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

Efficient Denting and Bending of Rigid Bodies

Saket Patkar[1†] Mridul Aanjaneya[2‡] Aric Bartle[1†] Minjae Lee[1†] Ronald Fedkiw[13†]

[1]Stanford University [2]University of Wisconsin - Madison [3]Industrial Light + Magic

Abstract

We present a novel method for the efficient denting and bending of rigid bodies without the need for expensive finite element simulations. Denting is achieved by deforming the triangulated surface of the target body based on a dent map computed on-the-fly from the projectile body using a Z-buffer algorithm with varying degrees of smoothing. Our method accounts for the angle of impact, is applicable to arbitrary shapes, readily scales to thousands of rigid bodies, is amenable to artist control, and also works well in combination with prescoring algorithms for fracture. Bending is addressed by augmenting a rigid body with an articulated skeleton which is used to drive skinning weights for the bending deformation. The articulated skeleton is simulated to include the effects of both elasticity and plasticity. Furthermore, we allow joints to be added dynamically so that bending can occur in a non-predetermined way and/or as dictated by the artist. Conversely, we present an articulation condensation method that greatly simplifies large unneeded branches and chains on-the-fly for increased efficiency.

1. Introduction

Realistic deformation and destruction of objects has become increasingly popular in modern day computer games providing immersive gameplay and visual feedback to the player's actions. Such effects can be accurately simulated using computationally intensive algorithms such as finite element methods [BW97], however, rigid objects require high stiffness and thus either more expensive implicit time discretization or a stringent explicit time step restriction. For efficiency, researchers have designed algorithms that use such methods only on a limited basis [MMDJ, PPG04, BHTF07] and otherwise use faster rigid body solvers for the bulk of the simulation. Although efficient finite element algorithms have also been designed for real-time systems [PO09, DBB11], it is often preferable to make the simulation as fast as possible freeing up resources for other tasks. More importantly, finite element simulations are not easy for the artist to control. In light of the above, [SSF09] proposed a prescoring algorithm for frame rate rigid body fracture which was optimized for real-time systems in [MCK13]. However, large scale denting effects are still missing from today's games and are only available for very specific scenarios such as bullet impacts using either precomputed textures or parallax mapping [Wes06].

We dent rigid bodies by deforming the triangulated surface of the target body using a dent map that is computed on-the-fly from the projectile body via a Z-buffer algorithm with varying degrees of smoothing. Our method has similarities with [SSF09] where a prescoring algorithm was used for fracture, and realistic fractures were obtained by centering the prescored fracture pattern at the point of impact. Similar to prescoring the entire space, our method computes a dent map for the entire space that is based on the projectile body's shape, orientation, and angle of impact. Our method allows for the efficient and controllable denting of arbitrary meshes, readily scales to thousands of rigid bodies (see Figure 10) and also works well in combination with prescoring algorithms for fracture. Moreover, it can be easily integrated into most rigid body simulation frameworks as a postprocess at the end of each time step.

Unlike denting which is a local change to the rigid body mesh, bending is more typically a global deformation and hence, is not easily achieved by using displacement maps, proxy meshes, etc. Thus, we propose a skeleton-based approach to bending where the skeleton can either be specified by an artist or computed using medial axes [Dey06, HWCO*13, MC14]. We use linear blend skinning for updating the rigid body's triangulated surface mesh, although other methods can also be used [KCvO08, DdL13, VBG*13]. Whereas bending is usually considered to be a plastic deformation, our method can also handle short term elastic behavior. Although various authors have considered two-way cou-

† e-mail: {patkar|abartle|mjlgg|rfedkiw}@stanford.edu
‡ e-mail: aanjneya@cs.wisc.edu

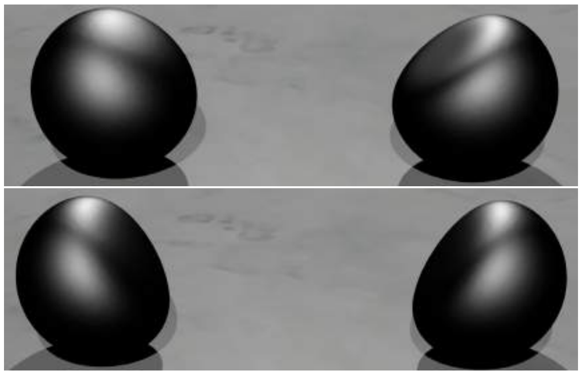
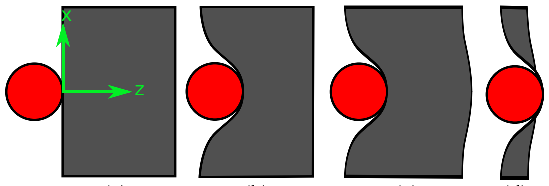


Figure 1: (Top) Denting in a Gauss-Seidel manner results in asymmetry. (Bottom) Symmetric denting can be achieved by using a Jacobi-style algorithm.

pling between articulated skeletons and deformable bodies, see e.g. [SSF08, KP11, LGS*11, TGTL11, LYWG13] (see also [JL11, MZS*11]), our method appears to be the first to use an articulated skeleton for defining the shape of a rigid body, and we obtain visually pleasing bends even with very crude skeletons. For added realism, we also allow for the dynamic augmentation of the skeleton with new joints whenever unforeseen impacts/collisions occur. For efficiency, we propose a condensation mechanism to dynamically collapse sub-bodies into a single rigid body cluster thereby simplifying large articulated branches and chains. Note that our method is not physically-based but still produces results that are comparable to finite element simulations; in fact linear finite element methods can have visual artifacts such as linearized rotations causing a plank to become thicker near its ends after bending (see [BHTF07]) - which our method avoids.

2. Rigid Body Simulation

We use the method of [GBF03] as our base solver, although our method is general enough to be incorporated into most rigid body simulation frameworks. Each rigid body has both a triangulated surface and a level set representation. We use



(a) (b) (c) (d)

Figure 2: a) A ball hitting a block in two spatial dimensions. The x and z axes represent the local placement of the coordinate system of the displacement map $\mathcal{D}$, b) A dent map applied to the collision surface points only, c) A dent map applied to all the points so that the back of the block bulges outwards. d) A thin wall gets inverted because of a very sharp attenuation function.

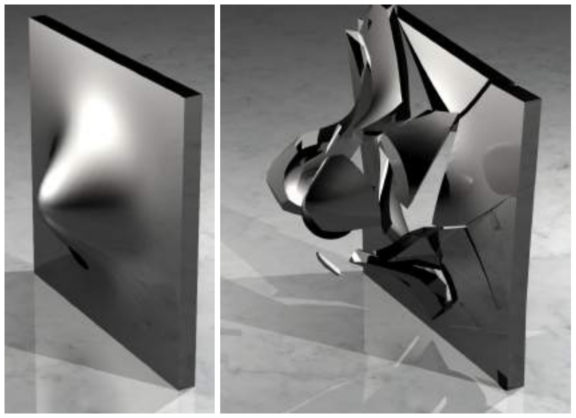


Figure 3: Ball fracturing a wall illustrating our denting algorithm in combination with a prescored fracture.

the second order accurate integration scheme from [SSF08] to explicitly update the position and velocity. For simulating articulated rigid bodies, we use the maximal coordinate framework of [WTF06], enforce the joint constraints with impulse-based prestabilization, and enforce linear and angular velocity constraints with poststabilization. Due to its impulse-based nature, the method can be easily integrated with most collision and contact algorithms for simulating rigid bodies.

3. Denting

Denting can be viewed as applying a displacement map $\mathcal{D}$ to every point in three dimensional space, $\mathcal{D} : (x,y,z) \rightarrow (\Delta x, \Delta y, \Delta z)$. Given such a map, one can center its coordinate axes at the point of impact and transform the points on the target body creating a dent. After denting the triangulated surface, other auxiliary rigid body information such as level sets and bounding boxes are updated. The projectile rigid body can be treated in several different ways: it can be moved into the space created by the dent, its velocity can be modified or left unaffected, it may also dent, etc. When the projectile rigid body is also denting, we use the predented information for each body when denting the other body in order to obtain symmetric results (see Figure 1).

3.1. Dent maps

Storing the displacement map $\mathcal{D}$ can be computationally inefficient requiring a huge memory footprint. Thus, we in-

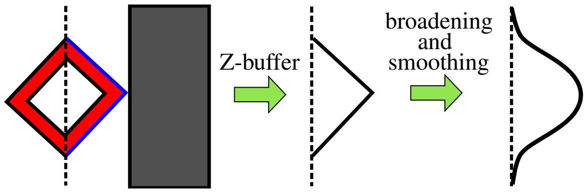


Figure 4: (Left) The dotted line shows the image plane used for the Z-buffer algorithm. (Middle) The profile of the projectile body causing the dent. (Right) Smoothed version of the profile.

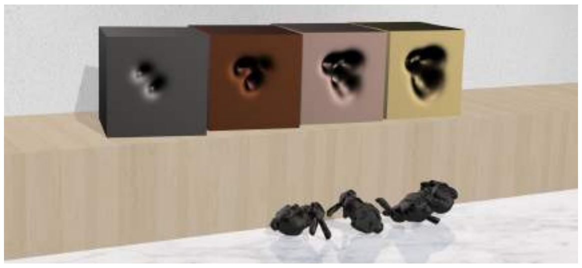
stead store a two dimensional *dent map* $\hat{\mathcal{D}} : (x,y,z=0) \rightarrow (\Delta x, \Delta y, \Delta z)$ which can be specified analytically, generated procedurally, created by an artist, taken from finite element simulations, etc. Consider a simple two dimensional example where a ball hits a block as shown in Figure 2. The ball leaves a dent in the block at the point of impact, and for this special case an analytic dent map with a Gaussian dent function is a suitable choice for a plausible dent. We place a coordinate system with its origin at the point of impact and z-axis along the direction of the inward normal of the body being dented (we refer to this inward normal as the *denting normal*). The dent map stores a displacement for each point on the x-axis. In this case a Gaussian is stored as a function of x and the width of the Gaussian is chosen as a function of the sphere's radius. The amplitude of the Gaussian, i.e. $|\hat{\mathcal{D}}(\vec{0})|$, can either be specified as a user-defined input parameter or computed based on the impact velocities during collision. We refer to this amplitude $|\hat{\mathcal{D}}(\vec{0})|$ as the *dent distance*. Applying the two dimensional dent map only moves the points on the collision plane as shown in Figure 2b. Additional realism can be obtained by including an attenuation function, for example, by specifying $\mathcal{D}(x,y,z) = a(z)\hat{\mathcal{D}}(x,y)$, where $a(0) = 1$ and $a(z)$ approaches zero for $z > 0$. In Figure 2c we chose $a(z) = 2/(1+e^{z/|\hat{\mathcal{D}}(\vec{0})|})$ and note that the deformation on the back side of the surface is an attenuated version of that on the colliding front side. One must be careful when choosing $a(z)$ such that the falloff is not so steep that points on the front surface cross over those on the back surface (see Figure 2d). If the falloff is less steep than $f(z) = 1 - z/|\hat{\mathcal{D}}(\vec{0})|$, then the dent map will not cause the object to self-intersect. A more steep falloff can be chosen to get a certain effect as long as the object is not too thin.

3.2. Arbitrarily shaped projectiles

Analytic dent maps can give implausible results for projectiles with more complex shapes. Although artists could hand craft object-specific dent maps, it can be cumbersome especially because the dent map depends on the orientation of the object as shown in Figure 4. To remedy this, we propose a novel algorithm to generate these dent maps procedurally. First we move the collision plane in the direction opposite the denting normal by the dent distance. Then we use a standard Z-buffer algorithm to compute the profile of the projectile that impacts the target (the maximum Z-depth from the plane towards the target is used). The size of the two dimensional grid is chosen such that it is large enough to have the dent map taper off to zero near the boundaries. The resolution of this grid is chosen based on the mesh resolutions of the target body and the projectile body to ensure that all small scale features of the projectile body are captured in the dent. Note that our algorithm readily handles non-convex shapes such as the bunny ears in Figure 5.

Using the profile generated by the Z-buffer algorithm alone may generate dents that are too crisp as shown in Figure 5b (far left). Thus, we smooth the results of the Z-buffer algorithm as shown in Figure 4 (far right), Figure 5b, and Figure 7. As an example consider Figure 6. The black curve is the profile generated by the Z-buffer algorithm using the uniform one dimensional grid that stores the dent map. To obtain a broader dent function, we place a Gaussian function of height z_i at location x_i for each point (x_i, z_i) on the black curve. The width of the Gaussian is computed based on the desired blurring of the final map. Then we compute the upper envelope of these Gaussians (shown in red). This upper envelope can have sharp corners when Gaussians from two local maxima intersect each other. To remove these and obtain the green curve, we use few iterations of an optimized heat kernel/convolution operator in an explicit fashion (many efficient GPU implementations exist). This removes the noise in the dent map near the peaks. Then we rescale the dent map such that the dent distance remains the same as before and again run few iterations of a modified heat kernel/convolution operator, but this time clamping the result so that Z-values are only allowed to increase (not decrease). This removes the sharp creases giving a smooth result. See Figure 7. Note that the broadening algorithm is crucial for producing realistic dents. Directly smoothing the Z-buffer profile using the heat equation will cause inter-penetrations when the projectile body is moved inside the target body, while using the modified heat equation on the Z-buffer profile retains the sharp corners (see the blue curve in Figure 6).

As discussed above, the algorithm requires one to compute the upper envelope of Gaussian functions defined on a two dimensional grid. We accomplish this efficiently using a technique from computational geometry (see for ex-


(a) (Left to right) shows the effect of increasing the dent distance.

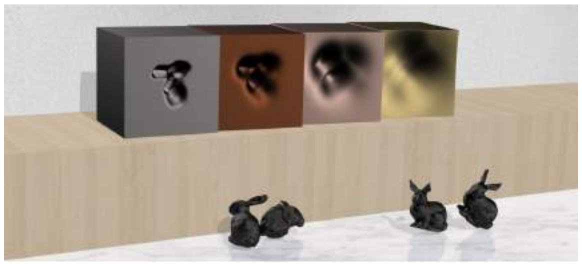

(b) (Left to right) shows the effect of increasing smoothing.

Figure 5: Different parameter values for the dent distance or Gaussian width allows us to obtain the illusion of simulating different material properties.

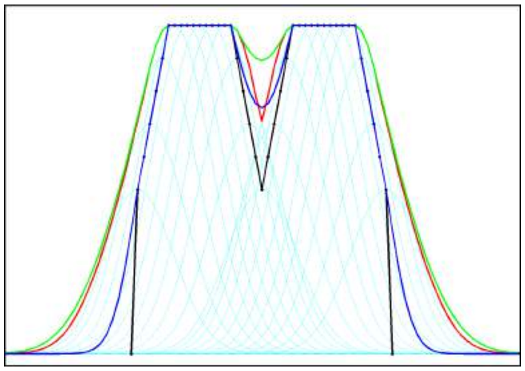


Figure 6: (Black) Result of the Z-buffer algorithm. (Cyan) Gaussians of height z_i placed at x_i for every point (x_i, z_i). (Red) Upper envelope of the Gaussians. (Green) Final result obtained by heat equation smoothing of the red curve. (Blue) Result obtained after running the modified heat equation directly on the black curve.

ample [dBCvKO08]). It is well-known that the upper envelope of planes in three spatial dimensions can be found in $O(N \log N)$ time by mapping the planes to points in the dual space and taking the lower convex hull of the dual points. Thus, given a set of Gaussians over the xy-plane defined as $z = z_i \exp\left(-c((x-x_i)^2 + (y-y_i)^2)\right)$, we compute their upper envelope by first mapping them to a set of planes with the same upper envelope in the following fashion: first take the logarithm of each Gaussian to obtain $\log z = \log z_i - c((x-x_i)^2 + (y-y_i)^2)$, then subtract off $-c(x^2+y^2)$ to obtain $\log z = 2cx_i x + 2cy_i y + (\log z_i - cx_i^2 - cy_i^2)$ which represents a plane in the coordinates $(x, y, \log z)$. Given these planes, we compute the convex hull of their dual points in $O(N \log N)$ time. Every edge in the lower convex hull corresponds to an intersection between two Gaussians that belong to the upper envelope. Note that two Gaussians with the same width intersect along a plane that projects to a line on the collision plane. Given a Gaussian, we find the corresponding point on the lower convex hull and compute the polygonal region in which it is the uppermost Gaussian by intersecting it with Gaussians corresponding to neighboring points. This gives us a polygonal partitioning of the two dimensional collision plane, which we triangulate to obtain a mapping from triangles to Gaussians (see Figure 7d). We rasterize each triangle onto the grid used by the Z-buffer algorithm and update the values of all grid cells (pixels) that lie inside that triangle with the value of the corresponding Gaussian at those cell centers. Note that this construction requires that all Gaussians have the same width c, which is fine for our application.

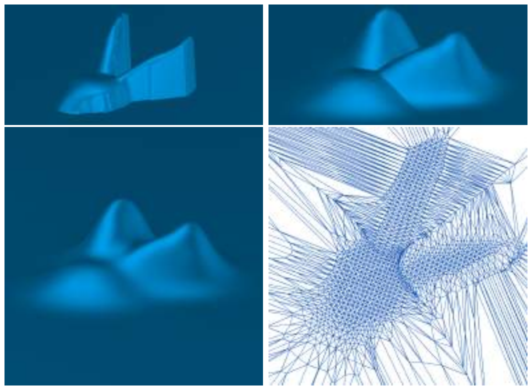


Figure 7: (Top left) Result of the Z-buffer algorithm for the bunny from Figure 5. (Top right) Result obtained by computing the upper envelope of the Gaussians - notice the sharp boundaries. (Bottom left) Final dent map after smoothing the upper envelope with the heat equation. (Bottom right) Partitioning of the collision plane based on the upper envelopes. Each triangle is uniquely associated with a Gaussian that is the uppermost Gaussian in that region.

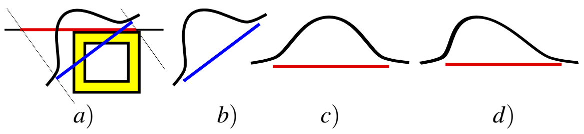


Figure 8: a) Ball hitting an object at an angle θ. b) Dent map along the relative velocity direction c) Scaling the dent map by $1/\cos(\theta)$ to account for the oblique projection and rotating to align it along the collision plane. d) Shearing the dent map while maintaining its base to orient it along the relative velocity direction.

3.3. Glancing impacts

The aforementioned discussion is valid when a projectile rigid body impacts a target rigid body head on, however, it requires modifications when the projectile impacts the target at an angle. Assuming the angle between the velocity of the

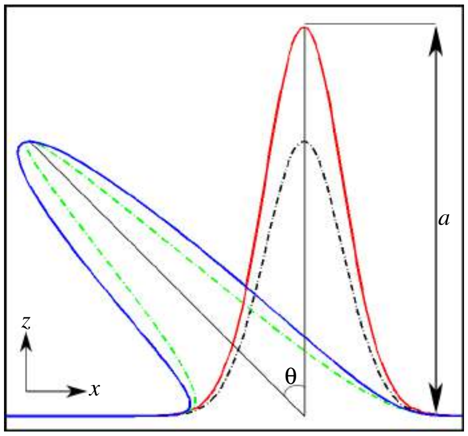


Figure 9: ***Shearing the dent map.*** (Red) Dent map for a ball hitting head on. (Black) Scaled dent map. (Green) Sheared dent map generated from the black curve corresponding to impacting at an angle. (Blue) Slightly widened version of the green curve that better matches the width of the red curve.

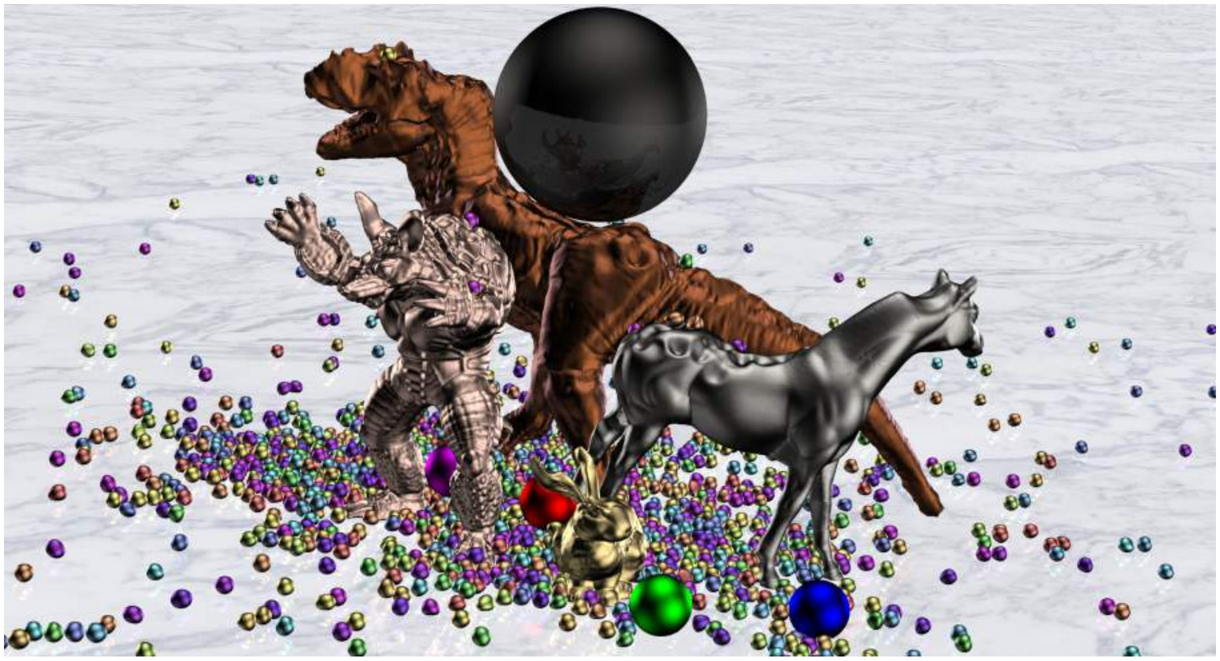


Figure 10: An armadillo, a bunny, a horse and a dinosaur being dented by a large number of spheres.

projectile and the denting normal is θ, we first use our Z-buffer algorithm along the direction perpendicular to the relative velocity as shown by the blue line in Figures 8a and 8b. Then we rotate the Z-buffer grid and scale it by $1/\cos(\theta)$ to align it with the collision plane - the scaling accounts for the oblique projection and widens the base of the dent map (see Figure 8c). Note that we clamp the denominator to avoid division by zero, although such cases are rare because the relative velocity along the normal direction is small for large values of θ. Finally, to obtain the result shown in Figure 8d, we shear the dent map by an angle θ as follows. For simplicity of exposition, consider a one dimensional dent map $\hat{\mathcal{D}}(x, z=0)$ as shown in red in Figure 9. Since the displacement is only along the denting normal, the dent map $\hat{\mathcal{D}}$ can be specified as $\hat{\mathcal{D}}(x) = (0, \hat{\mathcal{D}}_2(x))$. First we scale all points $(x, \hat{\mathcal{D}}_2(x))$ on the red curve by a factor $s < 1$ to obtain the black curve with points $(x, s\hat{\mathcal{D}}_2(x))$. Then we shear the points on the black curve by an angle θ along the x-axis to obtain the green curve with points $(x - \tan(\theta)s\hat{\mathcal{D}}_2(x), s\hat{\mathcal{D}}_2(x))$. The origin gets transformed to $(-\tan(\theta)sa, sa)$ on the green curve, where a is the original dent distance. Choosing s to keep the dent distance invariant implies that $\sec(\theta)as = a$ or $s = \cos(\theta)$. In summary, the green curve is computed by the transformation $(x,0) \to (x - \sin(\theta)\hat{\mathcal{D}}_2(x), \cos(\theta)\hat{\mathcal{D}}_2(x))$. Although the green curve has the correct angle and dent distance, it does not have the correct width - the width of the green curve reduces as the angle increases. The curve shown in blue can be obtained by the transformation $((x - 2\sin(\theta)\hat{\mathcal{D}}_2(x))/(1+\exp(tx)), \cos(\theta)\hat{\mathcal{D}}_2(x))$, where $t \geq 0$. Note that setting $t = 0$ gives back the green curve.

3.4. Timing

Table 1 shows the timing information on a single CPU for the different steps of our denting algorithm as the two-dimensional grid storing the dent map is refined. For all our examples, the resolution of this grid was between 50×50 and 200×200 depending on the mesh resolutions of both the projectile body and the target body. Note that the broadening and smoothing steps only depend on the grid resolution.

Grid		Broadening		
Resolution	Z-buffer	Ours	$O(N^2)$	Smoothing
25×25	.0086	.0053	.0148	$1.5e^{-4}$
50×50	.0094	.0118	.2392	$5.9e^{-4}$
100×100	.0104	.0696	3.949	.0025
200×200	.0125	.6643	65.88	.0105
400×400	.0178	8.031	1057	.0438

Table 1: Timing (in seconds) for the example from Figure 5a (far right) under grid refinement. The fourth column corresponds to the brute-force $O(N^2)$ approach for computing the upper envelope of a set of Gaussian functions.

As can be seen from Figure 11 a grid resolution of 25×25 gives results that are quite acceptable for a video game setting while grids of resolution 100×100 and higher give results that are completely free of artifacts. Note that a GPU implementation of the Z-buffer and convex hull algorithms (which form the major components of our broadening algorithm) can provide more than an order of magnitude speedup allowing one to create tens of dents in a real time environment (see for e.g. [TZTM12]). Finally, although we do use the expensive, albeit more accurate, level set method for sim-

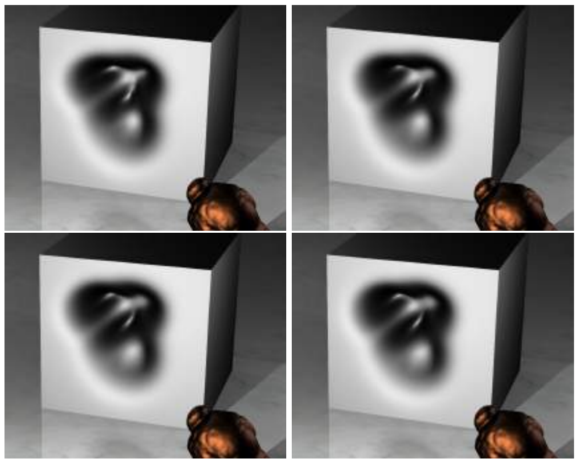


Figure 11: Bunny hitting a block with a grid of resolution (top left) 20×20, (top right) 50×50, (bottom left) 100×100, and (bottom right) 200×200 for the dent map.

ulating rigid bodies, one could certainly use a faster rigid body solver for video games as our algorithm can work with any black box rigid body solver.

Since our method operates only on the surface mesh, our work shares similarities with the recent work of [RJ07, DBB11, MC11]. Being more focused on denting effects, our algorithm does not require the storage of velocities for every mesh point and is local by design. However, it is more limited in scope as it cannot be used to obtain elastic effects such as squish, squeeze or twist - our main goal was to design a method that can easily be integrated in any rigid body solver and allows bodies to dent.

3.5. Results

Algorithm 1 gives the pseudo code for our denting algorithm. As is evident, denting is performed completely as a post-process and only requires some auxiliary collision information from the actual rigid body simulation. Note that step 21 can be done asynchronously and is very specific to level set based rigid body solvers.

Figure 3 shows a prescored wall being dented and subsequently fractured by a ball demonstrating that our method can be easily integrated with a prescoring algorithm for fracture. In this example we dented the wall over time instead of creating a single dent. To achieve this, we repeatedly used the predented state with increasing dent distances. Figure 5 demonstrates the ability of our method to mimic different material properties by changing the dent distance and Gaussian width. Figure 10 shows an armadillo, a bunny, a horse and a dinosaur being dented by 1500 small spheres, four medium sized spheres and three large spheres, which are themselves dentable, illustrating the scalability and robustness of our denting algorithm. Our method captures both the large deformation undergone by the dinosaur's back and the small scale details on the horse. The reader may observe that our method suffers from a noticeable amount of volume loss when the projectile body as well as the denting distance are large, as can be seen when the big sphere hits the dinosaur's back. We leave addressing this issue as interesting future work.

Algorithm 1 **Denting Pseudo code**

```
 1: function DENT(target $A$, projectile $B$)
 2:     Compute denting distance based on relative velocity.
 3:     Decide the resolution and size of the grid.
 4:     Compute the Z-buffer profile of B on the grid.
 5:     Broaden the Z-buffer profile using Gaussians.
 6:     Smooth the profile using the modified heat equation.
 7:     Shear the dent map to account for glancing impact.
 8:     Apply the dent map to $A$'s surface mesh.
 9: for every time step do
10:     Advance the rigid bodies by one time step. Store col-
11:     lision location, relative velocity, and collision nor-
12:     mal for every collision pair.
13:     for every collision pair (A,B) do
14:         if relative normal velocity $>$ threshold then
15:             if A is dentable then
16:                 DENT(A,B) and possibly move B to
17:                 make the dent look more plausible
18:             if B is dentable then
19:                 DENT(B,A) and possibly move A to
20:                 make the dent look more plausible
21:     Update surfaces and level sets for all dented bodies.
```

4. Bending

To model bending, we augment each rigid body with an articulated skeleton. The rigid body is then divided into *sub-bodies*, and each sub-body is associated with an articulated bone in the skeleton. For each bone, its triangulated surface is computed by assigning to it all triangles whose maximum skinning weight is associated with that bone. See for example, Figure 12 (top right) which shows a bunny with four articulated bones and (bottom left) which shows the triangulated surface associated with each bone. Note that the triangulated surfaces are open.

Similarly, we associate a level set with each articulated bone by assigning each voxel of the original rigid body's level set to one of the sub-body's level sets. The values of the level set function remain unchanged and we stress that the level set will not be a valid signed distance function in the regions where the triangulated surface has holes. We use a simple marching/coloring algorithm that first assigns all voxels adjacent to and closest to the triangles that have already been associated with the bones. Then we march inwards one voxel one ring at a time assigning those voxels as well, using a simple tie-breaker when necessary. The same process is carried out for the region outside the object where the level set values are positive, so that the whole grid is decomposed into subsections assigned to each articulated bone. In this fashion, a rigid body with N articulated bones will now have N

separate level set grids with level set values defined only on a subset of that grid. One can fill the empty cells by choosing a small positive number since we do not require valid level set data at the seams between sub-bodies. For added efficiency, one could prune this to a tighter fitting bounding box either by deleting grid cells far from the bone or by resampling to a different set of cell centers. Obviously, if needed, one could compute an accurate level set and even close the triangulated surfaces but we found this unnecessary for our purposes. Figure 12 (bottom right) shows the level sets associated with each of the four bones in the bunny's skeleton. At this point the level sets can be used to compute the masses, inertia tensors, velocities, angular velocities, etc., of the sub-bodies as in [SSF09].

The aforementioned process computes all information necessary in order to simulate the original rigid body as an articulated rigid body. However, significant changes in the joint angles will expose the seams in between the sub-bodies where the geometric information is not valid. Therefore, whenever necessary we update the rigid body's triangulated surface using the skinning weights, recompute a new level set for the deformed rigid body, and redivide the rigid body into sub-bodies as described above. This can be done asynchronously, i.e. if the CPU is being used for simulation then the skinning, level set construction, and segmentation into sub-bodies can be done separately on a GPU or other processors/cores - after which the simulated hybrid structure can be updated on-the-fly whenever these computations have finished. This updated information could be a bit out-of-date with the current state of the articulated rigid body, however, it is a better approximation than the one that was being used - the goal is simply to reduce the lag, although some amount of lag is readily tolerated. Note that these seams never appear in the final renderings at frame boundaries as the parent rigid body's mesh is updated/rendered using the skinning weights.

4.1. Plastic deformation

The joints should be unaffected by minor impacts but should bend when subjected to stronger ones. To achieve this requires modification of both the prestabilization and poststabilization steps of [WTF06]. Typically, prestabilization computes a linear and an angular impulse to maintain the joint constraints. We assume the joint is rigid and first compute only the angular impulse required to maintain the angular constraint. Then we clamp the magnitude of this impulse if it is larger than a predetermined threshold set by the user as the bending *strength*. [WTF06] applies prestabilization as an iterative process, and thus we clamp the accumulated impulse. This accumulated impulse either reaches the threshold in which case we clamp it and the joint bends, or the total impulse needed to maintain rigidity at the joint is less than the threshold in which case the collision causes no bending. Next, the linear impulse is computed and applied in standard fashion in order to maintain the linear constraint so that the articulated bones do not detach. Subsequently, we update

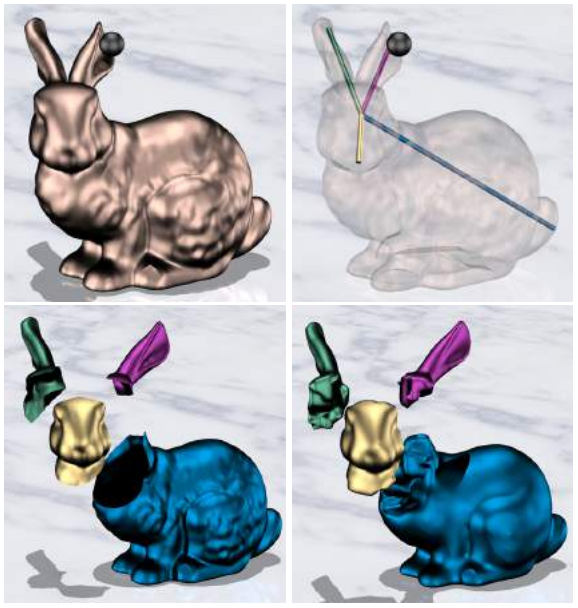


Figure 12: Different representations of the bunny including the articulated skeleton (top right), triangulated surfaces with holes (bottom left) and level sets (bottom right). Note that our algorithm works even with a very crude skeleton. Also note that the skeleton is not perfectly embedded inside the body and pops out near its neck region, which our algorithm robustly handles.

the target configuration to be the current configuration in the rigid joint - i.e., plastic deformation. Since we are treating the joints rigidly, standard poststabilization removes all relative velocities. While this is fine for the linear/prismatic velocities, we found that targeting a damped angular velocity is more realistic so that a bending joint will continue to bend somewhat for a short time duration after the collision. We found that a combination of constant and proportional damping gave good results.

4.2. Dynamic articulation

We allow for the dynamic addition of joints by taking the collision location into account. When a collision occurs with a body that has been specified to have dynamic articulation, we add a new articulation if the relative normal velocity during the collision is more than a set threshold. First we project the collision location onto the closest bone, and then break that bone into two separate bones with their own skinning weights and sub-bodies. Note that we use the rest configuration to compute the skinning weights so that the weights for the remaining bones do not change. See Figures 14 and 13.

For efficiency, we propose a condensation mechanism that dynamically collapses articulation. We do not carry out a full articulated rigid body simulation when the parent rigid body is not colliding or involved in any post-collision dynamics. Instead, we merge all sub-bodies into a single rigid body

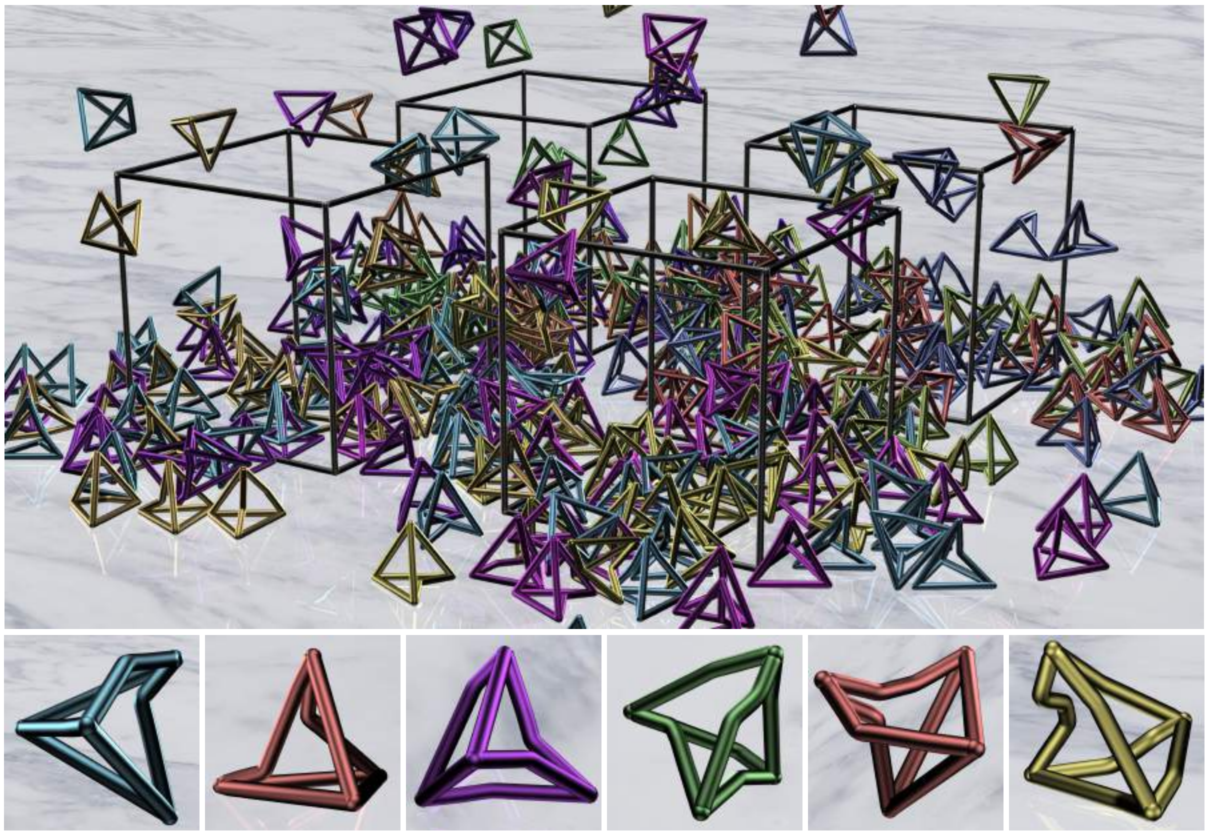


Figure 13: (Top) 400 articulated tetrahedra, each composed of ten rigid bodies (four balls and six cylinders) fall on four static cubes and bend using dynamically created articulations on the cylinders. The tetrahedra also bend each other when they collide. (Bottom) Six representative tetrahedra illustrating the variety of bending deformations that our method can achieve.

cluster with a single mass, center of mass, inertia tensor, linear and angular velocities, etc., similar to [RGL05, ELF13] (see also [BFA02]). It is inefficient to simulate the entire articulated rigid body when only one joint needs to bend. Therefore, we additionally collapse all rigid bodies on either side of the joint that is bending into single rigidified bodies so that the bending simulation can be highly optimal, dealing with only a single joint between two rigid bodies even for models with highly detailed articulation. However, in certain cases one wants to consider the bending of many joints, see for example, Figure 15.

4.3. Elasticity

To simulate elasticity, we add torsional springs in joints, where each joint is treated as a point joint and the axis of rotation is chosen such that the spring opposes the relative angular velocity. For simplicity of exposition, consider a joint with a single degree of freedom and let I_1 and I_2 be the respective inertia tensors of the bodies connected by the joint. This system can be reduced to a single mass spring system with an effective inertia tensor $I = (I_1 + I_2)^{-1} I_1 I_2$. Given a damping coefficient C and a spring constant κ, the spring equation is $I\ddot{\theta} + C\dot{\theta} + \kappa(\theta - \theta_0) = \tau_{ext}$, where θ, $\dot{\theta}$, $\ddot{\theta}$ are the relative joint angles, joint velocities, joint accelerations and θ_0 is the rest angle associated with the joint. This can be integrated analytically making it quite robust and efficient. After applying external forces such as gravity, we use the analytic solution of the spring equation to compute θ^{n+1} and ω^{n+1}. Then we apply equal and opposite angular impulses to the bodies in a manner that ensures the new joint angle is θ^{n+1} after integrating forward in time, i.e. the angular impulse is $j_\tau = I((\theta^{n+1} - \theta^n)/\Delta t - \omega^n)$. We have successfully used this approach for both angular and prismatic joints. See the video for an example.

4.4. Coupling with denting

For coupling bending and denting, we slightly modify our algorithm as follows: when the sub-body associated with any bone collides with another rigid body, instead of denting the associated triangulated surface, we dent the surface of the parent body. This requires updating the parent body to the current state. Once the parent has been dented, the sub-bodies are recomputed.

4.5. Results

Algorithm 2 gives the pseudo code for our bending algorithm. Note that steps 1 and 2 only need to be performed once before the simulation as a pre-process. Step 4 is only required if there are a large number of bendable bodies. Steps 8, 10 and 11 are optional and only required if one needs to perform dynamic articulation. Note that in step 10 the skinning weights are recomputed only for the parent body (and not every sub-body) in the rest pose, thus the skinning weights change only for the broken bone.

Figure 14 simulates the ball hitting a bar example from [BHTF07]. Initially the rod only has a single bone which breaks into two bones near the point of impact when the ball hits. Notice that unlike [BHTF07], the ends of the plank remain unchanged even under a large deformation. Figure 13 shows a pile of 400 tetrahedra falling on four static cubic frames. Each tetrahedra is made up of 6 cylinders at edges and 4 spheres at vertices with each cylinder having a single bone. At every collision we dynamically create joints if the relative collision velocity is greater than a threshold.

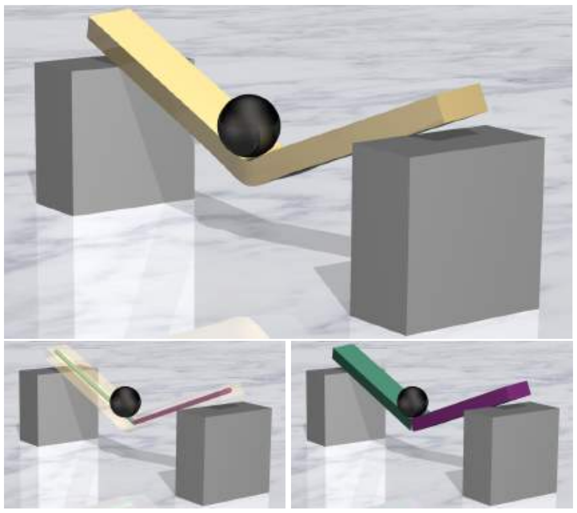


Figure 14: (Top) A ball hits a plank in the center where a joint is dynamically created. (Bottom left) Skeleton bones for the plank. (Bottom right) Triangulated surfaces associated with the bones of the plank.

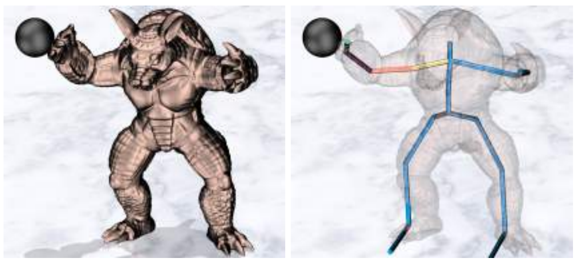


Figure 15: An armadillo simulated as an articulated rigid body, where all the blue bones are condensed into a single rigid body cluster and the four bones adjacent to the four joints in the right arm are simulated as individually articulated bodies to allow the bending of these four joints after the arm is hit by the ball.

Algorithm 2 Bending Pseudo code

```
1:  Compute skinning weights for the desired skeleton
2:  Break the rigid body into sub-bodies based on the skinning weights and articulate the parts together.
3:  for every time step do
4:      Advance the rigid bodies by a dummy step and
5:       gather collision information.
6:      for every collision pair do
7:          if relative normal velocity > threshold then
8:              Dynamically add joints at the collision loca-
9:               tion for dynamically articulated bodies.
10:             Recompute the skinning weights.
11:             Use the skinning weights to divide the parent
12:               body into sub-bodies and recompute mass
13:               properties.
14:             Condense/uncondense joints.
15:     Advance the rigid bodies by one time step.
16:     Update the rigid body surface mesh via skinning.
```

For the bunny mesh consisting of 34000 triangles, the skinning step took about 50ms on a single CPU core of a dual hexcore Dell T7500 workstation. Subdivision of the triangulated surface mesh and level set for the sub-bodies took about 5ms. As the reader may notice, the computation of the skinning weights in step 10 at every time step for the parent rigid body does slow down our algorithm for dynamic articulation. We are currently investigating methods for speeding this up by possibly interpolating skinning weights of simpler shapes obtained through a convex decomposition of the parent rigid body.

5. Limitations

Our method is not physically based by design. If the projectile body is too large or the denting distance is set to a large number, our denting algorithm can suffer from considerable amount of volume loss. We are currently investigating techniques to mitigate this problem by expanding the target body near the impact region to add back the lost volume. As mentioned before, our algorithm for dynamic articulation requires the computation of skinning weights at every time step which slows down our bending algorithm. We are currently investigating methods to speed up this step. Finally, both our bending and denting algorithms can produce self-intersections. The self-intersections in denting can be avoided by carefully choosing a falloff function while those in bending can be prevented by using a better skinning algorithm.

6. Conclusions and Future Work

We proposed a novel, efficient and controllable algorithm for denting and bending rigid bodies without the need for expensive finite element simulations. Our algorithm can be easily integrated into most rigid body simulation frameworks and can readily be used in real-time environments. In the future we would like to use the linearized velocity at the collision point in order to better account for the rotational motion of the projectile body, however, we found this unnecessary for our current examples. It would be interesting to explore other skinning models such as [VBG*13] that allow for self-collisions and contacts to obtain more realistic bending effects. One could automate our bending algorithm by using automatic rigging [BP07] combined with a medial axes algorithm for computing and embedding the skeleton inside the rigid body mesh. For handling more general scenarios during dynamic articulation, one could consider using more information such as external forces rather than just the collision location. We would also like to extend our algorithm to simulate creasing and folding of thin rigid sheets [NPO13].

7. Acknowledgements

Research supported in part by ONR N00014-13-1-0346, ONR N00014-09-1-0101, ONR N-00014-11-1- 0027, ONR N00014-11-1-0707, ARL AHPCRC W911NF-07-0027, and the Intel Science and Technology Center for Visual Computing. Computing resources were provided in part by ONR N00014-05-1-0479. S. P. was supported in part by a Stanford Graduate Fellowship, M. A. by @WalmartLabs and M. L. by a Samsung scholarship.

References

[BFA02] Bridson R., Fedkiw R., Anderson J.: Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. 21*, 3 (2002), 594–603. 8

[BHTF07] Bao Z., Hong J., Teran J., Fedkiw R.: Fracturing rigid materials. *IEEE TVCG 13* (2007), 370–378. 1, 2, 9

[BP07] Baran I., Popović J.: Automatic rigging and animation of 3d characters. *ACM Trans. Graph. 26*, 3 (2007). 10

[BW97] Bonet J., Wood R.: *Nonlinear continuum mechanics for finite element analysis*. Cambridge University Press, Cambridge, 1997. 1

[DBB11] Diziol R., Bender J., Bayer D.: Robust real-time deformation of incompressible surface meshes. SCA '11, pp. 237–246. 1, 6

[dBCvKO08] de Berg M., Cheong O., van Kreveld M., Overmars M.: *Computational Geometry: Algorithms and Applications*. Springer, third edition, 2008. 4

[DdL13] Dionne O., de Lasa M.: Geodesic voxel binding for production character meshes. SCA '13, pp. 173–180. 1

[Dey06] Dey T. K.: *Curve and Surface Reconstruction*. Cambridge University Press, 2006. 1

[ELF13] English R., Lentine M., Fedkiw R.: Fast interpenetration free simulation of volumetric and thin shell rigid bodies. *IEEE TVCG 19*, 6 (2013), 991–1004. 8

[GBF03] Guendelman E., Bridson R., Fedkiw R.: Non-convex rigid bodies with stacking. *ACM Trans. Graph. 22*, 3 (2003), 871–878. 2

[HWCO*13] Huang H., Wu S., Cohen-Or D., Gong M., Zhang H., Li G., Chen B.: L1-medial skeleton of point cloud. *ACM Trans. Graph. 32*, 4 (2013), 65:1–65:8. 1

[JL11] Jain S., Liu K.: Controlling physics-based characters using soft contacts. *ACM TOG 30*, 6 (2011), 163:1–163:10. 2

[KCvO08] Kavan L., Collins S., Žára J., O'Sullivan C.: Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph. 27*, 4 (2008), 105:1–105:23. 1

[KP11] Kim J., Pollard N. S.: Fast simulation of skeleton-driven deformable body characters. *ACM Trans. Graph. 30*, 5 (2011), 121:1–121:19. 2

[LGS*11] Lentine M., Gretarsson J. T., Schroeder C., Robinson-Mosher A., Fedkiw R.: Creature control in a fluid environment. *IEEE TVCG 17*, 5 (2011), 682–693. 2

[LYWG13] Liu L., Yin K., Wang B., Guo B.: Simulation and control of skeleton-driven soft body characters. *ACM Trans. Graph. 32*, 6 (2013), 215:1–215:8. 2

[MC11] Müller M., Chentanez N.: Solid simulation with oriented particles. *ACM TOG 30*, 4 (2011), 92:1–92:10. 6

[MC14] Ma J., Choi S.: Kinematic skeleton extraction from 3d articulated models. *CAD 46*, 0 (2014), 221 – 226. 1

[MCK13] Müller M., Chentanez N., Kim T.-Y.: Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph. 32*, 4 (2013), 115:1–115:10. 1

[MMDJ] Müller M., McMillan L., Dorsey J., Jagnow R.: Real-time simulation of deformation and fracture of stiff materials. In *Comput. Anim. and Sim. '01*, pp. 99–111. 1

[MZS*11] McAdams A., Zhu Y., Selle A., Empey M., Tamstorf R., Teran J., Sifakis E.: Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. 30*, 4 (2011), 37:1–37:12. 2

[NPO13] Narain R., Pfaff T., O'Brien J. F.: Folding and crumpling adaptive sheets. *ACM Trans. Graph. 32*, 4 (2013), 51:1–51:8. 10

[PO09] Parker E. G., O'Brien J.: Real-time deformation and fracture in a game environment. SCA '09, pp. 165–175. 1

[PPG04] Pauly M., Pai D., Guibas L.: Quasi-rigid objects in contact. SCA '04. 1

[RGL05] Redon S., Galoppo N., Lin M.: Adaptive dynamics of articulated bodies. *ACM TOG 24*, 3 (2005), 936–945. 8

[RJ07] Rivers A., James D.: FastLSM: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26*, 3 (2007). 6

[SSF08] Shinar T., Schroeder C., Fedkiw R.: Two-way coupling of rigid and deformable bodies. SCA '08, pp. 95–103. 2

[SSF09] Su J., Schroeder C., Fedkiw R.: Energy stability and fracture for frame rate rigid body simulations. SCA '09, pp. 155–164. 1, 7

[TGTL11] Tan J., Gu Y., Turk G., Liu K.: Articulated swimming creatures. *ACM TOG 30*, 4 (2011), 58:1–58:12. 2

[TZTM12] Tang M., Zhao J.-Y., Tong R.-F., Manocha D.: Smi 2012: Full gpu accelerated convex hull computation. *Comput. Graph. 36* (2012), 498–506. 5

[VBG*13] Vaillant R., Barthe L., Guennebaud G., Cani M.-P., Rohmer D., Wyvill B., Gourmel O., Paulin M.: Implicit skinning: Real-time skin deformation with contact modeling. *ACM Trans. Graph. 32*, 4 (2013), 125:1–125:12. 1, 10

[Wes06] West M.: Parallax mapped bullet holes. Game Developer Magazine, 2006. 1

[WTF06] Weinstein R., Teran J., Fedkiw R.: Dynamic simulation of articulated rigid bodies with contact and collision. *IEEE TVCG 12*, 3 (2006), 365–374. 2, 7