Available online at www.sciencedirect.com
SCIENCE DIRECT
Computers & Graphics 30 (2006) 451–460
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

A dynamic bounding volume hierarchy for generalized collision detection ☆

Thomas Larsson[a,*], Tomas Akenine-Möller[b]

[a]Department of Computer Science and Electronics, Mälardalen University, Sweden
[b]Department of Computer Science, Lund University, Sweden

Abstract

In this paper, we propose a new dynamic and efficient bounding volume hierarchy for breakable objects undergoing structured and/or unstructured motion. Our object–space method is based on different ways to incrementally update the hierarchy during simulation by exploiting temporal coherence and lazy evaluation techniques. This leads to significant advantages in terms of execution speed. Furthermore, we also show how our method lends itself naturally for an adaptive low memory cost implementation, which may be of critical importance in some applications. Finally, we propose two different techniques for detecting self-intersections, one using our hierarchical data structure, and the other is an improved sorting-based method.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Collision detection; Deformation; Data structures; Simulation; Animation; Three-dimensional graphics and realism

1. Introduction

Deformable objects occur frequently in animation, visualization, and simulation. As the complexity of such simulations increases, e.g., to include cutting, tearing, or fracture, improved collision detection (CD) algorithms, in terms of generality, execution speed, and memory cost, are required.

A great deal of effort has already been spent on solving the CD problem for various types of geometry and scenarios. We refer the reader to available surveys [1–4] for a broader view of this research area. Here, we will mainly focus on hierarchical CD methods suitable for deformable body simulation.

Because of their efficiency, bounding volume hierarchies (BVHs) have been widely adopted in the area of rigid body CD, e.g., BVHs of spheres [5–7], axis-aligned bounding boxes (AABBs) [8–10], oriented bounding boxes (OBBs) [11,12], discrete orientation polytopes (k-DOPs) [13,14], and convex pieces [15]. Interestingly, BVHs have also proven to be the preferred object–space acceleration technique for deformable body simulation [4]. Several factors, however, make efficient deformable CD more complicated. Preprocessed data structures must be updated on-the-fly, which may require both hierarchy restructuring and bounding volume refitting. Also, contact areas tend to involve more collision points. Still interactive performance is a fundamental requirement in many applications.

Deformable polygon meshes can be supported by different schemes for refitting the bounding volumes (BVs) in preprocessed hierarchies [16–19]. However, since these methods depend exclusively on preprocessed hierarchy structures that are refitted each frame, their performance breaks down when the accumulated deformation causes significant increases in the overlap among children BVs, and they do not address breakable objects.

For specific bounded types of deformation, methods can be designed to achieve output-sensitive CD [20–22]. Deforming necklaces [23] is another example of a type of deformable object for which efficient CD can be performed. These methods rely on clever usage of specific knowledge of the deformation type, but this also limits their usefulness to a much smaller subset of applications.

☆This paper is an extended version of the authors' vriphys'05 paper.

*Corresponding author.
E-mail address: thomas.larsson@mdh.se (T. Larsson).

0097-8493/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2006.02.011

Only a few general object–space methods that support highly deformable polygon soups undergoing unstructured motion are described in the literature. Smith et al. [24] present a method in which an octree is constructed lazily for all primitives in overlap regions each frame. This method, however, does not exploit temporal coherence, since the octree is always reconstructed top-down from scratch. Ganovelli et al. [25] use a full octree data structure, and reinsert all primitives in the leaves each frame. The stiff nature of the octree, however, makes this solution poorly adaptive to a non-uniform distribution of the primitives over the leaves.

Furthermore, uniform grids [26] and spatial hashing [27] have been utilized for CD. Spatial hashing is used to lower the otherwise costly memory requirement of three-dimensional arrays. However, spatial hashing is sensitive to the choice of the cell size, table size, and hash function. Also, uniform grids, like octrees, may suffer severely from clustering problems as well as the so-called *teapot in the stadium* problem [28]. In trying to overcome these problems, hierarchical hashing has been suggested as a more efficient alternative [29,30].

Recently, dynamic BSP-trees, updated by using a scheduling algorithm to alter the cutting planes in unbalanced regions, has been presented for broad-phase CD scenarios [31]. Also, deformable spanners have been proposed as an interesting graph data structure for CD of deforming point sets [32], which may turn out to be useful for interactive CD of deformable polyhedral models.

CD of massive models has also been addressed. By using an overlap graph, the search space can be pruned to reduce the memory requirements. Dynamic models are treated as floating nodes in the graph, and for these models, BVHs are constructed lazily as the collision queries progress [33].

Clothes are highly deformable objects which require the detection of complex collisions between the wearer and the clothes, self-intersections, and possibly tearing. None of the proposed methods, however, seem to have been designed to handle highly dynamic breakable objects efficiently [34–37].

Furthermore, the tremendous graphics hardware improvements over the past years have made hardware-accelerated image-based CD algorithms for breakable objects possible [4,38]. These methods seem to pose an interesting alternative to object–space methods. However, discretization errors can cause such algorithms report an incorrect subset of all collisions. For contemporary graphics hardware, this is due to the lack of rasterization algorithms that are *conservative*, i.e., rasterization that visits all pixels that are at least partially overlapped by a triangle. Increasing the frame buffer resolution may decrease these problems but can never fully avoid them. A notable exception to this is the extension of CULLIDE [39] presented by Govindaraju et al. [40], where a Minkowski sum of the triangle is used to "grow" each triangle enough to avoid the problems. Even so, many applications, e.g., games, cannot afford to use the graphics hardware for anything than rendering the scene for visual purposes. Furthermore, not all systems are equipped with capable graphics hardware.

To address these limitations, we propose dynamic bounding volume hierarchies (DBVHs) that generalizes the BVH approach to work efficiently for CD of deformable and breakable meshes undergoing structured as well as unstructured motion. The major contributions are as follows: (I) a new efficient object–space CD algorithm for objects consisting of independently moving geometric primitives. No limitations are imposed on the relative motion of the primitives. (II) A method for efficient balancing of lower level update work that amortizes the structural changes of our DBVHs over time. This makes feasible the interactive simulation of breakable objects with tens of thousands geometric primitives. Furthermore, the memory requirements for the BVHs adapts to the current complexity of the collision queries. (III) Demonstration of our algorithm in challenging scenarios. Experiments suggest that our method is significantly faster than the competing object–space algorithms for breakable objects. (IV) The introduced temporal coherent BV refit scheme, which is part of our new CD algorithm for breakable objects, can also be used to improve the performance of CD algorithms designed for non-breakable models. Particularly, in the CD algorithm introduced in [16], we can replace the proposed hybrid bottom-up/top-down update method with our new improved BV refit scheme to make the algorithm take advantage of temporal coherence. (V) Retained compatibility with previously suggested BVH-based methods for more specific types of deformation, and even for rigid bodies. This makes efficient CD of mixed types of objects much simpler. (VI) Minimal preprocessing, which e.g., makes interactive on-the-fly introduction of new dynamic objects in a simulated scene possible. (VII) A comparison of our DBVHs with an improved sorting-based method for the case of finding self-intersections.

2. Dynamic hierarchies

Without loss of generality, we consider DBVHs of AABBs constructed on polygonal meshes (see Section 2.6). Initially, at the start of a simulation, only a root node exists per dynamic object. Then, the trees are rebuilt incrementally as the current change of the spatial configuration of the objects requires. An overview of our algorithm is given in Fig. 1, which shows the two main phases that cooperate

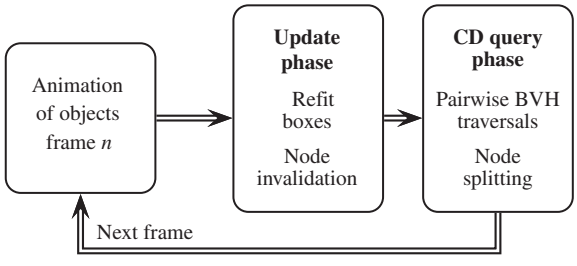

Fig. 1. Overview of the new CD algorithm using DBVHs.

to achieve fast generalized CD: (1) an *update phase*, in which each partly constructed hierarchy from the previous frame is updated and reused if possible, and, (2) a *CD query phase*, in which pairs of BVHs are traversed in parallel to sort out potentially colliding primitives.

The update phase involves *refitting* the currently active boxes, and deleting older or degenerate subtrees, a procedure we call *node invalidation*. In the subsequent query phase, invalidated nodes are rebuilt lazily in a top-down fashion (called *node splitting* hereafter) as they are encountered during the recursive traversal. Note that we refer to a non-leaf node as *invalid* if it has a set of geometry associated with it, but has no appropriate subtree yet. Next, we describe these two main phases of our algorithm in more detail.

2.1. The update phase

All nodes visited during the CD query phase (see Section 2.2) in the previous frame have been marked as *active*, and it is expected that they give a relatively good picture of which nodes will be visited during the next query. Therefore, the update phase starts at the deepest active nodes in each hierarchy and constructs minimal BVs directly from the geometry they cover. This is called *node refitting*. Then the nodes in the branches above them are refitted by merging the child boxes' extents iteratively upwards until the root node is reached. In Fig. 2, this efficient update scheme is illustrated. Note that this partial lazy update is highly adaptive to the current situation, which makes it much more efficient than a full bottom-up hierarchy update [9], and in most cases, it is expected to outperform other more selective update strategies as well (e.g., [16,17]).

In particular, we would like to point out that this novel BV refit scheme can, with advantage, replace *the hybrid bottom-up/top-down update scheme* in our previously proposed CD algorithm for non-breakable polygonal meshes, i.e., deforming meshes with a fixed topology (see [16]). The former refit strategy uses a static choice of always starting the update work at hierarchy level $d/2$, which is first updated completely, and then every node above this level is also updated by merging boxes. This is done in each frame, regardless of whether it is needed or not. In easy cases, too many nodes are updated, e.g., when it would have been enough to only update the root node's bounding box. Also, in hard cases, a lot of nodes need to be updated below the middle level, which then has to be done in the following collision traversals for each encountered non-updated node. In contrast to this, the new refit scheme proposed here is highly adaptive to the geometric situation at hand. Temporal coherence is used to make a qualified guess of where to start the incremental hierarchy update. This may, e.g., be the root node only, or along a few deep branches only, two cases which were particularly ill-suited for the former hybrid bottom-up/top-down update method.

We will now proceed by describing other issues that also must be handled efficiently, since we are addressing breakable objects. Due to too much unstructured motion, a subtree of a node may have become severely inefficient for CD. Therefore, those nodes are marked as *invalid* as follows. During the refitting of the BVs in the currently active nodes, the volume of the parent's bounding box, V_p, is compared to the sum of the child boxes' volumes, denoted V_i. If the following inequality is true,

$$r = \frac{V_p}{\sum_{i=0}^{k} V_i} < R, \tag{1}$$

the parent/children relationship is considered as degenerate and the parent node is marked as invalid, which means the node will be re-split if it is encountered in the CD query phase. The measure r can be seen as an extremely cost-efficient way to estimate the amount of overlap among the children volumes. In Fig. 3, some examples of different r-values are shown. Empirically, we have found that choosing $R = 0.9$ in Eq. (1) yields good results.

However, we note that there are geometric cases when Eq. (1) signals re-split unnecessarily, since when the re-split is executed it will not improve much on the overlap of the children boxes, and if this keeps happening frame after frame, re-split will always occur. For example, a single "giant polygon" might cause a significant overlap among children nodes causing the parent node to be invalidated every frame. Nevertheless, the low computational cost of our simple formula and the good experimental results it

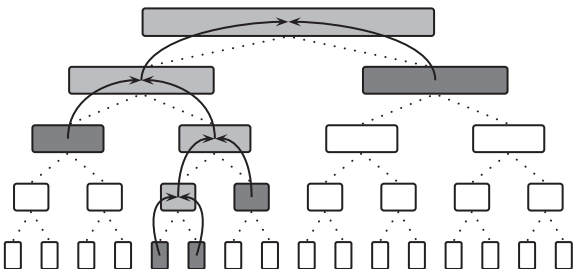

Fig. 2. Efficient temporal coherent BVH update scheme. The refitting starts with the boxes in the deepest active nodes, shown in dark gray, and then the boxes above them, shown in light gray, are refitted by merging their child boxes.

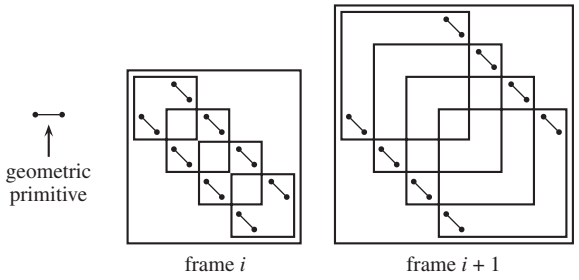

Fig. 3. Overlap among children boxes increases because of unstructured relative motion of primitives residing in the same boxes. Note that the severely overlapping child boxes in frame $i+1$ will never be used, since the parent will be re-split if encountered during a collision traversal.

gives for common scenarios show the usefulness of our strategy.

A possible solution to avoid unnecessary re-splits of the same node frame after frame, which forces re-splits also in reached sub-branches, thereby potentially destroying our goal of utilizing temporal coherence, would be to keep track of nodes which do not improve their parent/children volume relationship enough to satisfy Eq. (1) after a re-split, and keep them as they are during a few frames and then try re-split again. This would require some additional data to be stored in the nodes, as well as some extra calculations. Furthermore, this strategy seems more worthwhile for higher level nodes, with many primitives and a high re-split cost, than for deep level nodes, with few primitives. As an alternative solution, a more advanced node invalidation criteria can be designed, perhaps explicitly utilizing current node parameters such as depth, number of children, number of triangles, and volume ratio at rest state. Yet another solution would be to detect and subdivide too large polygons as they arise.

Any remaining nodes, not marked as active by the latest CD query phase are considered as too old and they are also invalidated. For example, in Fig. 2, the sub-trees below the dark gray nodes are considered too old. Any such sub-trees are traversed while the primitives are collected from their leaves and moved upwards to the closest active (dark gray) nodes. Note that this work is done together with the already described methods for BV refitting and node invalidation in a single traversal pass per BVH.

2.2. The CD query phase

Since the currently existing nodes in the hierarchies already have been updated, pairwise BVH overlap testing can proceed in the same way as done for rigid bodies until a node is reached that is invalid and thus need to be split.

To split a node, we simply classify the midpoint of all involved geometric primitives in relation to the midpoint of the parent box. After partitioning of the primitives, all empty sub-volumes are removed. Thus, a node's number of children, k, varies so that $2 \leqslant k \leqslant 8$. We also let the children take over the responsibility of recording where the primitives lie, i.e., we always store the primitive pointers at the deepest nodes in the existing branches. This simple split heuristic is chosen for its low computational cost per primitive, while at the same time, it rarely produces unacceptable splits. For robustness, however, we still catch too unbalanced splits, i.e., cases where more than 80% of the primitives goes into the same sub-volume, by escaping to specialized code that redo the split by first finding more appropriate split-planes (cf. Gottschalk et al. [11]). We propose the following simple procedure for this. Move the split-planes so that they pass through the midpoint of the crowded sub-volume instead of through the midpoint of the parent box, and then redo the split as described above, see illustration in Fig. 4. If the split problem still remains, although this is highly unlikely for all but contrived pathological cases, yet another re-split method can be executed, or even an arbitrary random primitive partitioning can be done as a last resort.

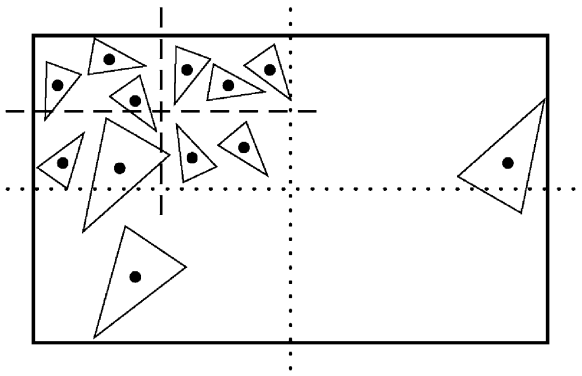

Fig. 4. To redo unacceptable splits, the split-planes are first moved from their old position (shown with dotted lines) so that they pass through the middle of the crowded sub-volume (shown with dashed lines).

After that, a minimal BV is constructed around each newly created non-empty node, and the pairwise traversal continues. In this way, the hierarchies are only split incrementally as the current situation requires, whereas previously constructed valid sub-trees can be reused. Thus, temporal coherence is used to our advantage, since it is expected that the majority of the visited nodes will already have updated BVs (from the update phase), and that only a small fraction of the non-constructed hierarchy need to be traversed using node splitting. Invalid nodes that are not visited during traversal are not split, which further improves performance and memory usage.

Finally, note that during the CD traversal phase, all visited nodes whose BVs were found to overlap with another node's BV are marked as active. This information guides the node refitting to be done in the next frame by the update phase. Also note that our methods automatically work for the case when a single breakable object is involved in multiple collisions with other objects. In this case, different branches will be marked as active in this object's BVH by different pairwise tree traversal, and then the update phase still works exactly as described (Section 2.1).

2.3. Cost function and expected performance

The total cost for CD between two breakable objects undergoing unstructured motion, can be expressed as

$$T = T_t + T_u + T_s, \tag{2}$$

where T_t is the cost of pairwise overlap and intersection testing, T_u is the cost of refitting BVs and invalidate nodes, and T_s is the cost of changing the structure of the hierarchies by node splitting and degenerate sub-tree deletion. Below, the involved relevant terms for calculating these costs are described.

The cost function for pairwise overlap tests is given by

$$T_t = N_p \times C_p + N_b \times C_b, \tag{3}$$

where N_p is the number of intersection-tested geometric primitive pairs, and C_p is the cost of testing one such pair. N_b is the number of overlap-tested BV pairs, and C_b is the cost of one such overlap test. See e.g., Gottschalk et al. for a good discussion of these parameters [11].

Node refitting can either be done directly from the geometric primitives contained in the node's sub-tree or by merging child volumes. Thus, the involved lower level operations is captured by the cost function

$$T_u = N_g \times C_g + N_u \times C_u, \tag{4}$$

where N_g is the number of geometric primitives of the breakable objects processed to update existing BVs and C_g is the cost of processing one primitive. N_u is the number of bounding boxes merged to form parent boxes and C_u is the cost of updating a parent box's extents considering one child box. The cost of making structural changes to the trees, i.e., node splitting and deletion, can be described by

$$T_s = N_s \times C_s + N_d \times C_d, \tag{5}$$

where N_s is the number of geometric primitives processed to split parent nodes into child nodes, and C_s is the cost of processing one primitive in this respect. N_d is the number of primitives processed during deletion of nodes and C_d is the involved average cost per primitive.

Note that all costs represented in the terms of these equations are $O(1)$. Our algorithm aims at minimizing the total cost, T, of a CD query, in particular by utilizing temporal coherence and lazy evaluation techniques to lower N_g, N_s and N_d. Furthermore, our adaptive BV refit scheme (Fig. 2) contributes to an effective balance between N_g, N_u, and N_s. The structural changes made to the hierarchy are further motivated by a strive of making the boxes tighter and avoiding too much overlap among children boxes. Thus, this work also aims at lowering N_p and N_b.

For an object with n primitives, T_u is always $O(n)$, since the total number of primitives processed to refit the deepest active nodes is n and the number of BVs merged to refit all their parents' BVs up to the root is bounded by the maximum number of nodes in an hierarchy, which is in $O(n)$. Furthermore, we expect T_s to be $O(n)$, because of the way temporal coherence is utilized to incrementally change the structure of the hierarchy from frame to frame. In the worst case, however, although highly unlikely, if we arrive at a situation requiring a fully constructed hierarchy, without having any partly constructed hierarchy to start from, then T_s would be $O(n \log n)$. Finally, T_t is output sensitive as for hierarchical rigid body CD.

2.4. Memory/speed trade-off

The algorithm, as we have discussed it so far, has an adaptive memory cost to the actual collision scenarios. For faster performance, however, it is possible to pre-allocate a full, but empty, hierarchy to maximize performance. We have implemented this approach, storing all the initially empty nodes in each hierarchy in a pre-allocated array. Then, the task of allocating and deleting nodes turns into simple assignments to mark nodes as non-empty, when splits occur, and invalid, when nodes get old or degenerate. By using this pre-allocation strategy, the memory requirements of our method becomes similar to the fully pre-processed BVHs normally used for rigid bodies.

Also, note that, even though the nodes are pre-allocated, deformable models with a dynamically varying n can still be supported, as long as the variation is reasonable. This is so because it is only the number of nodes that are predetermined and allocated, not the number of primitives. The primitive partitioning to the nodes can still be handled completely dynamically.

2.5. Front tracking for deformable models

In trying to improve performance further, we also examined the possibility to avoid re-traversing upper parts of the hierarchies which currently do not contribute to the geometry pruning, a technique that has been termed generalized front tracking [15,41]. However, the involved extra effort did not pay off in our case. One probable reason is that our way of constructing the BVHs gives flatter trees with fewer internal nodes than in the binary case (cf. [18]).

2.6. Extensions to other BVs

Our choice of using AABBs in the BVHs is well-motivated by the simplicity of the required procedure to dynamically re-compute the extents of the boxes, either by directly processing subsets of an object's geometry or by merging child boxes, which in both cases produces AABBs with optimal fits with respect to the used partitioning of the underlying primitives. Furthermore, overlap tests among AABBs is extremely fast.

However, our algorithm can with very small adjustments be extended to handle generalized boxes, k-DOPs, or spheres instead, since the only BV operations that our framework requires are efficient node splitting during pairwise traversals, and BV refitting, which in turn requires iterating over the underlying primitives and BV merging (cf. [17,18]). Using k-DOPs, instead of AABBs, would e.g., be a way to trade the more expensive cost involved in updating their extents and the overlap tests for tighter fitting volumes, possibly leading to earlier pruning during collision traversals. However, using spheres, and thus sphere merging in the update phase, produces a *layered hierarchy*, which may result in poor fitting BVs, as pointed out by James and Pai [21].

3. Detecting self-intersections

Although our presented method is primarily designed for efficient inter-object CD, it must be said that realistic deformable body simulation and animation inevitably

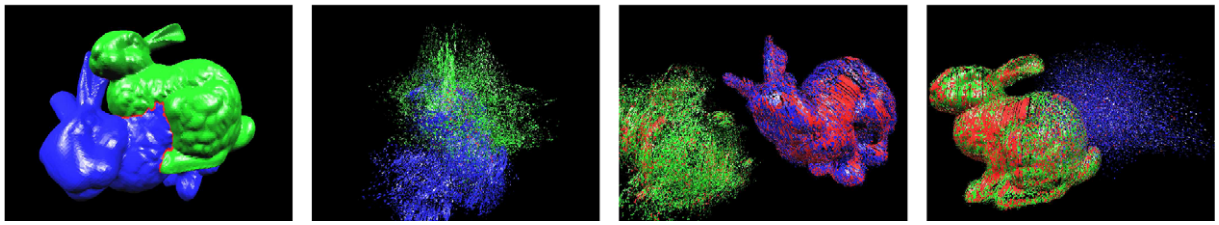

Fig. 5. Selected images from the *bunnies* scene. Inter-object collisions (left images) and self-collisions (right images) are shown in red.

involves handling of self-intersection or intra-object CD. For completeness, we will therefore discuss and evaluate our method for this case as well. This will show that although our method is not as efficient for this case, it is still quite usable. Furthermore, it should be noted that nothing hinders another complementary algorithm to be chosen for the intra-object case, if it is found advantageous.

Since, within a mesh, neighboring primitives are in contact with each other by sharing an edge or a vertex, this misleads the BVH-based search to also find all adjacent elements. Some previously proposed object–space methods for cloth animation have managed to gain efficiency by avoiding searching for most of these false self-intersections by using a local curvature criterion to interrupt the CD traversal [36,37]. Unfortunately, this approach is not possible for arbitrary breakable objects, since the curvature criterion can no longer be assumed to be fulfilled when an object break, i.e., the primitives become disconnected. This can also be seen in the two rightmost images of Fig. 5, where the self-intersections of the just broken bunnies are spread all over their surfaces. It should be noted, however, that for realistically animated fracturing objects, it may still be advantageous to use the curvature criterion in the non-fracturing parts of an object.

Since the curvature criterion is not applicable in the general case we consider, the performance of BVHs will degrade because of the false detection of adjacent (non-intersecting) elements [35]. However, when an object has broken so that the faces have become disconnected, a BVH-based approach can potentially work well.

To use our DBVHs for detecting self-intersections, the DBVH is updated as before, and then the collisions are simply detected by testing the DBVH against itself.[1] Note that most of the BVHs from the previous frame is still expected to be re-used in the next, since our update phase refits active parent BVs that are still valid by merging child BVs, and node invalidation makes sure degenerate branches get rebuilt on a per need basis.

3.1. Sorting-based self-CD

To put our DBVH algorithm for self-CD to test, we have improved a sorting-based algorithm (not based on hierarchies) so that it works for large amounts of primitives. Since a breakable object modelled by n separable geometric primitives can be considered as an n-body in itself, n-body algorithms might be applicable. In particular, we have examined the one-dimensional sweep and prune (SAP) method, which keeps a sorted list of intervals of each body per major axis x, y, and z [42]. Insertion sort is often used since it is $O(n)$ for nearly sorted lists, which often can be assumed.

We start by noting that some SAP algorithms have needed at least one boolean of storage for every possible pair of interval overlap. The table will consume $n(n-1)/2$ bits. For, e.g., 64K triangles, this amounts to 256 MB, which is highly impractical. Another possible implementation is to maintain all triangle pairs that overlap on all three axes in a balanced search tree [43]. This is the SAP method we compare our new algorithms against for finding self-intersections in Section 4.

We immediately found that for self-CD of larger scenes, existing SAP algorithms do not work well for two reasons: (I) the insertion-sort performance may degrade to the worst case of $O(n^2)$, and (II) the number of overlaps along an axis may be $O(n^2)$, which forces far too many insertions and removals in the search tree. Next, we describe our variant of SAP, called *SWIPER*, for overcoming these problems.

A simple way of avoiding the memory cost of previous SAPs is to traverse the sorted list for, say, the x-axis and at the same time detecting all the overlaps along this axis, a procedure done in $O(n+k)$ time, where k is the number of overlaps. We call this *overlap traversal*. Each interval has an identifier of which primitive it belongs to, and so, when an overlap is found, the other two axes can be immediately tested for overlap in $O(1)$ time in the other dimensions. This has $O(n)$ memory requirements, which is a significant improvement over the previous $O(n^2)$ behavior, and it efficiently exploits temporal coherency as previous SAP algorithms. To reduce the effect of problem II, we make sure that the number of overlaps for each axis is maintained, and efficiently updated during the insertion-sort as the order of intervals change. For efficiency purposes, the sorted list with fewest number of overlaps can be used for the overlap traversal described in the previous paragraph. In addition to being faster, it also avoids the problem with $O(n^2)$ overlaps along one axis for many cases.

By default each list is sorted with insertion-sort. However, if the sorting not succeeds in at least $O(n \log n)$ time, then it is interrupted, and quicksort takes over and

[1] We also tried testing each triangle of the object against the DBVH, but this variant reported consistently worse results.

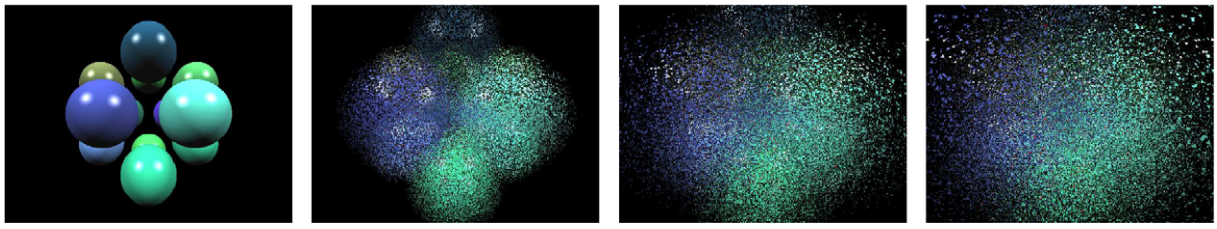

Fig. 6. Selected images from the *spheres* scene.

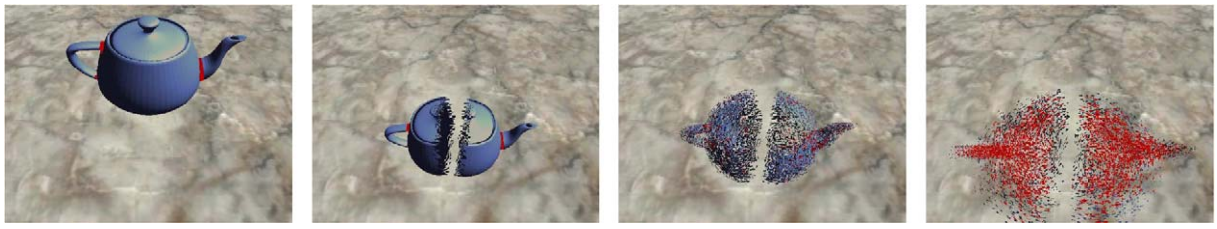

Fig. 7. Selected images from the *teapot* scene.

finishes the sorting.[2] Quicksort is used for m frames (we use $m = 20$), and then we attempt to switch back to using insertion-sort. However, during these m frames, it is only necessary to actually call quicksort in the last frame before switch back is attempted, unless insertion-sort fails to finish in time for all lists during the same frame. If this happens, a randomly selected list is quick sorted and used for overlap traversal. For reasonable scenes, at least one of the axes is sorted using insertion-sort. For those 1–3 axes using insertion-sort, we execute overlap traversal on the axis with the fewest number of overlaps. In this way, we avoid the $O(n^2)$ sorting problem, and by dynamically choosing a suitable axis, we expect the overlap traversal to become output sensitive to the actual number of overlaps along that axis (See Section 4).

We attempted to use the radixsort algorithm with guaranteed $O(n)$ behavior as well. However, for SAP, the intervals need to be sorted so that if interval coordinates are exactly the same, then all start coordinates need to precede the end coordinates. Radixsort cannot handle such comparisons (since it is not comparison-based). Therefore, the radixsort had to be followed by another call to insertion-sort. Quicksort also could be exchanged for heapsort with guaranteed $O(n \log n)$ worst-case behavior. However, in practice, quicksort performed better than both these variations in our test scenes.

4. Results

Our approach has been implemented and evaluated in several benchmark scenarios ranging from simpler cases to highly complex geometry constellations with intermixed geometry. Here we report results from three different benchmark scenarios referred to as the *bunnies* (Fig. 5), *spheres* (Fig. 6), and *teapot* (Fig. 7) scenes. Note that the animations in these scenarios have been designed to include highly unstructured motion, with initially neighboring primitives travelling in opposite directions, to become challenging for lazy hierarchy reconstruction approaches. To examine scalability, we run each scene using three different levels-of-detail (LODs). Both the spheres and bunnies scenes are used to study inter-object collisions, and the teapot and bunnies scenes are used to study self-intersections. Geometry statistics for these scenes, which includes the average number of intersecting primitive pairs per frame, are shown in Table 1. The simulations were run using a standard PC with a Pentium 4, 2.8 GHz CPU, and 512 MB.

Table 1
Geometry statistics for our three demo scenes, including the average number of colliding triangle pairs per frame

	Spheres	Bunnies	Teapot
Number of objects	12	2	1
Triangles/object LOD1	1280	4096	4032
Triangles/object LOD2	5120	16 384	16 256
Triangles/object LOD3	20 480	65 536	65 280
Inter-collisions/frame LOD1	445	218	0
Self-collisions/frame LOD1	0	1737	958
Inter-collisions/frame LOD2	889	442	0
Self-collisions/frame LOD2	0	4440	2524
Inter-collisions/frame LOD3	1823	955	0
Self-collisions/frame LOD3	0	12 757	4003

In Table 2, the experimental results for detecting inter-object collisions are shown. We compare the performance of our DBVHs against the octree method by Smith et al. [24], and we report the average per frame time as well as the best and worst-case frame times for all scenarios. As can be seen, in the bunnies scene, our DBVH method outperforms the octree method by approximately 4, 5, and 6 times on

[2] In practice, we terminate when $kn \log n$ operations have been executed, with $k \approx 8$ yielding best results for our experiments.

Table 2
Results for detecting inter-object collisions for spheres and bunnies

	Octree			DBVHs		
	LOD1	LOD2	LOD3	LOD1	LOD2	LOD3
Spheres						
Average time	171	766	2872	18 (5 + 12 + 1)	70 (20 + 48 + 2)	227 (65 + 140 + 22)
Worst time	382	1769	6799	36 (6 + 29 + 1)	146 (28 + 116 + 2)	458 (89 + 326 + 43)
Best time	1	4	14	1 (1 + 0 + 0)	4 (4 + 0 + 0)	14 (14 + 0 + 0)
Speedup						
Average [max]	1 [1]	1 [1]	1 [1]	9.5 [10.6]	10.9 [12.1]	12.7 [14.8]
Bunnies						
Average time	27	103	428	6 (1 + 2 + 3)	19 (7 + 7 + 5)	74 (31 + 35 + 8)
Worst time	73	281	1236	27 (2 + 7 + 18)	74 (14 + 49 + 11)	317 (78 + 219 + 20)
Best time	1	2	8	1 (1 + 0 + 0)	2 (2 + 0 + 0)	8 (8 + 0 + 0)
Speedup						
Average [max]	1 [1]	1 [1]	1 [1]	4.5 [2.7]	5.4 [3.8]	5.8 [3.9]

All results are in milliseconds. For DBVHs we also give the timings for the update phase, CD query phase and geometric primitive tests separately in parenthesis.

Table 3
Results for detecting self-collisions for teapot and bunnies

	Sweep and prune			SWIPER			DBVHs (self-CD)		
	LOD1	LOD2	LOD3	LOD1	LOD2	LOD3	LOD1	LOD2	LOD3
Teapot									
Average time	22	555	29211	16	118	2146	63	257	1022
Worst time	254	9194	322 532	34	199	5067	116	587	3108
Best time	6	24	91	10	85	1131	30	140	402
Speedup									
Average [max]	1 [1]	1 [1]	1 [1]	1.4 [7.5]	4.7 [46]	13.6 [63.6]	0.3 [2.2]	2.2 [15.7]	28.6 [103.8]
Bunnies									
Average time	87	1612	42 975	55	365	3593	69	224	785
Worst time	315	9217	196 264	74	1134	42 687	105	416	1996
Best time	12	65	263	35	247	2119	41	124	434
Speedup									
Average [max]	1 [1]	1 [1]	1 [1]	1.6 [4.3]	4.4 [8.1]	12.0 [4.6]	1.3 [3.0]	7.2 [22.2]	54.4 [98.3]

All results are in milliseconds.

average for the different LODs. In the spheres scene, the corresponding result is an achieved speed-up of approximately 10, 11, and 13 times on average. In particular, performance breaks down for the octree in hard CD cases, when the overlap regions among the objects' AABBs grow and a lot of geometric primitives get involved in the octree pruning phase. Note that we even used an improved version of the octree method, i.e., the size of the octree root box was dynamically set each frame to the minimum box covering the currently existing overlap regions. In particular, this helps the octree method in the bunnies scene. Furthermore, the fixed maximum depth of the octree was manually tuned to yield the fastest results for each scene and LOD. Indeed, these results show that our approach of only changing the structure of the BVHs incrementally between frames, by node invalidation and node splitting, pays off.

As previously mentioned, our algorithm was primarily designed for the inter-object CD case. However, if wanted, it can also be used for intra-object CD. The results for detecting self-collisions are shown in Table 3. We compare the sweep and prune (SAP) algorithm against both using our DBVHs for detecting self-collisions and against our SWIPER algorithm (see Section 3). For the teapot scene, the average speedup as well as the speedup in terms of the worst frame are substantial,[3] and the speedup increases with more triangles. For the lower LODs, the SWIPER algorithm performs best. However, for the highest LOD,

[3]Except for LOD1/DBVH.

DBVH wins, and we get about 29 times speedup on average, and about 104 times for the worst frame for DBVH. The algorithmic complexity of SWIPER vs DBVH starts to show off for the larger number of primitives, and thus for really large problems, the sorting-based algorithms should probably be avoided. It is interesting to see that SAP consistently reports the best frame time, but this is to be expected since the first 125 frames (of 272) for teapot consist of a simple translation, and thus nothing need to be sorted, and nothing inserted or removed from the search tree for SAP. However, for more complex frames, its algorithmic complexity starts to shine through, and our algorithms outperform SAP.

For self-CD for bunnies, similar behavior can be seen. However, DBVH now performs better than SWIPER for both LOD2 and LOD3. We tried increasing the temporal coherence in that scene, which would ameliorate the performance of the sorting-based algorithms (SAP and SWIPER). However, the performance did not change much. In further performance studies, we discovered that the reason is not the sorting, but rather the number of overlaps per axis, which is very high for the bunnies test scene. This is also the reason why the worst-case speedup slows down for SWIPER. In general, DBVH is a more reliable algorithm than SWIPER, and both DBVH and SWIPER outperforms SAP for large CD scenarios.

5. Discussion and future work

We have proposed a new efficient CD algorithm for highly deforming breakable objects. In particular, for inter-object CD, the introduced DBVH manages to balance the essential operations for hierarchy maintenance efficiently by using temporal coherence to guide the incremental changes between successive frames. Hence, we conclude that the bottleneck would most likely be in self-collision handling, which our experiments confirm. Therefore, much more research needs to be spent on this notoriously difficult problem. For instance, to be more competitive, it would be interesting to examine ways to integrate the approach by Volino et al. [36] for non-fracturing parts of fracturing objects on top of our DBVHs.

Although our test cases have been based on polygonal objects, we note that our approach easily can support breakable objects with other types of primitives such as tetrahedral meshes, which have proven to be an important representation when simulating different types of fracture (see e.g., [44]). Also, our DBVHs can be integrated easily with other BVH-based CD approaches in scenarios involving a mix of rigid and deformable bodies. Simply by providing a dual hierarchy traversal that operates on one rigid body, with an associated preprocessed BVH, and one deformable model, with an associated DBVH, efficient mixed type CD queries can be made.

Furthermore, based on our experiences in CD, we also believe similar DBVHs may be suitable to accelerate other types of queries on breakable objects, e.g., frustum culling, occlusion culling, ray tracing, and picking. Finally, we would find it appropriate to study ways to extend our approach to support continuous CD and more carefully considering the effects of using other BVs than boxes in the dynamic hierarchies.

References

[1] Lin MC, Manocha D. Collision detection. In: Goodman JE, O'Rourke J, editors. Handbook of discrete and computational geometry. Chapman & Hall; 2004. p. 787–807.
[2] Jiménez P, Thomas F, Torras C. 3D collision detection: a survey. Computers and Graphics 2001;25:269–85.
[3] Lin M, Gottschalk S. Collision detection between geometric models: a survey. In: Proceedings of IMA, conference of mathematics of surfaces; 1998. p. 602–8.
[4] Teschner M, Kimmerle S, Heidelberger B, Zachmann G, Raghupathi L, Fuhrmann A, et al. Collision detection for deformable objects. Computer Graphics Forum 2005;24(1):61–81.
[5] Quinlan S. Efficient distance computation between non-convex objects. In: Proceedings of the IEEE international of conference on robotics and automation; 1994. p. 3324–9.
[6] Palmer IJ, Grimsdale RL. Collision detection for animation using sphere-trees. Computer Graphics Forum 1995;14(2):105–16.
[7] Hubbard PM. Approximating polyhedra with spheres for time-critical collision detection. ACM Transactions on Graphics (TOG) 1996;15(3):179–210.
[8] Webb R, Gigante M. Using dynamic bounding volume hierarchies to improve efficiency of rigid body simulations. In: Proceedings of the 10th international conference of the computer graphics society on visual computing: integrating computer graphics with computer vision. New York: Springer; 1992. p. 825–42.
[9] van den Bergen G. Efficient collision detection of complex deformable models using AABB trees. Journal of Graphics Tools 1997;2(4):1–14.
[10] Haverkort HJ, de Berg M, Gudmundsson J. Box-trees for collision checking in industrial installations. In: SCG '02: proceedings of the 18th annual symposium on computational geometry; 2002. p. 53–62.
[11] Gottschalk S, Lin MC, Manocha D. OBBTree: a hierarchical structure for rapid interference detection. In: Computer graphics (SIGGRAPH '96 proceedings); 1996. p. 171–80.
[12] Redon S, Kheddary A, Coquillart S. Fast continuous collision detection between rigid bodies. Computer Graphics Forum 2002;21(3):279–88.
[13] Klosowski JT, Held M, Mitchell JSB, Sowizral H, Zikan K. Efficient collision detection using bounding volume hierarchies of k-DOPs. IEEE Transactions on Visualization and Computer Graphics 1998;4(1):21–36.
[14] Zachmann G. Rapid collision detection by dynamically aligned DOP-trees. In: Proceedings of the IEEE virtual reality annual international symposium; March 1998. p. 90–7.
[15] Ehmann SA, Lin MC. Accurate and fast proximity queries between polyhedra using convex surface decomposition. Computer Graphics Forum 2001;20(3):500–10.
[16] Larsson T, Akenine-Möller T. Collision detection for continuously deforming bodies. In: Eurographics conference; September 2001. p. 325–33.
[17] Brown J, Sorkin S, Bruyns C, Latombe J. Real-time simulation of deformable objects: tools and application. In: Proceedings of computer animation, November 2001.
[18] Mezger J, Kimmerle S, Etzmuss O. Hierarchical techniques in collision detection for cloth animation. Journal of WSCG 2003;11:322–9.
[19] Schmidl H, Walker N, Lin MC. CAB: fast update of OBB trees for collision detection between articulated bodies. Journal of Graphics Tools 2004;9(2):1–9.

[20] Larsson T, Akenine-Möller T. Efficient collision detection for models deformed by morphing. The Visual Computer 2003;19(2–3): 164–74.
[21] James DL, Pai DK. BD-tree: output-sensitive collision detection for reduced deformable models. ACM Transactions on Graphics 2004;23(3):393–8.
[22] Kavan L, Zara J. Fast collision detection for skeletally deformable models. Computer Graphics Forum 2005;24(3):363–72.
[23] Agarwal P, Guibas L, Nguyen A, Russel D, Zhang L. Collision detection for deforming necklaces. Computational Geometry Theory and Applications 2004;28(2–3):137–63.
[24] Smith A, Kitamura Y, Takemura H, Kishino F. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In: Proceedings of the IEEE virtual reality annual international symposium; 1995. p. 136–45.
[25] Ganovelli F, Dingliana J, O'Sullivan C. BucketTree: improving collision detection between deformable objects. In: Spring conference in computer graphics (SCCG2000); 2000. p. 156–63.
[26] Turk G. Interactive collision detection for molecular graphics. Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill; 1989.
[27] Teschner M, Heidelberger B, Mueller M, Pomeranets D, Gross M. Optimized spatial hashing for collision detection of deformable objects. In: Proceedings of vision, modeling and visualization; November 2003. p. 47–54.
[28] Haines E. Spline surface rendering, and what's wrong with octrees. Ray Tracing News 1988;1(2).
[29] Overmars MH. Point location in fat subdivisions. Information Processing Letters 1992;44:261–5.
[30] Mirtich B. Efficient algorithms for two-phase collision detection. In: Practical motion planning in robotics: current approaches and future directions. 1998. p. 203–23.
[31] Luque R, Comba J, Freitas C. Broad-phase collision detection using semi-adjusting BSP-trees. In: Proceedings of ACM siggraph interactive 3D graphics and games; 2005.
[32] Gao J, Guibas LJ, Nguyen A. Deformable spanners and applications. In: SCG '04: proceedings of the 20th annual symposium on computational geometry; 2004. p. 190–9.
[33] Wilson A, Larsen E, Manocha D, Lin MC. Partitioning and handling massive models for interactive collision detection. Computer Graphics Forum 1999;18(3):319–29.
[34] Bridson R, Fedkiw R, Anderson J. Robust treatment of collisions, contact and friction for cloth animation. ACM Transactions on Graphics 2002;21(3):594–603.
[35] Magnenat-Thalmann N, Cordier F, Volino P, Keckeisen M, Kimmerle S, Klein R, et al. Simulation of clothes for real-time applications. In: Tutorial proceedings of eurographics; 2004.
[36] Volino P, Thalmann NM. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. Computer Graphics Forum 1994;13(3):155–66.
[37] Provot X. Collision and self-collision handling in cloth model dedicated to design garments. In: Graphics interface; 1997. p. 177–89.
[38] Govindaraju NK, Lin MC, Manocha D. Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. In: IEEE VR; 2005.
[39] Govindaraju NK, Redon S, Lin MC, Manocha D. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In: Graphics hardware 2003. Eurographics Association; 2003. p. 25–32.
[40] Govindaraju NK, Lin MC, Manocha D. Fast and reliable collision culling using graphics hardware. In: VRST '04; 2004.
[41] 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; 1998. p. 139–44.
[42] Cohen JD, Lin MC, Manocha D, Ponamgi M. I-collide: an interactive and exact collision detection system for large-scale environments. In: Symposium on interactive 3D graphics; 1995. p. 189–96.
[43] Baraff D. Dynamic simulation of non-penetrating rigid bodies. PhD thesis, Cornell University; 1992.
[44] O'Brien JF, Hodgins JK. Graphical modeling and animation of brittle fracture. In: SIGGRAPH '99; 1999. p. 137–46.