Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions

Matthias Müller Nuttapong Chentanez Tae-Yong Kim

NVIDIA

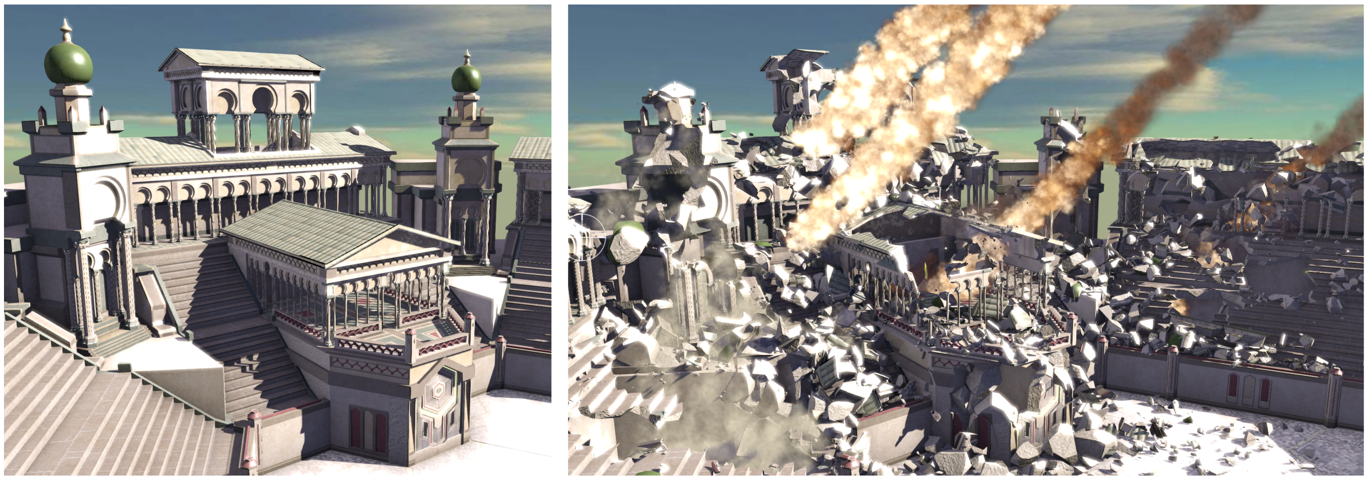

Figure 1: *Destruction of a Roman arena with 1m vertices and 500k faces. The simulation runs at over 30 fps including rigid body simulation, dust simulation and rendering until the end of the sequence where the original mesh is split up into 20k separate pieces.*

Abstract

We propose a new fast, robust and controllable method to simulate the dynamic destruction of large and complex objects in real time. The common method for fracture simulation in computer games is to pre-fracture models and replace objects by their pre-computed parts at run-time. This popular method is computationally cheap but has the disadvantages that the fracture pattern does not align with the impact location and that the number of hierarchical fracture levels is fixed. Our method allows dynamic fracturing of large objects into an unlimited number of pieces fast enough to be used in computer games. We represent visual meshes by volumetric approximate convex decompositions (VACD) and apply user-defined fracture patterns dependent on the impact location. The method supports partial fracturing meaning that fracture patterns can be applied locally at multiple locations of an object. We propose new methods for computing a VACD, for approximate convex hull construction and for detecting islands in the convex decomposition after partial destruction in order to determine support structures.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically Based Modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation and Virtual Reality

Keywords: fracture, real time, Voronoi, destruction

Links: DL PDF

1 Introduction

Destruction effects such as exploding buildings, shattering glass or the destruction of objects in a scene are becoming common in today's computer games. Such effects add significantly to the immersive experience of the player. To achieve real-time performance, game assets are typically pre-fractured during authoring time. During game play, when fracturing occurs, the original models are then simply replaced by their pre-fractured counterparts. This *static* method has been used successfully in many popular games due to its simplicity. However, pre-fracturing often requires careful preparation of the assets to control the amount and location of destruction within each object. In addition, to allow for repeated destruction of object pieces, artists have to provide a hierarchy of fracture levels. The increased authoring time can become a major bottleneck in game production because even without considering destruction the creation and preparation of game assets already constitutes a major part of cost and time in game development. In addition, each time the asset's geometry is changed during game development, the pre-fracturing step has to be repeated.

A major drawback of the static pre-fracturing approach is that there is no way to align fracture patterns with the impact location at run time. When a gamer shoots at a glass window, she expects the spider-web-shaped fracture pattern to be centered around the location where the bullet hit the glass as shown in Figure 5. Anything else clearly destroys the illusion of true fracturing. The only way to get the desired behavior is to generate the fracture pieces *dynamically*. Dynamic fracture has been largely avoided in computer games due to the additional complexity and computational cost. However, besides creating more interesting, non-repeating fracture behavior, dynamic fracture also significantly reduces model preparation time. With the method we propose in this paper, artists only have to prepare a set of generic fracture patterns which can then be assigned to a variety of assets instead of decomposing each model individually.

The idea of using generic fracture patterns and applying them at the impact location at run-time was recently proposed by Su et al. [2009]. In this context, a fracture pattern is a pre-computed decomposition of space, typically a large, pre-fractured block. This pattern is then applied to objects by performing boolean operations. Su et al. use a level set-based approach to apply the fracture pattern to a given object. However, in order to get sharp edges and resolve thin features, high resolution background grids are needed which slow down the simulation and increase the memory footprint, making the method impractical for the destruction of large scale environments in computer games.

We propose a set of new methods that allow dynamic massive destruction of large models in real time for the first time – to the best of our knowledge. Although our method is inspired by Su et al.'s work a variety of new ideas were necessary to develop a framework for dynamic fracturing specifically targeted at computer games.

It is important to distinguish between general real time simulations in which all of the compute power is dedicated to the simulation itself and games in which physical simulations only get a small fraction of computation time. The scene shown in Figure 1 belongs to the first category. This is mostly due to rendering and rigid body simulation. While the cost of these grows linearly with the complexity of the scene, the fracturing cost depends only on the size of the single object to be fractured which makes our method suitable for both, real time simulations and the use in computer games.

The key idea behind our approach is to represent the visual geometry as compounds of convex shapes, where each convex shape contains a unique part of the visual mesh. The decomposition of the model allows us to apply fracture patterns locally at low cost because only a small set of the convex shapes is typically affected by the operation. Su et al. noted that the boolean operations required to apply fracture patterns could be performed on the meshes directly but indicated that such operations would be complicated to implement robustly. Our decomposition allows us to perform the boolean operations directly on the geometry in a fast and robust way, enabling us to fracture highly detailed large scale meshes in real time.

We generate these decompositions using our novel Volumetric Approximate Convex Decomposition (VACD) algorithm and provide methods to fracture them in real time. In contrast to the approximate convex decomposition of *surfaces* often used for collision detection in games [Mamou and Ghorbel 2009], VACD splits the *volume* enclosed by a mesh into non-overlapping, touching convex regions [Lien and Amato 2007], [Tharp et al. 2012]. Our method leverages this property to assign visual geometry uniquely to the convex regions and to perform fast marching on the decomposition to determine the support structure of a partially fractured object. It is important to note that in contrast to pre-fracturing, the pre-computed convex decomposition is independent of the fracture pattern exposed at real-time. Our main contributions include

- A robust method to fracture VACDs with pre-defined fracture patterns which supports applying fracture patterns locally multiple times in different places.
- A method to determine the support structure of a partially fractured model.
- A method for computing a VACD of a visual mesh based on the Voronoi decomposition of space. This method allows user control by manipulating the locations of the Voronoi nodes.
- A fast method for the approximation of a convex hull.
- A generalization of VACDs to support initially overlapping sub-meshes.

2 Related Work

Terzopoulos et al. [1988] and Norton et al. [1991] were among the first to propose fracture models specifically designed for computer graphics. O'Brien et al. [1999] and Smith et al. [2001] focussed specifically on brittle and later on ductile fracture [O'Brien et al. 2002] as well. While Norton et al. used a mass-spring model and Smith et al. point masses connected with distance constraints, the other methods are based on continuum mechanics to compute internal stresses and to determine fracture propagation directions. Müller et al. [2001] and Bao et al. [2007] perform a static analysis only when an object is hit and simulate the objects as rigid bodies between fracture events. Although stress analysis generates more physically correct fracture behavior, we deliberately chose to use a geometric, pattern-based approach due to its controllability and speed. Pauly et al. [2005] devised a method for crack initiation and propagation based on a mesh-less discretization of objects. Zheng et al. [2010] synthesize sounds of dynamic fracture events via a quasi-static stress analysis.

Recently, Parker et al. [2009] proposed a fracture framework targeted at computer games. They use relatively coarse tetrahedral meshes that fracture only along tetrahedral boundaries. To hide the coarse discretization they use so-called splinters associated with each element to re-introduce geometric detail. Glondu et al. [2012] compute damped deformation waves based on modal analysis to estimate crack initiation. They also propose a crack propagation algorithm that is fast enough to be used in real time applications.

In addition to the cost of stress analysis and crack propagation, cutting the visual meshes along with the physical model puts another burden on the destruction system. Efficient and arbitrary cutting of general meshes remains a challenge. A variety of papers have focused on mesh cutting in connection with deformable objects such as thin shells [Kaufmann et al. 2009] [Turkiyyah et al. 2009], tetrahedralized volumetric objects [Sifakis et al. 2007] and meshless models [Steinemann et al. 2006]. Arbitrary cutting of meshes often introduces numerical instabilities due to sliver shapes. One way to alleviate this problem is to use separate meshes for simulation and rendering [Molino et al. 2005] [Sifakis et al. 2007]. We perform simulation and fracturing on proxies, the convex shapes, each of which contains a small part of the visual mesh.

As mentioned before, pre-fracturing is a popular approach used both in the gaming and the movie industry. Various methods have been proposed to split objects in to pieces. They include manual cutting of the geometry by artists, image guidance [Mould 2005], Voronoi fracturing of the surface [Raghavachary 2002] or solid [Baker et al. 2011], tetrahedralization [Parker and O'Brien 2009],

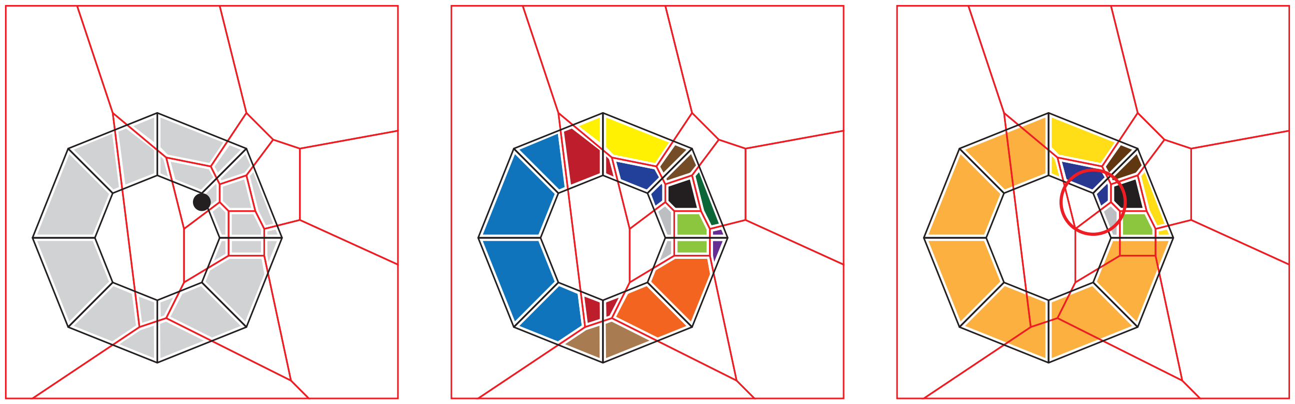

Figure 2: *Overview of the fracture algorithm. Left: The fracture pattern (red) is* ***aligned*** *with the impact location (black dot). Middle: All convex pieces are* ***intersected*** *with all cells. The green convex pieces can be* ***welded*** *to form a single piece because they cover the entire cell. Pieces within one cell become a new* ***compound*** *(coloring).* ***Island detection*** *finds that the dark red compound needs to be split. Right: Partial fracture. Pieces completely outside the impact sphere (orange) are not cut. All orange pieces plus the split pieces outside the impact sphere (yellow) form one new compound.*

BSP trees [Naylor et al. 1990] [Baker et al. 2011] or applying crack patterns created by simulations [Iben and O'Brien 2006].

Exact convex decomposition (ECD) of polyhedra is a well studied problem in computational geometry. It is NP-hard for two dimensional polygons with holes and for polyhedra [Lien and Amato 2006]. However for many computer graphics applications, approximate decompositions are often good enough. For example, for collision handling it is sufficient to decompose the surface of objects into convex regions that are allowed to overlap [Liu et al. 2008]. The most popular approach for *surface* decomposition is probably the hierarchical method of Mamou [2009]. However, for our purpose, we need a *volumetric* decomposition of a mesh into a non-overlapping convex covering of the underlying visual geometry. To our knowledge, the only method proposed to create such a decomposition is the recursive top-down approach by [Lien and Amato 2007]. They create the volumetric decomposition by recursively splitting the given mesh along a plane that minimizes the concavity of both sub-meshes. Their approach is fully automatic, but prone to producing inaccurate solutions. Again, our aim is to give the artist more control over the process. Our method allows the artist to guide the decomposition by manipulating Voronoi nodes in an intuitive way.

The basic building block of the VACD algorithm is the creation of an approximate convex hull (ACH) for a piece of the visual mesh. A simple and robust way to create such an approximate hull is to use k-dops, i.e. to create the hull from a reduced set of planes with predefined normal directions. Kavan [2006] proposed a fast method to tightly fit planes with given normals. Our method also fits a reduced set of planes to create the ACH but determines the normal directions based on the input. This way, the resulting hull fits the input geometry more tightly with a given number of planes.

3 Fracture Simulation

We first describe the simplest case in which we use only one mesh to perform the fracture operations, to handle collisions and for visualization.

3.1 The Basic Method

The meshes we work with have the following properties (Fig. 3):

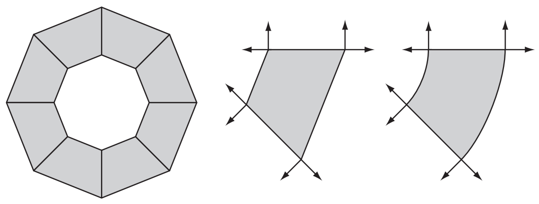

Figure 3: *The basic algorithm in 2d. Compounds are composed of a set of convex pieces that do not overlap. Connected pieces share co-planar faces. For smooth shading each piece stores one normal per vertex and face. Multiple normals per vertex are needed to render sharp edges.*

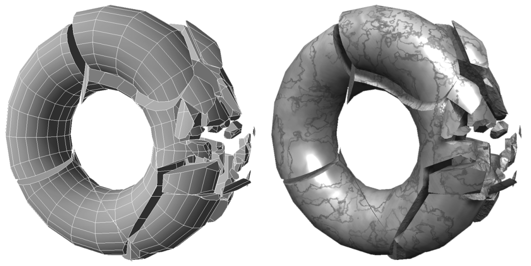

Figure 4: *The basic algorithm in 3d. The decomposition of the initial compound mesh does not become visible when a fracture patterns is applied.*

(a) They are composed of convex pieces.

(b) The pieces do not overlap.

(c) Two pieces are physically connected iff one piece has at least one face that (partially) overlaps a face of the other piece and the two faces lie in the same plane with opposite normals.

Meshes created via volumetric approximate convex decomposition meet these requirements by design. In what follows we use the term *compound* for such meshes. We will illustrate the steps of the fracture algorithm with the compound shown in Figures 3 and 4. It approximates the shape of a torus. To let the outside appear rounded, we assign a normal to each vertex of each face. This normal is only used for visualization.

As mentioned in the introduction, our fracture algorithm is based on the concept of fracture patterns. A *fracture pattern* is a decomposition of all of space into convex pieces. To avoid ambiguity we call the convex pieces of the fracture pattern *cells* and use the term *convex* for the convex components of the compounds. We use decompositions of a large axis-aligned box centered at the origin as fracture patterns. This way we do not have to handle the special cases of open cells. Our method does not depend on how a fracture pattern is generated. The only requirements we impose on its shape are that the impact is located at the origin and that all cells are convex. For the glass window examples shown in Figure 5 we created a spider-web-shaped pattern procedurally while we used Voronoi decomposition with decreasing node density away from the origin for all other scenes.

A key idea of our fracture method is to decompose objects into convex pieces and to use convex fracture patterns cells. This way, clipping a compound mesh against a fracture pattern yields convex pieces that can be re-assembled to form new compounds. In detail, to fracture a compound using a fracture pattern we perform the following steps at the time of impact also illustrated in Figure 2:

1. **Alignment**: First we translate the fracture pattern such that its origin aligns with the impact location. In addition to translation, arbitrary additional transformations can be applied to the pattern at this point as long as the origin is kept fixed and the cells convex.

2. **Intersection**: Next, we compute the intersections of all cells with all convexes. Any acceleration data structure used for convex - convex collision detection can be used to speed up the identification of overlapping cell-convex pairs. To compute the intersection of a single cell with a single convex, we clip the convex against all the planes of the cell one by one and not vice versa. This way, each clip plane creates new vertices on the edges of the partially clipped convex for which the interpolation of vertex attributes such as normals is straight forward. At the end of this step, we have a set of new convexes with new normals for rendering and each convex belongs to exactly one cell.

3. **Welding**: As an effective optimization step, we identify pieces that are completely covered with new convexes and replace these new convexes with one convex that has the shape of the piece. For this test, we compare the pre-computed volume of the cell with the sum of the volumes of all the convexes assigned to it. The pre-computed volume has to be scaled by the determinant of the pattern deformation matrix if the latter is not a rigid transformation.

4. **Compound formation**: Next, all the convexes belonging to one cell are combined to form a new compound. To support fracture patterns with non-convex pieces, a fracture pattern can be equipped with a coloring of its cells. If such a coloring is provided, we group all the new convexes belonging not to a single cell but to all cells of the same color to form one new compound.

5. **Island detection**: One last step is necessary to complete the fracture operation. It is possible that the previous step created compounds of two ore more disconnected islands of convexes. In order to avoid having disconnected fracture pieces moving as a single object, we detect disjoint islands and create individual compounds for them.

The last step is crucial. It is this step that makes sure that objects collapse in the correct way. For large objects it is essential to have a fast method for island detection to avoid frame rate drops. Therefore, the method we propose for this step is one of our core contributions.

To identify separate connected components we perform a flood fill on a newly created compound. The main challenge here is to identify the neighbor structure of its convexes. One method to identify neighbors would be to perform a convex-convex collision detection pass as we do for applying fracture patterns. However, while the application of a fracture pattern is a local operation in case of partial fracture, island detection is a global problem. Therefore, having a more efficient method is crucial. The technique we propose leverages property (c) of compounds. We first create a list with entries for all the faces of all the convexes of the compound. Each entry contains a link to the convex and the absolute value $|d|$ of the plane equation $\mathbf{n} \cdot \mathbf{x} + d = 0$ of the face. After sorting by the value $|d|$, entries with identical values (up to a numerical epsilon) can be identified during a single pass through the list. For each matching pair, we check whether property (c) holds. To compute the amount of overlap $s \in [0,1]$, we project the faces into their common plane and perform a planar convex-convex intersection. In addition to the connectivity test, s can be used for glass rendering. In the scene shown in Figure 5, we only render faces for which $s < 1$ to let the glass appear intact outside the impact radius.

3.2 Partial Fracture

When the basic fracture method described above is applied to a large mesh, an object is fractured along all the cells of the fracture pattern at once, independent of how far the cells are from the impact location as in the center image of Figure 2. One way to keep destruction local is to divide the cells of the fracture pattern into far cells which are outside a give impact sphere and near cells. When all the far cells are given the same color, they stay connected after the fracture operation. However, with this method, convexes are still clipped against the far cells. In order to speed the process up, we do not cut convexes that lie completely outside the impact sphere (orange in Figure 2 right) but integrate them as they are into one compound with the convexes that are outside the impact sphere but within a close cell (yellow in Figure 2 right). This optimization step is particularly important in connection with large meshes and explicit visual geometry as discussed in the next section. Figure 5 shows a typical use case of partial fracture.

3.3 Separate Visual Meshes

The basic method can already handle a variety of interesting scenarios in games such as the destruction of glass windows or objects with simple geometry such as cylindrical columns. For objects with more complex shapes, working with their meshes directly is not practical. For this case we use two separate meshes, a compound with the properties stated in Section 4.3 and a visual mesh for rendering. In Section 4 we will describe ways to create a compound for a given visual mesh. We require each convex to contain at most one manifold, watertight connected visual mesh that lies completely inside the convex. We call the convexes without visual mesh *internal convexes* and all others *surface convexes*. Internal convexes are handled as described in Section 4.3. For surface convexes we perform the additional sub-steps when clipping against a cell of the fracture pattern as shown in Figure 8 (c) and (d):

(a) **Mesh clipping**: If the new convex lies completely inside the visual mesh, it becomes an internal convex. Otherwise a new visual mesh is created by clipping the visual mesh of the par ent convex against the new convex.

(b) **Island detection**: Mesh clipping can generate more than one

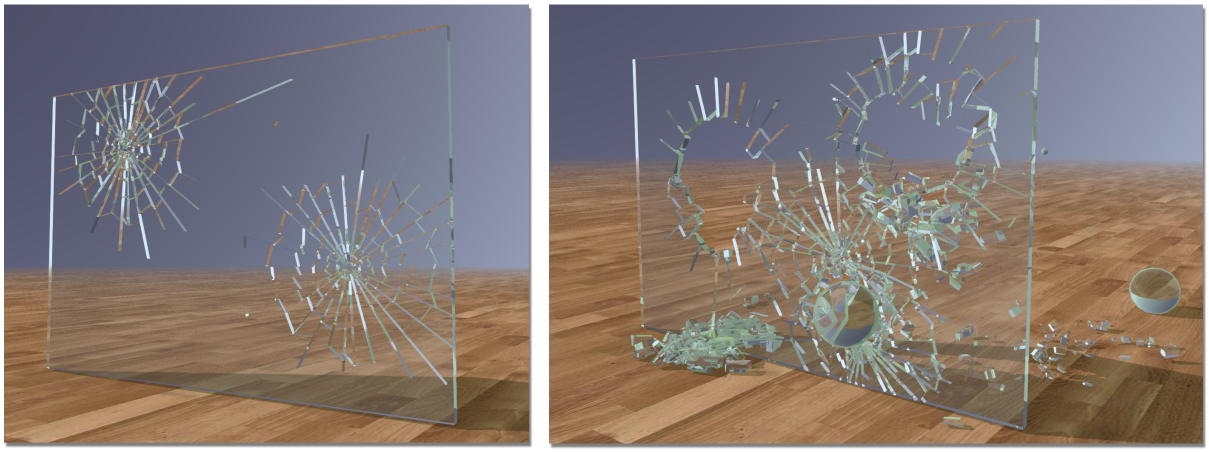

Figure 5: *A glass window is fractured locally multiple times. Our method makes sure the region outside the impact radii stays connected and the faces of the convexes in that region hidden.*

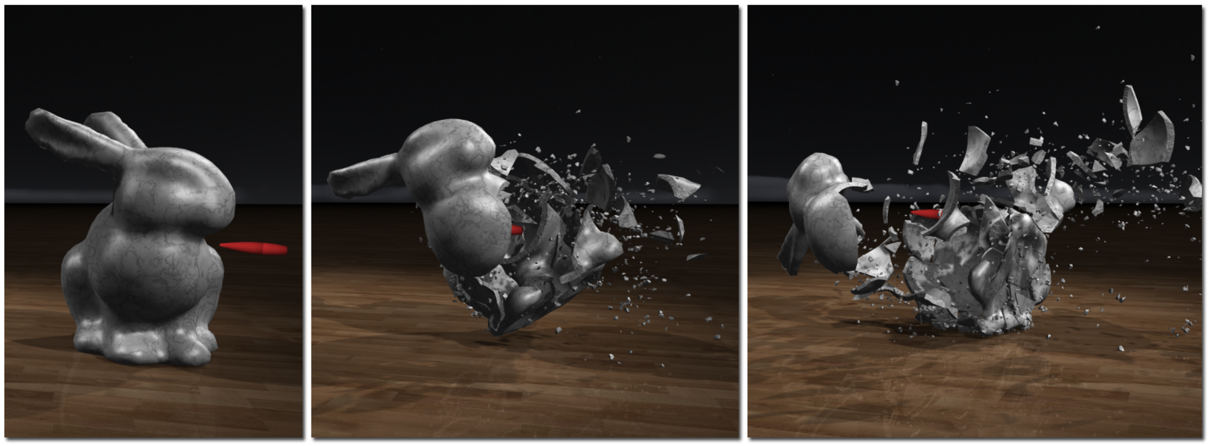

Figure 6: *Impact-based fracture triggering. Approximately 600 compounds and 800 convexes are created during the simulation.*

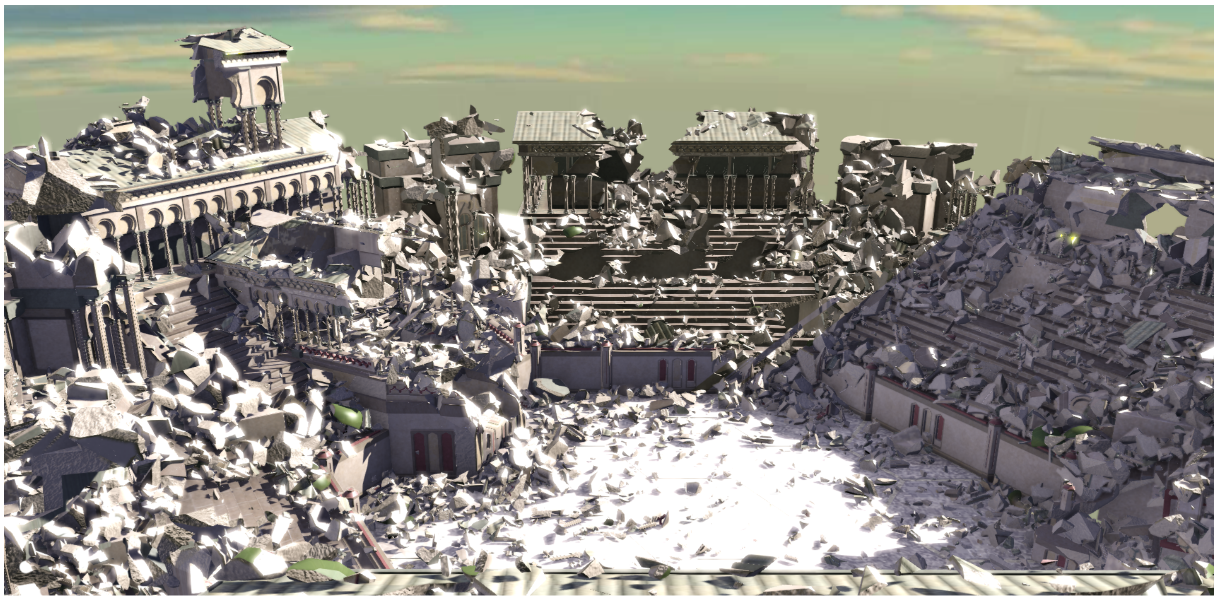

Figure 7: *The arena after serious destruction resulting in 20k compounds and 32k convexes. Throughout the simulation, the fracture time stays below 50ms.*

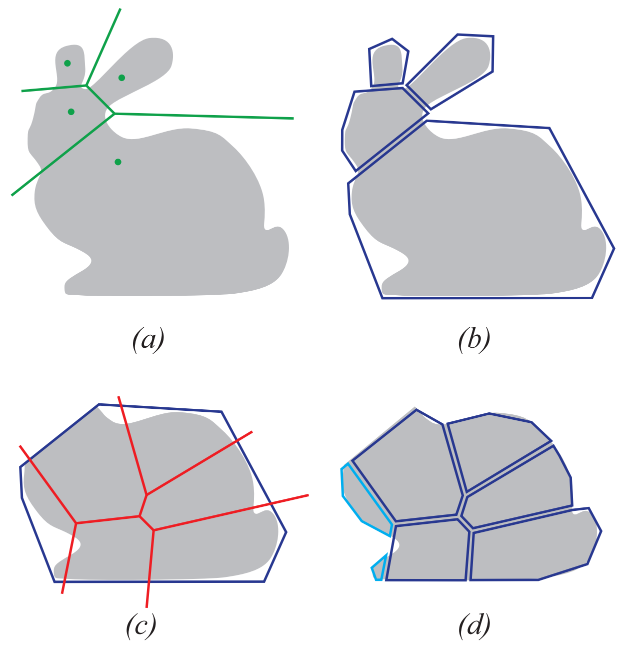

Figure 8: *Separate fracture and rendering meshes. (a): A compound mesh is created from a visual mesh (gray) via VACD. Interactive placement of Voronoi nodes provides intuitive control. (b): The mesh is split along the Voronoi cell boundaries and an approximate convex hull created for each sub-mesh. (c): At runtime a convex is fractured using a fracture pattern (red). (d): Each new convex is fit to the visual sub-mesh it owns. In the left-most cell two islands are detected and two new convexes created (light blue).*

connected visual mesh within a new convex. Therefore we perform island detection on the visual mesh and create separate new convexes if necessary. Here island detection is simple and fast because the visual mesh provides connectivity information.

(c) **Re-fit**: In general, after the previous steps, a new surface convex does not tightly enclose its visual mesh. Therefore, we re-compute the shape of the convex based on its visual mesh using the algorithm described in Section 4.2

Note that step (b) can potentially create overlapping convex pieces. This can happen because we allow the visual mesh to be concave within a convex. Overlapping pieces are only problematic if they end up in different compounds. In this case, the rigid body engine generates small separation impulse. We have not seen any disturbing artifacts in our test scenes due to this rare event.

Mesh clipping is the most complex step in the fracture algorithm and writing numerically stable code is far from trivial. The key decision that made our code robust was to work with general potentially non-convex polygonal faces instead of restricting the implementation to triangle meshes. Permanent triangulation of polygonal intersections of the visual mesh with cell boundaries creates a rapidly increasing number of ill-shaped triangles as shown in Figure 9. Of course, the polygonal approach can still fail due to numerical errors. In this case which we found to happen quite rarely, we simply remove the corresponding convex from the simulation. It should be noted that mesh clipping not just clips the given mesh against a cell but also creates geometry on the cell faces such that the mesh becomes watertight.

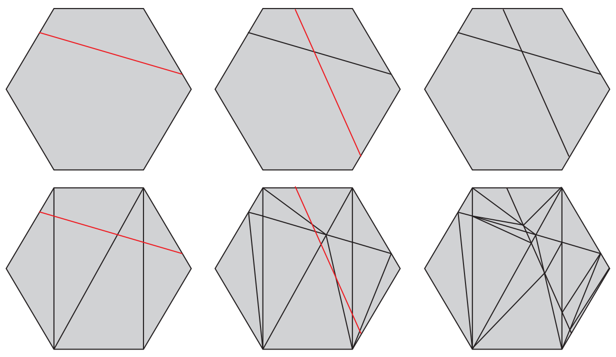

Figure 9: *The importance of working with polygons instead of triangles. The figure shows a hexagon split twice using general polygonal faces (top) and triangular faces only (bottom). In the latter case, many ill shaped triangles are created even after only two cuts.*

4 Mesh Preparation

As described in the previous sections, our fracture algorithm works on compound meshes, not on visual meshes directly. Therefore we have to create the compound mesh for a given visual mesh as a preprocessing step. We would like to emphasize again that this mesh preparation step is different from pre-fracturing models because it is independent of the fracture pattern used at runtime.

4.1 Voronoi-Based VACD

As input we expect a manifold and watertight polygonal mesh. Our VACD method is based on the Voronoi decomposition of space. It comprises the following steps (see Figures 8 (a) and (b)).

1. **Node placement** First, the user places nodes on the mesh.
2. **Voronoi decomposition** Given the nodes we compute the Voronoi decomposition of the bounding box of the mesh resulting in a set of convex shapes.
3. **Mesh clipping** Next we clip the mesh against the Voronoi cells as described in Section 3.3.
4. **Island detection** As mentioned in Section 3.3 each cell can potentially contain multiple mutually disconnected meshes. If this is the case, one convex is created for each sub-mesh.
5. **Convex fitting** As a last step, the convexes are reshaped to fit the enclosed visual mesh tightly using the method we will describe in Section 4.2.

Figure 10 shows the application of this procedure to a complex dragon model. A nice property of the method is that it works for meshes of any topology whereas traditional VACD methods need to treat meshes of genus one and higher in a special way. At the end we have a compound mesh and manifold watertight meshes per convex as required by the runtime fracture algorithm. An advantage of pre-clipping the original mesh is that for very large models, only local operations have to be performed at run time when a model is fractured locally. Another advantage is improved stability. Our VACD implementation uses double precision computation to avoid numerical problems with large meshes. However, at runtime we are restricted to use single precision. Since each convex contains a relatively small part of the visual mesh which we center at the origin, single precision is sufficient to handle most configurations stably.

The fact that our method only takes a fraction of a second to decompose a mesh of 10-100k triangles allows the user to interactively

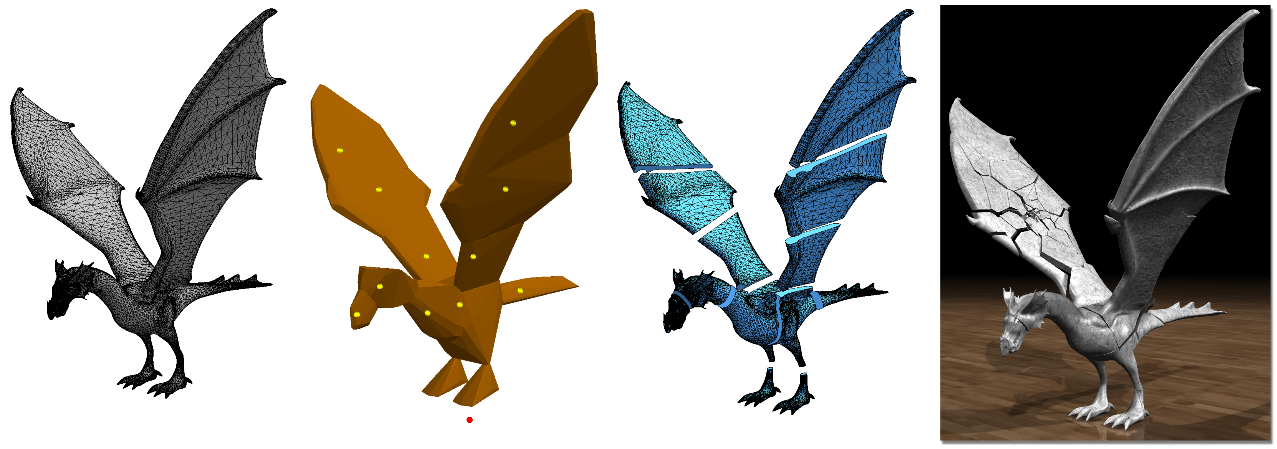

Figure 10: *Volumetric Approximate Convex Decomposition of a dragon model with 40k vertices and 80k triangles. Placing twelve nodes intuitively in a matter of seconds results in an tight fitting decomposition that makes fracturing this high resolution mesh possible in real time. From left to right: Original model, compound mesh, one split mesh per convex (shrunk for visualization) and destruction of the model at runtime.*

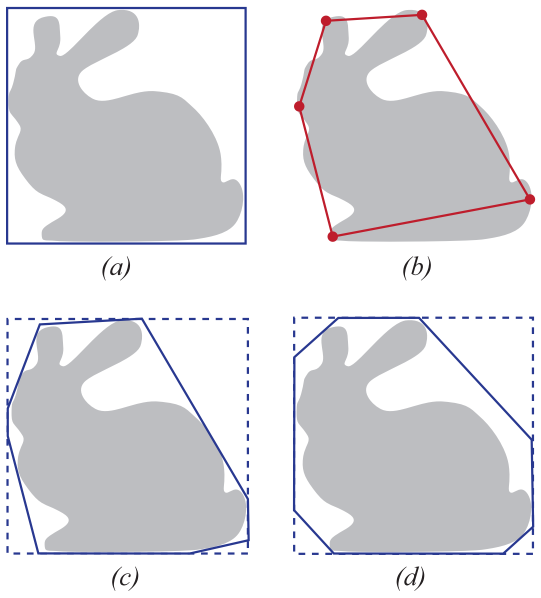

Figure 11: *Approximate convex hull algorithm. (a): initial convex. (b): incremental volume maximizing convex hull. (c): clipping of the initial convex using the plane directions. (d): clipping with fixed plane directions.*

manipulate the nodes while getting immediate visual feedback. We found that manual node placement and manipulation is fast even for complex models such as the dragon. However, if required, the algorithm above can be turned into a fully automatic decomposition method. What is needed is a way to procedurally place the nodes which is one direction of future work.

4.2 Approximate Convex Hull

At runtime as well as during off-line mesh decomposition, convexes have to be re-fitted to enclose a visual mesh tightly. The tightest fit would be to use the convex hull of the mesh. However, typically the convex hull of a detailed visual mesh contains a large number of faces and vertices slowing down both collision handling and further fracture operations. What we need is an approximate convex hull. The hull needs to satisfy the following constraints:

- It needs to enclose all the vertices of the visual mesh
- It must not overlap any other convexes of the compound
- Connected convexes must share co-planar faces.

In both the off-line and runtime case we already have a convex shape that meets all these constraints before re-fitting. A natural way to make the given convex tighter and not violate any of the above constraints is to choose a direction, move a plane perpendicular to this direction from the outside towards the convex until it hits a visual vertex and then clip the convex against this plane. At runtime where re-fitting needs to be fast we let the user choose between AABB, 14-dop and 26-dop fitting. For AABB fitting we use the following directions and their negatives

$$\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}. \tag{1}$$

In case of 14-dop fitting we additionally uses the following directions and their negatives:

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}-1\\1\\1\end{bmatrix}\begin{bmatrix}1\\-1\\1\end{bmatrix}\begin{bmatrix}1\\1\\-1\end{bmatrix}. \tag{2}$$

For 26-dop fitting the following directions and their negatives are used as well:

$$\begin{bmatrix}0\\1\\1\end{bmatrix}\begin{bmatrix}0\\-1\\1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix}\begin{bmatrix}1\\0\\-1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}-1\\1\\0\end{bmatrix}. \tag{3}$$

At run time, re-fitting is only used locally where a model is fractured. In contrast, re-fitting during mesh preparation yields the initial approximation of the entire visual mesh which is used for both fracturing and collision handling. Therefore, tighter fits are more essential.

For this purpose we propose a new method to choose the fit directions (see Figure 11). The basic idea is to apply the incremental convex hull algorithm [Kallay 1984] to the visual mesh and stop after a given number of included vertices. We then use the normals of the faces of this intermediate hull as fitting directions. What makes our algorithm effective is the order in which we process visual vertices. We do this in a greedy manner to optimize the volume of the intermediate convex. Algorithm 1 describes the procedure in more detail, where $v_1, .., v_n$ are the vertices of the visual mesh and $f_1, .., f_m$ the faces of the current convex hull. The function $vol(v_i, f_j)$ returns the volume of the tetrahedron formed by vertex v_i and face f_j. A face f_j is visible from vertex v_i if $vol(v_i, f_j) \geq 0$.

```
begin
    $C \leftarrow \{\arg\max_{v_i} v_i.x\}$;
    $C \leftarrow C \cup \{\arg\max_{v_i} \text{length}(C \cup v_i)\}$;
    $C \leftarrow C \cup \{\arg\max_{v_i} \text{area}(C \cup v_i)\}$;
    $C \leftarrow C \cup \{\arg\max_{v_i} \text{volume}(C \cup v_i)\}$;
    createConvex($C$);
    foreach $v_i \notin C$ do
        $V_i = \sum_j \max(vol(v_i, f_j), 0)$
    end
    while $|C| < N$ do
        $k \leftarrow \arg\max_i V_i$;
        $C \leftarrow C \cup \{v_k\}$;
        updateConvex($v_k$);
        foreach $v_i \notin C$ do
            $V_i \leftarrow V_i - \sum_{f_j \text{removed}} \max(vol(v_i, f_j), 0)$
            $V_i \leftarrow V_i + \sum_{f_j \text{added}} \max(vol(v_i, f_j), 0)$
        end
    end
end
```

Algorithm 1: Volume Maximizing Approximate Convex Hull

The procedures *createConvex()* and *updateConvex()* are the ones used in the traditional incremental convex hull algorithm. The key idea is to have each vertex v_i store the volume V_i it would add to the current convex if it was included. As mentioned above, the V_i can also be used for visibility tests needed in the base algorithm.

The algorithm can also be used outside of our context to compute a tight approximate convex hull that encloses all the vertices of a given mesh. One would simply start with the bounding box of the mesh. A nice property of the algorithm is that it converges to the true convex hull as the number of included vertices approaches the number of mesh vertices while the volume of the approximation converges towards the volume of the convex hull from above. Solutions using a fixed set of directions such as [Kavan et al. 2006] do not have this property.

4.3 Initial Convex Overlaps

Large objects such as the arena shown in Figure 1 and schematically in Figure 12 are typically not modeled as one connected water tight mesh. An artist would rather model each column as well as the roof as separate meshes. To simplify the pre-processing step we therefore relax the requirements for the initial compound mesh. Each convex still has to contain at most one connected manifold and watertight visual mesh but convexes and their visual meshes are allowed to overlap.

With this relaxation, the VACD algorithm iterating through all sub-meshes immediately finds the decomposition shown in Figure 12(a) even without any nodes and a simple AABB fit whereas finding an

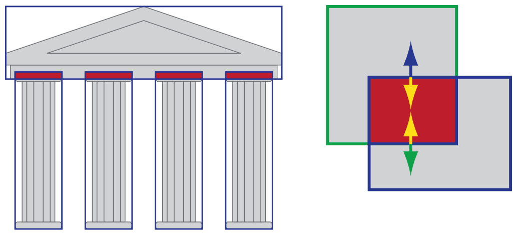

Figure 12: *The concept of ghost convexes. (a): We allow convex overlaps in the initial VACD so artists do not have to weld sub-meshes of large connected models into one manifold watertight mesh. (b): Initially, ghost convexes with negative volume and inward pointing normals (red) are created for each intersection. These are treated as regular convexes in the fracture process and ensure both correct volume computations and connectivity via coplanar faces.*

appropriate VACD for the combined mesh would require the placement of at least two nodes (to split the roof from the columns, the columns would still be split into separate convexes by island detection).

However, with overlapping convexes, two problems are introduced.

- In the welding step of the fracture algorithm we sum up the volumes of all convexes belonging to one cell and compare the sum with the volume of the cell. This test yields incorrect results near overlaps.
- It is important to require that two overlapping convexes are connected structurally otherwise they fly apart violently as soon as the simulation starts due to their mutual penetration. However, property (c) of compounds does not guarantee this. For instance, the two overlapping convexes shown in Figure 12 (b) do not have a pair of co-planar faces and would therefore end up in two different pieces.

We solve both problems at once by introducing *ghost convexes*. For each pair of overlapping convexes we create one ghost convex with the shape of their intersection (which is guaranteed to be convex). In contrast to regular convexes, ghost convexes are never rendered, have negative volumes and inward pointing face normals. Figure 12(b) shows how a ghost convex (red) solves the volume and connectivity problem. First, the sum of the volumes of the three convexes equals the volume of the compound mesh. Second, the green convex and the ghost convex have co-planar faces with opposite normals. The same is true for the blue convex and the ghost convex so the ghost convex connects the two original convexes.

Note that this is not true for a convex that lies completely inside another one. This is not an issue because such a configuration is not meaningful in our context. Multiple overlaps could be handled by testing for overlapping ghost convexes and introducing another level of them with positive volume. We have not investigated this generalization further because simple overlap handling was sufficient in all the models we used.

5 Results

To measure the effectiveness of Algorithm 1 to compute approximate convex hulls we tested it on the bunny mesh shown in Figure 6. The mesh contains 6k vertices and 11k faces with 645 vertices on the convex hull. Starting with the AABB of the mesh and including 4 and 6 vertices already produces a hull that has approximately the

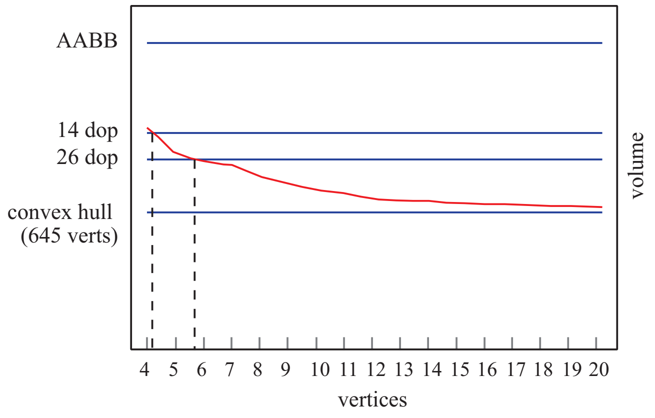

Figure 13: *Volume of the approximate convex hull of a bunny mesh with 6k vertices and 11k faces computed with Algorithm 1 (red). The approximate hull using 4 and 6 vertices fits the mesh as tightly as a 14-dop and the 26-dop hull, respectively. Including 20 vertices yields a volume only 5 percent larger than the true convex hull containing 645 vertices of the original mesh.*

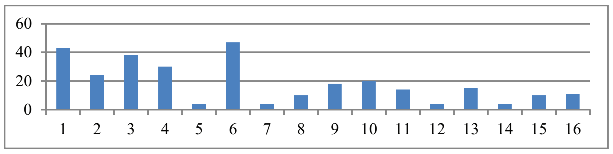

Figure 14: *Fracture times (ms) per fracture event (not frame) at the beginning of the arena destruction sequence.*

same volume as a 14-dop and a 26-dop fit, respectively as Figure 13 shows. With 20 vertices included, the resulting hull's volume is only five percent larger than the volume of the true convex hull.

We used an Intel Core i7 CPU at 3.07 GHz and an NVIDIA GeForce GTX 680 for all our simulations shown in the paper. As mentioned earlier, the frame rates in our examples are determined by rendering, rigid body simulation and dust simulation, typically using about 15, 45 and 40 percent of the computation time, respectively. The table in Figure 14 lists the fracture times at the beginning of the arena destruction scene shown in Figures 1 and 7 and the accompanying video. The time to fracture small to average sized objects is typically negligible, i.e. below 10 ms. Fracturing is most expensive when large buildings like the center stand with the temple are hit for the first time but still below 50ms throughout the simulation. A way to reduce performance hits further would be to distribute the fracture calculations over multiple time steps which is part of future work.

For our simulations we used the rigid body simulator of [Tonge et al. 2012] which runs completely in parallel on the GPU. For increased realism, we added a dust and debris simulation. After a fracture event, we identify the faces that became exposed. We then generate debris and dust particles uniformly on those faces with a user specified sampling density and ratio. To stabilize the solver, we do not create true rigid bodies below a certain size. The sizes of the debris particles cover the range below this threshold. Each debris particles is rendered by instancing one of a set of pre-defined visual meshes and simulated as a spheres with high angular damping to prevent unrealistic rolling on the ground.

For the simulation of dust we adapted the method of [Pfaff et al. 2010] adding additional orientation information to each particle. The orientation is used to add small scale turbulence effects to the coarse, grid based fluid simulation. Dust particles are rendered as rotating point sprites showing the small scale swirl of the flow.

6 Conclusion and Future Work

We have presented a novel method for the dynamic destruction of complex objects that is fast enough to be used in computer games. The basic idea is to represent visual meshes with a compound mesh of convex primitives which we compute using our new volumetric approximate convex decomposition technique. Applying a fracture pattern composed of convex cells to a compound mesh yields convex pieces that can be reassembled to form new compounds. A basic operation for mesh pre-processing and re-fitting convexes at run time is the computation of an approximate convex hull for which we have devised an effective technique that can be used in other contexts as well. We have demonstrated the efficiency of the fracture method in a variety of complex scenarios.

Using a purely geometric method, our results are typically not as physically accurate as those of stress-based methods. This, however, is not always the case. For common scenarios like the shattering of a window as shown in Figure 5, tens of thousands of finite elements would be needed to get the clean, sharp edged spider web, more than can be handled in real time. We chose the geometric, pattern-based approach because it is fast, still provides enough flexibility and because game developers like to have as much control as possible. Some artists are even reluctant to go from static to dynamic fracture due to the loss of precise control.

We have implemented collision based fracture triggering as shown in Figure 6 but so far, structures do not fracture under their own weight. Also, in nature, objects do not necessarily fracture at the impact location only. Both problems can be solved by performing a simplified stress analysis on the connectivity graph of a compound – a problem we will study as future work. In our examples, we have applied the same fracture pattern independent of the strength of the impact. This is not a limitation of our method in principle. One could pre-compute a variety of patterns and select them as well as the impact radius for partial fracture dependent on the impact force. Another line of future work is the extension of our method to handle ductile fracture. One idea to do this would be to re-assemble the convex pieces into new compounds after fracture only after a few steps of simulating them as separate bodies giving them time to change their relative poses. Turning our VACD algorithm into an automatic method is another interesting problem although we found that doing it manually is efficient and gives the user a high level of control to create a decomposition that optimally fits the scenario in a game.

Acknowledgements

We thank NVIDIA for supporting our research, the PhysX team for the valuable feedback, the anonymous reviewers for their comments and suggestions and Dane Johnston, Kevin Newkirk and Aron Zoellner for providing the arena model.

References

BAKER, M., CARLSON, M., COUMANS, E., CRISWELL, B., HARADA, T., KNIGHT, P., AND ZAFAR, N. B. 2011. Destruction and dynamic artist tools for film and game production. In *ACM SIGGRAPH 2011 course notes*.

BAO, Z., HONG, J.-M., TERAN, J., AND FEDKIW, R. 2007. Fracturing rigid materials. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (Mar.), 370–378.

GLONDU, L., MARCHAL, M., AND DUMONT, G. 2012. Real-time simulation of brittle fracture using modal analysis. *IEEE Trans. on Visualization and Computer Graphics*.

IBEN, H. N., AND O'BRIEN, J. F. 2006. Generating surface crack patterns. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '06, 177–185.

KALLAY, M. 1984. The complexity of incremental convex hull algorithms in rd. *Information Processing Letters 19*, 4, 197–.

KAUFMANN, P., MARTIN, S., BOTSCH, M., GRINSPUN, E., AND GROSS, M. 2009. Enrichment textures for detailed cutting of shells. In *ACM SIGGRAPH 2009 papers*, ACM, New York, NY, USA, SIGGRAPH '09, 50:1–50:10.

KAVAN, L., KOLINGEROVA, I., AND ZARA, J. 2006. Fast approximation of convex hull. In *Proceedings of the 2nd IASTED international conference on Advances in computer science and technology*, ACTA Press, Anaheim, CA, USA, ACST'06, 101–104.

LIEN, J.-M., AND AMATO, N. M. 2006. Approximate convex decomposition of polygons. *Comput. Geom. Theory Appl. 35*, 1 (Aug.), 100–123.

LIEN, J.-M., AND AMATO, N. M. 2007. Approximate convex decomposition of polyhedra. In *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, ACM, New York, NY, USA, SPM '07, 121–131.

LIU, R., ZHANG, H., AND BUSBY, J. 2008. Convex hull covering of polygonal scenes for accurate collision detection in games. In *Proceedings of graphics interface 2008*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, GI '08, 203–210.

MAMOU, K., AND GHORBEL, F. 2009. A simple and efficient approach for 3d mesh approximate convex decomposition. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, 3501 –3504.

MOLINO, N., BAO, Z., AND FEDKIW, R. 2005. A virtual node algorithm for changing mesh topology during simulation. In *ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, SIGGRAPH '05.

MOULD, D. 2005. Image-guided fracture. In *Proceedings of Graphics Interface 2005*, Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, GI '05, 219–226.

MÜLLER, M., DORSEY, J., AND MCMILLAN, L. 2001. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of (Eurographics CAS), Computer Animation and Simulation 2001*, Springer-Verlag Wien, 113–124.

NAYLOR, B., AMANATIDES, J., AND THIBAULT, W. 1990. Merging bsp trees yields polyhedral set operations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '90, 115–124.

NORTON, A., TURK, G., BACON, B., GERTH, J., AND SWEENEY, P. 1991. Animation of fracture by physical modeling. *Vis. Comput. 7*, 4 (July), 210–219.

O'BRIEN, J. F., AND HODGINS, J. K. 1999. Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '99, 137–146.

O'BRIEN, J. F., BARGTEIL, A. W., AND HODGINS, J. K. 2002. Graphical modeling and animation of ductile fracture. *ACM Trans. Graph. 21*, 3 (July), 291–294.

PARKER, E. G., AND O'BRIEN, J. F. 2009. Real-time deformation and fracture in a game environment. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, SCA '09, 165–175.

PAULY, M., KEISER, R., ADAMS, B., DUTRÉ, P., GROSS, M., AND GUIBAS, L. J. 2005. Meshless animation of fracturing solids. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 957–964.

PFAFF, T., THUEREY, N., COHEN, J., TARIQ, S., AND GROSS, M. 2010. Scalable fluid simulation using anisotropic turbulence particles. In *ACM SIGGRAPH Asia 2010 papers*, ACM, New York, NY, USA, SIGGRAPH ASIA '10, 174:1–174:8.

RAGHAVACHARY, S. 2002. Fracture generation on polygonal meshes using voronoi polygons. In *ACM SIGGRAPH 2002 conference abstracts and applications*, ACM, New York, NY, USA, SIGGRAPH '02, 187–187.

SIFAKIS, E., DER, K. G., AND FEDKIW, R. 2007. Arbitrary cutting of deformable tetrahedralized objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '07, 73–80.

SMITH, J., WITKIN, A., AND BARAFF, D. 2001. Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Forum 20*, 2, 81–91.

STEINEMANN, D., OTADUY, M. A., AND GROSS, M. 2006. Fast arbitrary splitting of deforming objects. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '06, 63–72.

SU, J., SCHROEDER, C., AND FEDKIW, R. 2009. Energy stability and fracture for frame rate rigid body simulations. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, SCA '09, 155–164.

TERZOPOULOS, D., AND FLEISCHER, K. 1988. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. *SIGGRAPH Comput. Graph. 22*, 4 (June), 269–278.

THARP, A., GHOSH, M., AND AMATO, N. M. 2012. Taming large 3d models : Approximate convex decomposition. In *Proceedings of summer undergraduate research 2012, Texas University*.

TONGE, R., BENEVOLENSKI, F., AND VOROSHILOV, A. 2012. Mass splitting for jitter-free parallel rigid body simulation. *ACM Trans. Graph. 31*, 4 (July), 105:1–105:8.

TURKIYYAH, G., KARAM, W. B., AJAMI, Z., AND NASRI, A. 2009. Mesh cutting during real-time physical simulation. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, ACM, New York, NY, USA, SPM '09, 159–168.

ZHENG, C., AND JAMES, D. L. 2010. Rigid-body fracture sound with precomputed soundbanks. In *ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, SIGGRAPH '10, 69:1–69:13.