Fast Collision Detection for Fracturing Rigid Bodies

Loeiz Glondu, Sara C. Schvartzman, Maud Marchal, Georges Dumont, and Miguel A. Otaduy

Abstract—In complex scenes with many objects, collision detection plays a key role in the simulation performance. This is particularly true in fracture simulation for two main reasons. One is that fracture fragments tend to exhibit very intensive contact, and the other is that collision detection data structures for new fragments need to be computed on the fly. In this paper, we present novel collision detection algorithms and data structures for real-time simulation of fracturing rigid bodies. We build on a combination of well-known efficient data structures, namely, distance fields and sphere trees, making our algorithm easy to integrate on existing simulation engines. We propose novel methods to construct these data structures, such that they can be efficiently updated upon fracture events and integrated in a simple yet effective self-adapting contact selection algorithm. Altogether, we drastically reduce the cost of both collision detection and collision response. We have evaluated our global solution for collision detection on challenging scenarios, achieving high frame rates suited for hard real-time applications such as video games or haptics. Our solution opens promising perspectives for complex fracture simulations involving many dynamically created rigid objects.

Index Terms—Physical simulation, collision detection, fracture, rigid body

1 Introduction

Fracture is commonplace in crashes and explosions, essential ingredients of action entertainment both in feature films and in video games [1]. When objects fracture, multiple object fragments collide and pile up, making fracture simulations extremely collision intensive. The recent advent of fast algorithms for fracture crack computation [2], [3], [4] has made collision handling a dominant cost in fracture simulation.

The simulation of fracture imposes two major challenges on collision handling. First, acceleration data structures for collision detection need to be created and/or updated at runtime, due to topology changes. Second, the newly created crack surfaces arise in parallel close proximity, which constitutes a worst case scenario for collision detection and response, with many surface primitives in contact, less chances for high-level culling, and no temporal coherence. Offline animations may afford spikes in the computational cost at fracture events, with the cost being amortized over the length of the simulation. But in interactive applications such as video games, simulation must comply with a maximum computational budget per time step, calling for efficient solutions at all simulation frames, particularly at fracture events.

In this paper, we present an efficient solution for collision detection among stiff objects undergoing brittle fracture. A first major observation for our solution is that, with very stiff objects, deformations are visually imperceptible; therefore, for collision handling purposes, the objects can be treated as rigid bodies between fracture events. Hence, our approach to collision detection, outlined in Section 3, relies on well-known efficient acceleration data structures for rigid body contact, namely distance fields and sphere trees.

However, distance fields and sphere trees typically rely as well on constant topology, and suffer a heavy preprocessing cost. A second major observation for our solution is that brittle fracture can be considered to be instantaneous [5]; therefore, collision detection data structures may be updated only at fracture events. In our work, we propose novel algorithms for fast reconfigurable distance fields and sphere trees. In Section 4, we present a novel method to compute an approximate interior distance field for fracture fragments. Our method exploits features of fracture simulation and collision response algorithms to optimize its storage and computational cost. In Section 6, we present an augmented sphere tree data structure, well suited for fast updates under fracture events.

The third major observation for our solution is that, at fracture events, the majority of the contacting primitives defines redundant contact constraints. As shown in Section 5, we propose a design of the sphere tree that lays the foundation for a simple self-adapting collision detection algorithm at runtime. It is executed as a part of hierarchical collision detection, not as a postprocess, thus enabling high-level pruning, and reducing the cost of both collision detection and response. Even though we apply our adaptive sphere tree in the context of fracture simulations, it is also applicable to more general simulations involving either rigid or deformable bodies.

In Section 7, we evaluate our data structures and algorithms on several challenging fracture simulations,

- *L. Glondu, M. Marchal, and G. Dumont are with IRISA/Inria, Campus de Beaulieu, Rennes Cédex 35042, France.*
- *S.C. Schvartzman and M.A. Otaduy are with URJC Madrid, Universidad Rey Juan Carlos, Edf. Ampl. Rectorado, D-0052, c/Tulipan s/n, E-28933 Mostoles, Spain.*

Manuscript received 8 Oct. 2012; revised 6 Feb. 2013; accepted 22 June 2013; published online 3 July 2013.
Recommended for acceptance by J. Lee and P. Kry.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2012-10-0223.
Digital Object Identifier no. 10.1109/TVCG.2013.98.

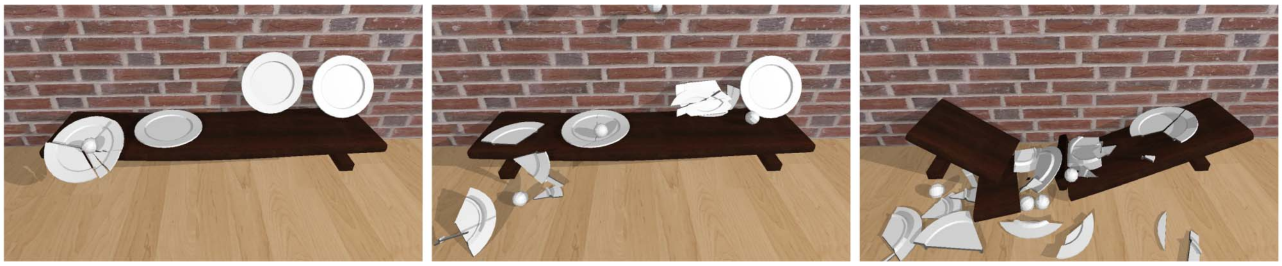


Fig. 1. *Smashing plates*. The user drops balls in real-time to smash the plates, and at the end of the simulation the scene consists of more than 45K triangles. The complete simulation runs at 7.4 ms per time step, on average, with a maximum of 13.2 ms.

demonstrating very high simulation frame rates, suited for hard real-time applications such as video games, as shown in Fig. 1.

2 Related Work

We focus our discussion of related work on the two main data structures used in our method, namely, distance fields and sphere trees, on adaptive collision detection methods, and on collision detection techniques particularly designed for fracture simulations.

Distance fields store in a grid distances to the surface of an object, and possibly the distance gradient. For rigid bodies, distance fields may be precomputed; hence, the computation of penetration depth of a point inside a rigid body becomes trivial [6]. Adaptive distance fields [7] store distances in an octree to reduce storage requirements. In some applications, it is even sufficient to store information only near the surface of the object [8]. Distance fields have also been used for deformable bodies by fast recomputation [9] or by approximating finite-element [10] or modal deformations [11]. In various applications of computer animation, distance fields have been approximated using front propagation algorithms [12] or graph-based distances [13].

Sphere trees are one of the classic types of bounding volume hierarchies for fast pruning in collision detection [14]. Fast culling is possible thanks to the sphere trees, while the adaptive distance field is used for accurate penetration depth queries. Weller and Zachmann [15] designed inner sphere trees for the fast computation of penetration volumes.

Adaptive and time-critical collision detection. One interesting use of sphere trees is time-critical collision detection [16]. The output of time-critical collision detection was later optimized by considering also collision response and adding adaptivity based on visual perception metrics [17]. Other ways to govern adaptivity in time-critical collision detection include error control based on potential intersection volumes [18]. With contact levels of detail [19], adaptive collision detection is extended to arbitrary types of bounding volumes, contact points are computed using surface approximations at each level, and various adaptivity criteria can be considered. Yet a different approach to limit the cost of hierarchical collision detection in an adaptive manner is stochastic sampling [20]. Kaufman et al. [21] augmented adaptive distance fields with sphere trees to design an adaptive collision detection algorithm with guaranteed error bounds. Barbič and James [11] performed time-critical collision detection based on sphere-tree versus distance field queries.

Collision detection for fracture. Acceleration data structures for collision detection need to be updated or recomputed at fracture events, because precomputed distances or bounds are no longer valid or tight, and new surfaces need to be considered. Larsson and Akenine-Möller [22] introduced the concept of selective restructuring of bounding volume hierarchies, according to fitting-quality metrics. Otaduy et al. [23] applied local restructuring operations to limit updates in progressive fracture. Recently, Heo et al. [24] have presented an algorithm that finds a good compromise between restructuring and fast recomputation.

All these approaches suffer two major limitations for simulations of brittle fracture. First, the quality of bounding volume hierarchies degrades immediately under brittle fracture, and full recomputations are needed. As a result, large computational spikes increase the simulation cost at fracture events. Such spikes can be amortized in offline simulations, but not in hard real-time applications such as video games.

Second, earlier research works on collision detection for fracture simulation [22], [23], [24] have typically focused on the problem of surface intersection, which unfortunately does not address the needs of collision response in rigid body engines. Robust and efficient rigid body engines in the video games and feature film industry rely on velocity level constraint-based solvers followed by stabilization or drift correction [25], [26]. These solvers need contact information in the form of penetrating points, distances, and directions, and our collision detection algorithm satisfies these needs.

In an earlier version of our work [27], we presented an algorithm for collision detection in fracture simulations that allowed fast update of acceleration data structures at fracture events, as well as adaptive contact selection for efficient constraint-based collision response. The fast update of data structures was based partly on a sphere tree that is augmented with interior nodes, hence such nodes do not need to be introduced dynamically when new fracture surfaces are exposed. However, the culling efficiency of the tree is reduced after fracture, due to notable imbalance. In this work, we present a fast restructuring of the fracturable sphere tree that largely improves culling efficiency. The contact selection approach in our earlier work was successful at reducing the cost of collision detection and response in comparison to a full contact sampling, but the computational overheads at fracture events were anyway noticeable. In this work, we present a maximally distant node traversal algorithm that leads to a more regular distribution of contact points, and therefore, the possibility to execute reliable collision response with fewer contact constraints.

Complementary to our approach, collision detection for fracture simulations can also benefit in several ways from fast parallel algorithms executed on graphics processors: for fast culling in piles of objects [28], collision detection queries with no need for preprocessing [29], or fast computation of data structures, either distance fields [9] or bounding volume hierarchies [30]. We have designed methods that achieve high performance by reducing computations (i.e., they are less computationally demanding than previous methods), and could also benefit from parallel implementations.

3 Overview of the Collision Detection Algorithm

We execute collision detection tests between pairs of objects A and B, which may be either original unfractured objects or fragments resulting from fracture events. Without loss of generality, let us refer to them as fragments. We augment each fragment A with two data structures for collision detection: a distance field $D(A)$ and a sphere tree $S(A)$. Section 4 describes our *fragment distance field*, its construction, and its update. Section 5 describes our novel *adaptive sphere tree* and its construction. Finally, Section 6 describes how we augment the sphere tree to allow fast updates under fracture, making it a *fracturable adaptive sphere tree*.

When broad-phase collision culling returns the pair (A, B) as potentially colliding, we query both $S(A)$ against $D(B)$ and $S(B)$ against $D(A)$. The result of a query $(S(A), D(B))$ is a set of contact constraints C, each defined by a tuple $(\mathbf{c}, d, \mathbf{n})$. $\mathbf{c} \in A$ is a contact point, d is the closest distance from $\mathbf{c}$ to the surface of B, and $\mathbf{n}$ is the contact normal. If $\mathbf{c}$ is inside B, d is negative and represents the penetration depth. After collision detection, we feed the complete set of contact constraints to a constraint-based contact solver with a velocity-level LCP (with friction), plus constraint drift correction. In our examples, we have used the off-the-shelf contact solver built in Havok Physics.

In our algorithm, a query $(S(A), D(B))$ builds on three elementary queries involving nodes of the sphere tree $S(A)$. Each node is represented by its center point $\mathbf{p} \in A$ and a radius r. Then, the three elementary queries are as follows:

- insideTest$(\mathbf{p}, D(B))$: it determines whether $\mathbf{p}$ is inside or outside B.
- penetration$(\mathbf{p}, D(B))$: when $\mathbf{p}$ is inside, it computes the penetration depth from $\mathbf{p}$ to the surface of B, as well as a penetration direction $\mathbf{n}$.
- sphereTest$(\mathbf{p}, r, D(B))$: when $\mathbf{p}$ is outside, it performs a conservative test for intersection between B and the sphere of radius r centered at $\mathbf{p}$.

These three elementary queries will be described in detail in Section 4. In all our descriptions, we assume that the point $\mathbf{p}$ has been transformed to the local reference system of fragment B.

Our collision detection algorithm, outlined in Algorithm 1, traverses a sphere tree $S(A)$ in a breadth-first manner, and prunes branches that are completely outside the other fragment B. Pruning is efficiently executed by comparing the radius of a sphere and the distance from its center to the surface of B. The algorithm can be easily modified to allow for a contact tolerance ϵ. A contact constraint is added to C if the distance is shorter than ϵ, and the query descends to the children if the distance is shorter than $r - \epsilon$.

Algorithm 1. Query sphere tree S against distance field D.
```
1: INPUT: $S$, $D$
2: OUTPUT: Set of contacts $C$
3: $\mathcal{Q}_1 = \{\text{root}(S)\}$
4: while $done =$ false do
5:   while $\mathcal{Q}_1 \neq \emptyset$ do
6:     $node \leftarrow$ pop_front($\mathcal{Q}_1$)
7:     $inside \leftarrow$ insideTest($node.\mathbf{p}$, $D$)
8:     if $inside$ then
9:       $(d, \mathbf{n}) \leftarrow$ penetration($node.\mathbf{p}$, $D$)
10:      $C = C \cup (node.\mathbf{p}, d, \mathbf{n})$
11:      if sufficientContacts($C$) then
12:        STOP
13:      end if
14:    else
15:      $intersects \leftarrow$ sphereTest($node.\mathbf{p}$, $r, D$)
16:    end if
17:    if $inside$ OR $intersects$ then
18:      $\mathcal{Q}_2 = \mathcal{Q}_2 \cup$ nextChild($node$)
19:      if nextChild($node$) $\neq \emptyset$ then
20:        $\mathcal{Q}_1 = \mathcal{Q}_1 \cup node$
21:      end if
22:    end if
23:  end while
24:  if $\mathcal{Q}_2 \neq \emptyset$ then
25:    $done \leftarrow$ true
26:  end if
27:  swap($\mathcal{Q}_1, \mathcal{Q}_2$)
28: end while
```

We augment the basic collision detection algorithm with *self-adapting contact selection*. As described in Section 5, we construct the sphere tree in a way that allows adaptive contact selection by simple breadth-first tree traversal, defining a contact constraint whenever we encounter a sphere whose center is inside fragment B, until a sufficient number of constraints is reached.

To optimize the efficiency of contact selection, we traverse the nodes of the sphere tree in a maximally distant order. This traversal order is achieved by appropriately ordering the nodes of the sphere tree when it is built, and populating the traversal queue by alternating children of colliding nodes. In Algorithm 1, we use two traversal queues, Q_1 and Q_2, which are swapped after each level of the sphere tree is processed. Full details about our self-adapting contact selection are given in Section 5.

4 Fragment Distance Field

Given a volumetric meshing of an object A, computed as a preprocess, we propose a fragment distance field data structure that is efficiently stored and updated even upon

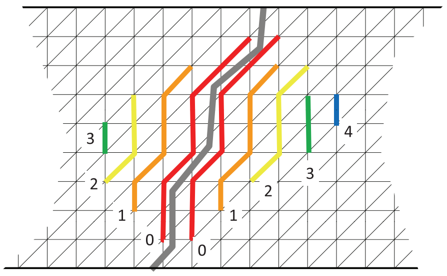


Fig. 2. Illustration of the front propagation algorithm for interior distance field computation.

multiple fractures of the object. This data structure stores an approximate interior distance field of all fragments created by the fractures, using a precomputed volumetric mesh, without any remeshing. Moreover, we exploit the connectivity of the mesh to compute approximate distances in a very fast manner using a front propagation approach.

In this section, we first describe the distance field data structure and its runtime update, and then we describe how it is used to answer the three elementary queries outlined in the previous section.

4.1 Mesh-Based Interior Distance

Our distance field data structure is motivated by features of fracture simulation algorithms. First, with very stiff objects, the deformations are visually imperceptible, and distances may only be updated at fracture events. Second, no energy is lost during brittle fracture, and the resulting fragments define an exact partition of the original object. Therefore, each point of the original object needs to store only one interior distance value even after multiple fractures. And third, popular approaches for fracture simulation use a volumetric mesh to compute an elastic deformation field and guide the propagation of crack surfaces [5], [31]. The virtual node algorithm and subsequent adaptations [3], [4], [32] limit the resolution of newly created fragments, forcing them to include one node of the original mesh. We exploit this feature and store one interior distance value at each node of the original volumetric mesh.

In our implementation, we have used a tetrahedral mesh as volumetric mesh for storing the fragment distance field. Specifically, each node of the mesh stores:

- f: an identifier of the fragment that contains the node.
- d: a value that approximates the shortest distance to the surface of fragment f.
- $\mathbf{n}$: a unit vector that approximates the direction from the node to the closest surface of f.

As a preprocess, we initialize the nodal information using an exact interior distance field.

In addition to nodal information, tetrahedra that are intersected by crack surfaces store exact local representations of those crack surfaces. Following the virtual node approach, each tetrahedral edge may be cut at most once, therefore, the storage requirements are limited to six plane equations.

4.2 Distance Updates upon Fracture

After each fracture event, we locally update the fragment distance field where needed, following a front propagation approach. The runtime computation of the exact distance field is computationally prohibitive, but we propose a fast approximation that fulfills desired properties. It is important to remark that the interior distance of a fragment is used in the computation of penetration depth and contact normal in the collision detection Algorithm 1. The amount of penetration depth is used by the drift correction algorithm during collision response, and the contact normal is used for the definition of nonpenetration contact constraints. Distances need to be accurate close to the surface of an object, grow monotonically in the interior, and locally approximate euclidean distance. Normal directions, on the other hand, must point outward of the object, and should vary smoothly to avoid competing contact constraints for nearby points. It turns out that the algorithm for consistent penetration depth computation of Heidelberger et al. [12] fulfills exactly these properties; hence, we have adapted this algorithm for interior distance field approximation.

Next, we summarize the application of Heidelberger's algorithm to our problem. When an object A suffers a fracture, we first visit all tetrahedra intersected by the newly created crack surfaces, and initialize distance field information at their nodes. This implies assigning a fragment identifier f, and computing a distance d and a direction $\mathbf{n}$, based on the exact surface information stored at the tetrahedra. For each fragment, we initialize a front with the visited nodes. Then, we iterate front propagation steps on the graph defined by tetrahedral edges, until no distances are reduced. Fig. 2 illustrates the front propagation inside a fragment.

If the front propagation reaches a node at position $\mathbf{p}$ in step $i+1$, we compute a distance d to the surface as an average propagation of distances from its neighbors reached in step i (denoted as $N_i(\mathbf{p})$), in the following manner:

$$d = \frac{\sum_{j \in N_i(\mathbf{p})} w_j \big(d(\mathbf{p}_j) + \mathbf{n}(\mathbf{p}_j)^T (\mathbf{p}_j - \mathbf{p})\big)}{\sum_{j \in N_i(\mathbf{p})} w_j}. \tag{1}$$

Following Heidelberger et al., we use as neighbor weights $w_j = 1/\|\mathbf{p}_j - \mathbf{p}\|^2$. If the distance d is shorter than the current value stored at $\mathbf{p}$, we update the distance and add $\mathbf{p}$ to the front at step $i+1$. We also update the direction at $\mathbf{p}$ as the weighted average direction of neighbors reached in step i:

$$\mathbf{n} = \sum_{j \in N_i(\mathbf{p})} w_j \, \mathbf{n}(\mathbf{p}_j), \tag{2}$$

followed by a normalization step.

Fig. 3 shows an accurate interior distance field for an object A, the accurate distance fields of its fragments after

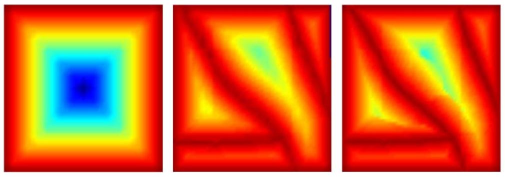


Fig. 3. From left to right: interior distance field of a 2D object, distance fields of its four fragments after fracture, and our approximate distance fields computed using a fast propagation method.

fracture, and our approximate distance fields. The image illustrates the monotonic growth of distances inside the fragments. Our distance field approximation does not require high-quality tetrahedral meshes in practice. In our examples, we used TetGen for mesh generation, with maximum radius-edge ratio between 1 and 2 and interior edges shorter than twice the length of the longest surface edge.

Even though we have used tetrahedral meshes, our data structure could be extended to other settings, such as hexahedral meshes or even meshless methods. The mesh is used in two ways: 1) Its edges define a graph for the propagation of distances; and 2) distances can be interpolated inside mesh elements. On hexahedral meshes, the graph may be constructed using the edges of the mesh or adding other connections, and interpolation can also be defined inside mesh elements. On meshless methods, a graph may be constructed using neighboring particle information which is easily updated upon fracture events [13], and interpolation can be defined based on neighboring nodes [33].

4.3 Inside-Outside Query: InsideTest($\mathbf{p}, D(f)$)

As a preprocess, we build a k-d tree with the tetrahedra of the mesh. To decide whether a point $\mathbf{p}$ is inside a fragment f, we first use the k-d tree to retrieve the tetrahedron that encloses $\mathbf{p}$. If all nodes of the tetrahedron are in the same fragment, the query is trivially answered. If only some nodes are in fragment f, we use the exact local representation of the crack surfaces to answer the inside-outside query.

4.4 Penetration Depth Query: Penetration($\mathbf{p}, D(f)$)

If the tetrahedron containing a point $\mathbf{p}$ is intersected by crack surfaces, we use the exact local surface information to compute the penetration depth and direction to the surface of fragment f. If the tetrahedron is completely inside the fragment, we use the fragment distance field. In particular, we use as neighbors $N_i(\mathbf{p})$ the four nodes of the tetrahedron, apply (1) to compute the distance to the surface of f, and (2) to compute the penetration direction.

Close to original surfaces of the object, where fracture does not modify distances, it is possible to obtain more accurate penetration information in a simple manner. As a preprocess, we compute a distance field on a dense regular grid. This regular-grid distance field is used for the initialization of the fragment distance field at nodal positions, but we also query it at runtime. We simply use the minimum of the distances returned by the (precomputed) regular-grid distance field and the (dynamically updated) fragment distance field.

4.5 Sphere Intersection Query: SphereTest($\mathbf{p}, \mathbf{r}, D(f)$)

The fragment distance field stores only interior distance information for the fragments. When the query point $\mathbf{p}$ is outside fragment f, the fragment distance field provides the distance d to the surface of some other fragment. This distance d is a lower estimate of the distance to f, and can be used for culling in Algorithm 1 if it is larger than the radius r of the sphere. To handle far fragments, we use the largest between d and the distance to a bounding box of fragment f.

The procedure described above performs well in most cases, but fails for large nonconvex fragments surrounded by small fragments, returning largely underestimated distances that produce little culling. We provide a less conservative solution for such situations. As a preprocess, we construct a multilevel grid on every object, and register pointers from the tetrahedra to their occupied cells. Every grid cell stores a bit mask indicating whether it contains each and every fracture fragment. Upon a fracture event, we traverse the tetrahedra of new fragments, mark the bit masks of their occupied cells, and then we perform a bottom-up update of the multilevel grid by simple logical AND operations. To test a sphere for intersection, we query the grid level with cell size immediately larger than $2\,r$. The sphere can be culled if none of the eight cells joining at the grid point closest to $\mathbf{p}$ contains fragment f.

5 Adaptive Sphere Tree

In this section, we introduce a sphere tree data structure that is suitable for adaptive collision detection. Our major goal was to reduce the cost of both collision detection and response in fracture simulations, in particular at collision-intensive fracture events, but our adaptive sphere tree is applicable also to more general simulations involving either rigid or deformable bodies. We achieve effective adaptivity thanks to a *maximum-distance ordering* of points, applied at two stages: at the construction of the tree, and during tree traversal. As a result, we propose a self-adapting collision detection algorithm that is easily integrated in typical hierarchical collision detection.

5.1 Ordering and Construction of the Sphere Tree

We build a sphere tree on a set of points $P = \{\mathbf{p}_i\}$ representing an object A. In this section, we assume that P is formed by the vertices in the surface of A, but in Section 6 we augment P with interior points to create a fracturable sphere tree.

Given m points visited as a part of a collision query, adaptive collision detection will be effective if the m points provide a uniform sampling of the surface of the object, for any value of m. Based on this observation, to construct the sphere tree we insert the points in P in an order that maintains a close-to-uniform sampling after every insertion. Specifically, we insert points following a maximum-distance ordering. During a collision query, described in detail in the next sections, we traverse the points in the same order as they are inserted.

We initialize an ordered list L_2 with the two furthest points in P. Then, given the ordered list with m points, L_m, we append the point that is furthest from its closest point in L_m, i.e., $L_{m+1} = (L_m,\ \mathbf{p}^*)$, with $\mathbf{p}^* = \arg\max_{\mathbf{p}i \notin P_m} \min_{\mathrm{p}j \in P_m} \|\mathbf{p}_i - \mathbf{p}_{j\|}$. Given the full ordered list, level l of a sphere tree, with 2^l nodes, is trivially constructed by selecting the first 2^l points in the list. Then, level $l+1$ is constructed using those same 2^l nodes and the following 2^l nodes in the list. We set as parent of a node in level $l+1$ its closest node in level l. This heuristic groups nodes based on proximity and increases the chances for pruning during runtime queries. Fig. 4 shows a 2D example with the maximum-distance ordering and the

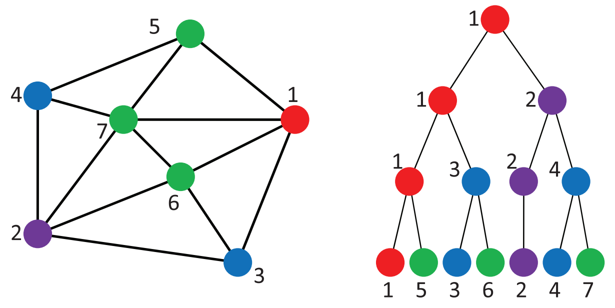


Fig. 4. A 2D polygon with surface vertices and interior nodes (left) and its sphere tree (right). The points are numbered according to maximum-distance ordering and colored according to their insertion level.

tree construction. The sphere tree construction is a preprocess, and we have followed an unoptimized $O(n^2)$ implementation based on pairwise distance computation, but accelerations are possible.

Our maximum-distance ordering heuristic does not guarantee uniform sampling. In our implementation, we used euclidean distance for the ordering, and results could be improved using geodesic distance. In addition, uniform sampling is not a sufficient condition for good adaptivity. The sampling could be improved by, for example, favoring convex features. Barbič and James [11] present a relaxation method for close-to-uniform sampling. However, they maintain good sampling on full levels of the tree, not after every individual insertion.

Each node of the tree must store the sphere center (i.e., the point position $\mathbf{p}$) and radius. For a node with center at $\mathbf{p}$, we precompute the radius as the distance to its furthest descendant. Each point may be present at multiple levels in the tree (but with different radii). We define a contact constraint only the first time the in Algorithm 1, and we point is queried cache its inside-outside status for subsequent queries down the tree. Note that choosing the point $\mathbf{p}$ as the center of the sphere does not yield optimally tight spheres. We tried instead approaches that produce tighter spheres with better culling, but the overall query times were worse as we could not exploit caching.

5.2 Self-Adapting Collision Detection

The fragment distance field and the fracturable sphere tree enable fast queries and fast data structure updates upon fracture. However, in situations with many penetrating points or with parallel surfaces in close proximity, the cost of collision detection is inevitably high, and collision response is affected by the large number of contacts. We have designed a self-adapting collision detection algorithm that produces a reduced set of contact constraints. Our algorithm relies on a velocity-level constraint-based contact solver plus drift correction, the gold standard solution in rigid body engines in video games. Under effective drift correction, our algorithm guarantees that final resting configurations are free of penetrations.

During breadth-first traversal of the sphere tree, we may output contact constraints high in the hierarchy as outlined in Algorithm 1. Thanks to the good sampling provided by the maximum-distance ordering, a few of the first encountered contacts are probably sufficient for the velocity-level constraint-based solver, while further contacts become

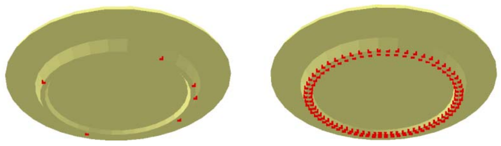


Fig. 5. Rolling plate on a transparent ground, with contacts output by (left) our self-adapting collision detection (up to 6), and (right) full collision detection (up to 128).

redundant. We initialize a collision query between two fragments f_i and f_j by setting a maximum number of contacts m (eight in our experiments), and if this number is reached we simply interrupt the query. Drift correction quickly resolves the contacts that have been detected, but if this number is m, then other contacts may have been missed. In that case, we increment $m \leftarrow m+1$, and continue the sphere tree traversal with a negative tolerance $-\epsilon$ (in our experiments $\epsilon = 0.2\%$ of the object radius), i.e., we search for contacts that penetrate further than ϵ. Effectively, with this approach collision detection self-adapts until the number of contacts guarantees nonpenetration up to a tolerance ϵ at resting configurations. Fig. 5 compares the number of contacts for a 5,392-triangle plate rolling on a transparent ground with our approach versus full collision detection. Self-adapting collision detection requires at most six contacts, while full collision detection outputs up to 128 contacts. In self-adapting collision detection, adaptivity could also be guided by error metrics of collision response, but existing approaches do not address the complex interactions of constraint-based collision response.

We found that, to be effective at fracture events, self-adapting collision detection requires a small positive detection tolerance ϵ, i.e., we output contacts closer than a small distance ϵ. The reason is that the tree traversal stops only when m contacts are output, and parallel surfaces just about to touch would allow little culling but produce no contacts.

5.3 Maximum-Distance Breadth-First Traversal

In a regular breadth-first collision query, nodes on one level of the sphere tree are processed before visiting any node in the next level. This traversal strategy is typically implemented using a queue [27]. When a node collides, its children are pushed into the queue, and the query continues by popping the first node in the queue. Sibling nodes are close-by in space; hence, the regular breadth-first traversal does not produce an appropriate visit order for efficient self-adapting collision detection on a given level of the tree. Here, we present a new traversal strategy that leads to efficient self-adapting collision detection both across levels of the tree and on a given level of the tree. As shown in Fig. 6, it leads to robust collision response with fewer contacts.

Our traversal strategy is based on the heuristic that nodes with different parents are more likely to be distant. Therefore, we push into the breadth-first traversal queue nodes with different parents first, and then we push their siblings. As shown in Algorithm 1, our strategy is implemented, in practice, by storing two different queues: a queue of nodes being visited at the current level, Q_1, and a

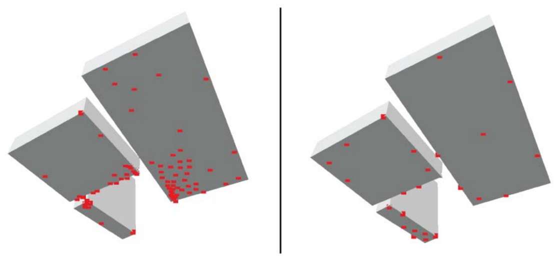


Fig. 6. Blocks on a transparent ground, showing the distribution of contact points under self-adapting collision detection. *Left*: with naïve restructuring and breadth-first traversal [27], many contacts are required to ensure a robust response, due to their nonuniform distribution. *Right*: with our tree restructuring algorithm and maximum-distance breadth-first traversal, the response is robust with fewer contacts, thanks to their improved distribution.

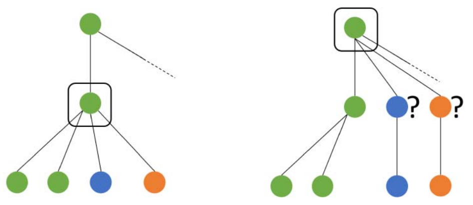


Fig. 7. Tree splitting, with colors of nodes indicating fragment identifiers. *Left*: The marked node is a splitting node, because its children belong to different fragments. *Right*: Two new copies of the splitting node (marked with "?") are created, and the splitting propagates upward. The new nodes will receive representative points as part of tree restructuring.

queue of nodes to be visited at the next level, Q_2. In addition, for each node, we maintain an identifier of the last child pushed into Q_2. When a node n collides, we push its next child into Q_2, and in case it has additional children, we push n itself into Q_1. In this way, n will be tested again after other nodes are pushed into Q_2, and its children will be pushed far apart.

6 Fracturable Sphere Tree

When a body fractures, new collision detection data structures must be created for the resulting fragments, and these data structures must account for both original surfaces and new fracture surfaces. We propose to leverage the volumetric mesh used by most fracture algorithms to build a fracturable sphere tree which can be efficiently updated. In addition, we propose to create sphere trees for the fracture fragments by quickly splitting and restructuring the original tree, while optimizing the maximum-distance ordering for efficient self-adapting collision detection.

6.1 Construction of the Tree

New fracture surfaces pass through the interior of the original object; therefore, we propose to augment the sphere tree with nodes of the volumetric mesh used by fracture algorithms. Specifically, we augment the set of points P described in Section 5.1 with the nodes of the volumetric mesh.

As shown in the example in Fig. 4, the maximum-distance ordering naturally places interior points low in the sphere tree. As a result, prior to fracture, interior parts are easily culled and produce almost no overhead, thanks to self-adapting collision detection. After fracture, on the other hand, interior parts are exposed with minor modifications to the sphere tree, and quickly accessed during tree traversal.

6.2 Tree Splitting and Restructuring

At a fracture event, we create a sphere tree for each resulting fragment. We decompose this process into three tasks: splitting the original tree, restructuring each fragment's tree, and adding new points.

To split the original tree, we first tag each point in P with an identifier of the fragment it belongs to. Tree nodes whose children belong to different fragments are marked as *splitting nodes*, and indicate where the tree should be split, as shown in Fig. 7. Then, we replicate each splitting node as many times as the number of fragments represented in its children, and connect each child with the copy of the splitting node in the same fragment.

We mark and replicate splitting nodes following a bottom-up pass of the fracturable sphere tree, because split operations may propagate upward in the tree. We also remove nodes with only one child. At the end of the splitting process, the original tree is naturally split into as many copies as fracture fragments.

After a tree is split, a node n, copied from a splitting node n' with representative point $\mathbf{p}'$, lacks its own representative point $\mathbf{p}$. We propose a tree restructuring approach that searches in the subtree rooted at n for a point that favors the maximum-distance ordering described in Section 5.1. In practice, we do this by selecting the point $\mathbf{p}$ in the subtree of n that is closest to $\mathbf{p}'$. Assuming that $\mathbf{p}$ is found at node n_p, $\mathbf{p}$ is set as the representative point for all nodes in the branch from n to n_p.

6.3 Addition of New Surface Points

When an object is fractured, we also add new surface points to the sphere trees, to appropriately sample the newly created fracture surfaces. In our implementation, we add two new points, each on a different fragment, for each edge of the volumetric mesh that is cut by the fracture surfaces.

To insert each new surface point $\mathbf{p}$ in the sphere tree, we favor again the maximum-distance ordering. The insertion requires picking a parent point $\mathbf{p}_p$ and a level in the hierarchy, because $\mathbf{p}_p$ may be present at multiple levels. Recall that $\mathbf{p}$ is the result of cutting an edge of the volumetric mesh; therefore, the edge-point in the same fragment as $\mathbf{p}$ is by construction close to it, and constitutes a good choice as parent $\mathbf{p}_p$.

The strategy to restructure the sphere tree and add new surface points in our previous work [27] is oblivious of the maximum-distance ordering. As a result, our novel approach leads to much more efficient self-adapting contact selection, as shown in Fig. 6 and further discussed in the next section.

To choose the level of insertion, note that, due to the maximum-distance ordering, each level of a sphere tree can be regarded as a level in a multiresolution representation of an object. Conceptually, we wish to insert $\mathbf{p}$ at the appropriate level of detail. The size of sphere bounds is a good indicator of the resolution of each level of detail; therefore, we choose to insert $\mathbf{p}$ as a child of $\mathbf{p}_p$ at the level

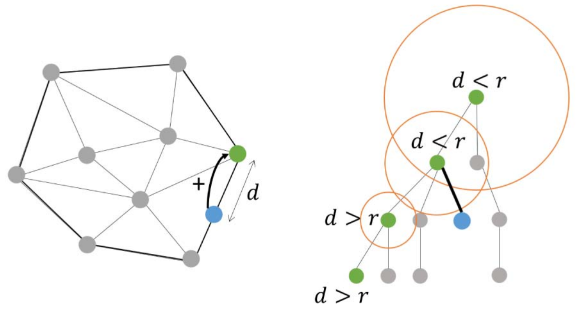


Fig. 8. Addition of a point into the sphere tree. A new point **p**, in blue, is added as a child of its adjacent point $\mathbf{p}_p$, in green, at the tree level where the sphere radius r is just larger than the distance d between **p** and $\mathbf{p}_p$.

where the radius of the sphere bound, r, is just larger than the distance $\|\mathbf{p} - \mathbf{p}_p\|$. This heuristic, depicted in Fig. 8, inserts **p** at the lowest level where there is some other descendant of $\mathbf{p}_p$ farther from $\mathbf{p}_p$ than **p**. If such a level does not exist, we simply insert **p** as a child of the highest instance of $\mathbf{p}_p$, and we update sphere radii bottom-up.

7 Experiments and Results

We evaluated our approach on five scenarios:

1. a piggy bank dropped on the ground (see Fig. 10),
2. twenty-seven bunnies dropped at different times (see Fig. 13),
3. thirty-two bricks crashed against the ground (see Fig. 14),
4. an interactive scenario where the user drops balls on four plates placed on a shelf (see Fig. 1), and
5. another interactive scenario where the user manipulates and fractures five bunnies (see Fig. 9).

The sizes of the surface and volumetric meshes of the different objects are summarized in Table 1. Our collision detection algorithm has been integrated with the rigid body engine of Havok, and we have used a fast fracture simulation method based on modal analysis [4]. The "freezing" utility of Havok Physics was deactivated in all experiments, for better analysis. All experiments were executed on a 3.4-GHz Intel i7-2600 processor with 8 GB of RAM, using only one core.

7.1 Overall Performance Analysis

Tables 2 and 3 report various simulation statistics and timings for the five benchmarks. The "piggy bank," "bricks," "plates," and "interactive bunnies" benchmarks are all real-time, including dynamics simulation, fracture simulation, collision detection, and collision response. The "drop bunnies" scenario, on the other hand, was executed with a subframe time step (5 ms) to illustrate robust contact handling with small fragments and high impact velocities.

Fig. 9. The user manipulates bunnies interactively with the mouse, producing fractures and collisions. The complete simulation runs at 2 ms per time step on average, with a maximum of 10 ms.

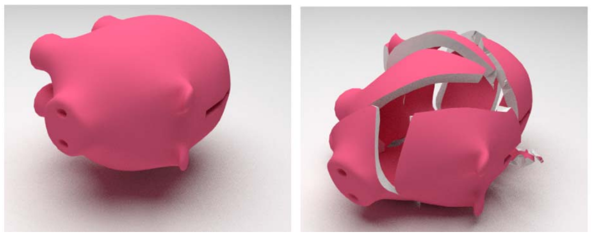


Fig. 10. Piggy bank used for comparisons and analysis.

Fig. 12 shows plots of timings and simulation statistics for the "drop bunnies" and "bricks" scenarios. In both examples, the cost of collision detection grows steadily as more objects are dropped. However, we can draw the important observation that collision detection does not suffer noticeable spikes at fracture events, despite the large number of colliding points, thanks to our self-adapting contact selection. Both scenarios show computational peaks at fracture events due to the cost of fracture computation. The cost of data structure updates was always smaller than the cost of fracture computation, and is not showed for clarity (but it is summarized in Table 3).

7.2 Influence of Resolution

We have analyzed the influence of resolution (both for the surface mesh and the interior sampling of the volumetric mesh) on data structure updates and collision detection queries (for the sphere in Fig. 11). The timings for a reference sphere (2.5K triangles and 4K interior points) are: 1.54 ms for updates upon fracture, and 1.16 ms on average (3.27 ms max) for queries. Changing the surface resolution (to 10K triangles), while keeping the interior sampling fixed, timings are: 1.8 ms for the update, and 1.37 ms on average (4.17 ms max) for queries. Changing the interior sampling (to 435 points), while keeping the surface fixed, timings are: 0.46 ms for the update, and 0.68 ms on average (2.26 ms max) for queries.

7.3 Analysis of Update Overhead

We have analyzed the overhead introduced in collision detection queries by our data structures, which trade fast updates upon fracture for not fully optimal culling. Fig. 15 plots several comparisons for the "piggy bank" scenario in Fig. 10. Our approach updates the distance field (D) and the

TABLE 1
Number of Triangles, Tetrahedra, and Points (Including Surface Vertices and Interior Points) for the Different Objects Used in the Experiments

Object	# Triangles	# Tetrahedra	# Points
Piggy bank	9,722	20,807	5,870
Bunny	7,940	18,767	5,089
Brick	468	594	224
Plate	5,392	8,617	2,711
Shelf	4652	10,200	2,989

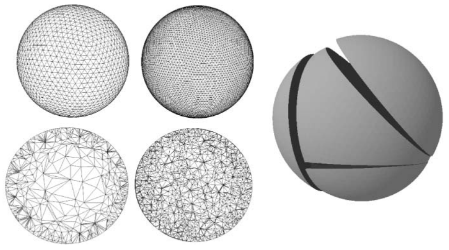


Fig. 11. Sphere used for the analysis of sampling resolution on update and query costs. The top left images show two different samplings of the surface, and the bottom left images show two different samplings of the interior.

sphere tree (S) when the piggy bank crashes. We have compared collision detection query times and the number of visited points in the sphere trees, with other combinations where we recompute the exact distance field and/or we recompute a sphere tree for the new surface (with no interior points).

7.4 Evaluation of Self-Adapting Collision Detection

We have carried out several experiments to evaluate the impact and performance of our self-adapting collision detection algorithm. First, we have analyzed the influence of the number of contact points allowed on the penetration error and the cost of collision detection on practical scenarios. The figure in the inset shows a plate which is dropped with random linear and angular velocities into a bowl. We have executed several trials, fixing the number of contacts in the range from 2 to 50. For each trial, we ran 10,000 simulations, measuring also the penetration error in each simulation frame with full collision detection. A plot of the results is shown in Fig. 16. We observe that the penetration error drops quickly in the range from 2 to 8 contact points, and adding more contact points barely improves quality. Intuitively, this can be explained because six nonredundant constraints may be sufficient to lock the motion of a rigid body. Our maximum-distance ordering and breadth-first traversal of the sphere tree ensure that the contact points are well distributed over the contacting areas; hence, using more than eight contact points will reduce the error only when the contact points are not optimally chosen.

Fig. 6 demonstrates that our novel tree restructuring and traversal algorithms, which favor maximum-distance

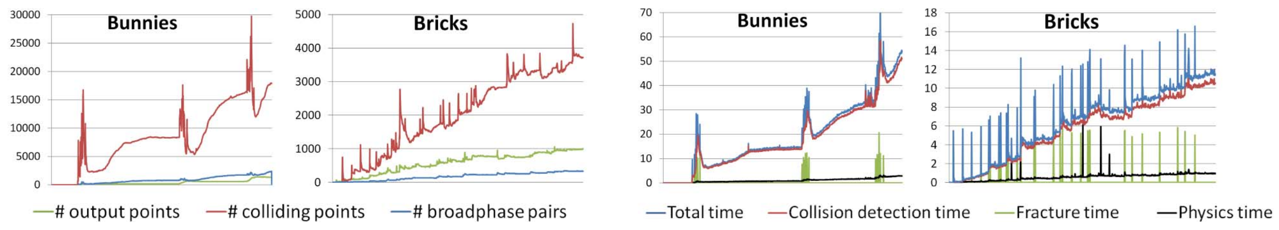


Fig. 12. Plots for the “drop bunnies” scene (first and third plots) and the “bricks” scene (second and fourth plots). The two left plots show collision detection statistics: number of points output by adaptive collision detection (green), number of actual colliding points (red), and pairs of bodies output by broad-phase collision detection (blue). The two right plots show timings per time step: total (blue), collision detection (red), fracture computation (green), and physics computations (black). Times to update collision detection data structures were always shorter than fracture computation, and are not included for clarity.

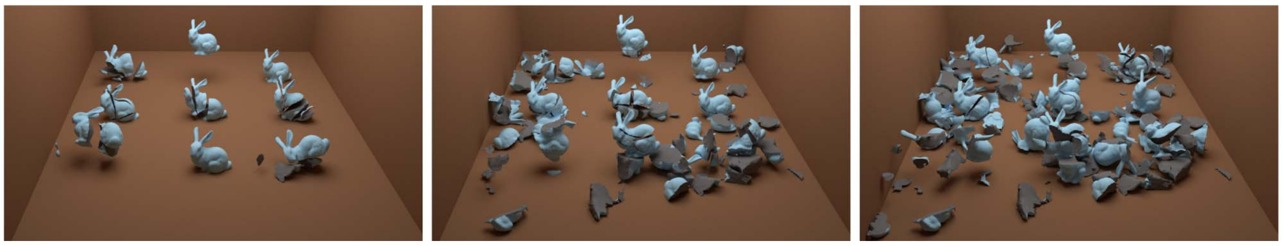


Fig. 13. Bunnies are dropped in three batches and fractured into 166 fragments and 435K triangles. The complete simulation runs at 24.5 ms per time step on average, with a maximum of 81.5 ms.

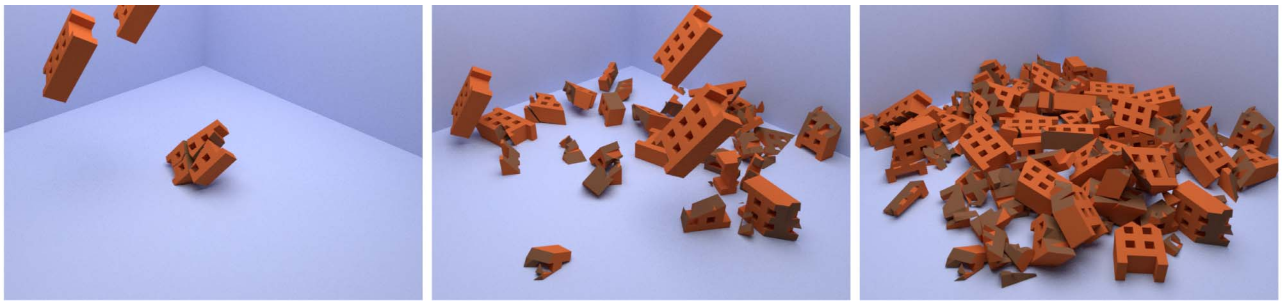


Fig. 14. Real-time demo of crashing bricks, totaling 156 fragments and 40K triangles. The complete simulation runs at 11.7 ms per time step on average, with a maximum of 29.5 ms.

TABLE 2
Simulation Statistics for the Different Scenarios: Number of Triangles of the Scene before and after Fracture; Total Number of Fragments; Number of Contacts Selected by Collision Detection for Collision Response; and Total Number of Colliding Points (Not Measured in the Interactive Scenarios)

Scenario	# Triangles Original/Fractured	# Fracture Fragments	# Output points Average (Max)	# Colliding points Average (Max)
Piggy bank	9,734 / 18,889	27	151 (914)	574 (7,281)
Drop bunnies	137,403 / 430,072	166	866 (2,393)	8382 (29,782)
Bricks	15,036 / 41,060	104	575 (1,055)	2,014 (4,726)
Plates	26,268 / 45,040	44	331 (567)	X
Interactive bunnies	39,724 / 44,064	15	88 (198)	X

TABLE 3
Break-Up of Timings for the Different Scenarios, All Given in Milliseconds, and Showing Average and Maximum Cost per Time Step: Time Step Size (with Frames Rendered Every 30 ms); Total Cost per Time Step; Time for Collision Detection Queries; Time for Physics Computations, Numerical Integration, and Collision Response; Time for Data Structure Updates; and Time for Fracture Computations

Scenario	Time step	Total time Average (Max)	Collision detection Average (Max)	Physics Average (Max)	Update Max	Fracture Max
Piggy bank	15	1.89 (13)	1.63 (11)	0.14 (0.59)	1.2	12.5
Drop bunnies	5	19.33 (73)	18 (58)	1.3 (3.66)	4.4	22
Bricks	30	6.4 (16.5)	5.6 (12.8)	1 (2.46)	1.05	2
Plates	30	7.37 (13.2)	5.8 (9.6)	0.36 (0.7)	0.7	8
Interactive bunnies	15	1.6 (7.3)	1.24 (2)	0.26 (0.41)	1	9

The last two times are measured only at fracture events.

ordering, produce a better sampling of contact surfaces, and hence allow self-adapting collision detection to exit with fewer contacts. With the naïve restructuring and traversal approaches in our earlier work [27], a restrictive self-adapting collision detection leads to excessive interpenetration, which cannot be eliminated by drift correction without noticeable artifacts.

We have also compared our novel restructuring and traversal methods with our earlier naïve approaches from a performance point-of-view. Fig. 17 shows timing comparisons for the "Drop bunnies" benchmark from Fig. 13. Our new methods lead to considerably better timings, thanks to more efficient high-level culling. Note also that the new maximum-distance traversal has barely any computational overhead. As a stress test, we have also evaluated the performance on the "Drop bunnies" benchmark with a fixed number of contacts, with no dynamic adaptation. From the observations made above, we have fixed the number of contacts to 8 for each pair of bodies. As expected, the computational cost is lower, almost constant during the intervals when the number of bodies in the scene does not change. With our earlier restructuring and traversal, the total average and maximum numbers of contacts per frame in this scene were 1,235 and 3,245, respectively (see result tables in [27]). With our novel restructuring and traversal, these numbers go down to 866 and 2,393 (see Table 2), and fixing the number of contacts per body pair to 8 the number go down to 755 and 1,974. In this benchmark, we observed no perceptible penetration errors when fixing the number of contacts, but this approach is not guaranteed to perform well in a general setting.

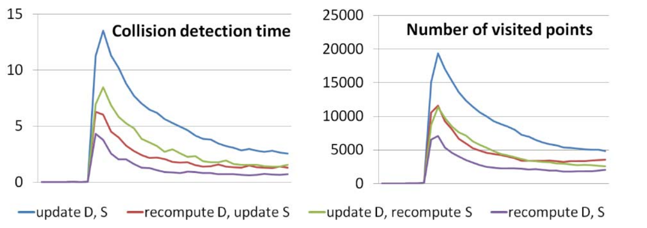


Fig. 15. Comparison of query times and statistics for the "piggy bank" scene. We compare our fast update of the distance field D and sphere tree S to full recomputation of an exact distance field and/or a sphere tree of the new surfaces. We achieve similar culling efficiency with much faster updates at fracture events.

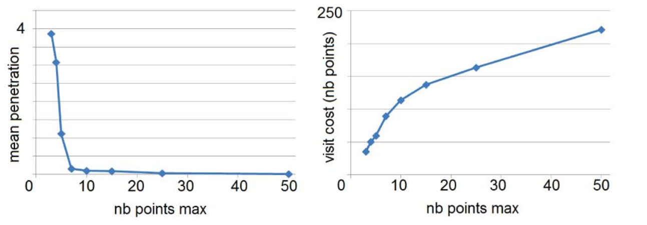


Fig. 16. Analysis of self-adapting collision detection. Plots of the mean penetration and the number of visited points versus the number of contact points allowed.

8 Discussion and Future Work

In this paper, we proposed an efficient solution for collision detection in simulations of fracturing rigid bodies. Our solution is composed of algorithms that address the two main challenges in such simulations: the update of

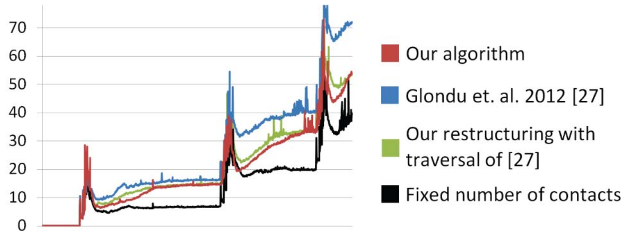


Fig. 17. Plots of total simulation times (ms) of the dropping bunnies scenarios, using different visiting and restructuring algorithms.

acceleration data structures upon topology changes, and the efficient computation of contacts between newly created crack surfaces. One of the main features of our solution is a maximum-distance ordering of nodes in a sphere tree, which serves as the basis for a self-adapting collision detection algorithm. We enforce maximum-distance ordering during the construction of the tree, but also during tree restructuring at fracture events, and during tree traversal as part of collision queries.

Our algorithms demonstrate high performance in challenging scenarios, including real-time user manipulation of fracturing objects, and scenes with hundreds of fragments and tens of thousands of triangles simulated at video game rates. Some limitations remain however, including the possibility to miss collisions with small features and robustness problems under large penetrations. Solving these limitations requires nontrivial extensions to incorporate continuous collision detection.

We envisage other interesting extensions as well. One is the design of parallel versions of our algorithms, to exploit the computing power of graphics processors. Another one is the application of our solutions to penalty-based collision response. The self-adapting collision detection was designed for constraint-based response algorithms and may not be trivially adapted, but other components, such as the fragment distance field, may be easily integrated. Yet another interesting extension is handling ductile and/or progressive fracture and elastic deformations. Since our approach already updates data structures at fracture events, it should also be possible to update those data structures as objects deform and fractures progress, but the extension is not straightforward. Distance fields and sphere trees could also serve for self-collision detection algorithms.

The results of our experiments open promising perspectives for the use of our solutions in real-time applications such as video games and haptic interaction.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This research was supported in part by the Spanish Ministry of Economy (TIN2009-07942 and TIN2012-35840) and by the European Research Council (ERC-2011-StG-280135 Animetrics).

References

[1] M. Baker, N.B. Zafar, M. Carlson, E. Coumans, B. Criswell, T. Harada, and P. Knight, *Destruction and Dynamics for Film and Game Production*, ACM SIGGRAPH Course Notes, 2011.

[2] Z. Bao, J.-M. Hong, J. Teran, and R. Fedkiw, "Fracturing Rigid Materials," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 2, pp. 370-378, Mar. 2007.

[3] E.G. Parker and J.F. O'Brien, "Real-Time Deformation and Fracture in a Game Environment," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, pp. 156-166, 2009.

[4] L. Glondu, M. Marchal, and G. Dumont, "Real-Time Simulation of Brittle Fracture Using Modal Analysis," *IEEE Trans. Visualization and Computer Graphics*, vol. 19, no. 2, pp. 201-209, Feb. 2013.

[5] J.F. O'Brien and J.K. Hodgins, "Graphical Modeling and Animation of Brittle Fracture," *Proc. ACM 26th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp. 137-146, 1999.

[6] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex Rigid Bodies with Stacking," *Proc. ACM SIGGRAPH Conf.*, 2003.

[7] S. Frisken, R. Perry, A. Rockwood, and R. Jones, "Adaptively Sampled Distance Fields: A General Representation of Shapes for Computer Graphics," *Proc. ACM 27th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp. 249-254, 2000.

[8] W.A. McNeely, K.D. Puterbaugh, and J.J. Troy, "Six Degrees-of-Freedom Haptic Rendering Using Voxel Sampling," *Proc. ACM 26th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp. 401-408, 1999.

[9] A. Sud, N.K. Govindaraju, R. Gayle, and D. Manocha, "Interactive 3D Distance Field Computation Using Linear Factorization," *Proc. ACM Symp. Interactive 3D Graphics and Games*, 2006.

[10] S. Fisher and M.C. Lin, "Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields," *Proc. IEEE/RSJ Int'l Conf. Intelligent Robots and Systems*, 2001.

[11] J. Barbič and D.L. James, "Six-DoF Haptic Rendering of Contact between Geometrically Complex Reduced Deformable Models," *IEEE Trans. Haptics*, vol. 1, no. 1, pp. 39-59, Jan.-June 2008.

[12] B. Heidelberger, M. Teschner, R. Keiser, M. Müeller, and M. Gross, "Consistent Penetration Depth Estimation for Deformable Collision Response," *Proc. Conf. Vision, Modeling and Visualization*, 2004.

[13] D. Steinemann, M.A. Otaduy, and M. Gross, "Fast Arbitrary Splitting of Deforming Objects," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, pp. 63-72, 2006.

[14] I.J. Palmer and R.L. Grimsdale, "Collision Detection for Animation Using Sphere-Trees," *Computer Graphics Forum*, vol. 14, no. 2, pp. 105-116, 1994.

[15] R. Weller and G. Zachmann, "Inner Sphere Trees for Proximity and Penetration Queries," *Proc. Conf. Robotics: Science and Systems*, 2009.

[16] P.M. Hubbard, "Approximating Polyhedra with Spheres for Time-Critical Collision Detection," *ACM Trans. Graphics*, vol. 15, no. 3, pp. 179-210, 1996.

[17] C. O'Sullivan and J. Dingliana, "Real-Time Collision Detection and Response Using Sphere-Trees," *Proc. 15th Spring Conf. Computer Graphics*, pp. 83-92, 1999.

[18] J. Klein and G. Zachmann, "ADB-Trees: Controlling the Error of Time-Critical Collision Detection," *Proc. Int'l Vision, Modeling and Visualization*, 2003.

[19] M.A. Otaduy and M.C. Lin, "CLODs: Dual Hierarchies for Multiresolution Collision Detection," *Proc. Eurographics Symp. Geometry Processing*, pp. 94-101, 2003.

[20] S. Kimmerle, M. Nesme, and F. Faure, "Hierarchy Accelerated Stochastic Collision Detection," *Proc. Conf. Vision, Modeling and Visualization*, 2004.

[21] D.M. Kaufman, S. Sueda, and D.K. Pai, "Contact Trees: Adaptive Contact Sampling for Robust Dynamics," *Proc. ACM SIGGRAPH Technical Sketches*, 2007.

[22] T. Larsson and T. Akenine-Möller, "A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection," *Computers and Graphics*, vol. 30, pp. 450-459, 2006.

[23] M.A. Otaduy, O. Chassot, D. Steinemann, and M. Gross, "Balanced Hierarchies for Collision Detection between Fracturing Objects," *Proc. IEEE Virtual Reality Conf.*, 2007.

[24] J.-P. Heo, J.-K. Seong, D. Kim, M.A. Otaduy, J.-M. Hong, M. Tang, and S.-E. Yoon, "FASTCD: Fracturing-Aware Stable Collision Detection," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, 2010.

[25] K. Erleben, "Velocity-Based Shock Propagation for Multibody Dynamics Animation," *ACM Trans. Graphics*, vol. 26, no. 2, 2007.

[26] D.M. Kaufman, S. Sueda, D.L. James, and D.K. Pai, "Staggered Projections for Frictional Contact in Multibody Systems," *Proc. ACM SIGGRAPH Asia*, 2008.

[27] L. Glondu, S.C. Schvartzman, M. Marchal, G. Dumont, and M.A. Otaduy, "Efficient Collision Detection for Brittle Fracture," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, 2012.

[28] F. Liu, T. Harada, Y. Lee, and Y.J. Kim, "Real-Time Collision Culling of a Million Bodies on Graphics Processing Units," *Proc. ACM SIGGRAPH Asia*, 2010.

[29] F. Faure, S. Barbier, J. Allard, and F. Falipou, "Image-Based Collision Detection and Response between Arbitrary Volume Objects," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*, pp. 155-162, 2008.

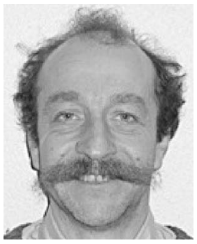
[30] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUS," *Proc. Eurographics*, 2009.

[31] M. Müller, J. Dorsey, L. McMillan, and R. Jagnow, "Real-Time Simulation of Deformation and Fracture of Stiff Materials," *Proc. Eurographics Workshop Animation and Simulation,* 2001.
[32] N. Molino, Z. Bao, and R. Fedkiw, "A Virtual Node Algorithm for Changing Mesh Topology during Simulation," *ACM Trans. Graphics,* vol. 23, no. 3, pp. 385-392, 2004.
[33] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa, "Point Based Animation of Elastic, Plastic and Melting Objects," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation,* pp. 141-151, 2004.

Loeiz Glondu received the PhD degree in computer science from the École Normale Supérieure de Cachan high school in Rennes, France, in 2012. His main research interests include physically based simulation, haptic rendering, and virtual reality.

Sara C. Schvartzman received the BS degree in computer science from Universidad Autónoma de Madrid in 2007, and the master's degree in graphics, games and virtual reality from Universidad Rey Juan Carlos (URJC Madrid), in 2010. She is currently working toward the PhD degree at URJC Madrid, and is a visiting research student in the Salisbury Robotics Lab at Stanford University. Her thesis work has been published at the ACM SIGGRAPH and SCA conferences, and her main research interests include collision detection, physically based animation, and fracture simulation.

Maud Marchal received the MS and PhD degrees in computer science from the University Joseph Fourier in Grenoble, France, in 2003 and 2006, respectively. She is an associate professor in the Computer Science Department at INSA (Engineering School) in Rennes, France. Her main research interests include physically based simulation, haptic rendering, 3D interaction and virtual reality.



Georges Dumont received the PhD degree in computer science from Rennes 1 University in 1990 and the habilitation degree in mechanical science in 2005. He is a professor in mechanical sciences at Brittany Antenna of École Normale Supérieure de Cachan in Rennes. His main research interests include physical simulation, mechanics, biomechanics, haptic rendering, interactive collaboration and virtual reality.

Miguel A. Otaduy received the BS degree in electrical engineering from Mondragón University, in 2000 and the MS and PhD degrees in computer science from the University of North Carolina at Chapel Hill, in 2003 and 2004, respectively. He is an associate professor in the Department of Computer Science's Modeling and Virtual Reality Group at Universidad Rey Juan Carlos (URJC Madrid). From 2005 to 2008, he was a research associate at ETH Zurich, and then he joined URJC Madrid. His main research interests include physically based computer animation and haptic rendering. He has published more than 60 papers in computer graphics and haptics. He is the conference cochair of the 2013 ACM Symposium Interactive 3D Graphics and Games and cochair of the editorial board of the 2013 IEEE World Haptics Conference. He also cochaired the program committee of the ACM SIGGRAPH/Eurographics Symposium Computer Animation in 2010.