# Clustering and Collision Detection for Clustered Shape Matching

Ben Jones
University of Denver

April Martin
University of Utah

Joshua A. Levine
Clemson University

Tamar Shinar
University of California, Riverside

Adam W. Bargteil*
Univeristy of Maryland, Baltimore County

## Abstract

In this paper, we address clustering and collision detection in the clustered shape matching simulation framework for deformable bodies. Our clustering algorithm is "fuzzy," meaning that it gives particles weighted membership in clusters. These weights are a significant extension to the basic clustered shape matching framework as they are used to divide particle mass among the clusters. We explore several weighting schemes and demonstrate that the choice of weighting scheme gives artists additional control over material behavior. Furthermore, by design our clustering algorithm yields spherical clusters, which not only results in sparse weight vectors, but also exceptionally efficient collision geometry. We further enhance this simple collision proxy by intersecting with half-spaces to allow for even better, yet still simple and computationally efficient, collision proxies. The resulting approach is fast, versatile, and simple to implement.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.6.8 [Simulation and Modeling]: Types of Simulation—Animation.

**Keywords:** Shape Matching, Clustering, Collisions

## 1 Introduction

Introduced a decade ago by Müller and colleagues [2005], *shape matching* is a geometrically motivated technique for animating deformable bodies. Figure 2 summarizes the method. The basic approach samples a deformable object with particles, which determine the degrees of freedom in the object. Each timestep, a best-fit rigid transformation of the rest *shape* of the object to the current configuration of particles is computed and Hookean springs are used to pull the particles toward the transformed shape. A powerful extension to this basic approach, also introduced by Müller and colleagues [2005], is to break the object into several overlapping clusters. We refer to this approach as *clusterd shape matching*. Having more than one cluster imbues the object with a richer space of deformation, while overlap keeps the object from falling apart. While this approach lacks a well-developed mathematical underpinning, it has a number of advantages that make it especially well-suited to interactive graphics applications, such as video games.

In this paper, we address two important aspects of the clustered shape matching framework, namely clustering and collision detection. We consider two previously introduced clustering algorithms and introduce a new algorithm tailored to our specific problem. Of particular note is that our new clustering algorithm is "fuzzy," meaning that particles have weighted membership in clusters. Weighted membership is a significant extension to the basic clustered shape matching framework and greatly increases the flexibility of the approach because the weights are used to determine how a particle's mass is distributed between clusters. We explore several weighting schemes and demonstrate that the choice of weighting scheme gives artists additional control over material behavior. Furthermore, by design our clustering algorithm forms spherical clusters that, unlike most other fuzzy clustering techniques, yield sparse and spatially localized weight vectors.

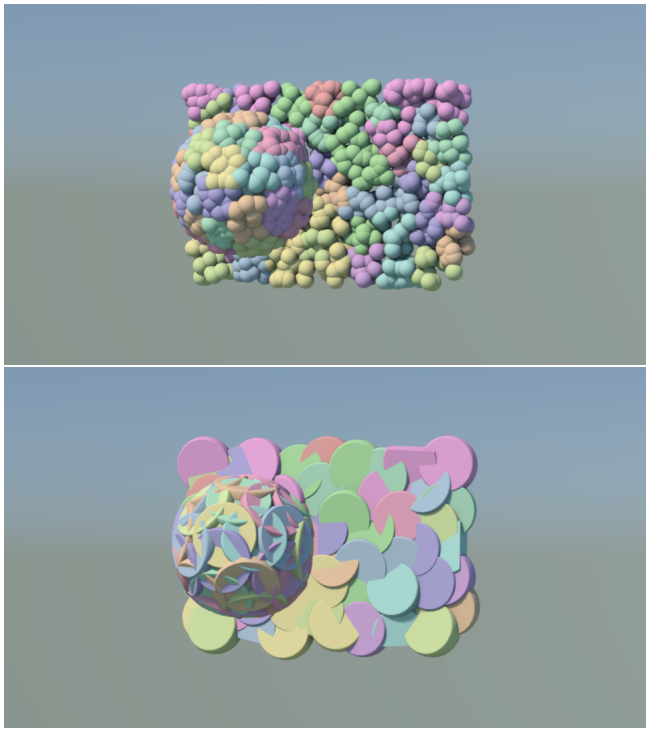

**Figure 1:** *Top: The particles in one of our scenes color-coded by the closest cluster. Bottom: The collision geometry for the same scene.*

Our clustering method also works well with our approach to collision detection. The spherical clusters result in exceptionally convenient collision geometry. We further enhance this simple collision proxy with half-spaces to allow for even better, yet still simple and computationally efficient, collision proxies. The resulting collision detection and handling code is very simple to implement *and* computationally efficient. Combined these extensions significantly enhance the power and versatility of the clustered shape matching framework.

*email: adamb@umbc.edu

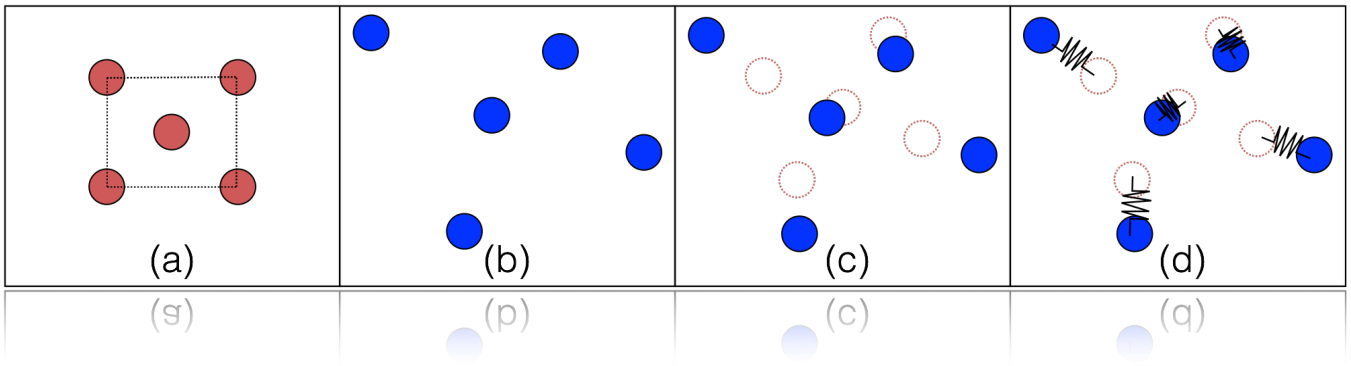


**Figure 2:** *Shape Matching Overview: (a) An object (here, a square) is sampled with particles, $p_i$, to get rest positions, $\mathbf{r}_i$. (b) As particles are subjected to external forces and constraints, their positions, $\mathbf{x}_i$, are updated in world space. (c) The best-fitting rigid transformation of the particles' rest positions, $\mathbf{r}_i$, to their world positions, $\mathbf{x}_i$ is computed. The dotted red circles are the* goal *positions, $\mathbf{g}_i$. (d) Hookean springs pull the world positions toward the goal positions.*

## 2 Related Work

The geometrically motivated shape matching approach was introduced by Müller and colleagues [2005], who demonstrated impressive results and described the key advantages of the approach: efficiency, stability, and controllability. Given these advantages, shape matching is especially appealing in interactive animation contexts such as video games. The authors also introduced several extensions including linear and quadratic deformations (in addition to rigid deformations), cluster-based deformation, and plasticity. Their clusters were generated by regularly subdividing the space around a given surface mesh into overlapping cubical regions. Spring forces are then accumulated over all the clusters to which a particle belongs, effectively imbuing particles in multiple clusters with greater mass. In contrast, we create clusters fully automatically using a variation of fuzzy c-means that favors spherical clusters and distributes a particle's mass among the clusters to which it belongs using a variety of weighting functions.

Two years after the introduction of shape matching, Rivers and James [2007] introduced lattice-based shape matching, which used a set of hierarchical lattices to define the shape matching clusters. They took advantage of the regular structure of the lattices to achieve extremely high performance.

Despite impressive results and substantial promise, the shape matching framework has been largely disregarded by the research community in favor of position-based dynamics [Müller et al. 2007; Bender et al. 2014; Macklin et al. 2014; Kelager et al. 2010], frame-based models [Gilles et al. 2011; Faure et al. 2011], and projective dynamics [Liu et al. 2013; Bouaziz et al. 2014]. Recently, however, Bargteil and Jones [2014] incorporated strain-limiting into clustered shape matching and pointed out several advantages of shape matching over position-based dynamics. Most significantly, shape matching respects Newton's laws of motion.

Clustering is well-studied in the machine learning literature and many techniques, such as $k$-means, have become standard tools in computer graphics. A complete survey of this literature is beyond the scope of this short paper, but for the sort of "fuzzy" clustering we propose in this paper, we recommend the survey by Nock and Nielsen [2006]. Similarly, collision detection is well-studied in the computer graphics and robotics literature. For a survey of real-time techniques we recommend the text by Ericson [2004] or the article by Teschner and colleagues [2005]. There has also been some more recent work that focuses on deformable bodies [Curtis et al. 2008; He et al. 2015].

## 3 Methods

For completeness and readability, we first briefly review the shape matching approach of Müller and colleagues [2005]. Figure 2 summarizes the approach.

### 3.1 Shape Matching

In the shape matching framework objects are discretized into a set of particles, $p_i \in \mathcal{P}$, with masses, $m_i$, and rest positions, $\mathbf{r}_i$, that follow a path, $\mathbf{x}_i(t)$, in world space through time. Shape matching takes its name from the fact that, each frame, we match the rest shape to the deformed shape by finding the least-squares best-fit rigid transformation from the rest pose to the current deformed pose by solving for the rotation matrix, $\mathbf{R}$, and translation vector, $\mathbf{x}_{cm} - \mathbf{r}_{cm}$, that minimizes

$$\sum_i m_i \|\mathbf{R}\left(\mathbf{r}_i - \mathbf{r}_{cm}\right) - \left(\mathbf{x}_i - \mathbf{x}_{cm}\right)\|^2. \tag{1}$$

The best translation is given by the center of mass in the rest and world space. Computing the rotation, $\mathbf{R}$, is more involved. We first compute the least-squares best-fit linear deformation gradient, $\mathbf{F}$. Specifically, we seek the $\mathbf{F}$ that minimizes

$$\sum_i m_i \|\mathbf{F}\left(\mathbf{r}_i - \mathbf{r}_{cm}\right) - \left(\mathbf{x}_i - \mathbf{x}_{cm}\right)\|^2. \tag{2}$$

Setting the derivative with respect to $\mathbf{F}$ to 0 and re-arranging terms we arrive at

$$\mathbf{F} = \left(\sum_i m_i \mathbf{O}(\mathbf{x}_i, \mathbf{r}_i)\right)\left(\sum_i m_i \mathbf{O}(\mathbf{r}_i, \mathbf{r}_i)\right)^{-1} = \mathbf{A}_{xr}\mathbf{A}_{rr}^{-1}, \tag{3}$$

where $\mathbf{O}(\cdot,\cdot)$ is the outer product matrix

$$\mathbf{O}(\mathbf{a}_i, \mathbf{b}_i) = \left(\mathbf{a}_i - \mathbf{a}_{cm}\right)\left(\mathbf{b}_i - \mathbf{b}_{cm}\right)^T, \tag{4}$$

and $\mathbf{A}_{**}$ is a convenient shorthand. We then compute $\mathbf{R}$ using the polar decomposition,

$$\mathbf{F} = \mathbf{RS} = \left(\mathbf{U}\mathbf{V}^T\right)\left(\mathbf{V}\mathbf{\Sigma}\mathbf{V}^T\right) \tag{5}$$

where $\mathbf{S} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T$ is a symmetric matrix and $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ is the singular value decomposition (SVD) of $\mathbf{F}$. While several researchers

(e.g. [Rivers and James 2007]) have pointed out that polar decompositions can be computed faster than the SVD, especially when warm started, the SVD requires a negligible portion of our computation time and, in our experiments, the optimized SVD in the Eigen library was faster than our implementations of polar decompositions. Furthermore, the SVD is more robust in the presence of degeneracies or inversions. We also note that we compute the polar decomposition of $\mathbf{F}$, not the left matrix ($\mathbf{A}_{xr}$) as done by Müller and colleagues [2005]. This modification is particularly important if the distribution of mass in the cluster is non-uniform and $\mathbf{F}$ is not a pure rotation.

Given $\mathbf{R}$ and $\mathbf{x}_{cm} - \mathbf{r}_{cm}$, we define goal positions, $\mathbf{g}_i$, as

$$\mathbf{g}_i = \mathbf{R}\left(\mathbf{r}_i - \mathbf{r}_{cm}\right) + \mathbf{x}_{cm}. \tag{6}$$

Hookean springs are then used to define forces that move the particles toward the goal positions.

### 3.2 Clustered Shape Matching

Handling multiple clusters is straightforward. When computing a particle's contribution to cluster quantities, we must divide the particle's mass among its clusters. To do so we introduce a weight $w_{i,c}$ for particle $p_i$ in cluster $c \in \mathcal{C}$ that describes how much of $p_i$'s mass is contributed to cluster $c$. These weights enter into equations (1)-(3) by multiplying the particle's mass. As an example, the center of mass of cluster $c$, $\mathbf{x}_{cm,c} = \mathbf{x}_c$ (dropping the $_{cm}$ for clarity), is

$$\mathbf{x}_c = \frac{\sum_{p_i \in \mathcal{P}_c} (m_i w_{i,c})\, \mathbf{x_i}}{\sum_{p_i \in \mathcal{P}_c} (m_i w_{i,c})} \tag{7}$$

where $\mathcal{P}_c$ is the set of particles in cluster $c$. We experimented with five weighting schemes in our implementation. The first, and simplest, weighting scheme divides the particles mass evenly among the $n_i$ clusters it belongs to, $w_{i,c} = 1/n_i$. This scheme correspnds to the "box" kernel, or constant weights,

$$\mathrm{box}(\mathbf{x}_i, \mathbf{x}_c, h) = 1. \tag{8}$$

Second is the well-known $\mathrm{poly6}(\cdot)$ kernel [Müller et al. 2003],

$$\mathrm{poly6}(\mathbf{x}_i, \mathbf{x}_c, h) = \frac{315}{64\pi h^9}\left(h^2 - \|\mathbf{x}_i - \mathbf{x}_c\|^2\right)^3, \tag{9}$$

where $h$ is the kernel width. Third is a blend of the $\mathrm{box}$ and $\mathrm{poly6}$ kernels

$$\mathrm{blend}(\mathbf{x}_i, \mathbf{x}_c, h) = \beta + \mathrm{poly6}(\mathbf{x}_i, \mathbf{x}_c, h), \tag{10}$$

where $\beta$ is a blend parameter. Fourth is a simple inverse distance squared kernel,

$$\mathrm{invsq}(\mathbf{x}_i, \mathbf{x}_c, h) = \frac{1}{\|\mathbf{x}_i - \mathbf{x}_c\|^2 + \epsilon}, \tag{11}$$

where $\epsilon$ prevents dividing by zero (we use $\epsilon = 0.0001$). Fifth is the fuzzy $c$-means weighting function [Dunn 1973; Bezdek 1981],

$$\mathrm{fcm}(\mathbf{x}_i, \mathbf{x}_c, h) = \frac{1}{\displaystyle\sum_{d \in \mathcal{C}} \left(\frac{\|x_i - x_c\|}{\|x_i - x_d\|}\right)^{\frac{2}{m-1}}}, \tag{12}$$

where $m$ is a user-specified parameter greater than one. Note that $\mathrm{fcm}$ is the only weighting function where the weight in one cluster depends on the positions of other cluster centers, which significantly complicates computation.

To ensure that the total mass of the clusters equals the total mass of the particles, for all kernels we normalize our weights to a partition of unity,

$$w_{i,c} = \frac{\mathrm{kernel}(\mathbf{x}_i, \mathbf{x}_c, h)}{\sum_{d \in \mathcal{C}} \mathrm{kernel}(\mathbf{x}_i, \mathbf{x}_d, h)} \tag{13}$$

As discussed in Section 4 the choice of kernel can have significant impact on the results; while we selected $\mathrm{invsq}$ as our default, our implementation makes it easy to switch between kernels to satisfy artistic goals.

As noted above, these weights show up in many of our calculations. As another example, when computing the goal position, $\mathbf{g}_i$, for a particle we perform a weighted average of the goal positions given by each cluster it is a part of. That is,

$$\mathbf{g}_i = \sum_c w_{i,c}\, \mathbf{g}_{i,c}, \tag{14}$$

where $\mathbf{g}_{i,c}$ is the goal position for particle $p_i$ in cluster $c$.

**Strain Limiting** To maintain stability we adopt the strain limiting approach advocated by Bargteil and Jones [2014].

### 3.3 Clustering

In our context, there are several desirable properities for a clustering algorithm. Of utmost importance is that the clusters overlap, otherwise the simulated object will fall apart. The clusters must also include all the particles, preferably with a modest number of clusters. Finally, if the clusters are well-approximated by spheres, collision handling becomes far simpler. While clustering is very well-studied in machine learning, these properties are unique to our problem and we are not aware of any algorithm tailored to these constraints.

In our implementation we experimented with three clustering algorithms. The first algorithm, $\mathrm{random}$, borrowed from Bargteil and Jones [2014], is a simple randomized scheme that iteratively chooses a random particle that is not a member of any cluster, uses its location as the center of a new cluster, and adds all particles within a user-specified distance to the cluster. The algorithm terminates when all particles are a member of at least one cluster. Weights, cluster center of mass, etc. are determined after the algorithm terminates. The user specifies the *neighborhood radius*—maximum distance, $d$, from the cluster center to particles included into the cluster; this $d$ is then used in a spherical range query to determine cluster membership. Note that in this algorithm, the user has no direct control over the number of clusters, $|\mathcal{C}|$.

The second algorithm, $\mathrm{kmeans}$, uses $k$-means to determine cluster centers; given $k = |\mathcal{C}|$ random seed locations for clusters, this two-step algorithm iteratively

1. updates cluster membership for each particle by choosing the nearest cluster center;
2. updates cluster centers to be the center of mass of the particles in the cluster.

Convergence is achieved when cluster membership is no longer changing. After the $k$-means algorithm terminates overlapping clusters are computed by including all particles within a user-specified distance, $d$, of the computed cluster centers. Finally, membership weights, cluster center of mass, etc. are computed.

Our final algorithm, $\mathrm{ours}$, more closely resembles *fuzzy c-means* [Dunn 1973; Bezdek 1981], but explicitly seeks greater sparsity in the membership weights by forcing weights for particles far

from a cluster center to zero. As in the previous algorithm, we choose initial cluster centers using $k$-means. However, we then perform an additional two-step iterative optimization:

1. update cluster membership *and weights*;
2. update cluster centers to be the *weighted* center of mass of the particles in the cluster.

Instead of computing the nearest cluster center for each particle as in $k$-means, our algorithm includes all particles within a given distance, $d$, using a standard spherical range query, which is accelerated with a grid data structure. Computing the weights (Equation (13)) requires evaluating the above kernels for each particle in each cluster and keeping a running sum of the weights for each particle. Updating the cluster centers simply requires computing Equation (7) for each cluster using the weights computed in the previous step. To faciliate satisfying the first requirement that all particles belong to at least one cluster, we add any particles that are not within the neighborhood radius, $d$, of any cluster center to the nearest cluster (in a similar manner to the $k$-means algorithm). Any such particles immediately signal that the algorithm has not converged. Otherwise, we declare convergence if cluster membership remains the same for two iterations and the cluster centers have not moved more than $0.1\%$ of the neighborhood radius. If the algorithm does not converge within a limited number of iterations we increase the number of clusters, $|\mathcal{C}|$, and/or the neighborhood radius, $d$, until convergence is achieved.

Compared to $k$-means our algorithm has the advantage of including the fact that clusters should overlap in the optimization itself, which leads to slower convergence, but also to a clustering that is better suited to our needs.

### 3.4 Collision Detection

We first describe the simple and efficient collision proxy geometry we use for clusters and then describe our collision detection and handling algorithm.

**Collision Geometry** Because we explicitly include the distance constraint during clustering, the resulting clusters are generally well approximated by spheres. However, for some input geometry, such as a thin sheet, this approximation is less than ideal. Consequently, we enhance our collision proxy by adding planes. Specifically, our geometric representation of the collision proxy is the intersection of a sphere with a set of half-spaces.

The radius of the sphere is given by the user-specified neighborhood radius, $d$. The center of the sphere is chosen as the geometric center of the range query used during clustering—the randomly chosen seed particles for $\mathrm{random}$ algorithm; the cluster centers given by $k$-means in the $\mathrm{kmeans}$ algorithm; or the cluster centers given by $\mathrm{ours}$. Note that with $\mathrm{random}$ and $\mathrm{kmeans}$ the algorithmic cluster centers and the physical cluster center of mass will generally not be the same point. One advantage of our algorithm is that, because the weighted center of mass is computed during the optimization, the algorithm's output is the physical center of mass.

As mentioned above we intersect these sphere with half-spaces. During initialization, after computing the clusters, we examine the Eigenvectors, $\mathbf{V}_i$, of the scatter matrix, $\mathbf{A}_{rr}$. These Eigenvectors give the principal directions that describe the distribution of particles in the cluster. We compute the planes normal to these Eigenvectors that bound the particles in the cluster. Specifically, for each Eigenvector we compute the dot product with each particle in the cluster, keeping track of the minimum and maximum values, yeilding six plane equations of the form

$$\phi(\mathbf{x}) = \mathbf{V}_i \cdot \mathbf{x} + a_j = 0. \tag{15}$$

We adopt the convention that points for which $\phi(\cdot) < 0$ lie *inside* the half-space. If a plane lies within a user-specified distance of the center of the sphere—that is it cuts off a significant portion of the collision volume—we add it to the collision geometry. This geometric representation is stored in the rest space of the objcet. More planes can be added at runtime by projecting them from world space to rest space using $\mathbf{F}^{-1}$.

For each cluster we also store, in world space, the maximum distance from the center of mass to a member particle, yielding simple collision proxy in world space.

**Collision Detection and Handling** During runtime, we check for collision between every pair of clusters, $c_1$ and $c_2$. We explicitly prohibit collisions between any two clusters that "overlap," or have a particle in common. To facilitate quickly culling these collisions we compute cluster overlap maps; each cluster stores a list of clusters that it cannot collide with. Computing the overlap maps is somewhat involved; for each particle, we inform every pair of clusters in the particle's membership list that there is an overlap. Luckily, these maps can be computed once during initialization if the clusters remain constant.

If two clusters do not have a membership overlap, we next check whether their world space sphere proxies overlap. If not, there is no collision and we continue to the next pair of clusters. If so, we consider each particle, $p_i$, in the first cluster, $c_1$. We first check whether the particle is inside the world space sphere proxy of the second cluster, $c_2$. If not, we continue to the next particle. If so, we transform the particle's world position into the second cluster's rest space, Specifically, we compute

$$\mathbf{x}_i' = \mathbf{r}_{c_2} + \mathbf{F}_{c_2}^{-1} \left( \mathbf{x}_i - \mathbf{x}_{c_2} \right) \tag{16}$$

We then project $\mathbf{x}'$ onto our simple collision geometry by taking the minimum of all projections onto the sphere and half-space components. If the particle lies outside the sphere or any of the half-spaces, we simply continue to the next particle. Otherwise, we transform the projected point, $\mathbf{y}_i'$, to world space to get $\mathbf{y}_i$. Finally we move the particle toward $\mathbf{y}$. Specifically, we update $\mathbf{x}_i$ as

$$\mathbf{x}_i = \mathbf{x}_i + \gamma \left( \mathbf{y}_i - \mathbf{x}_i \right). \tag{17}$$

With $\gamma = 1$ the collision should be completely resolved. In complicated collision scenarios, with multiple constraints on any given particle, choosing $\gamma = 1$ can be problematic, so we allow the user to specify the value.

## 4 Results and Discussion

Artists must specify two parameters for our clustering algorithm: the neighbor radius, $d$, and the desired number of clusters, $|\mathcal{C}|$. Additionally, artists may choose between all three of our clustering algorithms ($\mathrm{random}$, $\mathrm{kmeans}$, and $\mathrm{ours}$) and between all five weighting functions ($\mathrm{box}$, $\mathrm{poly6}$, $\mathrm{blend}$, $\mathrm{invsq}$, $\mathrm{fcm}$). The accompanying video includes a variety of examples that explore the effects of these choices with a simple animation of a cube that is stretched by a factor of 2 in the x-direction and released. In the first set of examples we show the effect of increasing the number of clusters, $|\mathcal{C}|$, from 1 to 50, while choosing the minimal neighbor radius that allows convergence. All these examples use $\mathrm{ours}$ algorithm and the $\mathrm{invsq}$ kernel. These examples clearly show how more clusters "soften" up the object. However, since the neighborRadius

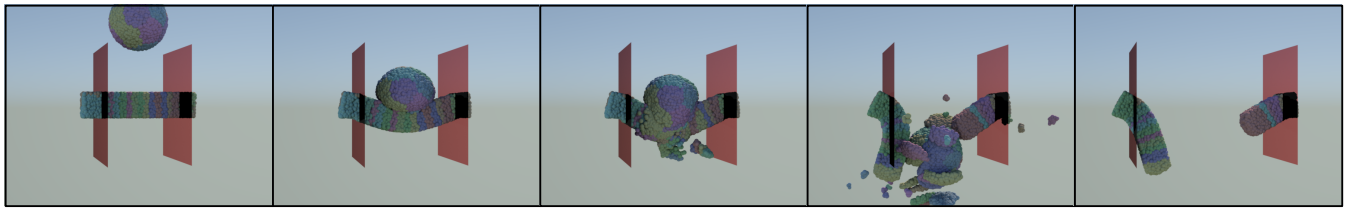

**Figure 3:** *A sphere falls on a beam, causing it to fracture.*

is minimal in some sense, there are still fairly sharp boundaries between the clusters. To illustrate the effect of the neighbor radius, $d$, we fix the number of clusters, $|\mathcal{C}|$, at 10 and increase $d$ from the minimal value needed for convergence. With a minimal radius, there are sharp boundaries between the objects. With a larger radius the result is smoother. Using an even larger radius is akin to using fewer clusters. Thus, there is a complex interaction between these two parameters. One approach to authoring would be to first choose the number of clusters and then increase the neighbor radius from the minimal value until a desired smoothness is acheived. Automatically tuning the neighbor radius would be an interesting area for future work.

We also show examples with the different algorithms and different kernels for cluster sizes of 10 and 50. It is clear from these examples that both the underlying clustering algorithm and the choice of weights have a significant impact on the material behavior. However, our clustering algorithm, because the weights are included in the optimization, is more sensitive to the choice of weighting function.

Next, we have a more practical example of a sphere colliding with a thin sheet. This example demonstrates our approach to collisions. Many of the clusters in the thin sheet include the automatically determined half-spaces demonstrating the efficacy of this approach. On a typical laptop this example ran at 55 frames per second with rendering, 161 frames per second without rendering. See Figure 1 for both the particle rendering as well as a visualization of the collision geometry.

The next example demonstrates our approach on more complex geometry. In this example four instances of the bunny model collide in zero gravity, see Figure 4. This example did not run in real-time.

Finally, we demonstrate our method's ability to update the collision proxies at runtime by including an example that exhibits fracture. In this example, the stress caused by the collisions induces the objects to break apart, see Figure 3.

Qualitatively, we found that the box kernel was too sharp to be satisfactory and that the falloff of poly6 was overly smooth, resulting in little resistance to deformation. blend performed better, but required setting an additional parameter. invsq and fcm performed similarly; because invsq is simpler and does not require setting an additional parameter, we chose it as our default kernel. When setting up examples, we typically chose the number of clusters, $|\mathcal{C}|$, first and then adjusted the neighbor radius, $d$, until we achieve the desired behavior. Too little overlap results in clusters that appear almost disconnected, while too much overlap produces results that could be achieved with fewer clusters. By default, our program automatically increases $|\mathcal{C}|$and $d$ if the clustering algorithm fails to converge.

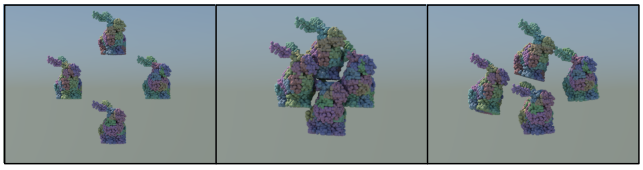

**Figure 4:** *Several bunnies collide in zero gravity.*

**Limitations and Future Work** Like other methods that lump mass at points in the object, the distribution of mass in the object is not uniform. For this reason, the stretching the floating cube does not result in uniform stress throughout the object and, even though momentum is preserved, the cube does not oscillate around its center of mass. Such lumping artifacts decrease at higher resolution. Shape matching is ill-suited to very stiff objects; the introduction of the $\alpha$ parameter essentially ties the maximum stiffness to the inverse of the timestep. This limitation is somewhat ameliorated by strain limiting, but excessive use of strain limiting reduces our method a version of position-based dynamics, with its attendant violations of Newton's laws of motion.

It would be interesting to consider additional clustering shapes, such as ellipsoids, that might better conform to complex geometry while remaining simple and computationally efficient. Our current implementation does not employ a bounding volume hierarchy resulting in an asymptotic runtime for collision detection that is quadratic in the number of clusters. Employing a variation of the hierarchy of He and colleagues [He et al. 2015] is a promising direction for future development and may allow the colliding bunnies example to run in real time. Finally, while nothing in our approach prevents spatially varying material properties, such variation is not currently supported by our implementation.

In conclusion we have significantly extended the clustered shape matching framework for animating deformable bodies by introducing a new "fuzzy" clustering algorithm that produces a highly flexible weight distribution for each particle and also leads to a simple collision proxy that we enhance with half-spaces. These approaches will certainly improve the power and versatility of the clustered shape matching framework.

## Acknowledgements

The authors wish to thank the anonymous reviewers for their time and helpful comments. We also thank Parasaran Raman and Suresh Venkatasubramanian Suresh for sharing their knowledge of clustering algorithms. This work was supported in part by National Science Foundation awards IIS-1314896, IIS-1314757, and IIS-1314813

## References

BARGTEIL, A. W., AND JONES, B. 2014. Strain limiting for clustered shape matching. In *Proceedings of the Seventh Inter-*

*national Conference on Motion in Games*, ACM, New York, NY, USA, MIG '14, 177–179.

BENDER, J., MÜLLER, M., OTADUY, M. A., TESCHNER, M., AND MACKLIN, M. 2014. A survey on position-based simulation methods in computer graphics. *Computer Graphics Forum*, 1–25.

BEZDEK, J. C. 1981. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA.

BOUAZIZ, S., MARTIN, S., LIU, T., KAVAN, L., AND PAULY, M. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. 33*, 4 (July), 154:1–154:11.

CURTIS, S., TAMSTORF, R., AND MANOCHA, D. 2008. Fast collision detection for deformable models using representative-triangles. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '08, 61–69.

DUNN, J. C. 1973. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics 3*, 3 (Jan.), 32–57.

ERICSON, C. 2004. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

FAURE, F., GILLES, B., BOUSQUET, G., AND PAI, D. K. 2011. Sparse meshless models of complex deformable solids. *ACM Trans. Graph. 30*, 4 (July), 73:1–73:10.

GILLES, B., BOUSQUET, G., FAURE, F., AND PAI, D. K. 2011. Frame-based elastic models. *ACM Trans. Graph. 30*, 2 (Apr.), 15:1–15:12.

HE, L., ORTIZ, R., ENQUOBAHRIE, A., AND MANOCHA, D. 2015. Interactive continuous collision detection for topology changing models using dynamic clustering. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 47–54.

KELAGER, M., NIEBE, S., AND ERLEBEN, K. 2010. A Triangle Bending Constraint Model for Position-Based Dynamics. In *Workshop in Virtual Reality Interactions and Physical Simulation*, K. Erleben, J. Bender, and M. Teschner, Eds.

LIU, T., BARGTEIL, A. W., O'BRIEN, J. F., AND KAVAN, L. 2013. Fast simulation of mass-spring systems. *ACM Transactions on Graphics 32*, 6 (Nov.), 209:1–7. Proceedings of ACM SIGGRAPH Asia 2013, Hong Kong.

MACKLIN, M., MÜLLER, M., CHENTANEZ, N., AND KIM, T.-Y. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph. 33*, 4, 104.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, 154–159.

MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. 24*, 3 (July), 471–478.

MÜLLER, M., HEIDELBERGER, B., HENNIX, M., AND RATCLIFF, J. 2007. Position based dynamics. *J. Vis. Comun. Image Represent. 18*, 2, 109–118.

NOCK, R., AND NIELSEN, F. 2006. On Weighting Clustering. *IEEE Trans. Pattern Anal. Mach. Intell. 28*, 8 (Aug.), 1223–1235.

RIVERS, A. R., AND JAMES, D. L. 2007. Fastlsm: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26*, 3 (July).

TESCHNER, M., KIMMERLE, S., HEIDELBERGER, B., ZACHMANN, G., RAGHUPATHI, L., FUHRMANN, A., CANI, M.-P., FAURE, F., MAGNENAT-THALMANN, N., STRASSER, W., AND VOLINO, P. 2005. Collision Detection for Deformable Objects. *Computer Graphics Forum*.