Physics-Aware Voronoi Fracture with Example-Based Acceleration

Sara C. Schvartzman
Stanford University

Miguel A. Otaduy
URJC, Madrid

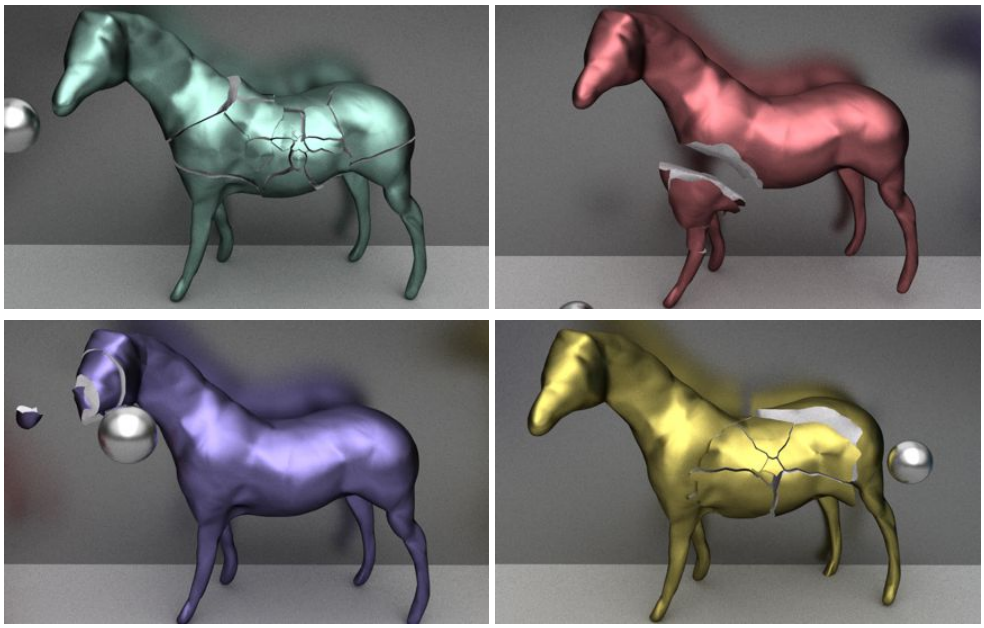

Figure 1. Horses fractured by metal balls. These fractures emphasize the richness and versatility provided by our method: object concavities are correctly resolved, cracks may be curved, and fracture patterns adapt to the impact and object properties.

Abstract

This paper provides implementation details of the algorithm proposed in Schvartzman and Otaduy [2014] to simulate brittle fracture. We cast brittle fracture as the computation of a high-dimensional *centroidal Voronoi diagram* (CVD), where the distribution of fracture fragments is guided by the deformation field of the fractured object. We accelerate the fracture animation process with example-based learning of the fracture degree and a highly parallel tessellation algorithm.

 ISSN 2331-7418

1. Introduction

Fracture animation creates spectacular effects in motion pictures, and it is lately making its way into video games and virtual reality applications. Physically-based simulation of fracture involves solving challenging mechanical problems, such as the computation of deformations, crack generation, and crack propagation. These operations require computationally demanding processes such as remeshing and fine time-stepping. In this work, we seek a faster, but plausible, solution for brittle, stiff objects.

Voronoi fracture methods offer a fast alternative to physically based simulation methods. They compute the locations of fragment centers, and then the fracture fragments are defined as the Voronoi cells of those centers. In contrast to physically based methods that compute fracture through local crack propagation, Voronoi methods compute fracture through the global distribution of fragments, thus giving the artist control over the global appearance of the fracture.

Although Voronoi methods are fast, they pose other challenges: mainly the versatile distribution of fragments according to arbitrary external forces, and the correct handling of object and crack concavities.

While most previous methods are oblivious of the forces acting on the fractured object, we introduce a new Voronoi fracture method that distributes fracture fragments based on the deformation field of the fractured object. Furthermore, our algorithm properly handles object and crack concavities, allows for intuitive artist control, and is guided by examples to accelerate computations.

Our method [Schvartzman and Otaduy 2014] produces fast animations where objects may be fractured in arbitrary non-scripted ways, showing rich and diverse fracture patterns even at close views, such as those in Figure 1.

The first step in the animation of fracture is the computation of elastic deformations. We adopt a common approach for stiff objects, simulating them as rigid bodies until an impact is detected, and then computing a quasi-static solution to the elastic deformation [Müller et al. 2001]. Using the deformation field, we then learn the fracture degree (number of fragments) from a set of precomputed fracture examples and construct a high-dimensional centroidal Voronoi diagram (CVD). Finally, we tessellate the fragments of the CVD and introduce the new fragments into the rigid body engine.

2. Deformation-Based Voronoi Fracture

In this section, we present a formulation of fracture as a CVD. Our approach is aware of the collision scenarios suffered by the fractured object and determines the size and distribution of fragments based on the distribution of deformation energy, as shown in Figure 2. In addition, we adopt an interior distance metric to handle robustly ob-

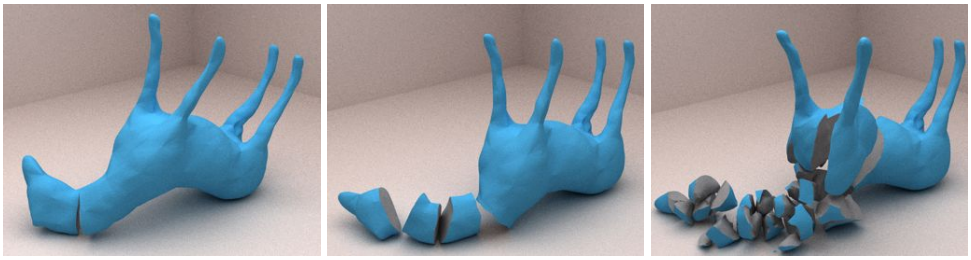

Figure 2. Three horses falling on their heads with different fracture energy thresholds, γ (highest γ on the left to lowest on the right).

ject and crack concavities. As shown in Schvartzman and Otaduy [2014], a CVD under such interior distance metric can be computed efficiently using the well-known Lloyd's method by lifting the object representation to higher dimensions. We conclude the section describing ways to introduce intuitive artist control into our fracture method.

2.1. Fracture as a Centroidal Voronoi Diagram

In brittle materials fracture propagates quickly, and the resulting fragments release their deformation energy and recover their initial shape. Based on this observation, we propose the following fracture criterion: an object will fracture if its deformation energy is larger than a certain threshold, γ.

We define the *distance-weighted deformation energy* of an object with volume Ω_i, center p_i, strain energy density $W(x)$, and points $x \in \Omega_i$, as

$$E_{D,i} = \int_{\Omega_i} \text{dist}(x, p_i)^2 \, W(x) \, dx.$$

In essence, $E_{D,i}$ accumulates strain energy, but penalizes the distance to the center of the object based on some suitable distance metric, $\text{dist}(x, p_i)$. In this way, an object with a large deformation concentrated in one place is more prone to fracture than an object with a moderate homogeneous deformation.

If an object is fractured, we place the centers of the new fragments such that the post-fracture deformation energy is minimized, i.e., the energy consumed by fracture is maximized. This procedure can be applied recursively until the maximum sustainable deformation energy γ is exceeded. With $P^* = \{p_i^*\}$, the locations of fragment centers and $N = |P^*|$, the fracture degree, we formulate fracture as the computation of the minimum number of fragments such that the distance-weighted deformation

energy does not exceed the fracture threshold:

$$N = \min |P^*| \quad \text{such that} \quad E_D(P^*) < \gamma. \tag{1}$$

$$P^* = \arg\min_P E_D(P), \tag{2}$$

$$\text{with } E_D = \sum_i E_{D,i}$$

It turns out that the solution to the optimization problem in Equation (2) is well known, and it corresponds to the CVD of object Ω. Then, the solution to the fracture problem in Equations (1)–(2) is given by the CVD with the fewest number of sites that satisfy $E_D < \gamma$.

2.2. CVD with Interior Distance Metric

For non-convex objects, the computation of the CVD using the Euclidean distance metric may produce small fragments topologically far away from the impact location. Instead, we correctly handle object concavities computing the CVD using an interior distance metric. We adopt the interior distance definition by Rustamov et al. [2009], which is based on a high-dimensional embedding that approximately preserves surface distances. Specifically, given two surface vertices v_i and v_j with interior distance $\text{dist}(v_i, v_j) = d_{ij}$, the Euclidean distance between their corresponding high-dimensional coordinates $\bar{v}_i$ and $\bar{v}_j$ is also $\|\bar{v}_i - \bar{v}_j\| = d_{ij}$. The high-dimensional coordinates of surface vertices are computed using a diffusion map, and the high-dimensional coordinates of interior points are computed through barycentric interpolation using mean-value coordinates. Given a surface mesh, Rustamov et al. [2009] provide Matlab code to compute the high-dimensional coordinates of a set of points located on or inside the mesh.

Once the interior distance metric is defined, we present an efficient algorithm to compute the CVD using such metric. Lloyd's method is a popular approach to compute the energy-weighted CVD in Equation (2) for the Euclidean distance metric. It iterates two steps until convergence: (i) computation of the Voronoi diagram for a given set of sites, and (ii) moving the sites to the centroids of their Voronoi cells.

The use of an interior distance metric makes the CVD problem highly nonlinear. However, Schvartzman and Otaduy [2014] demonstrate that for the interior distance of Rustamov et al. [2009], this nonlinear optimization problem admits a practical and efficient solution through the computation of an Euclidean CVD in the high-dimensional embedding space.

2.3. Preprocessing and Runtime Algorithms

Our full fracture algorithm proceeds as follows (See Algorithm 1). During runtime, given external forces on an object Ω, we compute the strain energy density $W(x)$ using a quasi-static finite-element formulation (line 9). Then, the strain energy field

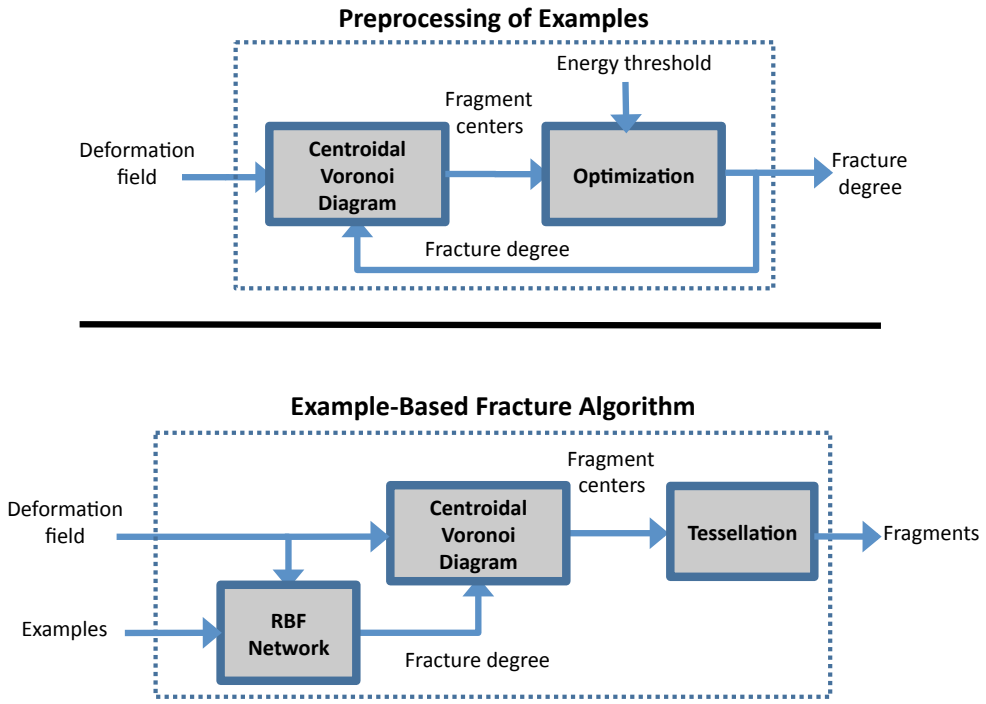

Figure 3. As a pre-process (top), we compute multiple fracture examples of the same object, allowing us to learn a relation between the deformation field and the fracture degree. At runtime (bottom), given the deformation field of the fracturing object, we compute the fracture degree from the examples, solve a deformation-aware centroidal Voronoi diagram, and tessellate the resulting fragments.

is used as input for the computation of the fracture fragments (lines 10 - 11). Finally, the surfaces of the resulting fragments are tessellated (line 12). We use a tetrahedral mesh to discretize the computations in all three steps.

As shown in Figure 3, as a preprocess we compute a set of example fractures for each object, and these examples are used at runtime to accelerate the computation of the fracture degree N (see Section 3 for full details).

Algorithm 2 shows the computation of the location of sites, both for preprocessing examples or during runtime fracture. We first introduce new Voronoi sites randomly using as probability function the strain energy field (line 2). The high-dimensional coordinate of the site is initialized to the coordinates of the random tetrahedral node selected. Then we solve a discrete version of the high-dimensional CVD on the nodes of the tetrahedral mesh using Lloyd's method. In the context of this discretization, the strain energy $W(x)$, used as distance weight in Equation (2), needs to be evaluated at mesh nodes. Once the deformation is computed, we integrate strain energy on tetrahedra, and we set node weights $W(x)$ by summing one fourth of the strain energy over their incident tetrahedra.

Algorithm 1 Overall algorithm.

preprocess:
Create Tetrahedral Mesh
Calculate High-Dimensional Coordinates
Create Radial Basis Network (RBN) ▷ Section 3

runtime:
while True **do**
    while No impact **do**
        Rigid Body Simulation
    Compute FEM Deformation
    Learn Fracture Degree from RBN ▷ Section 3
    Compute CVD ▷ Algorithm 2
    Tessellate New Fragments ▷ Section 4
    Insert New Fragments into Rigid Body Engine

Algorithm 2 High-dimensional CVD.

for all s_i Sites **do**
    $InsertSite(s_i)$
    $n \leftarrow NearestNode(s_i)$
    $n.closestSite \leftarrow s_i$
    $Q.Push(n)$
while not converged **do**
// Flooding algorithm
    while Q not empty **do**
        $Q.Pop(n)$
        for all n_j neighbours of n **do**
            $d \leftarrow \|n_j.coords - n.closestSite.coords\|^2 W(n_j)$
            if $n_j.distance > d$ **then**
                $n_j.distance \leftarrow d$
                $n_j.site \leftarrow n.site$
                Push n_j in Q
// Move to centroid
    for $i \leftarrow 1,$ Number Of Sites **do**
        $MoveSiteToCentroid(s_i)$ ▷ Equation 4

The execution of Lloyd's method in the discrete setting requires small changes. First, the update of Voronoi cells reduces to finding the closest site for each node. We speed up this computation by flooding Voronoi cells from the sites, exploiting the graph defined by the tetrahedral mesh and using a queue Q (lines 8 - 15). In addition, to accelerate the evaluation of interior distances, as a preprocess we compute the high-dimensional coordinates for all nodes. Second, the computation of high-dimensional centroids $\bar{p}$ for each Voronoi cell uses the high-dimensional coordinates of the nodes of the cell $\bar{x_j}$:

$$\bar{p} = \frac{\sum_{j=1}^{N} \bar{x_j} W(x)}{N},$$

where N is the number of nodes in the Voronoi cell.

We repeat the two steps of Lloyd's method until convergence. For our examples, we stop this iterative process when the sites move less than $1e-7$.

It turns out that our algorithm does not require the 3D positions of the Voronoi sites at any time, and it is sufficient to store their high-dimensional positions and the cell-classification of the nodes.

2.4. Artist Control

A major feature of our proposed fracture algorithm is that it can accommodate many types of artist control in very simple ways. Other than the trivial energy threshold γ that guides the overall toughness of the object, we have considered artist-driven fracture granularity, inhomogeneous material toughness, anisotropy, and smoothness, but other properties may also be controllable.

Fracture granularity and material inhomogeneity may be easily controlled by tweaking the strain energy, W. Using an exponential factor of the strain energy, W^{α}, affects the influence of the deformation on the fragment distribution (See Figure 4). Using a spatially varying multiplicative factor, βW, allows for controllable material inhomogeneity. The artist may "paint" fragile regions with $\beta > 1$ and tough regions with $\beta < 1$.

Fracture anisotropy and smoothness may be easily controlled by tweaking the distance metric. A simple way to do this is to apply a non-uniform transformation

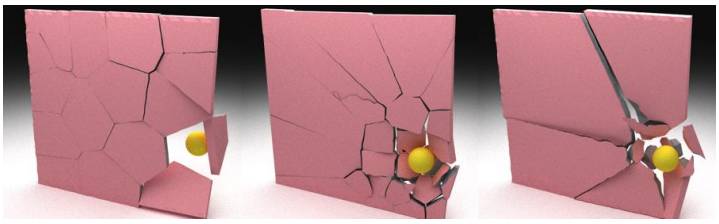

Figure 4. Artist control of fracture granularity through simple modification of the exponent of strain energy, W^{α}. From left to right: $\alpha = 0$ (the deformation is ignored), $\alpha = 0.5$, $\alpha = 1$.

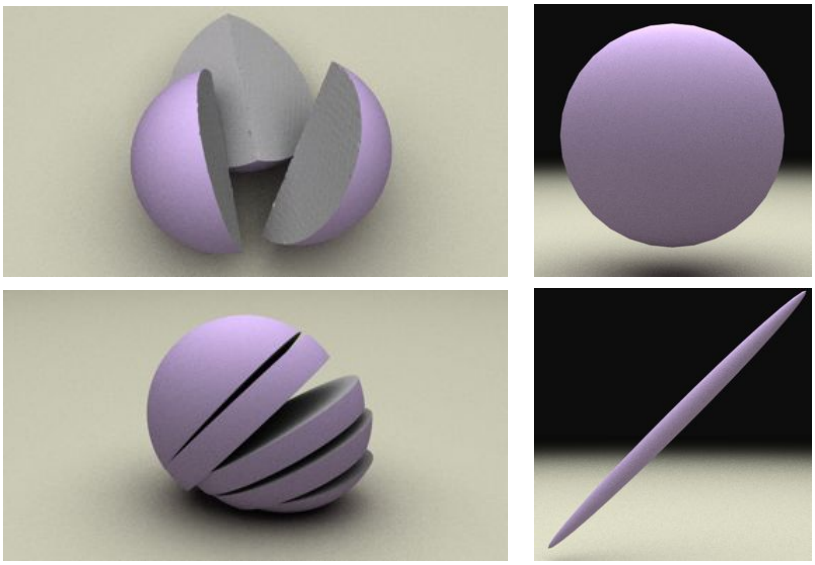

Figure 5. Artist control based on the modification of the rest shape. Top: Regular fracture of a sphere when its rest-shape is not modified. Bottom: Fracture with a preferred direction obtained by applying anisotropic scaling to the rest shape (on the right).

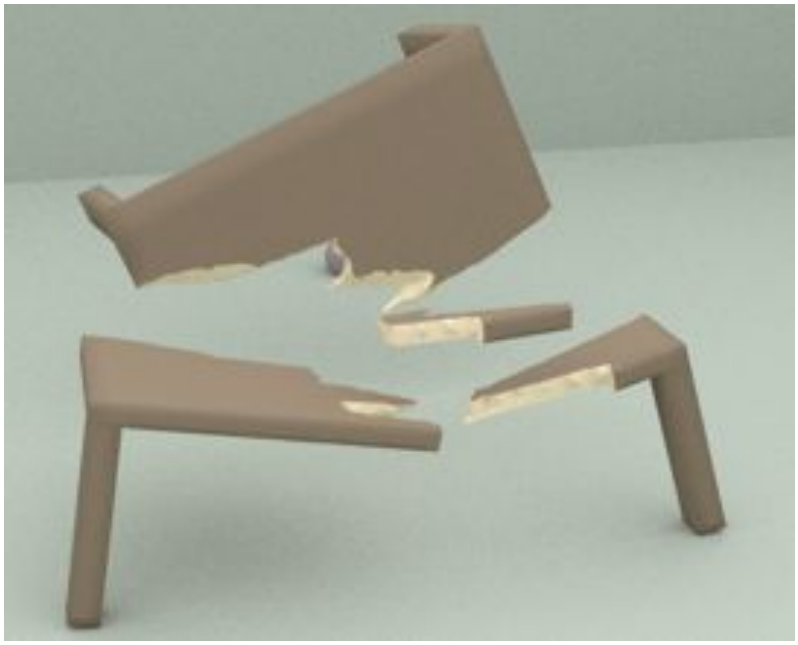

Figure 6. Fracture of a wooden table enabled by artist control. A wavy transformation is applied to the rest-shape of the table, and thus it produces concave wood-like fragments upon fracture.

to the undeformed reference object where distances are computed. Figure 5 shows an example of anisotropic material failure obtained by applying non-uniform scaling to the reference object. Figure 6 shows wood-like fracture of a table, obtained by applying a wavy transformation to its rest shape.

3. Learning Fracture from Examples

Our fracture model defines both the fracture degree N and the fragment centers P^* as a function of the strain energy W.

To generate a set of examples, we precompute the fracture degree following a simple optimization process. We initialize the fracture degree $N = 1$, and we double it until the distance-weighted deformation energy E_D is smaller than the fracture threshold γ. Then, we perform a bisection search on the fracture degree until we reach the smallest value for which the distance-weighted deformation energy is smaller than the fracture threshold.

With this method, computing the fracture degree requires solving a costly iterative optimization problem. In this section, we introduce a learning method that, based on a set of precomputed fracture examples, allows us to efficiently estimate the fracture degree at runtime as a function of the deformation field. We describe our specific learning method based on a radial basis function (RBF) network, and we discuss its approximation accuracy.

3.1. RBF Network

Let $\mathbf{W}$ be regarded as a vector that concatenates the strain energies of all nodes of the mesh. Our fracture model can be represented as a process that calculates the fracture degree N and the fragment centers P^* as a process

$$(N, P^*) = f(\mathbf{W}).$$

Conceptually, the fracture model can be divided into two functions; determining the fracture degree:

$$N = f_N(\mathbf{W}),$$

and computing the fragment centers as the CVD:

$$P^* = \mathrm{CVD}(N, \mathbf{W}).$$

These two conceptual functions correspond to the optimization problems in Equations (1) and (2), respectively.

Because of its iterative nature, computing the fracture degree constitutes the bottleneck of our fracture model. We propose an example-based approach to approximate

f_N in a fast manner. As a preprocess, we generate a set of example strain energy vectors $\{\mathbf{W}_i\}$, and compute the fracture degree N_i for each example. The example-based approximation of the fracture degree can be formalized as

$$N = \hat{f}_N(\mathbf{W}, \{\mathbf{W}_i, N_i\}) \approx f_N(\mathbf{W}).$$

To robustly learn the main aspects of the mapping between the deformation field and the fracture degree, without incurring overfitting, we perform a principal component analysis (PCA) of the strain energy data $\{\mathbf{W}_i\}$, and compute a reduced deformation basis. This can be easily achieved by using Matlab's tool *princomp*. Matlab will return the principal component coefficients in a matrix Π, which we use to project the strain energy vector $\mathbf{W}$ onto the reduced basis. The example-based approximation of the fracture degree can then be rewritten as

$$N = \hat{f}_N(\Pi\mathbf{W}, \{\Pi\mathbf{W}_i, N_i\}).$$

We have designed an example-based model to compute the fracture degree using an RBF network. Given a strain energy vector $\mathbf{W}$, the fracture degree is computed as

$$N = \sum_i^k w_i \phi(\|\Pi\mathbf{W} - c_i\|) + b.$$

Based on the training examples, we compute the k RBF centers $\{c_i\}$, the RBF weights $\{w_i\}$, and the bias b in Matlab using the function *newrbe*. We have modified this function so that the RBF is $\phi(r) = r$, with global support. The value k is calculated by incrementing the number of RBF centers until the net does not exceed a maximum error using the training examples.

The PCA projection matrix Π, together with the RBF centers $\{c_i\}$, the RBF weights $\{w_i\}$, and the bias b are precomputed and used during runtime to efficiently compute the fracture degree.

3.2. Training and Test Sets

The set of examples used for training the RBF network could depend on the types of interactions expected at runtime. For example, for the wall benchmark in Figure 4, we generated training examples by throwing balls with different velocities at different points on the wall. However, as a general procedure for example generation, we propose to drop an object from different heights and with different orientations. We found that this simple procedure is capable of producing diverse deformation distributions. The fractures in Figure 1 were produced by throwing balls at the horses, using examples produced by dropping horses.

To test the quality of the example-based fracture model presented above, we have compared its results to the iterative algorithm presented in Section 2.3. We have generated several training and test examples for both the horse and bunny objects shown

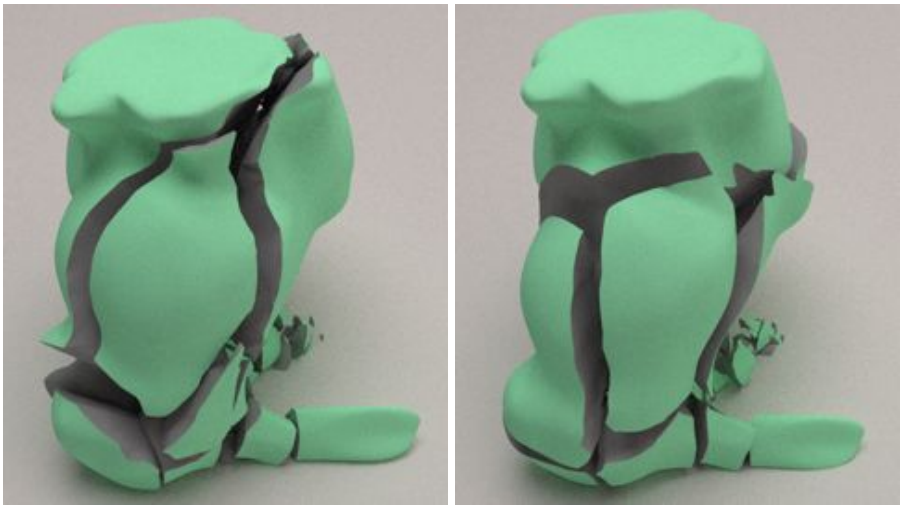

Figure 7. Simulation of the fracture of a bunny with iterative (left), vs. example-based computation of the fracture degree (right).

in Figure 2 and Figure 7. For training, we drop each object from six different heights and with 64 random, uniformly distributed orientations. We repeat each experiment three times, to account for the randomness in the initialization of Voronoi sites. For testing, we drop the objects from the same six heights, but with 18 different orientations.

Figures 8 and 9 show the learning results for the horse example (The bunny example had very similar results.). The two plots (Figure 8) show that the example-based fracture degree is very similar to the one computed through the full iterative method, both for the training and test data sets. On the top, we plot the mean and standard deviation of the fracture degree across all initial orientations of the horse for the same drop height. On the bottom, we plot the exact fracture degree for all the orientations and heights in the test set.

Figure 9 compares the results for various error settings in the PCA projection of the strain energy, for a drop height of 1 m. With no PCA projection, the method suffers from overfitting, which results in larger errors in the fracture degree (indicated by a larger standard deviation). With a PCA error of 10%, the quality of the fitting degrades too. Therefore, in all the examples in the paper, we used an error tolerance of 2% in the PCA projection. For the horse example, this error tolerance reduces the size of the strain energy vector $\mathbf{W}$ from 1595 to 54 components.

4. Tessellation of Fragments

Our fracture animation method allows concave crack surfaces through the artist control methods described in Section 2.4, but curved crack surfaces complicate the

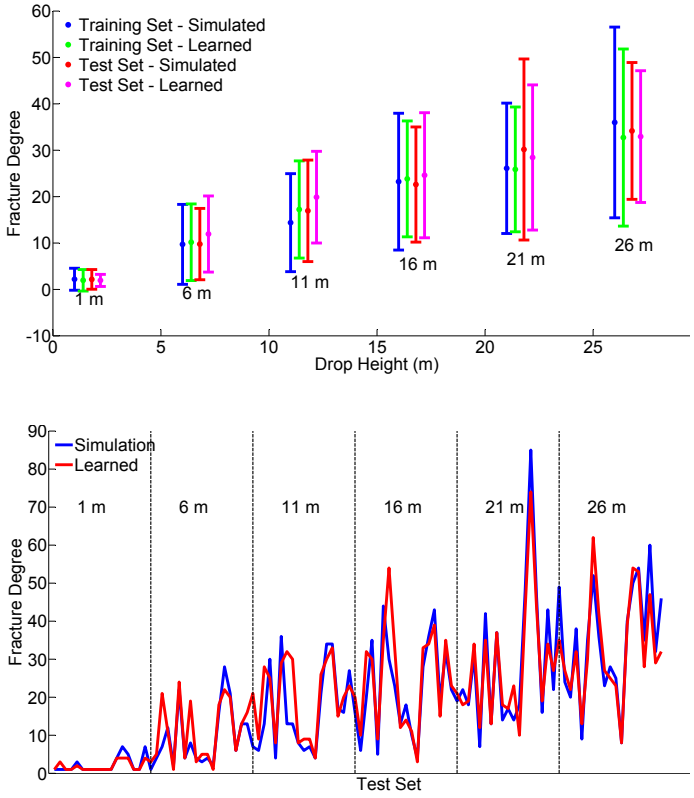

Figure 8. We study how well our algorithm learns the fracture degree when we drop the horse in Figure 2 from different heights with different orientations. Top: Mean and standard deviation of the fracture degree across different orientations, using the simulation and learning methods on the training and test sets separately. Bottom: Exact comparison of the fracture degree for all the heights and orientations in the test set.

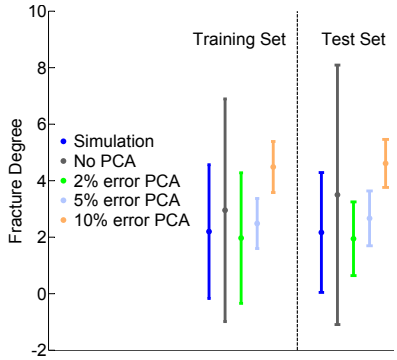

Figure 9. Mean and standard deviation of the fracture degree for the horse benchmark (Figure 2) across all orientations and a height of 1 m, for various PCA error settings.

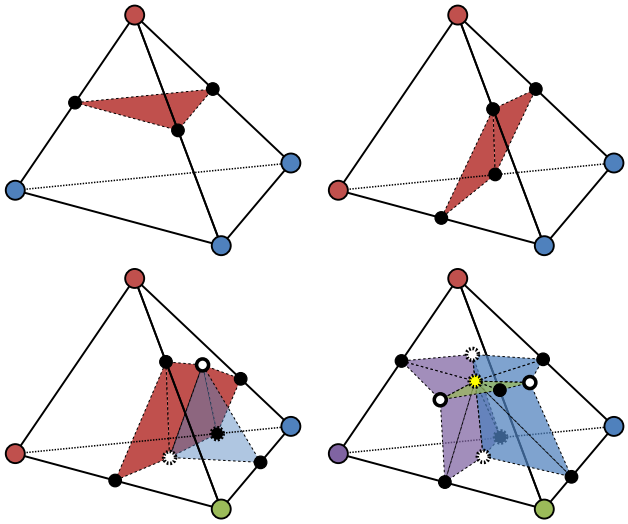

Figure 10. Four canonical tessellation schemes of a tetrhaedron. The nodes of the tetrahedron are colored based on the fragment to which they belong, and the fracture vertices are colored in the following way: black for intersections of Voronoi sheets and tetrahedral edges, white for intersections of Voronoi edges and tetrahedral faces, and yellow for Voronoi vertices.

tessellation step. In this section, we propose a highly parallel algorithm to tessellate the fragments resulting from the CVD. Our algorithm exploits the tetrahedral mesh used for deformation computations and defines a simple local tessellation procedure inside each tetrahedron independently (see Algorithm 3). Each tetrahedron is processed in parallel.

We tessellate the CVD by approximating its intersection with a tetrahedral mesh. We start by labeling the nodes of each tetrahedron according to their closest Voronoi site (i.e., fragment center), and then the tessellation remains local to each tetrahedron. Based on the labeling of nodes, we select one out of four canonical tessellation schemes (shown in Figure 10) that entirely define the arrangement of fracture triangles inside the tetrahedron (line 2 of Algorithm 3).

The intersection of the CVD with the tetrahedral mesh may produce three types of fracture vertices:

- If the two nodes of a tetrahedral edge are labeled differently, we compute a fracture vertex as the intersection of a Voronoi sheet with the edge.
- If the three nodes of a tetrahedral face are labeled differently, we compute a fracture vertex as the intersection of a Voronoi edge with the face.
- If all four nodes of a tetrahedron are labeled differently, a Voronoi vertex defines a fracture vertex inside the tetrahedron.

Once all the nodes have been labeled, we proceed with the parallel tessellation of all tetrahedra. For each tetrahedron, based on the number of nodes closer to each of the four possible different sites, the four canonical tessellation schemes can be summarized as follows (see Figure 10):

- $(3,1,0,0)$: There are three fracture vertices at edges and one fracture triangle.
- $(2,2,0,0)$: There are four fracture vertices at edges and two fracture triangles.
- $(2,1,1,0)$: There are five fracture vertices at edges, two more at faces, and five fracture triangles.
- $(1,1,1,1)$: There are six fracture vertices at edges, four more at faces, one due to a Voronoi vertex, and 12 fracture triangles.

All three types of fracture vertices can be computed following a common definition: Given a simplex with all its M nodes closer to M different Voronoi sites, the fracture vertex is given by the barycentric combination of the nodes which is equidistant to all sites (line 3 of Algorithm 3). We propose an approximate computation of the fracture vertex, based on the barycentric interpolation of distances to the Voronoi sites. We construct a matrix D of size $M \times M$, where each term d_{ij} stores the distance from node n_i to the Voronoi site p_j, which is the site closest to node n_j. Then, the barycentric coordinates b of the fracture vertex and the distance d to the Voronoi sites are given by the solution to the system

$$\begin{pmatrix} D & -[1] \\ [1]^T & 0 \end{pmatrix} \begin{pmatrix} b \\ d \end{pmatrix} = \begin{pmatrix} [0] \\ 1 \end{pmatrix}.$$

$[0]$ and $[1]$ represent column vectors of 0s and 1s of size M.

For an edge, the fracture vertex is guaranteed to lie inside, but this is not true for a face or a tetrahedron. In such cases, we propose an approximation of the fracture vertex that optimizes the smoothness of the approximate Voronoi sheets, but is constrained to lie inside the face or tetrahedron, therefore guaranteeing the locality of the tessellation algorithm. Instead of considering arbitrary positions inside a tetrahedron, we reduce the complexity of the problem by choosing the approximate fracture vertex out of the set of fracture vertices at lower-dimensional simplices or their centroid.

Algorithm 3 Tessellation.

```
1: for all Tetrahedra do
2:     Select tessellation scheme
3:     Calculate fracture vertices
4:     Store fracture surface
5: Triangulate intersection of fracture surface and triangle mesh
```

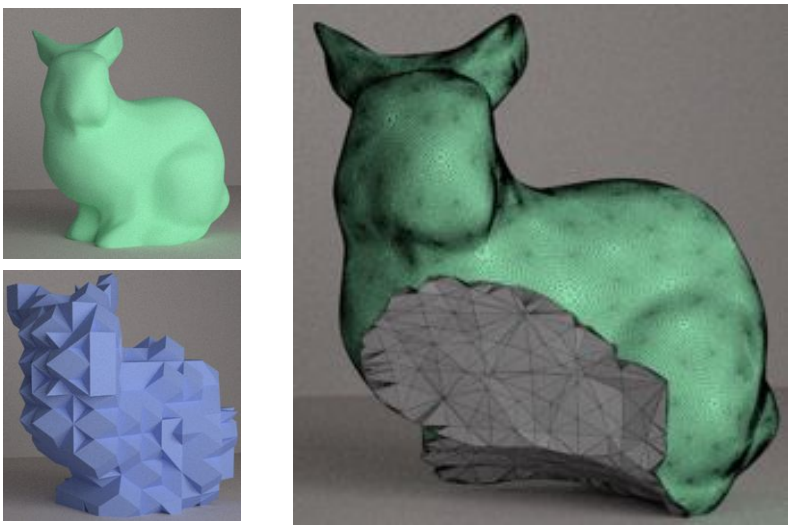

Figure 11. A triangle mesh (top left), its corresponding tetrahedral mesh (bottom left) and one fragment after fracture (right). The fracture surface is created from the low resolution tetrahedral mesh. The clipping process creates small triangles to merge the low resolution fracture surface and the original high resolution triangle mesh.

Constraining the tessellation to each tetrahedron simplifies the computational process, but also improves the smoothness of fracture surfaces.

If the algorithm uses only a high-resolution tetrahedral mesh for simulation and rendering, then the tessellation would not need any more processing. However, in our examples, we use tetrahedral meshes of moderate resolution that embed the high-resolution triangle surfaces. Then, the tessellation requires a final step of clipping fracture surfaces against the original triangle-based surface (see Figure 11). To accelerate this clipping operation, as a preprocess we store in each tetrahedron a list of the surface triangles it intersects. Then, when a tetrahedron is fractured, we test for intersections of its fracture triangles and its embedded surface triangles (line 5 of Algorithm 3), and we triangulate the surface triangles using the Triangle library [Shewchuk 1996].

The triangulation method is called for each fracture triangle that intersects with surface triangles, and vice versa. The Triangle library receives as an input 2D points and segments (that will be included in the final triangulation) and 2D hole points (that define the concavities of the surface). The whole triangulation process takes place in the plane defined by the triangle T that is being divided. The segments of the triangulation are defined by the edges of T and the intersection segments of other triangles t_i with T. We set hole points on the midpoint of each intersection segment, with a small displacement in the direction of the 2D projection of the normal of t_i. In our examples, we displace the hole point by $1e-7$ m.

The triangulation might generate thin triangles, as shown in Figure 11. This is a result of trying to merge a high-resolution triangle mesh, with a lower resolution fracture surface. Small or thin triangles could lead to numerical errors, especially if the fragment is fractured again. These numerical errors could be avoided by post-processing the fracture surface to increment its resolution.

It is important to note that the interior of the original model lacks material detail or texture parameterization. For the creation of our videos, we have exported each fragment into two separate files: one containing the crack surface and another one with the rest of the surface triangles. However, for interactive applications, one may also use methods such as 3D texturing, procedural materials, or dynamically generated parameterization and texture. Some examples of these techniques are shown in Pietroni et al. [2010] and Takayama et al. [2010].

5. Results

We have integrated our fracture algorithm in the Bullet Physics simulation engine. The objects are simulated as rigid bodies until a collision is detected, and then we compute the resulting deformation field using a quasi-static finite element formulation. We then apply our fracture algorithm, and insert the resulting fragments as rigid bodies in the Bullet scene.

We have executed several benchmarks in order to analyze different aspects of our algorithm. All experiments were executed on a 3.4GHz Intel i7-2600 processor with 8GB of RAM, using 8 cores for the parallel tessellation of fragments.

The various examples in the paper highlight the richness and versatility of fracture patterns with our method, as well as the influence of collision scenarios. As expected, smaller fragments are concentrated in the zone of impact. Moreover, if an object falls on a thin part (e.g., when the bunny in Figure 7 falls on its ears), the object naturally shatters into more fragments than when an object falls on a large part (e.g., when the horse in Figure 2 falls on its head). Please see the accompanying video. Also, as expected, the fracture degree grows on average as objects are dropped from a greater height, as shown in Figure 8. Figure 2 demonstrates the expected behavior when the material is modified. When the horse is dropped from the same height and with the same orientation, it breaks into more and smaller fragments with a choice of weaker material (lower threshold γ).

We have already discussed the accuracy of the example-based approach for computing the fracture degree in Section 3.2. Figure 7 shows a comparison of a bunny whose fracture degree is computed iteratively versus a bunny whose fracture degree is quickly computed based on examples. We observe that the distribution of fragments with the example-based approach is very similar to the iterative method, but it reduces the computation of the final CVD from more than one second to under 70 ms. Fig-

	#tris	#tetras	#nodes
Horse (Figs. 1 & 2)	38016	5256	1595
Wall (Figure 4)	4800	1408	582
Bunny (Figure 7)	163584	5301	1477
Mushrooms (Figure 12)	768	1309	433

Table 1. Resolution details of our benchmarks. We indicate the resolution of the triangle meshes and their corresponding tetrahedral meshes.

	high-dim	RBFs	Frac. deg.	CVD (iters)	CVD (ms)	Tessel. (ms)
Horse	74	38	7 / 67	4 / 15	15 / 37	65 / 233
Wall	-	18	3 / 18	7 / 3	7 / 7	24 / 58
Bunny	147	45	3 / 48	13 / 18	49 / 67	213 / 272
Mushrooms	250	90	5 / 81	10 / 18	26 / 21	23 / 45

Table 2. Simulation statistics for several benchmarks. First, we indicate the size of their high-dimensional embedding space for interior distance computation and the number of RBF centers used for example-based computation of the fracture degree. Then, for two different fracture examples with each model, we indicate the fracture degree, the number of iterations needed to compute the CVD, and the time to compute the CVD and the tessellation (in ms.)

ure 8 indicates the same results for a large battery of tests executed on the horse in Figure 2.

In Tables 1 and 2, we show some statistics and timings to compute a single fracture event in our benchmarks. For a small fracture degree, the cost is dominated by the CVD computation, but as the fracture degree grows, the cost is rapidly dominated by the tessellation, in particular by the tessellation of the original surface, as noted by the data from the bunny benchmark. Our implementation takes advantage of multi-core CPU architectures to parallelize the tessellation, but further performance improvements would be possible by using the general triangulation routines in Triangle [Shewchuk 1996] only in complex cases. It is worth noting that, for the wall model, which is originally convex, we simply used the Euclidean metric as a suitable interior distance metric.

The mushroom benchmark in Figure 12 shows a practical application of our algorithm in an interactive scene. A user throws and drags around the mushrooms interactively, producing fractures and collisions. As indicated in Table 2, fracture events in this scene reach a cost of up to 66 ms with our algorithm. The complete simulation, including collision handling with the Bullet engine, runs at an average rate of 14.80 fps, and it contains 250 fragments. The interactive horse benchmark in Figure 1 shows how fracture patterns adapt to the various collision scenarios. This example also demonstrates the plausibility of example-based fracture. The animation runs at an average rate of 21.82 fps, and it consists of 69 fragments at the end.

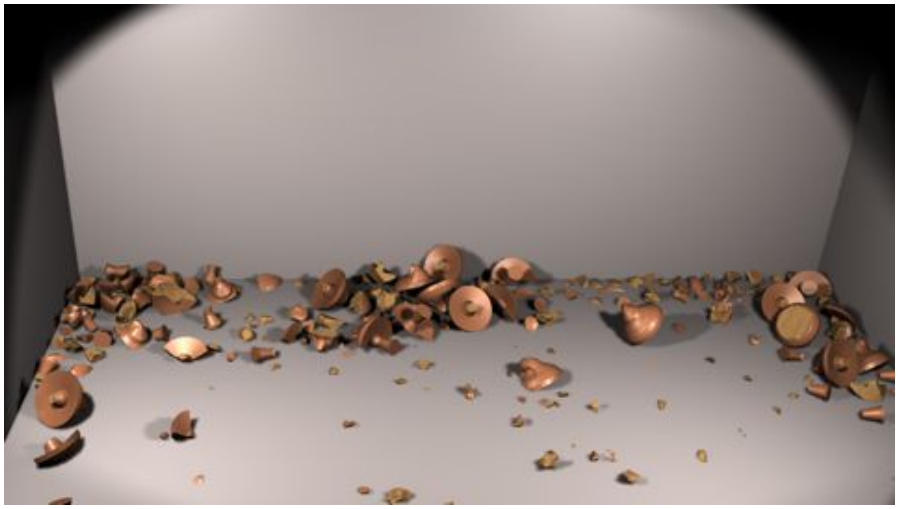

Figure 12. Breaking Mushrooms. The user throws and drags around mushrooms interactively. The complete simulation runs at an average rate of 14.80 fps, and it consists of 250 fragments at the end.

6. Discussion and Future Work

We have presented an algorithm to simulate brittle fracture that combines the flexibility and plausibility of physically based methods, with the efficiency and artist control of Voronoi-based geometric methods. The major components of our algorithm are a formulation of fracture as the computation of an interior CVD, an example-based learning method to accelerate the computations, and a highly parallel local tessellation method based on a tetrahedral lattice. In our implementation, the tetrahedral lattice is the same tetrahedral mesh used for computing elastic deformations, but our algorithm could easily be extended to other deformation algorithms. It would be sufficient to evaluate deformation energies at the nodes of the tetrahedral lattice.

The major limitation of our approach is that it relies heavily on preprocessing, at several levels. First, the computation of the high-dimensional embedding space for the evaluation of interior distances requires a precomputation step [Rustamov et al. 2009]. Second, the example-based computation of the fracture degree requires the precomputation of multiple simulation examples for each object. In our benchmarks, we have computed up to 1152 examples per object (6 heights $\times$ 64 orientations $\times$ 3 instances). With an average cost of two seconds per example, this amounts to a preprocessing cost of 38 minutes. Note that this preprocessing can be trivially parallelized. Fortunately, our results indicate that a general training procedure, based on dropping objects from various heights and with various orientations, is sufficient even for runtime simulations with very different dynamics and collisions.

From our experiments, we can easily identify tessellation, in particular the tessellation of the fine surface, as the major bottleneck of the runtime algorithm. As

mentioned in the previous section, further optimizations would be possible by identifying simple cases that do not require the general triangulation solution of Triangle [Shewchuk 1996].

In conclusion, we believe that our work also poses interesting questions about the outreach of current geometry-based and physically based methods for simulating fracture. A combination of both methods, as done in our algorithm, produces plausible results with high flexibility for artist control. It would be interesting to extend artist control tools to handle data from real-world fractures. Another open question is whether Voronoi methods could be used for handling partial and/or ductile fracture.

Acknowledgements

This research is supported in part by the Spanish Ministry of Economy (TIN2012-35840) and by the European Research Council (ERC-2011-StG-280135 Animetrics). We would also like to thank Ming C. Lin and Alberto Sánchez for initial discussions.

References

MÜLLER, M., MCMILLAN, L., DORSEY, J., AND JAGNOW, R. 2001. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic Workshop on Computer Animation and Simulation*, Springer-Verlag New York, Inc., New York, NY, USA, 113–124. URL: http://dl.acm.org/citation.cfm?id=776350.776361. 36

PIETRONI, N., CIGNONI, P., OTADUY, M., AND SCOPIGNO, R. 2010. Solid-texture synthesis: A survey. *Computer Graphics and Applications, IEEE 30*, 4 (July), 74–89. URL: http://vcg.isti.cnr.it/Publications/2010/PCOS10/, doi:10.1109/MCG.2009.153. 50

RUSTAMOV, R. M., LIPMAN, Y., AND FUNKHOUSER, T. 2009. Interior distance using barycentric coordinates. In *Proceedings of the Symposium on Geometry Processing*, Eurographics Association, Aire-la-Ville, Switzerland, SGP '09, 1279–1288. URL: https://sites.google.com/site/raifrustamov/publications. 38, 52

SCHVARTZMAN, S. C., AND OTADUY, M. A. 2014. Fracture animation based on high-dimensional voronoi diagrams. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '14, 15–22. URL: http://doi.acm.org/10.1145/2556700.2556713, doi:10.1145/2556700.2556713. 35, 36, 37, 38

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Selected Papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, Springer-Verlag, London, UK, UK, FCRC '96/WACG '96, 203–222. URL: http://dl.acm.org/citation.cfm?id=645908.673287. 49, 51, 53

TAKAYAMA, K., SORKINE, O., NEALEN, A., AND IGARASHI, T. 2010. Volumetric modeling with diffusion surfaces. In *ACM SIGGRAPH Asia 2010 Papers*, ACM, New York,

NY, USA, SIGGRAPH ASIA '10, 180:1–180:8. URL: http://doi.acm.org/10.1145/1866158.1866202, doi:10.1145/1866158.1866202. 50

Index of Supplemental Materials

Please see the attached video for examples of our algorithm.

Author Contact Information

Sara C. Schvartzman
19-23 Wells Street
London
W1T 3PQ, UK
sara.schvartzman@gmail.com
http://www.gmrv.es/~sschvartzman

Miguel A. Otaduy
Universidad Rey Juan Carlos
Calle Tulipán, S/N
E-28933 Móstoles, Spain
miguel.otaduy@urjc.es
http://www.gmrv.es/~motaduy

Sara C. Schvartzman, Miguel A. Otaduy, Physics-Aware Voronoi Fracture with Example-Based Acceleration, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 3, 35–54, 2014
http://jcgt.org/published/0003/01/01/

Received:	2013-10-22		
Recommended:	2013-12-09	Corresponding Editor:	Cindy Grimm
Published:	2014-10-07	Editor-in-Chief:	Morgan McGuire