Balanced Hierarchies for Collision Detection between Fracturing Objects

Miguel A. Otaduy* Olivier Chassot† Denis Steinemann* Markus Gross*

Computer Graphics Laboratory, ETH Zurich

ABSTRACT

The simulation of fracture leads to collision-intensive situations that call for efficient collision detection algorithms and data structures. Bounding volume hierarchies (BVHs) are a popular approach for accelerating collision detection, but they rarely see application in fracture simulations, due to the dynamic creation and deletion of geometric primitives. We propose the use of balanced trees for storing BVHs, as well as novel algorithms for dynamically restructuring them in the presence of progressive or instantaneous fracture. By paying a small loss of fitting quality compared with complete reconstruction, we achieve more than one order of magnitude speedup in the update of BVHs.

Keywords: Collision detection, AVL-trees, fracture.

Index Terms: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object Hierarchies

1 INTRODUCTION

Action videogames and feature films often display spectacular crashes that lead to the fracture of cars, walls, etc. In medical simulation, surgical interventions often involve cutting of soft tissue. After fracture or cutting, the newly created objects or surfaces move independently, and are free to collide with each other, thereby leading to collision-intensive situations.

At fracture events, the precomputed data structures for accelerating collision detection become obsolete, and need to be recomputed. This recomputation carries a cost that cannot be afforded in interactive applications. One alternative, commonly followed in videogames, is to predefine the fractured pieces, thus their collision detection data structures can be precomputed. Though practical, this predefinition limits the richness of the effects that can be achieved, and is not appropriate for applications that involve arbitrary fracture or cutting.

In this paper, we present an approach for dynamically adapting data structures for collision detection at fracture or cutting events. We build our approach on bounding volume hierarchies (BVHs), which have proved to be a successful acceleration data structure for collision detection between rigid bodies [14, 10, 16, 8], collision detection between deformable bodies [32, 7, 18, 15, 19, 31, 34], and also as a first step of culling in combination with other methods [12].

We leverage AVL-trees [1, 17] for constructing BVHs, and we design operations for *dynamically restructuring* BVHs during fracture, based on elementary operations for rebalancing AVL-trees. We also discuss metrics and operations for maintaining good fitting of bounding volumes. To the best of our knowledge, no previous work addresses the problem of restructuring BVHs as a result of fracture. We show that, in cases where objects fracture progressively, dynamically updating BVHs with our approach is up to 20 times faster than fully reconstructing them from scratch. Moreover, complemented with existing efficient techniques for refitting BVHs [19, 34], our dynamic restructuring algorithms can provide these update speedups with almost no penalty on the performance of collision detection.

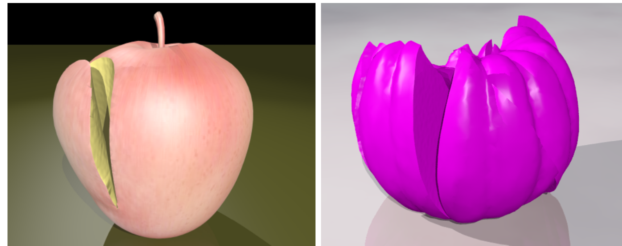

Figure 1: **Collisions in Fractured Objects.** *Left:* Self-collisions on an apple that is progressively cut. *Right:* Collisions between multiple pieces of a pumpkin that hits the ground and breaks. When new surface primitives are created due to fracture, we dynamically restructure the BVHs for collision detection. This operation is up to 20 times faster than full reconstruction of the BVHs.

In §2, we discuss related work, including a description of characteristics of BVHs that are relevant for design decisions in our algorithm. In §3, we overview AVL-trees and their basic rebalancing operations for insertion and deletion of nodes. With this background knowledge, in §4 we present our algorithms and data structures for dynamic update of BVHs during fracture. Finally, we present experiments and results in §5, before concluding with discussions in §6.

2 RELATED WORK

Collision detection is a problem that has been largely addressed in computer graphics, robotics, or computational geometry. We refer the reader to recent surveys [20, 31] for extensive discussions on the topic. In this paper, we are interested in collision detection of objects that undergo topological changes, thus surface primitives (i.e. triangles) are created, deleted, and reconnected dynamically. Some of the popular techniques for collision detection between deforming objects, such as visibility-based culling [11], distance fields [9, 29], layered depth images [13], or spatial hashing [30], assume no pre-processing of the surfaces. Bounding volume hierarchies (BVHs) have also proved to be competitive when updated efficiently, for complete collision detection between deforming objects [32, 7, 18, 15, 19, 34], or as a first culling stage in combination with visibility-based algorithms [12].

The cost of collision detection using BVHs can be decomposed into: the cost of primitive-level tests, the cost of BV tests, and the cost of BV updates (for deforming objects) [10]. The choice of type of BV is driven by the minimization of the overall cost. AABBs [3, 7] offer a good trade-off for collision detection between deforming objects, as each BV can be updated in $O(1)$ time from information of its children. Other types of BVs include spheres [25, 14], k-DOPs [16], OBBs [10], or convex hulls [8]. Recently, Weller et

*e-mail: {otaduy,deniss,grossm}@inf.ethz.ch
†e-mail: olivier.chassot@alumni.ethz.ch

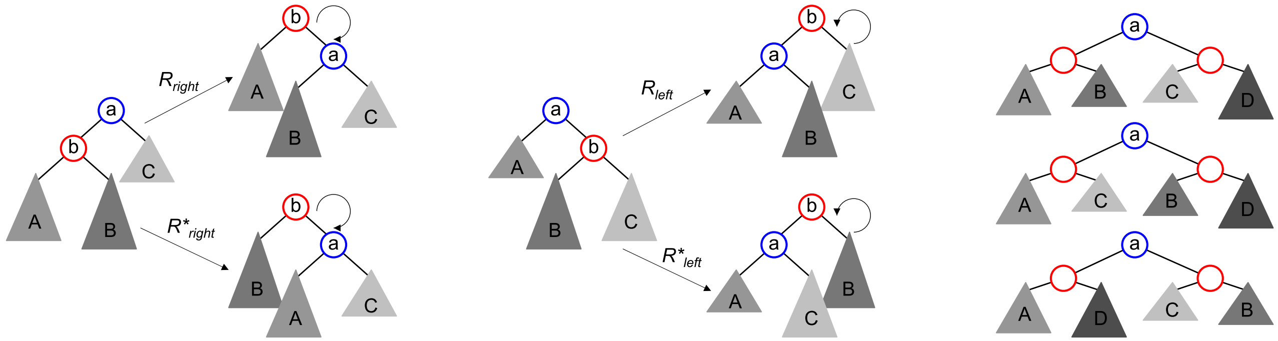

Figure 2: **Local Restructuring Operations.** *Left:* Right rotation R_{right} of the edge (a,b), and modified right rotation R^*_{right} with pre-swap of subtrees A and B. *Middle:* Left rotation R_{left} of the edge (a,b), and modified left rotation R^*_{left} with pre-swap of subtrees B and C. *Right:* The three different permutations of the grandchildren of node a.

al. [33] have investigated the expected time for collision detection using AABB trees.

The design of the BVH is also driven by the minimization of the overall collision detection cost. To guarantee efficiency, the BVH of a surface with n primitives must maintain several properties: (i) the depth of the tree must be $O(\lg n)$, (ii) each node must have $O(1)$ children, and (iii) the size of BVs should be minimized. Typically, these properties are achieved by appropriately constructing the BVH during preprocessing. One option is to construct the BVH in a bottom-up manner, by pairwise merging of BVs. Another option is to construct it in a top-down manner, by successive splitting along the longest axis [10]. Splitting at the midpoint or mean position yields a construction cost of $O(n \lg n)$, but may violate the properties of BVHs, as it does not guarantee a balanced tree of logarithmic depth. Splitting at the median position, on the other hand, ensures a balanced tree, but yields a construction cost of $O(n \lg^2 n)$.

With deforming objects, the BVH must be updated at runtime. We consider two different types of updates: (i) refitting of BVs, and (ii) restructuring of the tree. Efficient refitting has been extensively studied, and it includes approaches such as hybrid top-down and bottom-up updates [18], lazy update with enlarged BVs [21], fully top-down update for reduced deformable models [15], or event-based refitting in the framework of kinetic data structures [34]. Efficient restructuring of the trees, however, is a less explored problem. Larsson and Akenine-Möller [19] present a fitting quality metric and a method for dynamically splitting BVs in situations with unstructured motion. In this paper, we investigate dynamic restructuring of BVHs for fracturing objects.

Our work is also related to previous work on dynamic mesh reclustering by Carr and Hart [5]. They define imbalance functions to measure the quality of mesh clusters, and apply node rotations and swapping operations to improve balance. Our BVH restructuring approach builds upon balanced trees, such as AVL trees [1, 17] or red-black trees [6], and employs elementary rebalancing operations similar to those of Carr and Hart [5].

The algorithms presented in this paper focus on the problem of detecting collisions between fracturing objects, not in the simulation of fracture itself. We rely on existing techniques in computer graphics for physically-based simulation of fracture [24, 23], and for handling the geometric aspects of topological changes in cutting or fracture simulations [22, 28].

3 AVL Trees for Collision Detection

Given an object A, described by a set of triangles $\{\tau_i\}$, we wish to design a BVH that will be efficiently adapted under fracture events. We make no assumptions on the topology of A (i.e. it may be a triangle soup), but for simplicity we will refer to the set $\{\tau_i\}$ as a triangle mesh. Fracture produces dynamic changes to the triangle mesh, as some triangles are decomposed into new triangles, and parts of the mesh are newly synthesized when cracks evolve. The BVH for accelerating collision detection queries with A should remain balanced at all times, and the BVs should maintain a good fitting quality. Although the basic algorithms and data structures we propose support any type of BV, we have chosen AABBs for their $O(1)$ update cost.

We use AVL trees for the implementation of BVHs, and thus we provide efficient operations for guaranteeing balanced BVHs and for optimizing fitting quality of BVs, as the triangle meshes are updated during fracture. In this section, we introduce notation and parameters of AVL trees, we describe the basic operations of insertion and deletion, as they will serve us to account for updates in the geometry of the meshes, and we describe restructuring operations for rebalancing and local optimization of fitting quality.

3.1 Description of AVL Trees

An AVL Tree is a self-balancing binary tree [1]. Each node a stores pointers to its parent, $a.parent$, and left and right children, $a.left$ and $a.right$. The height $h(a)$ of a node a is defined recursively as

$$h(a) = \begin{cases} 0 & \text{if } a \text{ is a leaf,} \\ 1 + \max(h(a.left), h(a.right)) & \text{otherwise.} \end{cases} \quad (1)$$

And the *balance factor* $\beta(a)$ of a node a is defined as

$$\beta(a) = \begin{cases} 0 & \text{if } a \text{ is a leaf,} \\ \|h(a.left) - h(a.right)\| & \text{otherwise.} \end{cases} \quad (2)$$

AVL trees self-balance by guaranteeing a balance factor $\beta \in \{0,1\}$ on all nodes. In other words, the height difference of right and left subtrees will be at most one for all nodes. Insertion and deletion of nodes may imbalance the tree (see §3.2), but after a single insertion or deletion the balance factor of a node may be at most $\beta = 2$, and the node may be rebalanced with a single local operation (see §3.3).

For our purpose of collision detection with triangle meshes, each leaf of the tree stores one triangle τ_i and its corresponding BV. Internal nodes store pointers to their children, and a BV that bounds all triangles in the subtree. Insertion and deletion operations are performed on leaves, not on internal nodes. We initialize the AVL tree (i.e. the BVH) by top-down successive sorting and splitting of the input triangle mesh [10]. Note that, by successively sorting a set of triangles and splitting them at the median, we ensure a balanced tree. Then, for each leaf we compute an AABB, and we fit AABBs to all tree nodes in a bottom-up manner [7].

3.2 Insertion and Deletion of Triangles

An individual insertion operation $Insert(\tau_i, \tau_j)$ of triangle τ_i at the position of triangle τ_j replaces τ_j with an internal node a and sets τ_j

and τ_i as children of a. An individual deletion operation $Delete(\tau_i)$ of triangle τ_i replaces the parent of τ_i with its sibling.

Both insertion and deletion operations locally modify the height of the tree, and this may result in imbalance. After insertion or deletion, the tree must be rebalanced by means of node rotations (see §3.3), and heights need to be recomputed along the path to the root of the tree. For an AVL tree with n leaves, an individual insertion needs, in the worst case, one rotation and $O(\lg n)$ height recomputations; and an individual deletion needs, in the worst case, $O(\lg n)$ rotations and $O(\lg n)$ height recomputations [17]. As discussed in §4, fracture simulation is dominated by insertion operations, which require fast height recomputations and only a few rebalancing operations.

3.3 Restructuring Operations

Here we describe restructuring operations on AVL trees that enable both dynamic tree rebalancing and BV refitting.

3.3.1 Tree Edge Rotations

Given a node a with imbalance factor $\beta(a) = 2$, the node can be rebalanced by applying a local tree rotation. We refer as the *higher* child of a node a to the child with larger height value. If the left child is higher, i.e. $h(a.left) = h(a.right) + 2$, we apply a right rotation R_{right}. If the right child is higher, i.e. $h(a.right) = h(a.left) + 2$, we apply a left rotation R_{left}. Tree rotations are depicted schematically in Figure 2.

Simple rotations may not rebalance the node a if the higher child has a balance factor $\beta = 1$. We exploit the fact that in our application AVL trees do not store sorted data, hence the order of siblings is irrelevant. Then, if the balance factor of the higher child is $\beta = 1$, we swap its children appropriately before applying the rotation. This pre-swapping yields modified rotations R^*_{right} and R^*_{left}, as shown in Figure 2.

3.3.2 Grandchildren Permutations

The possibility to swap siblings can be exploited further, in order to locally optimize the fitting quality of BVs. Given a node a, we propose local restructuring by permuting its grandchildren, as shown in Figure 2. We only consider as valid those permutations that do not produce imbalance.

We can easily test for valid permutations due to the following property: If the node a and its descendants are balanced, the node a will also be balanced after a permutation of its grandchildren *if* its children are balanced. Therefore, it is sufficient to test for the balance of $a.left$ and $a.right$.

In §4.3 we explain the local optimization of fitting quality employing grandchildren permutations.

4 Dynamic Update of BVHs

In this section we describe how to dynamically update the BVHs of fracturing objects, using the data structures and algorithms discussed in the previous section. We distinguish two types of fracture processes: progressive fracture, where cracks evolve incrementally between two time steps (see Figure 5), and instantaneous fracture, where cracks start and terminate during one single time step (see Figure 6). After discussing the different update algorithms for progressive and instantaneous fracture, we describe local optimization operations for improving the fitting quality of BVs.

4.1 Progressive Fracture

In the simulation of fracture in computer graphics, crack surfaces must be dynamically synthesized. Given the path swept by a crack during a time step of simulation, dynamic crack surface synthesis involves the following steps [28]: (i) triangulate the surface swept by the crack during the time step, (ii) intersect the crack surface with the original surface of the object, (iii) trim and decompose

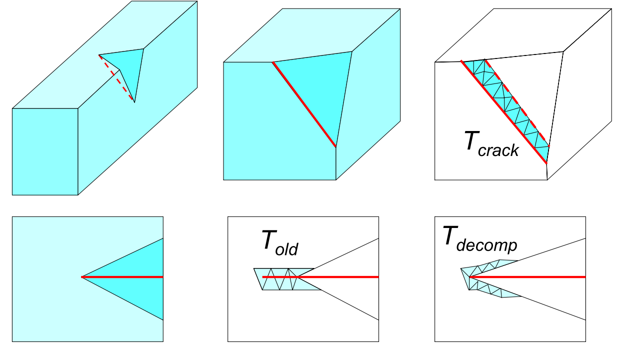

Figure 3: **Progressive Crack Growth.** *Top:* As the crack front (in red) evolves, new triangles T_{crack} are synthesized on the crack surface. *Bottom:* Top view of the crack growth, showing old triangles T_{old} that break and are decomposed into new triangles T_{decomp}.

the triangles that intersect, and (iv) stitch the crack surface and the original surface together.

For the purpose of updating BVHs for collision detection, at every time step we are interested in the set of old triangles T_{old} that break and are decomposed into new triangles T_{decomp}, as well as new triangles T_{crack} on the crack surface (see Figure 3). The old triangles T_{old} must be deleted from the BVH, while the new triangles T_{decomp} and T_{crack} must be inserted.

Every old triangle $\tau_i \in T_{old}$ is decomposed into a (typically small) set of triangles $\{\tau_j\} \subset T_{decomp}$. Instead of deleting τ_i from the BVH with the $Delete()$ operation described in §3.2, it is more efficient to directly replace it by one of the triangles in $\{\tau_j\}$. Therefore, we design an operation $Replace(\tau_j, \tau_i)$ that places τ_j at the position of τ_i in the AVL tree. Note that this operation requires no rebalancing or height recomputations. We refer as $T_{replace} \subset T_{decomp}$ to the subset of decomposed triangles that are added to the BVH simply by replacing old triangles T_{old}.

The rest of new triangles $\tau_i \in T_{insert} = (T_{decomp} - T_{replace}) \cup T_{crack}$ are inserted in the AVL tree using the $Insert()$ operation discussed in §3.2. For each of them, we need to identify a location for insertion τ_k, and we do it in the following way. We first add to a queue all triangles neighboring $T_{old} \cup T_{crack}$ that are not fractured. Then, we do a breadth-first search (BFS) from this queue, and whenever we visit a triangle $\tau_i \in T_{insert}$, we insert it in the BVH by the procedure $Insert(\tau_i, \tau_k)$, where τ_k is the parent of τ_i in the BFS. This guarantees that τ_k is present in the BVH when τ_i is inserted. Also, since τ_k and τ_i are adjacent to each other, the insertion procedure favors tight fitting BVs. In case we are dealing with a triangle soup instead of a connected triangle mesh, the BFS procedure may be substituted by a greedy search of nearby triangles.

After each insertion operation we recompute heights and rebalance nodes if necessary, as described in §3.3. Once all new triangles are inserted, the BVs may be updated, and we perform local optimizations described in §4.3.

4.2 Instantaneous Fracture

In some materials the propagation of cracks is much faster than the frequency at which collision detection is typically performed. Therefore, a complete object can be considered to fracture in just one time step of simulation (see Figure 6). We refer to this phenomenon as *instantaneous fracture*.

In the case of instantaneous fracture, it would not be efficient to let the BVHs self-adjust by local optimization operations, as is the case with progressive fracture. After restructuring of the BVH, all the triangles belonging to one connected component should lie under one single AVL subtree, but the number of restructuring operations required to achieve this may easily grow beyond the cost

of simply rebuilding the BVHs from scratch. This is especially true if the initial object breaks into many small components.

Here, we propose a method for restructuring the BVHs that is appropriate for instantaneous fracture into several components. Based on our experiments (see §5.4), the method is competitive for objects of about 10K triangles that fracture into less than 10 components (For objects with more triangles, it will be competitive even if they fracture into more components). In cases where an object fractures into many small pieces, full reconstruction of the BVHs is more efficient. Our algorithm is also applicable for handling the disconnection of components at the end of a progressive fracture. Given an object with n triangles that breaks into a small number of pieces, our restructuring algorithm has $O(n)$ complexity.

To describe our method for restructuring BVHs after instantaneous fracture, let us assume that an object A breaks into two components B anc C. The initial set of triangles T_A is decomposed into three subsets: triangles fully in B, T_B, triangles fully in C, T_C, and triangles that break T_{old}. After synthesis of the crack surfaces, we also have two sets of new triangles for each component, T_{crackB} and T_{crackC}.

We first clone the BVH of object A, which yields initial BVHs for B and C. Then, we delete $T_C \cup T_{old}$ from the BVH of B, and delete $T_B \cup T_{old}$ from the BVH of C. For this, we use the $Delete()$ operation described in §3.2. Finally, we insert T_{crackB} into the BVH of B, and T_{crackC} into the BVH of C, using the $Insert()$ operation described in §3.2. In order to determine the location of insertion for new triangles, we follow the same BFS-based flooding approach introduced for progressive fracture.

4.3 Optimization of Fitting Quality

Before describing optimization strategies, we will define the fitting quality $q(a)$ of a node a of the BVH, and the fitting quality $Q(A)$ of the complete BVH of object A. Larsson and Akenine-Möller [19] define a fitting quality metric that evaluates overlap among sibling nodes in the BVH. In contrast, the fitting quality Q that we define here is an absolute metric that serves for comparing BVHs. As it will be discussed in §5.2, the depth of the BVH is another possible quality metric.

Given a node a of the BVH, we measure its fitting quality $q(a)$ as the sum of squared volumes of its children:

$$q(a) = Vol^2(a.left) + Vol^2(a.right). \tag{3}$$

Given an object A, we measure the fitting quality $Q(A)$ of its BVH as the sum of squared normalized volumes of all nodes in the BVH:

$$Q(A) = \frac{1}{Vol^2(BV_{root})} \sum_{BV_i \in BVH(A)} Vol^2(BV_i). \tag{4}$$

As reference, in a balanced binary BVH where each BV is divided into two equal children with half the volume of their parent, the fitting quality for n triangles is $Q = \sum_{i=0}^{\lg n} \frac{1}{2^i} = \frac{2n-1}{n}$. Note that smaller values of q and Q denote better fitting.

After each time step, once new fracture geometry is inserted into the BVH, we perform a bottom-up update of the BVs, followed by local refitting optimization. For every node a of the BVH, we test if some grandchildren permutation (see §3.3) improves the fitting quality $q(a)$. We only allow permutations that do not imbalance the children of a, in order to avoid cascading rotations. Carr and Hart [5] proposed local optimization operations based on tree edge rotations in the context of mesh reclustering, but their operations would induce cascading rotations in our application. By limiting local optimizations to those that do not produce imbalance, we also limit the flexibility of improving the fitting quality of the complete BVH, $Q(A)$, but we ensure that the cost of the optimization remains the same as for the BV refitting, i.e. $O(n)$.

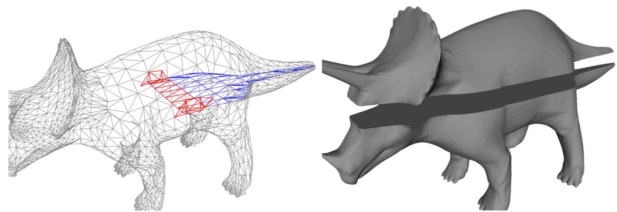

Figure 4: **Progressive Splitting of Triceratops Model.** These images depict the benchmark for the results described in §5.2. *Left:* Triangle mesh of a triceratops as it is progressively split. The blue triangles denote the evolution of the crack, while the red triangles denote the decomposed triangles T_{decomp} and new crack triangles T_{crack} for the last splitting step. *Right:* Final fractured triceratops.

5 Results

In this section we describe experiments that we have carried out for testing the performance of our restructuring algorithms. These experiments include static benchmarks for scalability analysis, and dynamic simulations of progressive and instantaneous fracture. The fracture simulations (See Figures 5 and 6) present very challenging situations of collision and self-collision, with large areas in parallel close proximity, which is a worst-case scenario for collision detection [10].

5.1 Implementation Details

Our algorithms have been implemented by extending the collision detection library SOLID [27], based on AABBs. We modified SOLID to support AVL trees as the underlying data structure, and added median-based splitting for the construction of a balanced BVH. Our dynamic simulations also employ existing methods for contact handling of rigid and deformable objects [2, 4].

All experiments have been executed on a dual Pentium-4 3.0 GHz processor PC with 2.0 GB of memory.

5.2 Analysis of Performance and Scalability

We have performed a progressive fracture of a static triceratops model (see Figure 4) with different mesh resolutions, for comparing the cost of dynamic restructuring of the BVH Vs. the cost of full reconstruction, as well as for evaluating the evolution of the fitting quality. Table 1 shows average timings across the different fracture steps, and the overall performance gain obtained by our algorithm, for the different mesh resolutions. Figure 7, on the other hand, depicts the evolution of the BVH update time, the fitting quality Q, and the depth of the BVH, for the densest mesh (90 560 triangles).

As can be deduced from Table 1, the time required for replacing and inserting new primitives is negligible in comparison with the time required for refitting and optimizing the BVH. The reason is that the amount of triangles to be inserted per step is small compared to the size of the mesh, and each insertion only requires one tree rotation at most, as explained in §3.2. Height updates, on the other hand, are extremely fast.

Notice also from Figure 7 that the time for dynamic restructuring per step (approximately 100 ms) is up to 4 times smaller than for full reconstruction through midpoint splitting, and up to 18 times smaller than for full reconstruction through median splitting. As discussed in §4.3, the cost of refitting and optimization is $O(n)$, asymptotically better than the cost for full reconstruction through both midpoint splitting, $O(n \lg n)$, and median splitting, $O(n \lg^2 n)$. Note also that, with reconstruction through midpoint splitting, the quality of the BVH degrades largely, as denoted by the depth of the tree (see Figure 7), which almost triples the depth of the balanced AVL trees when the fracture ends.

As one would expect, with dynamic restructuring the quality Q of the BVH decays as cracks evolve, but the local optimization

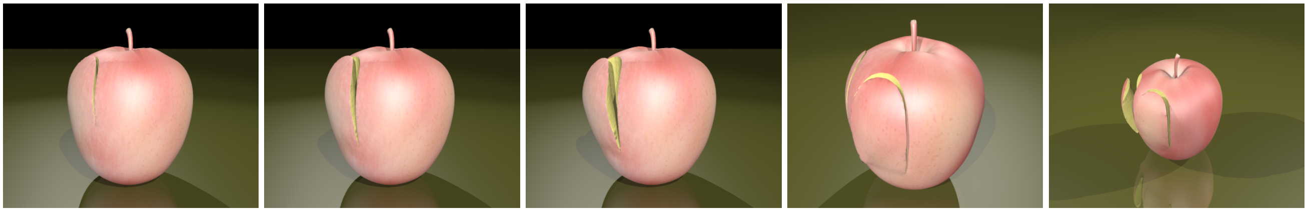

Figure 5: **Progressive Peeling of an Apple.** Benchmark for our BVH update algorithm with progressive fracture and challenging self-collisions.

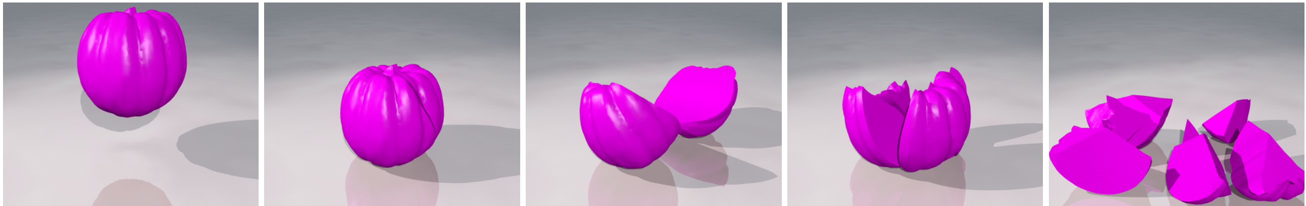

Figure 6: **Instantaneous Fracture of a Pumpkin.** A pumpkin is dropped on the ground and it fractures into a variable number of pieces (2 and 8 in the snapshots). At the instant of fracture, there are large areas in parallel close proximity, a worst-case scenario for collision detection.

operations (see §4.3) improve the fitting quality by approximately 30%, as shown in Figure 7. The fitting quality and depth plots also suggest that further investigation should be devoted to designing a unique quality metric that accounts for both the size of bounding volumes and the depth of the tree.

5.3 Progressive Fracture

Figure 5 shows an example of dynamic scene with progressive fracture. An apple with 6 124 triangles is progressively peeled in several parts, and it undergoes challenging self-collisions when the peels retract. We have applied our BVH restructuring algorithm, and we have compared its performance with the performance of full reconstruction through median splitting. We have also compared the time spent on self-collision detection with both approaches, for a query that returns all intersecting primitives.

The times for BVH update and collision detection for the first 800 frames of simulation are shown in Figure 8. At the end of the sequence, the apple mesh has already grown to 11 152 triangles. Note that we only rebuild or restructure the BVH when some fracture takes place, not when there is only deformation. The time for restructuring the BVH remains consistently under 12 ms, with up to 20 times speedup compared to full rebuilding at some frames.

In our experiments, we found that the quality of the BVH can degrade significantly when many new primitives are added (as is the case in the example, where the size of the mesh almost doubles). One conclusion that can be drawn from this is that our dynamic restructuring algorithm can be complemented with existing work for efficiently refitting BVs or rebuilding parts of the BVH [34, 19]. Therefore, we decided to fully rebuild the BVH of the apple whenever the fitting quality $Q > 2.3$. This explains the 9 spikes in the graph for BVH restructuring, whose cost is amortized over the rest of the simulation. With this approach, the average time for self-collision detection with the restructured BVH is 142.8 ms, only 3.2% more than with a rebuilt BVH (138.3 ms).

5.4 Instantaneous Fracture

We have also applied our dynamic BVH restructuring to instantaneous fracture simulations. Figure 6 shows a pumpkin model (10 000 triangles) that is dropped and fractures instantaneously into a variable number of rigid pieces (i.e., between 2 and 12 in our tests).

As depicted in Figure 8, in this example our dynamic restructuring algorithm outperforms full reconstruction when the resulting number of pieces is smaller than 8. From our experience, this number highly depends on the size of the original mesh and the amount of triangles that are created during fracture (In our case, for 8 pieces the pumpkin mesh grows to 16 136 triangles). The time for restructuring is dominated by the time needed for cloning the trees, and insertion and deletion operations are more costly than refitting and local optimizations due to the large growth of the number of triangles.

As shown also in Figure 8, the fitting quality degrades as the number of fractured pieces increases, and here as well it would be worth incorporating other refitting strategies [19]. However, the time for collision queries degrades less than the fitting quality. As a reference, in the simulation with 8 fractured pieces, the time for a collision query that returns all intersecting triangles reaches a maximum of 61.3 ms with fully rebuilt BVHs, and it grows only to 71.5 ms with restructured BVHs. During the first 200 frames of simulation (when most of the collisions occur), the average collision detection time is 18.6 ms with rebuilt BVHs, and 23.7 ms with restructured BVHs, an average increase of 27%.

6 Conclusions

We have presented novel algorithms for dynamically restructuring BVHs in the presence of fracture simulations, as opposed to reconstructing them from scratch when geometric primitives must be inserted and/or deleted. Our algorithms for progressive and instantaneous fracture rely on the implementation of BVHs using AVL trees as the underlying data structure, which guarantees balanced hierarchies through simple local operations.

Our approach for dynamic restructuring performs particularly well during progressive fracture, when the amount of added triangles remains as a small factor (approximately less than 10%) of the original mesh size. We have demonstrated speedups of up to a factor of 20 with small degradation of the fitting quality. If the size of the mesh continues growing, our local optimization operations cannot guarantee a good fitting quality, and our approach must be complemented with local rebuilding and refitting strategies [19, 34].

In situations of instantaneous fracture, our restructuring algorithm is competitive only when the resulting number of fractured pieces is relatively small. As part of future work, we plan to investigate approaches for bulk insertion and deletion of AVL subtrees [26], which could lay the grounds for more efficient restructuring algorithms for instantaneous fracture. In that case, it is also necessary to design algorithms for grouping triangles of a same

fractured piece under a common AVL subtree.

To conclude, further work should also be devoted to handling self-collision situations during progressive fracture. Simulation information may be exploited to accelerate the culling of newly created crack surfaces that hinder self-collision detection.

ACKNOWLEDGEMENTS

This work was supported by the NCCR Co-Me of the Swiss National Science Foundation.

REFERENCES

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. (Russian) English translation by M. J. Ricci in Soviet Math. Doklady, 3:1259-1263, 1962.

[2] AGEIA PhysX SDK. http://www.ageia.com/developers/downloads.html.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *Proc. of ACM SIGGRAPH*, 2002.

[5] N. A. Carr and J. C. Hart. Two algorithms for fast reclustering of dynamic meshed surfaces. *Eurographics Symposium on Geometry Processing*, 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[7] G. V. den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graphics Tools*, 2(4):1–14, 1997.

[8] S. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum (Proc. of Eurographics'2001)*, 20(3):500–510, 2001.

[9] S. Fisher and M. C. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. *Proc. of EG Workshop on Computer Animation and Simulation*, pages 99–111, 2001.

[10] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[11] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32, 2003.

[12] N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstoff, R. Gayle, M. C. Lin, and D. Manocha. Interactive collision detection between deformable models using chromatic decomposition. *Proc. of ACM SIGGRAPH*, 2005.

[13] B. Heidelberger, M. Teschner, and M. Gross. Real-time volumetic intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*, 2003.

[14] P. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Brown University, 1994.

[15] D. L. James and D. K. Pai. Bd-tree: Output-sensitive collision detection for reduced deformable models. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 2004.

[16] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37, 1998.

[17] D. E. Knuth. *The Art of Computer Programming Vol. 3, Sorting and Searching, Second Edition*. Addison-Wesley, Reading, MA, 1998.

[18] T. Larsson and T. Akenine-Möller. Collision detection for continuously deforming bodies. *Eurographics*, pages 325–333, 2001.

[19] T. Larsson and T. Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. *Workshop on Virtual Reality Interaction and Physical Simulation*, 2005.

[20] M. C. Lin and D. Manocha. Collision and proximity queries. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, 2nd Ed.*, chapter 35, pages 787–807. CRC Press LLC, Boca Raton, FL, 2004.

[21] J. Mezger, S. Kimmerle, and I. Etzmuss. Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG*, 11(2):322–329, 2003.

[22] N. Molino, Z. Bao, and R. Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *Proc. of ACM SIGGRAPH*, 2004.

[23] J. F. O'Brien, A. W. Bargteil, and J. K. Hodgins. Graphical modeling and animation of ductile fracture. *Proc. of ACM SIGGRAPH*, pages 291–294, 2002.

[24] J. F. O'Brien and J. K. Hodgins. Graphical modeling and animation of brittle fracture. *Proc. of ACM SIGGRAPH*, 1999.

[25] S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[26] E. Soisalon-Soininen and P. Widmayer. Amortized complexity of bulk updates in avl-trees. *Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 2368, Springer*, pages 439–448, 2002.

[27] SOLID Collision Detection Library. http://www.dtecta.com/.

[28] D. Steinemann, M. A. Otaduy, and M. Gross. Fast arbitrary splitting of deforming objects. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2006.

[29] A. Sud, N. K. Govindaraju, R. Gayle, I. Kabul, and D. Manocha. Fast proximity computationa among deformable models using discrete voronoi diagrams. *Proc. of ACM SIGGRAPH*, 2006.

[30] M. Teschner, B. Heidelberger, M. Müeller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. *Proc. of Vision, Modeling and Visualization*, 2003.

[31] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Furhmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1), 2005.

[32] P. Volino and N. Magnenat Thalmann. Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces. In D. Terzopoulos and D. Thalmann, editors, *Computer Animation and Simulation '95*, pages 55–65. Eurographics, Springer-Verlag, Sept. 1995. ISBN 3-211-82738-2.

[33] R. Weller, J. Klein, and G. Zachmann. A model for the expeceted running time of collision detection using aabb trees. *Eurographics Symposium on Virtual Environments*, 2006.

[34] G. Zachmann and R. Weller. Kinetic bounding volume hierarchies for deforming objects. *ACM Int'l Conf. on Virtual Reality Continuum and its Applications*, 2006.

Init Size	End Size	T_{old}	T_{decomp}	T_{crack}	Replace	Insert	Refit	Optim.	Total	Midpoint Split	Median Split
5660	6702	16.27	49.00	62.00	0.01	0.09	2.60	4.08	6.78	25.48 (3.76)	49.42 (7.28)
22640	24672	29.73	89.36	125.09	0.02	0.18	9.48	15.99	25.68	100.10 (3.90)	226.07 (8.80)
90560	94560	57.45	172.55	248.55	0.04	1.40	37.02	62.27	100.74	410.60 (4.08)	1491.59 (14.81)

Table 1: **Statistics of Dynamic BVH Restructuring.** A triceratops model of three different resolutions (5660, 22640, and 90560 triangles) is split in 11 steps as shown in Figure 4. The table shows the initial and final size of the mesh; the average number of old split triangles T_{old}, new decomposed triangles T_{decomp}, and new crack triangles T_{crack} for each splitting step; and the average time spent in replacement, insertion, refitting, and optimization operations for each splitting step (all in ms.). The last columns also compare the average total time for dynamic restructuring of the BVH Vs. the time needed for rebuilding the BVH from scratch with midpoint or median splitting. The numbers in parenthesis indicate the average speedup obtained with our dynamic restructuring algorithm.

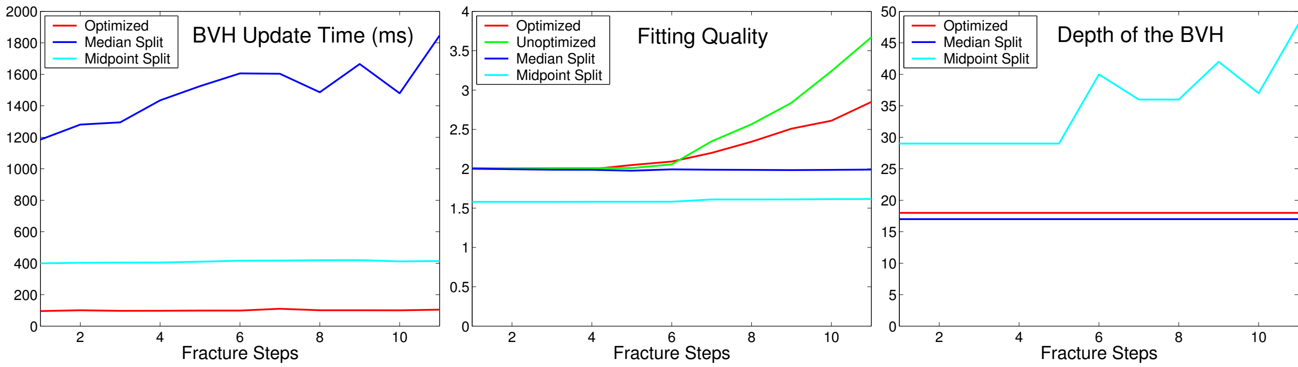

Figure 7: **Performance and BVH Quality in a Static Benchmark.** The data corresponds to 11 steps of progressive fracture for the highest resolution triceratops model (90560 triangles, see Figure 4). *Left:* Comparison of time for dynamic restructuring, time for full reconstruction through median splitting, and time for full reconstruction through midpoint splitting (all in ms.), during the different fracture steps. *Middle:* Evolution of the fitting quality Q with full reconstruction and dynamic restructuring (with and without local optimization operations). *Right:* Evolution of the depth of the BVH.

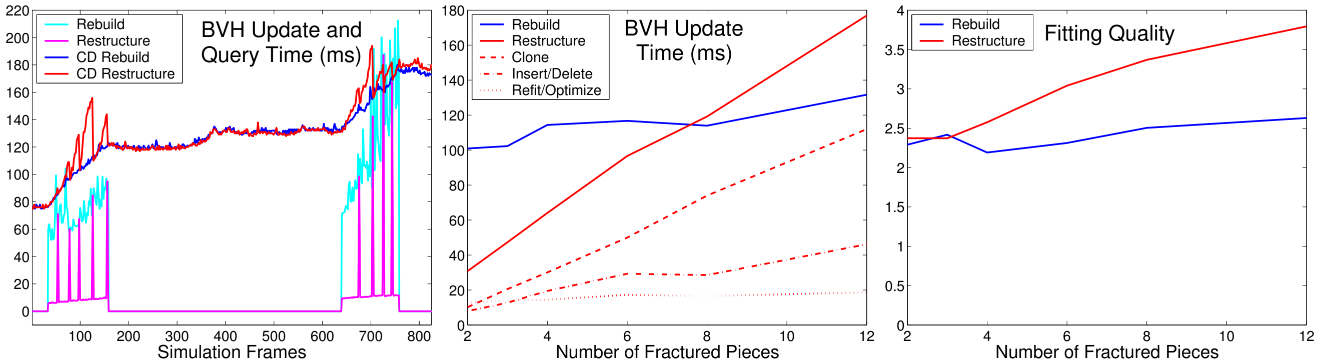

Figure 8: **Performance and BVH Quality in Dynamic Benchmarks.** *Left:* Time spent on BVH update and self-collision detection query (CD) during the progressive peeling of an apple (see Figure 5). The graph compares the times with our dynamic restructuring algorithm and with full BVH rebuilding (in ms.). *Middle:* Performance of dynamic restructuring for instantaneous fracture of a pumpkin into a variable number of pieces (see Figure 6). The graph compares the time spent on different steps of the algorithm and the time for full BVH rebuilding. *Right:* Comparison of the average BVH quality Q for the resulting pumpkin pieces.