Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2014)
Vladlen Koltun and Eftychios Sifakis (Editors)

A Peridynamic Perspective on Spring-Mass Fracture

J. A. Levine[1], A. W. Bargteil[2], C. Corsi[1], J. Tessendorf[1], and R. Geist[1]

[1]Clemson University, USA
[2]University of Utah, USA

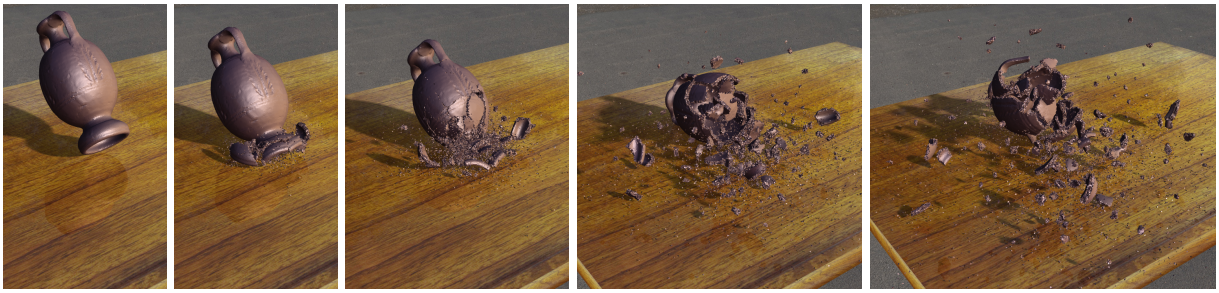

Figure 1: Vase of 637027 particles undergoing fracture at frames 153, 175, 199, 264, and 300.

Abstract

The application of spring-mass systems to the animation of brittle fracture is revisited. The motivation arises from the recent popularity of peridynamics *in the computational physics community. Peridynamic systems can be regarded as spring-mass systems with two specific properties. First, spring forces are based on a simple strain metric, thereby decoupling spring stiffness from spring length. Second, masses are connected using a distance-based criterion. The relatively large radius of influence typically leads to a few hundred springs for every mass point. Spring-mass systems with these properties are shown to be simple to implement, trivially parallelized, and well-suited to animating brittle fracture.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations;

1. Introduction

Physically-based modeling and rendering of natural phenomena has gained increasing importance as demands for realism and detail in animations, particularly within the visual effects industry, have escalated at a rapid pace. One particularly important phenomenon is the process of solids undergoing fracture. A simple approach to animating fracture, demonstrated more than twenty years ago [TF88, NTB*91], is to use a spring-mass system and dynamically remove overly extended springs. Despite significant advances and the advent of more sophisticated techniques, this simple approach remains popular. Motivated by recent work in the computational physics community on *peridynamics* [Sil00, ELP13], we describe a particular variation on this approach that is simple to implement, trivially parallelized, and well-suited to animating brittle fracture.

Peridynamic systems can be characterized as spring-mass systems with two particular properties. First, spring forces are based on a simple strain metric, ε, which normalizes a spring's deformed length by its rest length, thereby decoupling spring stiffness from spring length. Specifically, the traditional spring force exerted on node i by node j is

$$\mathbf{f}_i = -k(\|\mathbf{x}_i - \mathbf{x}_j\| - d_{ij})\frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}, \qquad (1)$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are the deformed positions of nodes i and j, d_{ij} is the spring's rest length, and k is the spring stiffness.

DOI: 10.2312/sca.20141122

Instead, peridynamics defines a simple strain metric

$$\varepsilon = \frac{\|\mathbf{x}_i - \mathbf{x}_j\| - d_{ij}}{d_{ij}} = \frac{\|\mathbf{x}_i - \mathbf{x}_j\|}{d_{ij}} - 1 \tag{2}$$

and then defines the force as

$$\mathbf{f}_i = -k\,\varepsilon\,\frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}. \tag{3}$$

This seemingly small change (note that ε is dimensionless) has the effect of decoupling a spring's stiffness, k, from its rest length. Consequently, springs of different lengths will share the same stiffness.

The second variation from a standard spring-mass system involves the spring connection topology. In a standard spring-mass system, nodes (point masses) are connected based on a mesh topology, typically defined using 1-ring neighborhoods. In peridynamic systems, spring connections are set between all pairs of nodes that are within a particular distance from one another. Specifically, given a *length scale*, λ, we place a spring between any pair of nodes that are within a distance $\delta = N\lambda$ from one other, where N is typically an integer between 2 and 6. This choice creates networks that can involve hundreds of springs per node, which is far beyond the connectivity found in typical spring-mass systems.

To model fracture, a spring is removed from the system whenever its strain exceeds a threshold, τ. It is worth noting that k and τ depend not only on properties of the material, but vary with δ. Specifically,

$$k = \frac{18K}{\pi\delta^4} \qquad \text{and} \qquad \tau = \sqrt{\frac{5G}{K\delta}} \tag{4}$$

where K is the *bulk modulus* and G is the *fracture energy* of the material [SA04]. See Silling and Askari [SA05] for the derivation of the spring constant. Note that spring stiffness decreases quickly with larger values of δ. To model non-uniformity in solids, one can vary each springs's τ (we use a normal distribution with a standard deviation of 2% from the above mean). To model the fact that some brittle materials become stronger under compression, the effective threshold, τ', can be updated after a round of force calculations by

$$\tau' = \tau - \alpha \min(0, \varepsilon_{min}), \tag{5}$$

where α is a user-specified constant (we use 0.25) and ε_{min} is the minimum strain over the incident springs [SA04].

Finally, we adopt a simple explicit integration scheme, velocity Verlet. While much maligned for their onerous timestep restrictions, explicit integration is not only simple to implement, but, as in solid-fluid coupling, the small timesteps greatly simplify the modeling of fracture dynamics. With large timesteps, one must model the fracture propagation and the resulting energy release that occurs over the timestep. In contrast, with small timesteps, we are able to model fracture propagation by simply remove a subset of the active springs every timestep and the resulting energy release is accounted for in the next timestep.

While our point-based geometric representation offers great simplicity, this simplicity does come at a cost—it is difficult to generate render geometry. Although the particle skinning problem has received a great deal of attention (e.g. [Bli82, BGB11, YT13]), these solutions have largely targeted *smooth* materials, like liquids. To model realistic fracture, we require sharp edges in the visualization geometry. This naturally motivates the constructive solid geometry (CSG) approach used in this paper. Solving these geometric problems is critical for computer graphics applications and our solutions comprise the main technical novelty of this paper.

Because these techniques focus on simplicity, they complement each other and, taken together, provide a simple and effective approach for realistic animation and rendering of brittle fracture that is straightforward to implement on the GPU.

2. Related Work

Fracture has been a topic of interest in computer graphics since the work of Terzopoulos and Fleischer [TF88] more than 25 years ago. Shortly thereafter, Norton and colleagues [NTB*91], described a fully three-dimensional spring-mass model of fracture that closely resembles our peridynamics-inspired approach.

As computer hardware improved, researchers turned to more sophisticated approaches, such as the finite element method (FEM), to animate deformable bodies and fracture. In this context, the focus has centered on accommodating the discontinuities that arise from fracture. O'Brien and colleagues [OH99, OBH02] used dynamic remeshing techniques to align mesh elements with the discontinuities created during fracture. More recently, Busaryev et al. [BDW13] applied a Delaunay remeshing strategy to fracture of thin surfaces. Müller and Gross [MG04] avoided remeshing by limiting fracture to follow the faces of the simulation elements, only requiring the duplication of vertices. Parker and O'Brien [PO09] also adopted this approach, but introduced "splinters" to hide the resulting artifacts.

Instead of remeshing and creating new elements, Kaufmann and colleagues [KMB*09] used the extended finite element method (X-FEM) to modify the underlying basis functions to account for discontinuities. Similarly, in their point-based method, Pauly and colleagues [PKA*05] updated the weight functions that determine how particles interact. Instead of simply disallowing interaction between separated particles, as we do, weights were reduced as points moved farther from the crack tip. Yet another alternative, known as the virtual node method, duplicates elements and treats them as partially filled [MBF04]. The modal analysis based method of Glondu et al. [GMD13] also adopted

this approach, but employed an implicit representation of the fracture surface.

Numerous non-physical techniques have also been employed, typically relying on clever geometric decompositions instead of physical definitions of cracks. Smith et al. presented a constraint-based system based on points and distances [SWB01], which offers simplicity both in implementation and user control. Ragavachary employs Voronoi diagrams to produce fractures [Rag02]. Martinet et al. employed procedural modeling based on hybrid trees [MGDA04]. Schvartzman and Otaduy casted fracture as an optimization using centroidal Voronoi diagrams of high dimension [SO14]. Müller and colleagues aligned approximate convex decompositions to fracture surface meshes in real time [MCK13]. Their algorithm applies dynamically computed fracture patterns, alleviating the need for prefracturing.

Despite the availability of these more sophisticated techniques, there has been continued interest in spring-mass models. In terms of fracture, Hirota et al. use spring-mass models for both surface [HTK98] and volume [HTK00] crack generation. More broadly, Lloyd and colleagues [LSH07] and Natsupakpong and colleagues [NCc10] have explored techniques to approximate finite element methods using spring-mass systems with carefully tuned spring constants. Both of these techniques, like peridynamics, utilize material-driven spring constants. More recently, Liu and colleagues [LBOK13] investigated fast integration of spring-mass models.

An interesting connection appears in the popularity of spring-mass systems for modeling hair. Selle and colleagues [SLF08] went beyond connecting only nearest neighbors and added virtual points and a variety of springs to address torsion in straight hair, while Iben and colleagues [IMP*13] defined several sophisticated springs for simulation of curly hair. Peridynamics also goes beyond simply connecting nearest neighbors, though we connect point masses based on a spatial range query that is independent of any mesh structure.

Peridynamics Over the last fifteen years, the computational physics community has been exploring a formulation of fracture referred to as "peridynamics" [Sil00, ELP13]. This research has resulted in several useful insights. First, many of the alternative approaches discussed above for dealing with discontinuities are somewhat complex and are not without drawbacks, particularly in difficulty of implementation and ensuring numerical robustness. By treating objects as collections of point masses inter-connected by springs, we can simply remove springs to achieve fracture in our systems. Second, as discussed above, by using a strain metric to measure the deformation of springs, we can set the spring constants on the basis of material properties rather than in an ad-hoc manner. Third, by connecting particles over long distances, we can capture non-local effects.

3. Implementation

Preprocessing We first compute the minimum inter-particle distance, λ, and then place springs between pairs of particles that are within distance $\delta = N\lambda$ from one another. Spatial queries use a kD-tree of particle positions. The value of N is thus one of the controls on material properties. We typically use 2-6, which yields a few hundred springs per particle. For each particle, we maintain its current position, its original position, its velocity, the forces upon it, the list of particles to which it is currently connected, and its current strain threshold τ.

Time Integration Our implementation uses velocity Verlet [SABW82] for numerical integration. Given that we use an explicit scheme, velocity Verlet provides a good trade off between speed, stability, and accuracy; symplectic Euler or leapfrog, which aside from initialization is equivalent to velocity Verlet, would be reasonable alternatives. If $\mathbf{x}, \mathbf{v}, \mathbf{f}$, and m are particle position, velocity, forces, and mass, and the time step is Δt, an update is given by:

$$\mathbf{v}(t+\frac{\Delta t}{2}) \leftarrow \mathbf{v}(t) + \frac{\Delta t}{2}\frac{\mathbf{f}(t)}{m}$$
$$\mathbf{x}(t+\Delta t) \leftarrow \mathbf{x}(t) + \mathbf{v}(t+\frac{\Delta t}{2})\Delta t$$
$$\text{Calculate } \mathbf{f}(t+\Delta t) \text{ from } \mathbf{x}(t+\Delta t)$$
$$\mathbf{v}(t+\Delta t) \leftarrow \mathbf{v}(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2}\frac{\mathbf{f}(t+\Delta t)}{m}$$

GPU Implementation The implementation is in CUDA with one particle per thread. We use seven CUDA kernels, which together comprise only 175 lines of code. One of the kernels handles position updates, and one handles velocity updates. These are entirely straightforward, as the velocity Verlet algorithm would suggest. The other five kernels are devoted to calculation of forces.

The five force kernels, *initialization, collection, collisions, springforces,* and *bodyforces*, are called in sequence, once each per timestep. The *initialization* kernel loads new strain limits based on any updates from the previous iteration and resets the hash bins used in the subsequent *collisions* kernel to empty. The *collection* kernel is a preliminary step required for collision detection. We use spatial hashing [THM*03] to put particle positions into bins. Since hashing executes in parallel (each particle determines its own bin and adds itself to that bin), we use the CUDA atomic operations to prevent conflicts.

Each particle executing the *collisions* kernel then locates its own bin and checks distances from itself to other particles in its bin and in the surrounding 26 bins, although some of these bin checks may be eliminated a priori. The force model for collision is quadratic repulsion. Specifically, the force exerted on particle p_i by particle p_j is

$$\mathbf{f} = K_c\left(\|\mathbf{x}_i - \mathbf{x}_j\| - \frac{\lambda}{2}\right)^2 \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} \tag{6}$$

Figure 2: Projectile (force $K_S = 1e15$ Pa) through a glass plate modeled with 131072 particles at frames 18, 20, 23, and 31.

where K_c is a model parameter. We use $K_c = 1e8$ Pa. Particles in the same connected component do not collide because collision forces are only applied at distances less than $\lambda/2$ and the minimum inter-particle spacing is λ.

The *springforces* kernel essentially follows that of Silling [SA05], where the particles are regarded as point masses inter-connected by springs. The *springforces* kernel, executed by each particle, p_i, in parallel, carries out three tasks:

1. It determines, for each particle, p_j, to which p_i is connected, the current strain, ε, between p_i and p_j. If $\varepsilon > \max(\tau_i, \tau_j)$ the spring is removed.
2. It updates the effective fracture threshold for the next timestep, $\tau' = \tau - \alpha \min(0, \varepsilon_{min})$.
3. It accumulates spring forces induced on p_i by the particles to which it is connected.

The *bodyforces* kernel accumulates force due to gravity, collision with the floor, and, for the glass example, the force of the projectile. Upon contact with the floor, the y-component of the velocity of a particle is reflected. We use a coefficient of restitution of 0.99 (99% of the y-component is retained). For the projectile, we achieved the best results using a repulsion force that we formulated experimentally. The direction of this force is halfway between the unit velocity vector of the projectile and the unit direction of pure repulsion from the projectile center, which can be computed by adding these vectors together and normalizing the result. If $\mathbf{c}$ is the center of the projectile and r is its radius, then the magnitude of the force acting on particle p_i is

$$f = K_S \left(16 \left(\|\mathbf{x}_i - \mathbf{c}\| - \frac{3r}{4} \right) \left(\|\mathbf{x}_i - \mathbf{c}\| - \frac{3r}{2} \right)^2 \right)^{2/3} \quad (7)$$

for $\|\mathbf{x}_i - \mathbf{c}\| \in [3r/4, 3r/2]$. This gives a force that is quadratic in distance, peaks at r with value $K_S r^2$, has relatively slow rise from $3r/4$ to r, and yet falls off quickly, but not discontinuously, away from r. The constant, K_S, is another model parameter. We provide video comparisons for $K_S = 1e10, 1e15$, and $1e20$ Pa.

4. Modeling Geometry

We experiment with three different choices for generating particle sets from input objects. The first and simplest is a regular grid of particles, each representing a cube of material of side length λ. While useful for modeling simple, regular shapes (such as a plate of glass), more complex objects require more sophisticated techniques.

Mesh-Based Geometry Given an input surface mesh, we construct a tetrahedral mesh of a fattened shell using a two step process. First, we initialize a collection of layers using the input surface as the most exterior, with each layer a fixed distance inward from the surface. We used six layers of particles for our vase model, and in practice we found that the number of layers should be larger than N. On each layer, we distribute vertices using the variational technique of Meyer et al. [MKW07]. Working layer-by-layer allows us to control both the inter-vertex distance in each layer as well as distance between layers without requiring a 3D optimization.

Finally, we construct a tetrahedral mesh from this point set using a Delaunay triangulation. We then exclude all tetrahedra whose vertices are either all from the most exterior surface or all from the most interior surface. The barycenters of remaining tetrahedra become the simulation particles.

Voronoi-Based Geometry While the mesh-based technique is fairly straightforward, it has two limitations. First, it means that fracture surfaces are restricted to faces of the underlying tetrahedral mesh, which results in well-known artifacts [MG04, PO09]. Second, it ties the resolution of the (surface) render geometry to the resolution of the (volumetric) simulation geometry, strictly limiting visual quality.

We employ a third technique to decouple the surface mesh resolution from the particle density. Given an arbitrary surface triangulation, we first compute a point cloud contained within the surface. To keep inter-particle distances bounded, but still introduce irregularity, we use blue noise sampling [CGW*13]. Next, we use clipped Voronoi diagrams to construct the volume decomposition [YWLL13].

Figure 3: Welsh dragon (473380 particles) fracturing on floor, sequence at frames 58, 78, 188, and 258.

Each particle is then associated with a set of triangles (possibly empty) and a set of planes (Voronoi faces). Consequently, we can use an arbitrary surface mesh, and rely on the Voronoi description to build geometry for our particles.

Alignment As the particles move during the simulation, we track which of those springs connecting adjacent geometric elements have broken. Using the adjacencies of neighboring geometric elements, we can then compute (with breadth-first search) the set of connected components at any iteration of the simulation. The graph of these adjacencies is strictly contained within the initial spring network, and edges are removed only when the springs are broken. To align geometry for rendering, we need only compute one (rigid) transformation for each connected component. For any component of n particles, given its initial point set (represented as an $n \times 3$ matrix) $\mathbf{P}$ and its current point set $\mathbf{Q}$ we compute the transformation from $\mathbf{P}$ to $\mathbf{Q}$ using Procrustes superposition [Kab78, TKA10].

5. Rendering

Each particle serves as a base point for an oriented, primitive geometric element. For the three examples provided here, we used cubes (glass plate, Figure 2), tetrahedra (falling vase, Figure 1), and Voronoi polytopes with embedded triangular mesh surfaces (falling Welsh dragon, Figure 3). The objects are constructed by the raytracer using CSG, and so elements that abut or overlap one another will appear as joined in a single piece.

Primarily, the raytracer executes on an NVIDIA K20 GPU with a standard allocation of one ray per thread. Each particle has a radius that describes the extent of its associated geometric element, and the particles and their radii are used to construct a kD-tree on the CPU. The kD-tree is flattened to a 1-dimensional array and loaded into the K20 memory. Raytracing on the GPU then uses the *short-stack* algorithm due to Horn et al. [HSHH07].

Raytracing either transparent cubic elements or opaque tetrahedral elements is straightforward and needs little explanation, except to note that for transparency, the last geometric element in the connected collection of elements that is first hit by the ray must be identified so that its orientation can be used to determine the exiting refractive ray and internal reflective ray. Raytracing a Voronoi polytope with internal triangular mesh (a *trimtope*) is only slightly more involved. The important cases are shown in Figure 4.

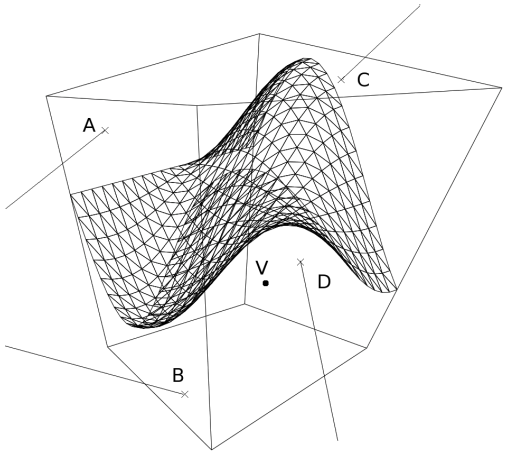

Figure 4: Raytracing polytope with internal mesh.

The Voronoi site of the polytope (point **V**) determines the forward/backward orientation of the triangle mesh. The mesh lies above **V**, in the forward direction. The ray **A** will enter and exit the trimtope without intersecting the triangle mesh. A secondary ray from the entry point of **A** to **V** will intersect a forward facing triangle, and so the ray is declared a "miss". The ray **B** will also enter and exit without intersecting the mesh, but a secondary ray from the entry point to **V** will either miss the mesh entirely or hit a backward facing triangle. In either case, the entry point to the trimtope is declared the "hit" location. The ray **C** will hit a forward facing triangle, and that is taken as the "hit" location. The ray **D** will hit a backward facing triangle, and so the entry point on the trimtope is again declared the "hit" location.

In all renderings, we used environment lighting for the diffuse lighting component and a normalized Blinn-Phong

shader for the specular component. The environment lighting technique employs the approximation due to Ramamoorthi and Hanrahan [RH01], which is based on the first 9 spherical harmonic coefficients. The HDR environment "parking lot" is from HDRI Hub [HDR13], and the woodgrain texture is from TextureX [Tex13].

6. Results

To evaluate our methods, we created three experimental setups, which are highlighted in Figures 1 (falling vase), 2 (glass plate / projectile), and 3 (falling Welsh dragon). For each simulation, our rendering framework varied as described above. To better understand peridynamics, we experimented with four parameters: K_S, K, δ, and τ. In addition, we made adjustments to the simulation framework based on both physical and artistic controls. Table 1 summarizes each of the models used in our simulations.

Model	δ	#Parts.	Type	# Tris.
Glass	3λ	131072	Cubes	73728
Vase	2λ	637027	Tetrahedra	80008
Welsh Dragon	6λ	473380	Trimtopes	499820

Table 1: Model summary, showing values for δ, the number of particles in each simulation, type of modeling primitive, and the number of triangles on the surface representation.

Performance Table 2 shows a quantitative analysis of the performance of each of the three simulation codes. Our Δt for all simulations was set to $2^{-23} = 1.192e{-}7$ seconds, the nearest power of two to the value of $1e{-}7$ used by Silling and Askari [SA04]. For the glass plate example, 500 iterations were computed for each rendering frame, while for the vase and dragon 1000 iterations were used. Consequently, all videos are in slow-motion.

We used the CUDA 5.5 profiler to measure the time spent in each of the seven CUDA kernels, and we aggregated the data across three different iteration ranges to give an estimate of mean occupancy and execution time before, during, and after impact within each simulation. Across all runs, the majority of the time (95-99%) was spent in either the *collisions* kernel or the *springforces* kernel, the balance depending on how much collision was occurring. Note that because the glass simulation can assume the particles are initially distributed in a regular grid, we can leverage this in our implementation of certain kernels. Collision was ultimately treated slightly differently in each simulation as a result.

A significant difference among the runs was the size of the radius, δ, used for computing the initial set of springs. In the dragon example, δ was set significantly higher ($\delta = 6\lambda$), resulting in a maximum of 517 initial springs, whereas the maximum number of springs were 121 and 185 for the glass and vase examples. The net result was a significantly larger

Model	Iter. Range	Occupancy	Total Time (s.)
Glass	0 - 1k	0.570	30.171
	50k - 51k	0.725	226.888
	100k - 101k	0.726	239.864
Vase	120k - 121k	0.622	268.888
	160k - 161k	0.622	268.236
	200k - 201k	0.626	265.479
Welsh Dragon	100k - 110k	0.395	525.335
	200k - 201k	0.395	511.017
	300k - 301k	0.393	505.449

Table 2: Mean GPU occupancy and execution time for each simulation at 1000 iteration ranges before, during, and after fracture.

time spent in the *springforces* kernel as opposed to the *collisions* kernel. This affects mean occupancy, since *collisions* achieved an occupancy of 0.75 vs. 0.375 for *springforces*.

Projectile Force Figure 5 shows a comparison of projectile force (K_S, $1e10$, $1e15$, and $1e20$ Pa) both at the point of impact as well as during impact (approximately $5.96e{-}5$ seconds later). In this case, while the apparent energy increases with K_S, the dominant crack patterns of the glass are identical because we selectively weakened a set of springs in the glass/projectile simulation. While we did this by using a random walk, an artist could use this same control to determine where cracks occur (selective weakening was not used in the other examples.)

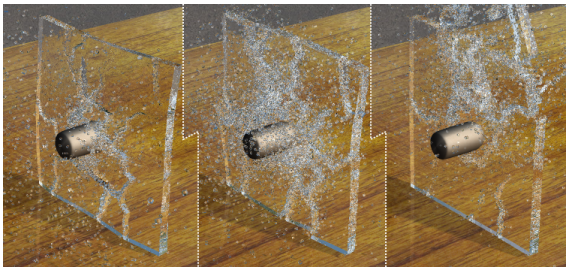

Figure 5: Comparison of projective impact at frame 21 for varying projectile force $K_S = 1e10, 1e15, 1e20$ Pa from left to right.

A visual comparison of the three runs shows that when the projectile force is weaker, the glass behaves as if it is being struck with a blunt hammer. Nevertheless, at the highest force, the projectile punches through the glass, effectively disintegrating a hole along the way. Controlling these parameters, as well as other body forces, allows for a fairly diverse set of fracture behaviors.

Spring Force In a second experiment, we varied the bulk modulus, K, of the falling vase. Instead of targeting a specific value (such as $K = 32.81e9$ Pa for glass), we varied the value of K for the vase and dragon examples until we

Figure 6: Comparison of vase impact at frame 227 for varying bulk modulus $K = 2.94e8, 2.94e9, 8.82e9$ Pa from left to right.

Figure 7: Comparison of dragon impact at frame 177 for varying radius $\delta = 2\lambda, 3\lambda, 6\lambda$ from left to right.

achieved a desired look. Our default, $K = 2.94e9$ Pa for both the vase and dragon, creates a less brittle object, similar to clay earthenware. To demonstrate the parameter's effect, we show two additional fractures of the vase for $K = 2.94e8$ and $K = 8.82e9$, a 10-fold decrease and a 3-fold increase. See Figure 6.

With less brittle material, i.e., lower K and hence weaker springs, the vase quickly crumbles into dust. The object acts almost as a fluid. With a higher K, we get more dust and larger chunks, indicative of the resistance to stretching from the stiffer springs. Note that if we increased K further, the vase would explode on impact, as the brittleness of the object would produce instabilities relative to any major collision.

Spring Radius In general, using only the 1-neighbors ($\delta = \lambda$) for the peridynamic spring network leads to unrealistic phenomena. Our springs are set to be relatively stiff compared with other uses in computer graphics, and there is no damping. External body forces cause the mesh to flex and break quickly. A larger radius of springs leads to a beneficial network large enough to provide non-local effects and rigidity to the system.

Figure 7 shows a comparison of the dragon mesh shortly after impact when using two radii, $\delta = 2\lambda$ and $\delta = 3\lambda$, that are smaller than the nominal radius of 6λ used in Figure 3. These radius values correspond to a maximum number of bonds between masses of 24 and 72, compared with the 510 used at $\delta = 6\lambda$. The network with the fewest number of springs disintegrates upon contact with the floor, slowing the eventual rotation of the object. While the value of 3λ does lead to some larger chunks, there is also a large cloud of dust emerging.

Figure 8: Comparison of vase impact at frame 190 with strain limits $\tau = 0.01, 0.005, 0.001, 0.0005, 0.0002$ decreasing from left to right.

Strain Limits Varying the strain limit, τ, is similar to varying the toughness parameter used in FEM-based fracture [OH99]. Using the formula defined in Eq.5, a reasonable value of τ for a glass object would be 0.0005 [SA04]. Nevertheless, for the dragon and vase we wanted to reduce fragility, and so we varied this threshold to produce a variety of effects shown in Figure 8. In particular, at the highest value, the vase simply bounces off the floor, whereas each decreasing value causes less bounce and more fragile crumble. At the lowest value, the impact with the floor causes the entire vase to crumble almost instantaneously.

7. Discussion

Undoubtedly, we are not the first computer graphics researchers to use spring-mass systems to animate fracture, to connect springs over long distances, or to base spring forces on a strain metric. Nevertheless, motivated by peridynamics, we are the first to combine these techniques and show their viability for animating brittle fracture in computer graphics. The power of our approach is its simplicity. Discontinuities arising from fracture are trivially handled and the method is highly amenable to GPU implementations.

Limitations and Future Work Foremost among limitations is the difficulty of generating geometry for rendering from the underlying point based simulation data. This challenge is not unique to our approach but plagues all point-based approaches. Nevertheless, while particle skinning has been studied in the case of smooth surfaces, very little work has been done for generating surfaces with sharp features from particle data. Using Procrustes superposition can lead to minor popping artifacts when large chunks are connected by only a few remaining springs and those springs break, leading to a sudden change in the Procrustes transform. These artifacts could be addressed by smoothly transitioning to the new transforms using a method analagous to a proportional-derivative controller.

Second, the simplicity of spring-mass systems does come at a cost—it is far more difficult to handle arbitrary material models. Indeed, our approach is limited to a single Poisson ratio, 0.25. However, recent work in peridynamics promises to circumvent this restriction [ELP13]. Moreover, we are currently limited to modeling brittle fracture. While some ductility has been shown to be important for animation of

fracture [OBH02], inclusion of a volume-preserving plasticity model in a spring-mass framework is non-trivial.

Third, as in most physics-based animation methods, tuning parameters can be a tedious task. While a variety of phenomena can be achieved, the parameters, K, K_c, K_S, δ, τ, α, and ρ (mass density, we use 2200 kg/m^3 in our experiments), all must be tuned. Fortunately, some of these can be set to physically measured quantities, while others have recommended values from the peridynamics community [SA04].

Our force model for collision is approximate, and can lead to missed collisions when small, fast moving elements penetrate through each other. This situation is avoided, in general, by using longer springs and having the object break into fewer, larger chunks.

While peridynamics makes no explicit assumptions about the particle sampling, the relationship between sampling density and the simulation quality remains unexplored. We have explored both regular and irregular sampling techniques, but in general we have maintained uniform density requirements. There is an expectation that in the presence of nonuniformity, the appropriate value of N might also have to vary. Both of these design choices currently have an intimate relationship with debris size as well as the noise/graininess of the resulting solids. Understanding these relationships further would benefit the model, as would modifying the rendering stage to take advantage of more complex sampling techniques. Moreover, the level of detail in our fracture patterns is tied to the initial particle sampling. Adapting the particle sampling to better resolve cracks and other simulation features could dramatically improve the results.

An important area of future work involves doing a full comparison to both other meshless methods as well as finite element techniques. In particular, it would be interesting to better understand the differences between our approach and FEM. This comparison could be used to evaluate both the computational differences as well as different definitions of strain and their efficacy under various sampling strategies.

Finally, our current implementation uses explicit integration. While this choice admits straightforward and highly parallel implementation, it does require many timesteps per frame. Taking larger timesteps would not only require implicit integration, but also new techniques to propagate fracture over a single timestep and a different approach to collision detection. Alternatively, our simple explicit scheme could be improved by employing asynchronous integration similar to Harmon et al. [HVS*09]. Addressing these limitations and improving the performance are promising directions for future work.

Acknowledgments

We would like to thank the US National Science Foundation for its support under awards CNS-1126344, IIS-1314757, and IIS-1314896 as well as Toyota Racing Development for its support of our research programs. We would also like to thank Jacob Hinkle of NREL and Leonhard Gruenschloss of Weta Digital Ltd. for their insightful comments.

References

[BDW13] BUSARYEV O., DEY T. K., WANG H.: Adaptive fracture simulation of multi-layered thin plates. *ACM Trans. Graph. 32*, 4 (2013), 52. 2

[BGB11] BHATTACHARYA H., GAO Y., BARGTEIL A. W.: A level-set method for skinning animated particle data. In *Symposium on Computer Animation* (2011), pp. 17–24. 2

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (1982), 235–256. 2

[CGW*13] CHEN J., GE X., WEI L.-Y., WANG B., WANG Y., WANG H., FEI Y., QIAN K.-L., YONG J.-H., WANG W.: Bilateral blue noise sampling. *ACM Trans. Graph. 32*, 6 (2013), 216. 4

[ELP13] EMMRICH E., LEHOUCQ R. B., PUHST D.: Peridynamics: A nonlocal continuum theory. In *Meshfree Methods for Partial Differential Equations VI*, vol. 89. Springer Berlin Heidelberg, 2013, pp. 45–65. 1, 3, 7

[GMD13] GLONDU L., MARCHAL M., DUMONT G.: Real-time simulation of brittle fracture using modal analysis. *IEEE Trans. Vis. Comput. Graph. 19*, 2 (2013), 201–209. 2

[HDR13] HDRI HUB: HDR parking lot 2, 2013. URL: http://www.hdri-hub.com/hdrishop/freesamples. 6

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 167–174. 5

[HTK98] HIROTA K., TANOUE Y., KANEKO T.: Generation of crack patterns with a physical model. *The Visual Computer 14*, 3 (1998), 126–137. 3

[HTK00] HIROTA K., TANOUE Y., KANEKO T.: Simulation of three-dimensional cracks. *The Visual Computer 16*, 7 (2000), 371–378. 3

[HVS*09] HARMON D., VOUGA E., SMITH B., TAMSTORF R., GRINSPUN E.: Asynchronous contact mechanics. *ACM Trans. Graph. 28*, 3 (2009). 8

[IMP*13] IBEN H., MEYER M., PETROVIC L., SOARES O., ANDERSON J., WITKIN A.: Artistic simulation of curly hair. In *Symposium on Computer Animation* (2013), pp. 63–71. 3

[Kab78] KABSCH W.: A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A 34*, 5 (Sep 1978), 827–828. 5

[KMB*09] KAUFMANN P., MARTIN S., BOTSCH M., GRINSPUN E., GROSS M.: Enrichment textures for detailed cutting of shells. *ACM Trans. Graph. 28*, 3 (2009), 50:1–50:10. 2

[LBOK13] LIU T., BARGTEIL A. W., O'BRIEN J. F., KAVAN L.: Fast simulation of mass-spring systems. *ACM Trans. Graph. 32*, 6 (Nov. 2013), 209:1–7. 3

[LSH07] LLOYD B., SAKELY G., HARDERS M.: Identification of spring parameters for deformable object simulation. *IEEE Transactions on Visualization and Computer Graphics 13*, 5 (2007), 1081–1094. 3

[MBF04] MOLINO N., BAO Z., FEDKIW R.: A virtual node algorithm for changing mesh topology during simulation. *ACM Trans. Graph. 23*, 3 (2004), 385–392. 2

[MCK13] MÜLLER M., CHENTANEZ N., KIM T.-Y.: Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph. 32*, 4 (2013), 115. 3

[MG04] MÜLLER M., GROSS M. H.: Interactive virtual materials. In *Graphics Interface* (2004), vol. 62, pp. 239–246. 2, 4

[MGDA04] MARTINET A., GALIN E., DESBENOIT B., AKKOUCHE S.: Procedural modeling of cracks and fractures. In *Shape Modeling International* (2004), pp. 346–349. 3

[MKW07] MEYER M. D., KIRBY R. M., WHITAKER R. T.: Topology, accuracy, and quality of isosurface meshes using dynamic particles. *IEEE Trans. Vis. Comput. Graph. 13*, 6 (2007), 1704–1711. 4

[NCc10] NATSUPAKPONG S., CENK ÇAVUŞOĞLU M.: Determination of elasticity parameters in lumped element (mass-spring) models of deformable objects. *Graph. Models 72*, 6 (Nov. 2010), 61–73. 3

[NTB*91] NORTON A., TURK G., BACON R., GERTH J., SWEENEY P.: Animation of fracture by physical modeling. *The Visual Computer 7*, 4 (1991), 210–219. 1, 2

[OBH02] O'BRIEN J. F., BARGTEIL A. W., HODGINS J. K.: Graphical modeling and animation of ductile fracture. *ACM Trans. Graph. 21*, 3 (2002), 291–294. 2, 8

[OH99] O'BRIEN J. F., HODGINS J. K.: Graphical modeling and animation of brittle fracture. In *SIGGRAPH* (1999), pp. 137–146. 2, 7

[PKA*05] PAULY M., KEISER R., ADAMS B., DUTRÉ P., GROSS M. H., GUIBAS L. J.: Meshless animation of fracturing solids. *ACM Trans. Graph. 24*, 3 (2005), 957–964. 2

[PO09] PARKER E. G., O'BRIEN J. F.: Real-time deformation and fracture in a game environment. In *Symposium on Computer Animation* (2009), pp. 165–175. 2, 4

[Rag02] RAGHAVACHARY S.: Fracture generation on polygonal meshes using voronoi polygons. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 187–187. 3

[RH01] RAMAMOORTHI R., HANRAHAN P.: An efficient representation for irradiance environment maps. In *SIGGRAPH* (2001), pp. 497–500. 6

[SA04] SILLING S., ASKARI E.: Peridynamic modeling of impact damage. In *ASME Conference Proceedings* (2004), pp. 197–205. 2, 6, 7, 8

[SA05] SILLING S., ASKARI E.: A meshfree method based on the peridynamic model of solid mechanics. *Computers & Structures 83*, 17âĂŞ18 (2005), 1526–1535. Advances in Meshfree Methods. 2, 4

[SABW82] SWOPE W. C., ANDERSEN H. C., BERENS P. H., WILSON K. R.: A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics 76*, 1 (1982), 637–649. 3

[Sil00] SILLING S.: Reformulation of elasticity theory for discontinuities and long-range forces. *Journal of the Mechanics and Physics of Solids 48*, 1 (2000), 175 – 209. 1, 3

[SLF08] SELLE A., LENTINE M., FEDKIW R.: A mass spring model for hair simulation. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 64:1–64:11. 3

[SO14] SCHVARTZMAN S. C., OTADUY M. A.: Fracture animation based on high-dimensional voronoi diagrams. In *Symposium on Interactive 3D Graphics and Games* (2014). 3

[SWB01] SMITH J., WITKIN A. P., BARAFF D.: Fast and controllable simulation of the shattering of brittle objects. *Comput. Graph. Forum 20*, 2 (2001), 81–90. 3

[Tex13] TEXTUREX: red grain wooden panel, 2013. URL: http://www.texturex.com. 6

[TF88] TERZOPOULOS D., FLEISCHER K. W.: Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *SIGGRAPH* (1988), pp. 269–278. 1, 2

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANTES D., GROSS M. H.: Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the Vision, Modeling, and Visualization Conference* (2003), pp. 47–54. 3

[TKA10] TWIGG C. D., KAČIĆ-ALESIĆ Z.: Point cloud glue: Constraining simulations using the procrustes transform. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), pp. 45–54. 5

[YT13] YU J., TURK G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Trans. Graph. 32*, 1 (Feb. 2013), 5:1–5:12. 2

[YWLL13] YAN D.-M., WANG W., LÉVY B., LIU Y.: Efficient computation of clipped voronoi diagram for mesh generation. *Computer-Aided Design 45*, 4 (2013), 843–852. 4