenance operations at lines 5, 7, both of which are essential functions for a robust and adaptive framework. Yet, we can alleviate this negative impact by simply categorizing the front nodes *front(i, j)s* according the their second components. Those *front(i, j)s* with *j* denoting an external (leaf) node of a BVH are grouped in an *external front*, otherwise, they result in an *internal front*. This design tactic is based on the fact that, for every front node in the *external front*, its sprout function generates next to zero overhead because, there are no child nodes left to traverse. Even if the thread detects an intersection and enters the sprout branch,

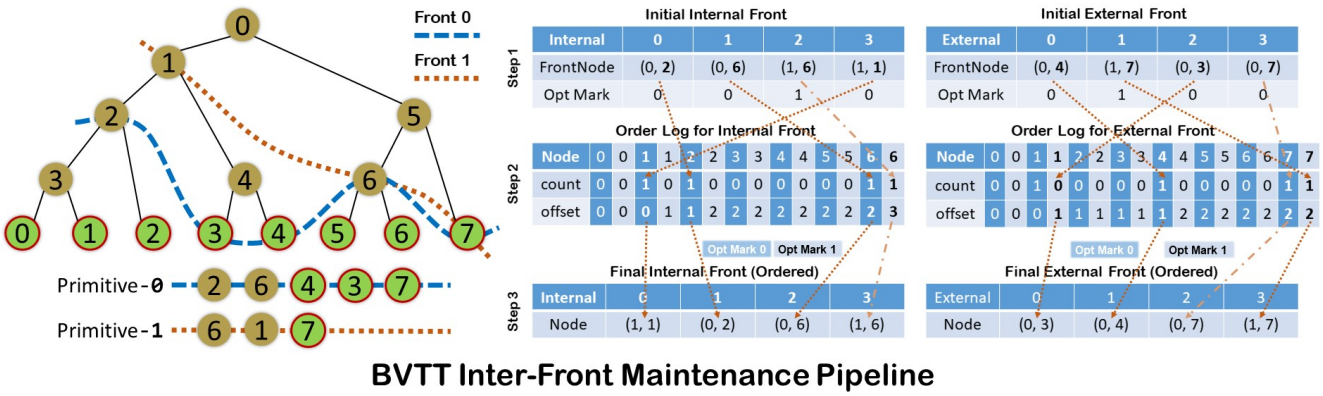


Figure 3: *In this example, we are checking BV overlaps between two primitives and this BVH. All the front nodes are displayed in the left half with respect to its primitive. The right half exhibits our 3-step ordering scheme. In step 1, front nodes categorized into internal front and external front are originally in arbitrary order and marked with a 1-bit tag* ***Opt Mark*** *indicating which branch in maintenance function is executed. In step 2, we count the number of node in each segment and compute prefix sums. In step 3, each front node first refer to its corresponding segment by its second component and the* ***Opt Mark*** *tag, then check the offset log to acquire its segment starting position, finally inserted in this segment.*

the performance loss due to serialization is negligible. Moreover, the strategy of separating external BVH nodes from internal nodes mentioned Subsection 3.2 helps increase the bandwidth efficiency during respective collision queries within both *external fronts* and *internal fronts*.

A more thorough practice of this strategy is illustrated in Figure 3. While preparing BVTT fronts for the next frame, we mark each node *front(i, j)* according to its maintenance branch. Those front nodes that execute *prune* branch are marked 0, otherwise they are marked 1. We combine this 1-bit mark with the BVH node index *j* to get the segment key of node *front(i, j)* and use it for BVTT front ordering. Due to spatio-temporal coherence, all front nodes that execute a specific operation will likely enter the same branch in the following frame, and these nodes are further arranged in a depth-first order layout for better cache utilization. In practice, the mark is not embedded in the segment key, but rather indicates the direction in which to insert corresponding elements, i.e. 1 means to insert from back to front and 0 the opposite. Therefore, the quantity of segments in the *external order log* equals the number of external (leaf) nodes (e.g. 7 in Figure 3) and is thus not doubled.

Finally, we observe that the speedup from updating fronts in each frame does not make up for the overhead caused by memory operations at lines 24, 42 in Algorithm 1. Thus, the front update is only active once every few frames in our scheme and we call this maintenance operation *preserve fronts* (see pipeline **b** in Figure 4).

4. Restructuring Scheme

The BVH-based collision detection pipeline equipped with our ordering scheme gains remarkable speedup over the previous state-of-the-art approach [TMLT11]. However, the BVH quality degeneration is still an unsettled problem. A simple workaround is reconstructing BVHs and BVTT fronts periodically to manage degener-

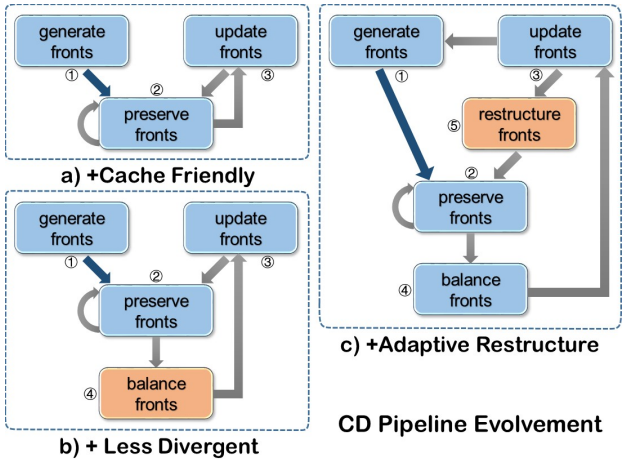


Figure 4: *During each frame, our CD framework executes one of the numbered operations ranging from 1 to 5. Three kinds of collision detection pipelines are presented above.*
Pipeline a*. This pipeline focuses on GPU cache optimization. We add a BVTT front ordering phase in circle* ***1*** *and circle* ***3****. For further speedup, we disable front maintenance in circle* ***2****. Note that whenever the BVH is rebuilt from scratch, this pipeline is reset to circle* ***1****.*
Pipeline b*. Circle* ***3*** *usually takes a lot more time than circle* ***2****. In addition to the overhead of front memory operations, the accumulated deformations in circle* ***2*** *also bring down the CD efficiency. Therefore circle* ***4*** *is injected into the cycle to alleviate thread divergence in circle* ***3****.*
Pipeline c*. The complete pipeline is integrated with our quality inspection and restructuring. The decision for circle* ***3*** *to go to* ***1*** *or* ***5*** *depends on the current degeneration scale measured through our metric.*

ations. But rebuilding from scratch is slow enough that it may lead to performance stuttering. Even worse, setting the period to be too narrow results in unnecessary overhead, while setting it to be too wide frequently neglects notable degenerations. Therefore we propose a novel collision quality metric for frequent quality inspection and use the less costly restructuring operation for BVH structure optimization.

4.1. Quality Inspection

A robust BVH-based CD pipeline should be able to detect degenerations in a timely manner and pick out corresponding BVH subtrees before performing further operations. The previous method [LAM06] measured the culling efficiency of a node by evaluating the overlapping ratio of both children. Each calculation involves all attributes of three bounding boxes (3 * 6 = 18 float variables in total for AABB bounding volume type). The significant flaw in this metric is that the collision status related to the BVH node is omitted. For example, if an internal node covers a subset of primitives that rarely collide with other primitives, then the degeneration of this node is tolerable because only a few collision query tasks are slowed down by it. As a matter of fact, there is usually not enough CD performance speedup from restructuring to make up for its overhead in our experiments on this metric.

A better metric should also consider the magnitude of collisions associated with each BVH node. In our front-based CD, whenever a node degenerates, its related BVTT front nodes most likely *sprout* during an *update front* (see Figure 4). More precisely, this *front(i, j)* will expand into several *front(i, k)s* whose k indicates a node in the subtree of node j. However, the swelling of front nodes could also be the result of an increasing number of collisions with the subset of primitives that node j covers during regular animations. Based on these patterns, we design a metric function that takes the quantity of BVTT front nodes and collision pairs whose second components belong to the same subtree as independent variables. For simplicity, the latter parameter is approximated to the number of external front nodes, so that both items can be measured through the already computed *order log* for front ordering, referred to as *BVTT front log*. Since the overall CD performance is fundamentally determined by the former item, our metric should also grow linearly with it. The optimal metric value needs to remain stable as its collision status changes. Our resulting metric Formula 1 is as follows:

$$Q(i) = \frac{\sum_{j=a}^{b} intcnt_j}{\sum_{k=s}^{t} extcnt_k} = \frac{intoffset_{b+1} - intoffset_a}{extoffset_{t+1} - extoffset_s} \qquad (1)$$

i is the index of an internal BVH node. $Q(i)$ is our improved quality metric of node i. a, b are the smallest internal node index and the largest in subtree i (subtree rooted at node i), respectively. s, t are the smallest external node index and the largest in subtree i, respectively. *intcnt(j)* is the j-th value of *internal front count log*, while *extcnt(k)* is the k-th value of *external front count log*. *intoffset(i)* and *extoffset(i)* stand for the i-th value of *internal front offset log* and *external front offset log*, respectively. Because of our BVH ordering, indices of all the nodes in the same subtree are consecutive; therefore the sum of front node counts within a subtree is simply the subtraction of two prefix sums in *offset log* (see Figure 3).

We integrate it with several CD pipelines and test our metric on two benchmarks where fronts are activated. The curve (see Figure 5) strongly proves the validity of our metric in the way that the overall collision time has a positive correlation with this quality curve. The lower the curve is, the faster its *update front* is executed, which matches the description of our metric function, i.e. the approximate average number of front nodes it takes to compute an AABB collision pair (mostly between 3 and 4). Another proof is that the reference quality curve of every benchmark remains steady no matter what its collision status is, while the static curve rises under the influence of BVH quality degenerations.

Since each calculation involves only four integers from the *offset log* already computed in the front ordering phase, the overhead of quality inspection is almost negligible. In practice, it occupies less than 5 percent of overall CD time in all benchmarks. The detection is performed whenever the BVH structure changes, more specifically right after *generate fronts* or *update fronts* (see Figure 4).

However, there are several regions of interest in Figure 5. Range 1 shows that the optimal BVH structure upon a set of primitives may not be the one built by [Ape14], as the reference curve is higher than the static curve in this range. This is primarily because the construction algorithm does not build the theoretically best quality tree for CD. Range 2 reveals the general pattern of the correlation between the overall CD performance and the quality curve of a simulation.

4.2. BVH Restructuring

The subtree candidates for restructuring are selected during quality inspection in *update front*. In the beginning of the next frame, the quantity of BVH nodes and related front nodes is then evaluated for further instructions (see Formula 2).

$$Opt_{BVH} = \begin{cases} refit & num_{restr} < threshold_{refit} \\ rebuild & threshold_{refit} <= num_{restr} < threshold_{build} \\ restructure & num_{restr} >= threshold_{build} \end{cases} \qquad (2)$$

The implementation of BVH restructuring is generally the same as in [Ape14], except that we only rebuild the subtree candidate in a bottom-up fashion until its root is met.

4.3. BVTT Front Restructuring

After a BVH is restructured, a portion of its related BVTT front nodes, the second components of which map to restructured BVH nodes, becomes invalid. The rest are valid ones that can still be directly used in front-based CD. As shown in Figure 5, reconstructing both BVHs and BVTT fronts is about one order of magnitude slower than *preserve fronts* (see Figure 4). Therefore, a BVTT front restructuring is expected, as in BVH maintenance.

The central idea of *restructure fronts* (see pipeline **c** in Figure 4) is that all valid front nodes are marked, compacted, and later handled by Algorithm 1 as usual, while the invalid front nodes are recalculated. This procedure is similar to the previous ordering scheme, except that the invalid front nodes are filtered out and supplemented in an additional phase, during which each invalid front

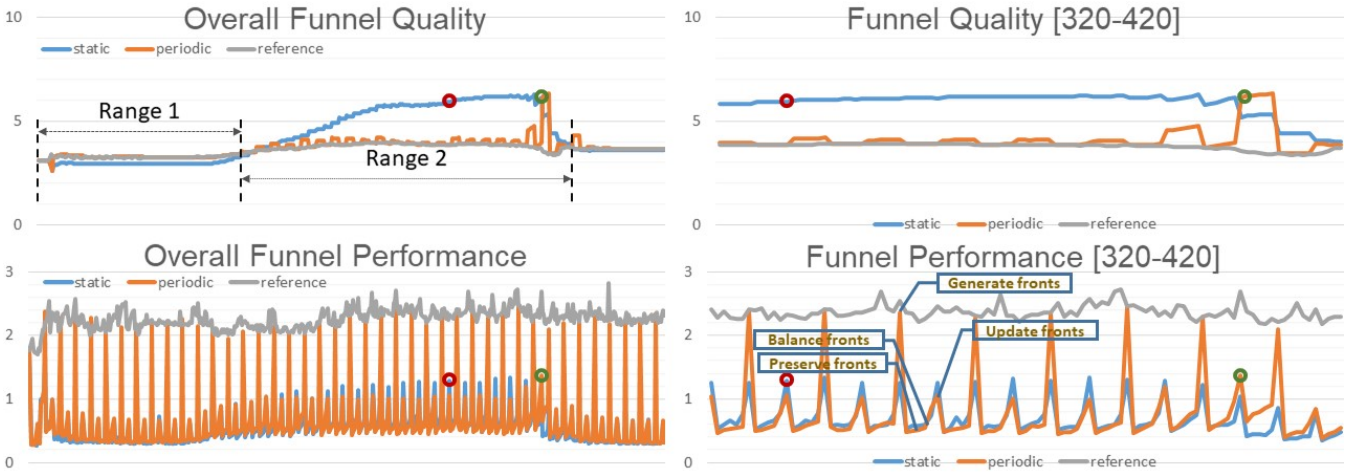


Figure 5: *Three quality curves (a static curve, a periodic curve and a reference curve) and corresponding CD performance curves (total CD time per frame on GTX1080) of benchmark **funnel**. The static curve represents pipeline **b** in Figure 4, and the periodic curve is the variation that replaces "update fronts" operation with "generate fronts" every second cycle. The reference curve rebuilds BVHs and BVTT fronts (i.e. "generate fronts") every frame in order to retrieve the ideal quality under the assumption that the construction algorithm [Ape14] builds the optimal tree structure for CD. The periodic curve shows that "preserve fronts" and "balance fronts" are approximately one order of magnitude faster than "generate fronts", and the cost of "update fronts" which executes Algorithm 1 is about twice as much as "preserve fronts". The two circles marked in red and green display the statistics of "update fronts". The differences in the static curve with the periodic curve suggests that whichever has a higher quality metric takes more time on the CD query.*

node *front(i, j)* is examined for further instruction. Normally, If node *j* covers the leftmost primitive of its subtree, the thread will traverse from *front(i, root(j))*, otherwise it is discarded. A particular case in self-collision front is that, if node *i* is also among the *root(j)*, then node *j* certainly cannot cover the left boundary (its index is smaller than *i*) of the restructured subtree. Instead, if node *j* covers primitive *i+1*, then the thread will traverse from *front(i, i+1)* within this subtree. In the end, the resulting front in arbitrary layout is sorted as in Subsection 3.3.

5. Comparisons and Analysis

5.1. Effectiveness of Histogram Sort

As described in Subsection 3.3, the BVTT front is sorted based on the value of the second component of each front node, and we use histogram sort as our sorting algorithm. To verify its efficiency, we compare it with the implementation of radix sort in Thrust, which is considered to be one of the fastest sorting algorithms for short keys on a GPU. To perform radix sort, the integer keys are extracted and then sorted using the *sort_by_key* function from Thrust library in CUDA toolkit 8.0. Figure 6 displays the overheads of two sort algorithms in three benchmarks. The table shows that radix sort is approximately one order of magnitude slower than histogram sort.

The improvement from ordering is as expected. Figure 7 compares the execution statistics of the most frequently used kernel *preserve-fronts* (see Figure 4), the performance of which is governed by our ordering scheme. It implies that both global memory access and warp execution efficiency benefit from the ordering.

	BVTT Front Length Variation (Internal + External)	Histogram Sort (ms)	Radix Sort (ms)
Flamenco	(644429 + 486981) ~ (1442704 + 753844)	0.40 ~ 0.70	8.20 ~ 12.2
Funnel	(205651 + 149119) ~ (697200 + 214470)	0.14 ~ 0.27	1.53 ~ 2.50
Cloth Ball	(982110 + 755381) ~ (4371698 + 1675945)	0.55 ~ 1.70	9.50 ~ 13.25

Figure 6: *Histogram Sort vs. Radix Sort.*
Experiment conducted on GTX1080. The time spent on sorting depends linearly on the size of BVTT front, therefore it fluctuates as the front length varies during animations.

	L2 Cache Hit Rate (L1 Reads)	Global Load L2 Transactions/Access	Maximum Divergence
Unordered	88%	31.7	99.9%
Ordered	92%	23.4	65.7%

Figure 7: *Impact of Ordering on ClothBall Benchmark.*
Tested on GTX780. All statistics are gathered through NVidia Visual Profiler 8.0.

5.2. Analysis of Our Scheme

We implement two versions of our CD pipeline. The first version is embedded in the ARCSim simulator in [TWT*16]. All primitives are categorized into a deformable BVH and a rigid BVH. Collision queries are composed of an intra-CD inside the deformable BVH and an inter-CD between the deformable BVH and the rigid BVH. We experiment the following four scenarios.

- **Flag:** A single piece of hanging cloth (80K triangles, deformable) flutters in constant wind and with gravity.

- **Sphere:** A piece of hanging cloth (66K triangles, deformable) is hit by a forward/backward moving sphere (1K triangles, rigid).
- **Victor:** A character (104K triangles, rigid) holds still in a red dress (54K triangles, deformable).
- **DressBlue:** A character (27K triangles, rigid) dances in a blue dress (34K triangles, deformable).

The other version is a standalone broad-phase collision detection application. Five publicly available benchmarks also used in *OTG* [WLZ14] or *kDet* [WDZ17] are our test candidates. Note that in each benchmark, we store the primitives of all objects in a single BVH and employ a self collision detection scheme to search for every AABB overlapping pair. No other optimizations are applied. The experiments are conducted on both GTX780 and GTX1080. In certain benchmarks marked with suffix *, their whole animations consist of only a few frames and lack spatio-temporal coherence, so we disable BVTT fronts in these tests and only perform collision queries on our ordered BVH.

- **Funnel (18.5K triangles):** A piece of cloth is squeezed by a ball and falls through the funnel.
- **Cloth Ball (aka Whirling Cloth in *kDet*) (92K triangles):** A piece of cloth is draped over a rotating sphere.
- **Flamenco (49K triangles):** A female character dances a flamenco in a red dress with multiple layers of cloth.
- **N-body* (146K triangles):** Hundreds of spheres and five cones collide in a cuboid space.
- **Dragon* (252K triangles):** A bunny bumps into a dragon and breaks it into pieces.

To verify the effectiveness of our ordering and restructuring techniques, we test the following five strategies.

1. **Unordered curve**. Constructs the BVH once, refits it in the rest of the frames. Fronts are active without ordering.
2. **Static curve**. Constructs the BVH once, refits it in the rest of the frames. Fronts are active and ordered.
3. **Periodic curve**. Replaces *update fronts* with *generate fronts* in Strategy **2** every second cycle.
4. **Restructure curve**. Enables restructuring scheme on BVTT fronts in addition to Strategy **2**. Fronts are rebuilt/restructured along with BVH restructuring.
5. **Reference curve**. Rebuilds BVHs and BVTT fronts (i.e. "generate fronts") every frame to retrieve the ideal quality.

Figure 8 demonstrates the CD performances of the above strategies. We can see that front-based CD algorithms (i.e. *update fronts*, *preserve fronts*, etc.) are significantly faster than BVH-based CD (i.e. *generate fronts*), and applying our ordering scheme applied on both BVH and BVTT front further brings considerable speedup to front-based CD based on the comparison between an *unordered curve* and a *static curve*.

Although our restructuring scheme has a relatively small effect on the overall CD performance among these benchmarks, its *restructure curve* is smoother and steadier than its *periodic curve*. It precisely detects degenerations and only performs restructuring/reconstruction when necessary. Moreover, the quality of the restructured tree is comparable to the reconstructed one. Therefore the *restructure curve* is faster than the *periodic curve* and more robust than the *static curve* due to its adaptivity to degenerate situations.

Figure 8: *Five curves of five strategies for **Whirling Cloth** benchmark tested on GTX1080. The red circles indicate a BVH restructuring event. Since the number of related front nodes are above the threshold, the BVTT fronts at both events are reconstructed right before further degenerations. The time interval between these two events is small, so the benefits are unfortunately soon offset by their overhead.*

5.3. Comparison

The summary of performance comparison between several methods is shown in Figure 9. Both ordering and restructuring improve the overall CD performance including the narrow phase (same implementation as *gProximity* also used in [WLZ14] and [WDZ17]), and our CD framework is thus generally faster than the other two state-of-the-art spatial subdivision methods. The speedup is more noticable compared to [TMLT11]. Even in benchmarks without BVTT fronts, i.e. **N-body** and **Dragon**, our total CD time (including the narrow phase) is comparable or superior to that of *kDet* and *OTG* due to our ordered BVH. In other benchmarks where the BVTT front is enabled, our CD scheme can adapt to BVH quality degenerations during animations aided by our quality metric, thus providing robustness to the overall system.

There are three benchmarks in which our method exhibits marginal performance. In the benchmarks with suffix * (i.e. **N-body** and **Dragon**), the overall performance benefits solely from the BVH ordering. Since most primitives are uniformly distributed in the scene, GPU workloads are balanced in *OTG* and *kDet*, while in our algorithm each thread starts from a different leaf node location and ends up searching all at the rightmost branch of the BVH (see Figure 1), resulting in divergent workloads and slowdown. The BVTT front is enabled in the **ClothBall** benchmark, yet it results in insignificant speedup over *OTG*. According to Figure 8, the performance is, on average, 1.7ms over the first 70 frames (on GTX1080). Then the efficiency drops significantly during the last few frames when the cloth twists around the ball. Besides the increase in collisions, the inability of our scheme to adapt to such coherence changes affects the performance. In other benchmarks, our scheme takes full advantage of coherence, and is therefore much faster.

Benchmark (Frame Counts)	Triangles ×1,000	Tang's (ms)	Ours (ms)	Speedup (times)
Flag (3000)	80+0	49.35	6.48	7.62
Sphere (4100)	66+1	33.66	4.92	6.84
Victor (1300)	54+104	14.57	2.77	5.26
DressBlue (525)	34+27	18.98	5.40	3.51

a) ARCSim Benchmarks broad-phase CD excluding BVH maintenance(Tesla k40c)

Benchmark (Frame Counts)	Triangles ×1,000	OTG CCD (ms)	Ours (ms)	Speedup (times)
Funnel (500)	18	3.4	1.4	2.43
ClothBall (94)	92	9.2	6.0	1.53
Flamenco (506)	49	10.8	3.8	2.85
N-body*(76)	146	19.1	10.5	1.82

b) Standalone Benchmarks (GTX 780)

Benchmark (Frame Counts)	Triangles ×1,000	kDet (ms)	Ours (ms)	Speedup (times)
Dragon* (16)	252	9.3	6.8	1.37
WhirlingCloth (94)	92	6.1	2.3	2.65
N-body*(76)	146	4.4	4.3	1.02
Funnel (500)	18	1.7	0.6	2.83

c) Standalone Benchmarks (GTX 1080)

Figure 9: *Overall CD Performance. We disable BVTT fronts in certain benchmarks marked with suffix * due to a lack of spatio-temporal coherence. In other benchmarks, we use pipeline **b** or **c** in Figure 4.*

The memory footprints of our method are due to BVHs and BVTT fronts. When the number of primitives is 524288 and the upper limit of BVTT front size is set to 12000000 (7000000 for the internal front and 5000000 for the external front), the total amount of memory overhead of our entire CD algorithm is approximately 900MB. However, the largest memory budget (from **ClothBall**) among all standalone benchmarks is actually less than 700MB, when the number of primitives is 100000. Although the memory overhead is larger than the reported 512MB for a nine-level counter octree plus 64MB for the type octree in *OTG* [WLZ14] and the spatial hashing method *kDet* [WDZ17], it fits within the memory capacity of commodity GPUs.

6. Discussions and Future Work

We present a BVH-based broad-phase collision detection framework on GPU. It demonstrates a significant speedup over prior collision detection algorithms due to our ordering scheme, as well as adaptivity to performance-corelative degenerations due to our novel quality metric and efficient restructuring operations. However, there are certain limitations in our scheme and room for improvement.

6.1. Discussions

Limitations: The inherent issue in our scheme is the large memory consumption of BVTT fronts, especially for high resolution scenes. Even though other techniques, e.g., normal cone [WLT*17], can reduce redundant BVTT nodes to a certain degree, the problem is only partially solved. This may limit its use in applications provided with a low memory budget.

Room for improvement: The metric currently used to decide restructuring and to pick out degenerated subtrees is entirely local in terms of time. It performs well in smooth animations, but in cases where motions are large, the benefit of restructuring may soon be offset by the overhead of restructuring itself (see Figure 8). It is a common challenge for restructuring to evaluate the trade-off between operation overhead and resulting performance gain. In such scenarios, the ideal timing of restructuring should be at critical events when a long-term deformation happens, e.g., inelastic motions. Therefore in event-based simulations, the information of events can also be utilized as a supplement to our metric for performance guarantee.

Another defect is deciding whether the use of BVTT fronts for collision detection is static throughout the whole animation. Once the extent of spatio-temporal coherence quickly changes, our scheme is not able to dynamically switch BVTT fronts on/off. Even if the evaluation concludes that the pipeline should proceed with BVTT fronts enabled, the fixed length of the maintenance cycle (the length of *preserve fronts* loop in Figure 4) is not able to make adjustments to such coherence variations.

Although the above issues have little negative impact on consistently spatio-temporal coherent benchmarks, they may become performance barriers and need managing for more general applicability.

6.2. Future Work

Topology operators on models are not currently viable in our GPU version of the ARCSim simulator. Even though we believe our restructuring scheme can support topology changes by inserting/deleting primitives, this feature is subject to the establishment of mapping between previous and current primitives and thus has not been realized yet. We intend to continue this work, and apply it to a wider range of scenarios.

During research, we discovered that BVHs have recently been a hot topic in the field of ray tracing. Several GPU-based restructuring techniques aiming for higher ray throughput have been proposed [KA13, DP15, BWWA17]. The key difference between these techniques and our method is the scale of structure for optimization, i.e. their target of restructuring is *treelets* (local neighborhoods of BVH nodes) and has a size limit. Moreover, *treelet restructuring* has a controllable workload and is applicable to collision detection as well. Meanwhile, the gap between collision detection and ray tracing is closing because of BVH re-use. We expect to develop a simulator that integrates both collision detection and ray tracing based on BVHs in the near future.

Acknowledgements: We would like to thank the reviewers for their constructive comments, Tongtong Wang and Zhongyuan Liu for the assistance in the paper submission. All benchmarks are from the UNC Dynamic Scene Benchmarks collection. The project is supported in part by the National Key Research and Development Program (2017YFB1002700), NSFC (61572423, 61572424, 61732015), the Science and Technology Project of Zhejiang Province (2018C01080), and Zhejiang Provincial NSFC (LZ16F020003). Dinesh Manocha is supported in part by the 1000 National Scholar Program of China and NSFC (61732015).

References

[Ape14] APETREI C.: Fast and simple agglomerative LBVH construction. In *Computer Graphics and Visual Computing (CGVC)* (2014), Borgo R., Tang W., (Eds.), The Eurographics Association. 3, 4, 5, 7, 8

[BWWA17] BENTHIN C., WOOP S., WALD I., ÁFRA A. T.: Improved two-level bvhs using partial re-braiding. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, ACM, pp. 7:1–7:8. 10

[Dam07] DAMKJÆR J.: Stackless BVH collision detection for physical simulation. 3, 4, 5

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG '15, ACM, pp. 13–20. 2, 10

[DZPW15] DU P., ZHAO J.-Y., PAN W.-B., WANG Y.-G.: GPU accelerated real-time collision handling in virtual disassembly. *Journal of Computer Science and Technology 30*, 3 (2015), 511–518. 2

[EL01] EHMANN S. A., LIN M. C.: Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Computer Graphics Forum* (2001), vol. 20, Wiley Online Library, pp. 500–511. 2

[Gar08] GARANZHA K.: Efficient clustered BVH update algorithm for highly-dynamic models. In *2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 123–130. 2

[GNRZ02] GUIBAS L., NGUYEN A., RUSSEL D., ZHANG L.: Collision detection for deforming necklaces. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry* (New York, NY, USA, 2002), SCG '02, ACM, pp. 33–42. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20. 1

[HSK*10] HEO J.-P., SEONG J.-K., KIM D., OTADUY M. A., HONG J.-M., TANG M., YOON S.-E.: FASTCD: Fracturing-aware stable collision detection. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2010), SCA '10, Eurographics Association, pp. 149–158. 2

[JP04] JAMES D. L., PAI D. K.: BD-tree: Output-sensitive collision detection for reduced deformable models. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 393–398. 2

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99. 2, 10

[Kar12] KARRAS T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. Eurographics Association, pp. 33–37. 2, 3

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 197–204. 2

[LAM06] LARSSON T., AKENINE-MOLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics 30*, 3 (2006), 450 – 459. 2, 7

[LC98] LI T.-Y., CHEN J.-S.: Incremental 3d collision detection with hierarchical data structures. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 1998), VRST '98, ACM, pp. 139–144. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 1, 3

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166. 1

[NPK*10] NAH J.-H., PARK J.-S., KIM J.-W., PARK C., HAN T.-D.: Ordered depth-first layouts for ray tracing. In *ACM SIGGRAPH ASIA 2010 Sketches* (New York, NY, USA, 2010), SA '10, ACM, pp. 55:1–55:2. 2

[NT05] NIELS THRANE L. O. S.: A comparison of accelerating structures for GPU assisted ray tracing. *Master's thesis, University of Aarhus* (2005). 3

[OCSG07] OTADUY M. A., CHASSOT O., STEINEMANN D., GROSS M.: Balanced hierarchies for collision detection between fracturing objects. In *Virtual Reality Conference, 2007. VR'07. IEEE* (2007), IEEE, pp. 83–90. 2

[Sob05] SOBOTTKA G.: A.: Collision detection in densely packed fiber assemblies with application to hair modeling. In *In Conference on Imaging Science, Systems and Technology: Computer Graphics, Las Vegas* (2005), CSREA Press, pp. 244–250. 2

[SPO10] SCHVARTZMAN S. C., PÉREZ Á. G., OTADUY M. A.: Star-contours for efficient hierarchical self-collision detection. In *ACM Transactions on Graphics (TOG)* (2010), vol. 29, ACM, p. 80. 2

[TMLT11] TANG M., MANOCHA D., LIN J., TONG R.: Collision-streams: Fast GPU-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 63–70. 2, 4, 6, 9

[TMT10] TANG M., MANOCHA D., TONG R.: MCCD: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models 72*, 2 (2010), 7 – 23. 3

[TTSD06] TROPP O., TAL A., SHIMSHONI I., DOBKIN D. P.: Temporal coherence in bounding volume hierarchies for collision detection. *International Journal of Shape Modeling 12*, 02 (2006), 159–178. 2, 3

[TWT*16] TANG M., WANG H., TANG L., TONG R., MANOCHA D.: CAMA: Contact-aware matrix assembly with unified collision handling for GPU-based cloth simulation. *Computer Graphics Forum 35*, 2 (2016), 511–521. 2, 4, 8

[WDZ17] WELLER R., DEBOWSKI N., ZACHMANN G.: kdet: Parallel constant time collision detection for polygonal objects. In *Computer Graphics Forum* (2017), vol. 36, Wiley Online Library, pp. 131–141. 1, 9, 10

[Wel13] WELLER R.: *New Geometric Data Structures for Collision Detection and Haptics*, 2013;1; ed. Springer Verlag, DE, 2013. 1

[WLT*17] WANG T., LIU Z., TANG M., TONG R., MANOCHA D.: Efficient and reliable self-collision culling using unprojected normal cones. *Computer Graphics Forum* (2017), n/a–n/a. 2, 10

[WLZ14] WONG T. H., LEACH G., ZAMBETTA F.: An adaptive octree grid for GPU-based collision detection of deformable objects. *The Visual Computer 30*, 6 (2014), 729–738. 1, 9, 10

[YCM07] YOON S.-E., CURTIS S., MANOCHA D.: Ray tracing dynamic scenes using selective restructuring. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGSR'07, Eurographics Association, pp. 73–84. 2

[YM06] YOON S.-E., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum* (2006), vol. 25, Wiley Online Library, pp. 507–516. 2