Simplification of Node Position Data for Interactive Visualization of Dynamic Datasets

Paul Rosen and Voicu Popescu

Abstract—We propose to aid the interactive visualization of time-varying spatial datasets by simplifying node position data over the entire simulation as opposed to over individual states. Our approach is based on two observations. The first observation is that the trajectory of some nodes can be approximated well without recording the position of the node for every state. The second observation is that there are groups of nodes whose motion from one state to the next can be approximated well with a single transformation. We present dataset simplification techniques that take advantage of this node data redundancy. Our techniques are general, supporting many types of simulations, they achieve good compression factors, and they allow rigorous control of the maximum node position approximation error. We demonstrate our approach in the context of finite element analysis data, of liquid flow simulation data, and of fusion simulation data.

Index Terms—simplification of node positions, trajectory simplification, trajectory clustering, rigid body decomposition, interactive visualization, simulation data compression.

As compute power and simulation algorithm sophistication continue to advance, scientists and engineers are inspired to simulate complex phenomena with increased fidelity. These simulations often take the form of time-varying spatial datasets with high spatial and temporal resolution. Such a dataset records the 3-D position of every simulation node (e.g. finite element vertex, particle center) for each simulation state. The result is a large volume of data that is challenging for transmission to remote parties and for interactive visualization. The fact is that the movement, storage, and visualization of large datasets continues to be a significant challenge in visualization as a whole [1].

Until now, much of the research in interactive visualization has focused on simplifying a static dataset or a single state of a time-varying dataset. However, such an approach does not take advantage of the temporal coherence exhibited by time-varying datasets. The simplification of node position data over all simulation states has received minimal attention from the visualization community in spite of how common this form of data happens to be.

In this paper we propose techniques for simplifying node position information in time-varying simulation datasets by considering all simulation states. The techniques take advantage of motion redundancy characteristic to most simulations. A first technique is based on the straightforward observation that the trajectory of most nodes can be described well without recording the position of the node for each intermediate state. For example, the trajectory of a node that does not move or that moves on a straight line should be described with only a starting and an ending node position. This technique, dubbed individual trajectory simplification or ITS, uses a specialized form of polyline simplification to achieve significant dataset compression factors given a user specified maximum node position error. We define the maximum node position error as the maximum Euclidean distance between the position of a node in the simplified and in the original datasets, over all nodes and all states. The compression factor is defined as the ratio between the storage size of the original and of the simplified datasets.

A second and a third technique are based on the observation that in many simulations there are groups of nodes that move together as semi-rigid bodies. For such a group, the motion of the nodes from one state to the next can be described well with a single transformation, which is more compact than storing the intermediate positions of the nodes explicitly.

Simulation data can come in many forms, from triangle and tetrahedral elements of animation or finite element crash simulations to the connectivity-free point-cloud data of dam breaks and other fluid simulation, magnetic field lines of a fusion reactor, or celestial motion simulation. The type and availability of connectivity information associated with these types of data can vary widely from domain to domain and even between simulations within a single domain. Relying too heavily upon any connectivity data reduces the generality of any simplification technique to at best *only* simulations with connectivity information, despite the widespread usage of datasets lacking any connectivity information. Whereas specific simulation information, such as any neighborhood or connectivity information, can be used to help determine these groups, the only input we use are node positions. One reason for this is that initial groups of nodes can deform or break apart under the extreme conditions that are typically simulated, so groups have to be validated and further refined using simulation data anyway. A second reason is that many groups of nodes cannot be determined a priori from the

- *P. Rosen is with the Scientific Computing and Imaging Institute, The University of Utah, Salt Lake City, UT 84112. E-mail: prosen@sci.utah.edu*
- *V. Popescu is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: popescu@purdue.edu*

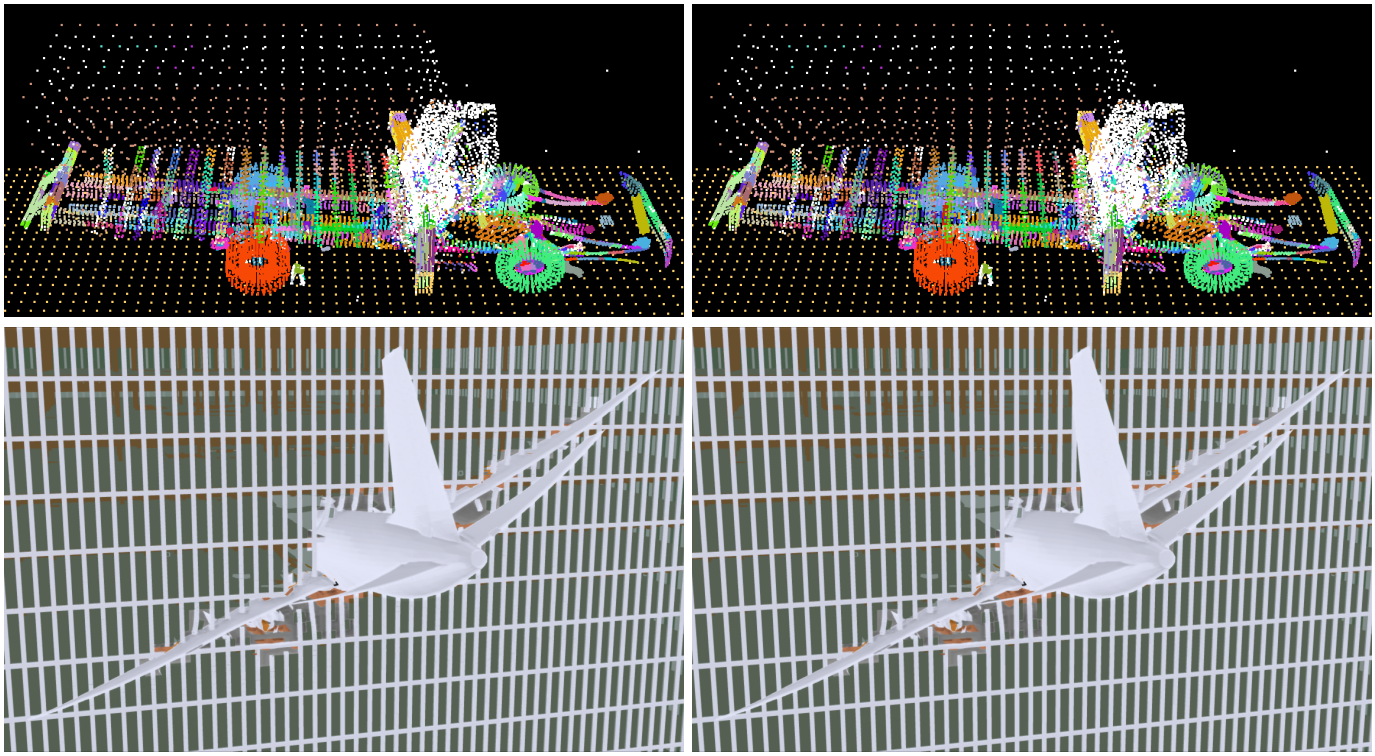


Fig. 1. Top: Dataset simplified using rigid body decomposition (left) and original dataset for comparison (right). Nodes not assigned to a rigid body are shown in white. The dataset size is 15m x 5m x 3.3m, the maximum node error is 10mm, and the data compression factor is 11:1. Bottom: Dataset simplified using trajectory clustering (left) and original dataset (right). The dataset size is 110m x 90m x 60m, the maximum node error is 10mm, and the data compression factor is 14:1.

simulation input data, and can only be determined a posteriori by examining the simulation output data: pieces of a concrete wall that is breaking apart, sections of a metal structural column undergoing deformation, or groups of liquid particles moving in unison. A third reason for only relying on node data is that in this way the resulting technique is general, independent of domain, simulation approach, and simulation scene specifics.

The first technique based on node grouping, dubbed trajectory clustering or TC, only considers translations from state to state. The second technique, dubbed rigid body decomposition or RBD, allows for a full 6 degree of freedom transformation from one state to the next. For both techniques, the node group is defined by the initial positions of the nodes and by a sequence of transformations. The transformations are applied to the initial node positions to decode intermediate positions. Compared to RBD, TC has the advantage of faster encoding and of more compact transformation storage (i.e. it does not need to encode rotation angles). RBD has the advantage of detecting rotating rigid bodies, which are missed by TC, yielding better compression factors for datasets where rotating rigid bodies are significant. Both TC and RBD can be used in conjunction with ITS.

Figure 2 shows RBD applied to a liquid simulation on subsequences of 10 states at a time. The dataset covers a 1m x 0.18m x 0.2m region, and the maximum error is 1mm, or 0.1%. The top row of Figure 1 shows an example of RBD simplification using a point-based visualization. The trajectories of nodes that are not assigned to a rigid body are simplified using ITS. A good compression factor is achieved for a maximum node error of 10mm that corresponds to a relative error of 0.06% (10mm/15m). The bottom row of Figure 1 shows an example of TC simplification, where cluster and residual trajectories are simplified with ITS. The mesh-based visualization of the simplified dataset is virtually indistinguishable from the visualization constructed from the original dataset. Figure 7 illustrates TC simplification on a fusion simulation dataset. The dataset covers a 2.5m x 2.5m x 0.5m region and the maximum error is 10mm, or 0.04%. We also refer the reader to the accompanying video.

In summary, our paper contributes node data simplification techniques that provide:

- good compression factors,
- strict enforcement of a maximum node position error specified by the user,
- fast decoding, and
- support for many simulation types by relying exclusively on node data.

The remainder of the paper is organized as follows. Section 1 discusses prior related work. Sections 2, 3, and 4 outline the individual trajectory simplification, trajectory clustering, and rigid body decomposition simplification techniques, re-

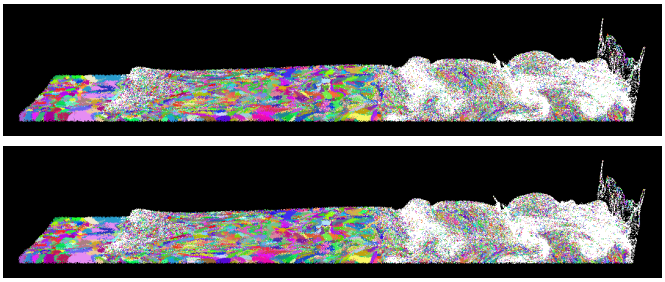


Fig. 2. Simplification of liquid simulation dataset using rigid body decomposition (top) and original dataset (bottom). Compression factor is 8.5:1 for a maximum error of 0.1%.

spectively. Results are discussed in section 5, and section 6 will conclude the work.

1 Prior Work

The vast majority of work in simplifying simulation position data has focused on compressing the data using high-level structures such as triangles meshes for a single static simulation state. A smaller body of work, primarily coming from the field of animation, exists on the simplification of time-varying node positions. For completeness, we review both.

The fields of both computer graphics and visualization have seen many approaches for reducing the storage size of index and node positions for static meshes. The broad majority are focused on compressing vertex and/or connectivity data of triangular meshes [2], [3], [4], [5], [6], [7], [8] or in some cases polygonal meshes [9], [10], [11] for the purpose of reducing storage and transmission costs or for streaming geometric detail [12], [13], [14]. There has been work on compressing volumetric structures as well, such as tetrahedral meshes [15], [16] and hexahedral meshes [17]. Some methods work on more general unstructured data, such as point-clouds, by reordering the nodes [18].

In contrast to the majority of these methods, our method works independently of the higher level structures. Furthermore, these methods are only concerned with simplifying data in the spatial domain while our approach simplifies in both the space and time domains. While you could, for example, use these techniques to compress each state of a simulation independently of the others, such an approach would ignore very important temporal data correlations that can be made between states. We do however see the usefulness of some these approaches as potential enhancements to the techniques we present. All of our techniques currently store the initial node positions in a completely uncompressed state. Efficient coding of this data would produce even higher compression rates than those presented.

For time-varying datasets, approaches have employed the simplification of individual trajectories for applications including compressing and managing trajectories of moving object databases [19], in a vain similar to ours. Other work has also been done on clustering trajectories [20], [21], some using individual trajectory simplification as an aid [22]. Whereas we are concerned with simplification of large datasets, this work has mostly focused on finding clusters of trajectories as a means of identifying features of data. The approach is therefore similar to ours, but focuses on design decisions such as robustness to finding features while our primary goal is compactness.

Some of the earliest work on time-varying datasets advocated for manually subdividing a mesh into sets of rigid bodies [23], [24] which would then have their positions updated as a group. The automatic segmentation [25], [26], [27] of meshes has also received a lot of attention. Later approaches use principal component analysis to automatically detect and compress rigid-body [25], [28] or soft-body [29] animations. Still others have automatically calculated skeletons to compress animation [30]. Shamir and Pascucci [31] generated level-of-detail animations for meshes by combining low-frequency motion encoding affine transformations with residuals for encoding high-frequency motion. Others have forgone mesh segmentation and instead used Fourier or wavelet compression [32], [33], [34] or space and time predictors and connectivity graphs [35] to enable predicting new vertex positions from neighbor positions in the previous and current states to reduce the size of time-varying datasets. Many of these approaches in one way or another, rely on an underlying mesh structure for their simplification.

Finally, there are a wide variety of techniques for compressing floating-point numbers [36], [37], [38], [39], [40] to reduce the size of simulation datasets. These techniques are complementary to our approach. We store our simplified data as full-sized uncompressed 32-bit floating-point numbers. It is very likely that the addition of one or more of these techniques would further enhance our results.

2 Individual Trajectory Simplification

The underlying shape of a node's trajectory in time-varying simulations is a curve, which, in the process of computing and saving simulations, is discretized into a set of piecewise linear segments, a.k.a a polyline. A polyline is sampled at regular time intervals regardless of the shape and amplitude of the motion of the node. Polyline simplification can be used to remove redundant intermediate node positions, reducing the size of simulation data.

A polyline simplification will find a representative subset of points from an original polyline which closely matches the shape of the original polyline. In conventional polyline simplification, the simplified polyline is found by using the orthogonal distance to measure error and match the shape. In our context we are interested in measuring the error at the simulation states, which are regularly spaced time intervals, so we modify the error function as shown in Equation 1, where the two polylines are represented as functions of time $f(t)$ and $g(t)$.

$$Diff(f(t),g(t)) = MAX(\quad \|f(t_1)-g(t_1)\|,\ldots, \quad \|f(t_M)-g(t_M)\|) \tag{1}$$

2.1 Method

Given an input polyline and an error threshold, an optimal polyline simplification finds a polyline (P_{simp}) that satisfies two conditions. The first condition is that there should be no polyline with fewer points than P_{simp} that satisfies the error threshold. The second is that out of all polylines with the same number of points as P_{simp}, P_{simp} should have the smallest error. However, a brute force search for the optimal polyline simplification is prohibitively slow, as the number of possible polylines is exponential in the number of states.

The algorithms of Ramer [41] and Douglas–Peucker [42] are fast methods for finding near-optimal polyline simplifications. Their methods take a greedy approach to building a simplified polyline by incrementally inserting into the simplification the point where error is largest. The algorithm adapted to our context is given below.

Algorithm: RamerDouglasPeucker

Input : Polyline $P = \{p_1, \ldots, p_N\}$ and threshold ε
Output: Simplification P_{simp} with error $E_{simp} <= \varepsilon$

1 Insert p_1 and p_N into P_{simp}
2 **repeat**
3 Find the point p_i with the maximum error e_i
4 $E_{simp} = e_i$
5 **if** $E_{simp} > \varepsilon$ **then**
6 Insert p_i in P_{simp}
7 **end**
8 **until** $E_{simp} <= \varepsilon$;

The algorithm starts from a segment connecting the two end points and keeps adding points of largest error until the error condition is met. Figure 3 right shows an example of this algorithm while Figure 3 left shows the optimal polyline simplification. Initially the polyline simplification contains the two endpoints $\{p_1, p_8\}$. The point with the largest error (p_3) is then inserted into the approximation resulting in polyline $\{p_1, p_3, p_8\}$. Subsequently, p_5 and then p_7 are inserted into the approximation, resulting in the final polyline $\{p_1, p_3, p_5, p_7, p_8\}$. This algorithm produces a polyline simplification which may be suboptimal (see rows 2 and 3), but it does find quality approximations quickly.

2.2 Encoding, Storage, and Decoding

Polylines are encoded in two parts. First, all polylines contain the first and last position, which are stored directly. Second, all intermediate positions are stored as {position, state} pairs. The data is stored in sorted order by time.

The size of the data storage first assumes 3-D points and floating point storage. Each position then costs $3*4$ bytes to store for a total storage of $3*4*p$ bytes for p positions. The other component to store is the state number for each point. This can be stored in a byte for simulations with fewer than 256 states, otherwise a short is used. Assuming the usage of a single byte, the total storage cost is $3*4*p+1*(p-2)$. Note, the first and last state numbers need not be stored, since

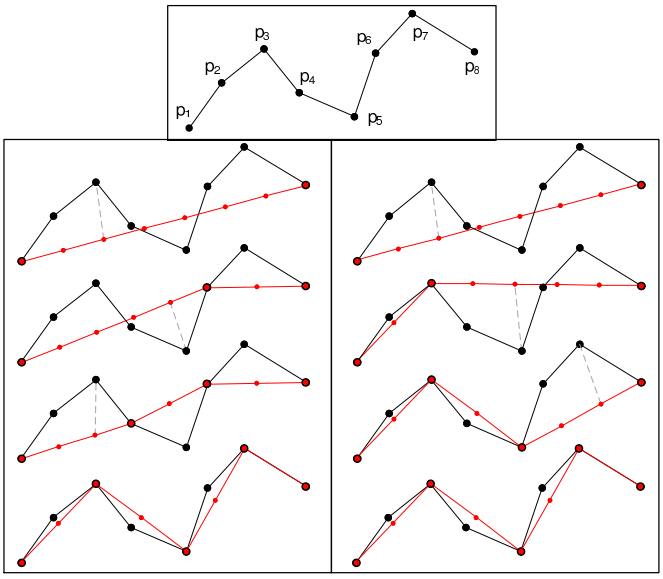


Fig. 3. The input polyline (top) is simplified shown with the optimal representation (left) and the greedy Ramer / Douglas-Peucker method (right) which is significantly faster but may not always find the best representation.

they are implicit. This is compared to the cost of storing the original polyline of s states which is $3*4*s$. Consider an example polyline with 100 states. The original polyline would cost 1,200 bytes to store. Compressing the line with 92 points (1,194 bytes) would result in a compression ratio of approximately 1:1, while 25 points (323 bytes) would lead to a 3.7:1 compression ratio.

Decoding node positions is accomplished by performing linear interpolation with respect to time across the interval containing the current time. The next problem is finding the interval. For sequential access in time (either forward or backward) an index is stored for the most recently used interval, resulting in constant time access. For random access, a binary search is performed on the intervals to find the interval containing the current time, resulting in logarithmic access time.

3 Trajectory Clustering

Trajectory clustering takes advantage of data redundancy caused by groups of nodes moving in unison. The method works by locating and clustering trajectories with similar motion using a greedy algorithm that attempts to minimize cluster entropy. This idea parallels that of the principal component analysis methods for meshes [25], [28], [29] and that of k-means clustering of points [43].

3.1 Method

Before trajectory clustering can proceed, node trajectories need to be transformed into initial-position invariant trajectories, which is simply achieved by subtracting the position of the node at state 0 from the positions of the node at all the other states. A metric is needed for estimating the difference

between two trajectories. We have chosen the infinity norm of the Euclidean distance between corresponding points of the two trajectories. This is done using Equation 1, replacing $f(t)$ with trajectory $\overrightarrow{x}$, defined by points x_i, and $g(t)$ with trajectory $\overrightarrow{y}$, defined by points y_i.

A cluster of trajectories (T) is defined as a set of trajectories $\{\overrightarrow{t_1}, \overrightarrow{t_2}, \ldots, \overrightarrow{t_N}\}$ where the first trajectory ($\overrightarrow{t_1}$) is the basis for the simplified motion of the entire cluster. Each cluster of trajectories also has an entropy value assigned to it. Equation 2 describes the entropy value for a cluster as the 2-norm of the difference between the basis trajectory ($\overrightarrow{t_1}$) and all other trajectories in the cluster.

$$Entropy(T) = \left(\sum_{i=2}^{n} Diff\left(\overrightarrow{t_1}, \overrightarrow{t_i}\right)^2 \right)^{1/2} \quad (2)$$

The set of all trajectory clusters is found as follows.

Algorithm: FindClusters

Input : A set containing a single cluster, $\mathbf{T} = \{T_0\}$
Output: A set of clusters, $\mathbf{T} = \{T_1, \ldots, T_U\}$

```
repeat
    Remove cluster T_i with highest entropy from T
    Subdivide T_i into 2 new clusters T_j and T_k
    Insert T_j and T_k into T
until size(T) = desired number of clusters;
```

The input is a set ($\mathbf{T}$) containing a single cluster holding all trajectories for the dataset. The algorithm subdivides the cluster with highest entropy repeatedly until the desired number of clusters is reached. Cluster subdivision is performed by the *SubdivideCluster* algorithm.

Algorithm: SubdivideCluster

Input : Cluster of trajectories $T = \{\overrightarrow{t_1}, \overrightarrow{t_2}, \ldots, \overrightarrow{t_N}\}$
Output: Two new clusters, A and B

```
Select 2 trajectories a and b randomly from T
foreach t_i in T do
    if Diff(a, t_i) < Diff(b, t_i) then
        Insert t_i in A
    else
        Insert t_i in B
    end
end
```

The algorithm first selects 2 trajectories ($\overrightarrow{a}$ and $\overrightarrow{b}$) randomly from the input cluster (T). These trajectories become the basis trajectories for new clusters A and B. Each trajectory ($\overrightarrow{t_i}$) in the input cluster (T) is compared with both $\overrightarrow{a}$ and $\overrightarrow{b}$, using Equation 1. The trajectory ($\overrightarrow{t_i}$) is inserted into the new cluster with the most similar basis trajectory.

The choices used in this algorithm can be seen as a compromise between speed, optimality, and compactness. As an alternative, the k-means approach could be used to subdivide the set of trajectories into 2 or more clusters. However k-means requires many sweeps through the data to find optimal sets, a cost which because difficult to bear as the data becomes larger. Further, we could have used the k-means approach to calculate a mean for the cluster basis trajectory (this is in fact the first approach we attempted). In practice, we found that using a trajectory from the original dataset gave us similar errors for cluster members with the added benefit of avoiding the need to save any residual information for the basis trajectory. As the number of clusters increase and cluster sizes decrease, this leads to significantly fewer residuals needing to be calculated.

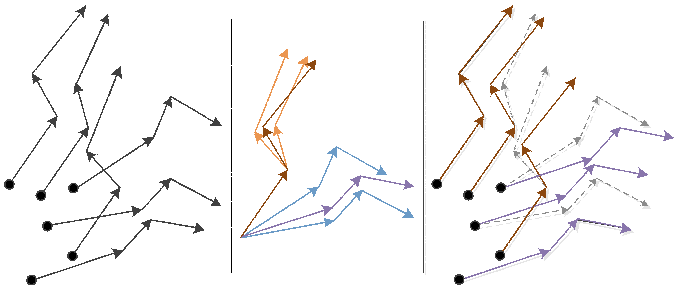


Fig. 4. Trajectory clustering example. The input nodes (left) are converted to clusters and simplified (middle). The clustering is then applied to the original nodes (right) resulting in a lossy 3:1 compression ratio.

When contemplating this approach one might also expect to follow an error-bounded approach similar to that of ITS. Unfortunately, this approach does not work particularly well due to outlier trajectories. There tend to be a few trajectories which travel in very different directions from most other trajectories. These outliers are difficult to detect and can lead to many unnecessary cluster subdivisions.

Figure 4 shows a trajectory clustering example. Six node trajectories (left) are first converted to initial-position invariant trajectories (middle). Two basis trajectories (brown and purple) are selected at random to seed two new clusters (middle). The remaining trajectories are added to the new cluster with the closest basis trajectory (orange with brown and blue with purple, middle). The result is a 3:1 compression factor, as six trajectories have been reduced to only two. The basis trajectories of the new clusters is applied to the initial node positions to approximate positions at intermediate states (right). Trajectory clusters obtained with our algorithm are visualized in Figure 5.

3.2 Encoding, Storage, and Decoding

Each cluster has a single trajectory associated with it. These trajectories are first simplified using the ITS method presented in section 2. For every node, an initial position and index to a trajectory cluster is stored.

The storage space for each cluster trajectory is relatively small in size (refer to section 2), however one must be stored for each cluster. The storage space for the initial positions is $3*4*p$ for p nodes (assuming 3-D floating point values). The raw storage space for unsigned integer indices is $4*p$.

Decoding a node position first requires using the trajectory cluster index to find the associated cluster. An offset position is the calculated using the ITS decoding method. The final position is then found by adding the initial node position to the offset position for the cluster.

4 Rigid Body Decomposition

Consider a dataset with n nodes and s states. The goal of rigid body decomposition is to partition the nodes into groups such that the motion of the nodes belonging to a group can be approximated well with a sequence of rigid body transformations. We define a rigid body decomposition of the dataset as a triplet (G, S_0, Q):

- $G = \{g_1, g_2, \ldots, g_m\}$ is a partition of nodes into m groups, with each group containing at least 3 nodes,
- $S_0 = \{(x,y,z)_1, (x,y,z)_2, \ldots, (x,y,z)_n\}$ is the set of initial 3-D positions for all nodes, and
- $Q = \{q_1, q_2, \ldots, q_m\}$ is a set of m sequences of $s-1$ rigid body transformations.

Not all simulation nodes have to be assigned to a group. A rigid body transformation consists of a rotation followed by a translation. Let j be a node assigned to group i. The rigid body decomposition approximates the position of node j at state k according to Equation 3.

$$node_j^k = \begin{cases} (x,y,z)_j, & \text{if } k = 0 \\ q_i^{k-1}.R * (x,y,z)_j + q_i^{k-1}.T, & \text{if } k > 0 \end{cases} \quad (3)$$

4.1 Method

Given a user-selected maximum node position error ε, a rigid body decomposition of a dataset is computed with the following algorithm.

Algorithm: RigidBodyDecomposition

Input : All node positions and an error threshold ε
Output: Rigid body decomposition (G, S_0, Q)

```
1  G = {}, Q = {}, S0 = {}
2  foreach node i do
3      foreach group j in G do
4          if Err(i, q_j) < ε then
5              g_j += {i}
6              S0 += {(x,y,z)_i}
7              next i
8          end
9      end
10     if NewRigidBody(i, ε, j_out, k_out, q_out) then
11         G += {(i, j_out, k_out)}
12         Q += {q_out}
13         S0 += {(x,y,z)_i, (x,y,z)_j_out, (x,y,z)_k_out}
14         next i
15     end
16     Mark i as unassigned
17 end
```

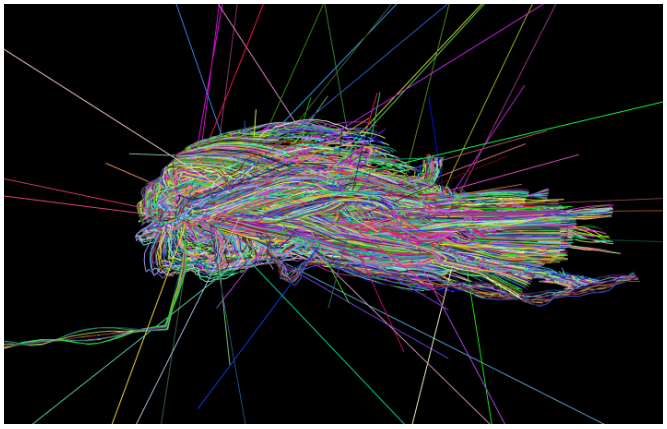


Fig. 5. A visualization of the trajectory clustering for the truck dataset shown in Figures 1 and 8. The trajectories are all centered at the origin and colored using their associated cluster ID.

The algorithm considers each node in turn. The algorithm first tries to assign the current node to an existing group (line 3). The function $Err(i, q_j)$ returns the maximum node position approximation error over all states; to do so, the position of node i is estimated at all states using the sequence of rigid body transformations q_j. If node i cannot be assigned to an existing group, the algorithm attempts to construct a new rigid body with the given node (line 10). If that fails, the node remains unassigned (line 16).

A new rigid body with three nodes is constructed as follows.

Algorithm: NewRigidBody

Input : A node i and an error threshold ε
Output: Construction success and j_{out}, k_{out}, q_{out}

```
1  foreach node j do
2      if distance (i,j) varies > ε then
3          next j
4      end
5      foreach node k do
6          if distance (i,k) or (j,k) varies > ε then
7              next k
8          end
9          ComputeRBX(i, j, k, q)
10         if Err(i,q) & Err(j,q) & Err(k,q) < ε then
11             (j_out, k_out, q_out) = (j, k, q)
12             return true
13         end
14     end
15 end
16 return false
```

Given a node i with which to construct a new rigid body, the algorithm first finds a second node j that remains approximately at the same distance from i throughout the simulation (line 2). This condition is implemented by computing the

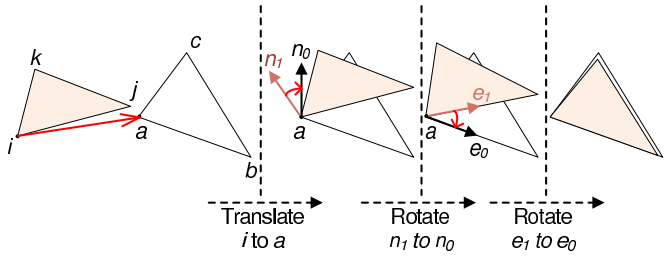


Fig. 6. Illustration of 3 step rigid body transformation construction described by algorithm ComputeRBX.

distance d between nodes i and j at state 0, and then by checking whether the distance remains within ε of d for the subsequent states.

Once a second node is found, the algorithm searches for a third and final node k (line 5). Nodes whose distance to the previous two nodes does not remain approximately constant throughout the simulation are early rejected (line 6). The three candidate nodes are used to construct a sequence q of $s-1$ rigid body transformations (line 9), as described below. If q places i,j,k within ε of their true positions for all states (line 10), the new rigid body (i,j,k) with its transformation sequence q are returned (line 11).

Given the 3 nodes i, j, k, a sequence of transformations q_{out} is constructed as follows.

Algorithm: ComputeRBX

Input : i,j,k
Output: q_{out}

1 Make (i,j,k) the triangle with nodes i,j,k at state 0
2 **foreach** *state t = 1 to s-1* **do**
3 Let triangle (a,b,c) be defined by i,j,k at state t
4 X_1: Translate (i,j,k) by $a-i$
5 X_2: Rotate (i,j,k), align planes (i,j,k) & (a,b,c)
6 X_3: Rotate (i,j,k), align edges (i, j) & (a, b)
7 $q_{out}^{t-1} = X_3$ x X_2 x X_1
8 **end**

Given 3 nodes, a transformation is constructed for each state t in 3 steps that align the triangle defined by the 3 nodes at state 0 with the triangle defined by the nodes at state t (also refer to Figure 6). The first step is a translation that aligns one vertex of the two triangles. The second step is a rotation that aligns the normals of the two triangles. Finally the third step is a rotation in the now common plane of the two triangles to align a pair of corresponding edges. The triangles are not congruent thus the two triangles will not overlap perfectly. By construction, the approximation error at vertex i is 0. The transformations of the individual steps are combined to compute the final rigid body transformation from state 0 to state t.

4.2 Encoding, Storage, and Decoding

Once a rigid body decomposition of a dataset has been computed, we encode it by storing:

1) for each node, the initial position and an index pointing to the group to which the node is assigned (or -1 if the node is not assigned);
2) for each group, a rigid body transform for every state except state 0; a transform is encoded with 3 translations and 3 Euler angles;
3) for each unassigned node, the original positions of the node for all states except state 0.

Consider a group with p nodes and s states. The raw cost of the nodes is $3*ps*4$ bytes. The group cost is $p*4$ to encode the group index for the nodes, plus $3*p*4$ for state 0, plus $(3+3)*(s-1)*4$ for the group's sequence of rigid body transformations, totaling $16p+24(s-1)$. The group brings storage savings for any $p>=3$. A group with 100 nodes is over 30 times more compact than its corresponding raw data, assuming 100 states. For large groups, the compression factor approaches $0.75s$, or 75 for 100 states and 750 for 1,000 states.

For the initial state, decoding straightforwardly returns the node position which is stored explicitly. For an intermediate state first one finds the group to which the node is assigned by using the group index. If the node is unassigned the position is directly looked up in the data containing unassigned nodes. If the node is assigned to a group, the rigid body transformation for the group and the current state are looked up and the node position is computed using Equation 3. This amounts to computing a rotation matrix from Euler angles, multiplying the position vector by the rotation matrix, and finally adding the translation to the result.

This method allows random node and state decoding queries. It is frequently the case of course that one decodes the positions of all nodes for a given state, when the Euler angles to rotation conversion is done only once per group, which results in an insignificant amortized cost. Moreover, the rigid body decomposition is well suited for graphics APIs which allow transforming all nodes in a group by placing the transformation for the current state on the model-view matrix stack.

5 Results and Discussion

We have applied our techniques to 4 datasets from different application domains. The first dataset, *Truck* (Figures 1 and

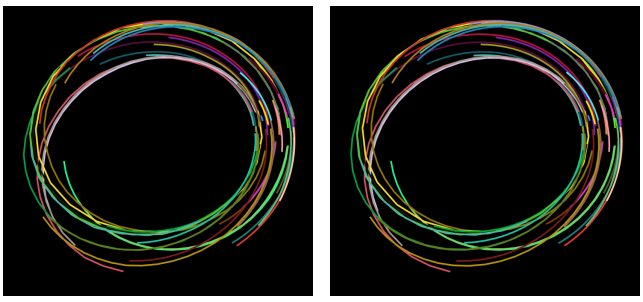


Fig. 7. Simplification of fusion simulation dataset using trajectory clustering (left) and the original dataset (right). The trajectories of only 50 randomly selected nodes are shown. Compression factor is 5:1 for a maximum error of 0.4%.

TABLE 1
Compression performance using optimal configurations for various datasets, methods, and error thresholds.

Error Threshold		ITS		TC+ITS			RBD+ITS		
Absolute [mm]	Relative [%]	Compression Factor	Average Error [mm]	Compression Factor	Clusters	Average Error [mm]	Compression Factor	Rigid Bodies	Average Error [mm]
Truck (26.6 MB uncompressed) 15m x 5m x 3m									
1	0.0066	1.8	0.86	2.4	8K	0.86	2.7	1,809	0.33
5	0.033	4.3	4	6.3	4K	4.1	7.0	573	1.0
10	0.066	6.4	7.8	9.8	4K	7.8	11	353	1.7
50	0.33	15	36	24.8	4K	36	25	139	6.4
100	0.66	22	61	33.8	2K	60	33	92	14
Airplane (643 MB uncompressed) 110m x 90m x 60m									
1	0.00091	4.0	0.76	4.1	128K	0.76	4.1	530	0.10
5	0.0045	8.7	3.4	9.6	64K	3.4	9.0	429	1.0
10	0.0091	12	6.3	14	64K	6.3	13	286	2.2
50	0.045	25	25	29	32K	25	27	51	11
100	0.091	34	41	39	32K	41	38	31	20
Liquid (9.4 GB uncompressed) 1m x 0.18m x 0.2m									
1	0.1	7.5	0.95	12	512K	0.94	—	—	—
5	0.5	19	4.5	31	512K	4.5	—	—	—
10	1	30	8.8	47	256K	8.8	—	—	—
Fusion (4.2 GB uncompressed) 2.5m x 2.5m x 0.5m									
1	0.04	1.5	0.89	1.7	128K	0.97	—	—	—
5	0.2	3.4	4.9	3.6	128K	4.9	—	—	—
10	0.4	4.4	9.7	5	128K	9.8	—	—	—

8), is a finite element analysis (FEA) simulation of a truck colliding with a barricade. The dataset also contained connectivity information which was used for the surface rendering, but the cost of storage was not included in our analysis. The simulation contains 28.5K nodes simulated over 80 states, and the simulation first state axis aligned bounding box (AABB) is 15m x 5m x 3m. The second dataset, *Airplane* (Figure 1), is also an FEA dataset containing 370K nodes in total, simulated over 170 states, including 60K smooth particle hydrodynamics (SPH) elements, with the first state AABB of 110m x 90m x 60m. Once again, this dataset contains surface connectivity information which was excluded from analysis. The third data set, *Liquid* (Figure 2), is a dam break simulation calculated using SPH containing over 2.1M nodes simulated across 360 states, with an AABB of 1m x 0.18m x 0.2m. The final dataset, *Fusion* (Figures 7 and 10), is a simulation of the magnetic field lines of fusion tokamak. This dataset contains 500K nodes simulated over 750 states, and an AABB of 2.5m x 2.5m x 0.5m. All datasets were simplified using small error thresholds, which results in visualizations that are difficult to distinguish from visualizations of the original data, as shown in the figures throughout this paper and in the accompanying video.

5.1 Individual Trajectory Simplification

Individual Trajectory Simplification (ITS) treats nodes independently, working on a single node trajectory at a time. The runtime memory required for a dataset with n nodes and s states is only the size of one node worth of data, or O(s). The construction time is n multiplied by the time needed to process a trajectory. The worst case asymptotic running time for ITS is O(ns^2), which corresponds to the case when simplified trajectories have O(s) points and when simplified trajectories grow from one endpoint to the other, one position at a time, requiring O(s^2) updates to the point errors. The expected running time is O($ns\ log\ s$), which corresponds to a case when simplified trajectories are constructed in more balanced fashion. The best case running time is O(ns) when all simplified trajectories have only 2 points, which still requires checking that the error is below the threshold at the intermediate positions. In practice, the running time was always below 40 minutes for all our tests and was usually in the 1 to 10 minute range. All running times reported in this paper were measured on a single processor, without any form of parallelism.

Table 1 gives the compression factors and the average errors achieved by ITS for our 4 datasets and for various node position error thresholds (also see Figure 8). The relative error threshold is computed as the absolute threshold value over the largest dimension of the AABB of the dataset, times 100. ITC achieves good compression of all datasets for these small relative error threshold values.

The average error is comparable to the error threshold (i.e. the maximum error). As expected, compression performance is very good for the Truck and Airplane datasets. We were

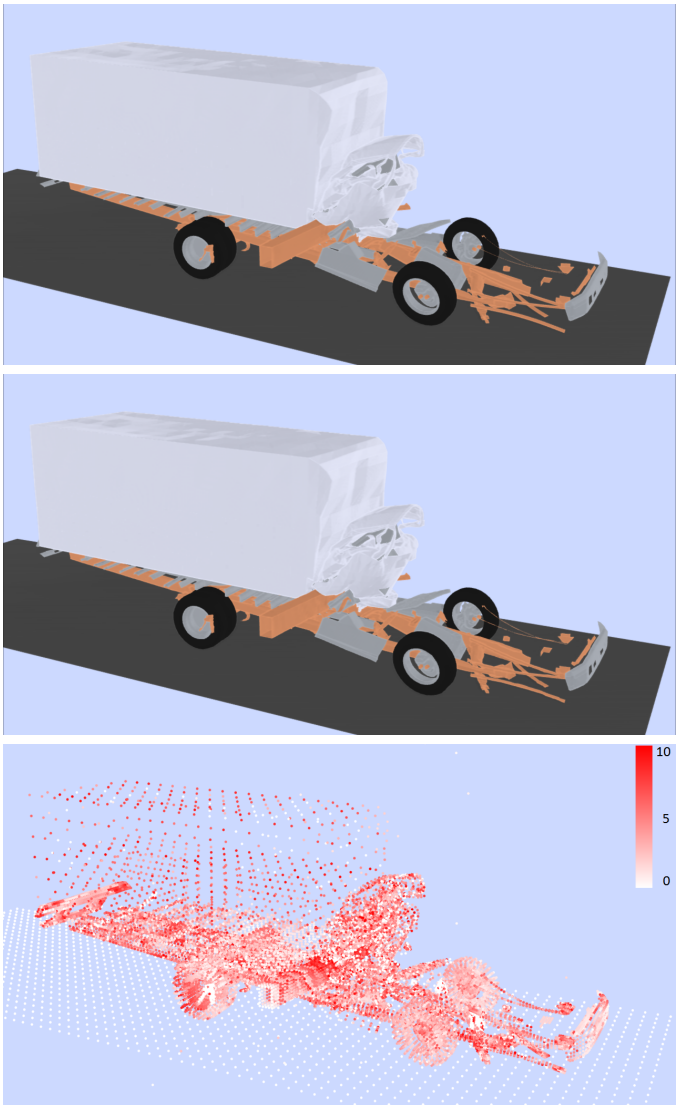


Fig. 8. Individual trajectory simplification on Truck dataset (middle) and original dataset (top). The error at each point is shown (bottom), while the maximum error is 10 mm for compression factor 12:1.

particularly pleased with the performance on the liquid simulation, where a relative error of 0.1% (1mm) was enforced with a compression factor of 7.5:1. As expected, the fusion dataset with its chaotic motion was the most challenging, but even there, ITS achieves a 4.4:1 compression factor for a relative error of 0.4% (10mm).

5.2 Trajectory Clustering

Trajectory clustering (TC) starts out with a single cluster with all n trajectories, which is then subdivided into two clusters repetitively. Subdividing a cluster with t trajectories takes O(ts) work and requires an O(ts) runtime memory footprint. Consequently, if the subdivision is balanced, completely subdividing the initial cluster to singleton clusters is done in O($ns\ log\ n$) time with shrinking runtime memory footprints of size O($ns * 2^{-level}$). An unlikely unbalanced subdivision takes O($n^2 s$) time. If data streaming is used, the runtime memory footprint can remain small at the cost of loading the data O($log\ n$) expected and O(n) worst case number of times. In practice, the running time was on the order of a few minutes and never exceeded 45 minutes.

Although TC could be trivially implemented to stop subdivision when an input error threshold is met by changing the termination condition in Line 5 of Algorithm FindClusters, doing so would generate an excessive number of clusters which hurts the compression factor. This is due to the fact that TC does not optimize the choice of basis trajectories for the subclusters. Much better compression factors are generated if TC is run up to a user specified number of clusters, followed by ITS on the cluster basis trajectories, and finally followed by ITS on individual node residual trajectories, where the residual trajectory is simply a trajectory containing the error for each simplified state.

The results of this TC+ITS method are given in Table 1 (also see Figures 1 (bottom), 7, and 10). The number of clusters in this case were chosen to be near-optimal, producing the highest compression ratio. TC+ITS further improves over the compression factors of ITS alone. For the Liquid dataset for example, the compression factor improves from 7.5 to 12 for the 0.1% threshold. TC+ITS average error is virtually identical to that of ITS, which is explained by the fact that it is ITS in both cases that brings the error of individual nodes below the threshold.

Choosing the optimal number of clusters is a challenging problem. We approached the problem by progressively increasing the number of clusters until the most compact representation was found. Fortunately, TC is a refinement process that can reuse previous results. For example, the result of 1K clusters can be used as input for determining 2K clusters. This means that when using TC+ITS, determining the ultimate storage cost at each progression only requires running through an additional ITS phase per progression.

TC compression performance depends on the number of clusters used, as seen in Figure 9. The dotted lines show the size of the dataset simplified using ITS alone. Solid lines correspond to TC+ITS. The error threshold is specified by the suffix of the name of the series. The graphs show that performance improves for a while with increasing numbers of clusters. More clusters model the trajectories better, allowing for smaller residual entropies in the clusters, which are more easily encoded by the subsequent ITS. The storage cost achieved by TC+ITS eventually dips below that achieved by ITS alone. However, once the number of clusters becomes too large, the overhead of the additional clusters starts to exceed the benefits they bring. Once a certain number of clusters is sufficient for each cluster to model the replaced trajectories well, further increasing the number of clusters only adds overhead without considerably reducing cluster entropy. Also as the number of clusters becomes large, the average number of trajectories per cluster becomes small, and the cluster payoff, which is dependent on the cluster basis trajectory replacing many trajectories, decreases.

5.3 Rigid Body Decomposition

Rigid body decomposition (RBD) seeds rigid bodies by finding triples of nodes (i, j, k) that move semi-rigidly together. As validating a candidate triple takes s work to check the triangle edge lengths at each state, an upper bound on the worst case performance is $O(n^3 s)$. However, many triples are trivially rejected when pairs of nodes (i, j) do not respect the rigid body constraint. It is difficult to imagine a dataset where any pair of nodes moves like a rigid segment yet no 3 nodes move like a rigid triangle. On the other hand, one can easily imagine the case when no pair of nodes moves rigidly, which provides a lower bound for the worst case of $\Omega(n^2 s)$. When the algorithm finds rigid bodies, performance is good. In the extreme case of a dataset with a single rigid body, the algorithm runs in $O(ns)$, as checking whether a node can be added to a rigid body takes $O(s)$ time. In practice, all Truck and Airplane simplifications took less than 1 and 70 minutes, respectively.

As for runtime memory requirements, optimal performance is achieved when the entire dataset is loaded into memory. The *RigidBodyDecomposition* algorithm scans through the entire dataset only once, hinting towards a data streaming method. However, the *NewRigidBody* algorithm also scans $O(n^2)$ times, and may be executed up to $O(n)$ times. Therefore, for the data streaming method, the worst case is that the dataset needs to be loaded $O(n^2)$ times.

To further improve the compression factor of RBD alone we run ITS on the unassigned nodes (note the difference with the TC+ITS method described above, which runs ITS on the residuals of all nodes). RBD+ITS performance is given in Table 1 (also see Figures 1 (top) and 2). Regarding compression factors, RBD+ITS improves over ITS, RBD+ITS improves slightly over TC+ITS for the Truck dataset, and TC+ITS has a slight edge on the Airplane dataset. RBD+ITS is well suited for datasets with spinning rigid bodies, which TC+ITS does not find. These could be mechanical parts (e.g. wheels) or debris resulting from the fracture of brittle materials. When the semi-rigid bodies are not spinning, the overhead of RBD+ITS is not warranted and TC+ITS should be preferred.

While somewhat counter intuitive, increasing the error threshold decreased the number of rigid bodies detected. One might expect a larger number of rigid bodies to be found with higher errors. Instead the size of the rigid bodies (i.e. number of elements) increased caused by a more or less merging of similar rigid bodies.

Regarding average errors, RBD+ITS clearly outperforms TC+ITS. The difference is even more pronounced when RBD is used alone, encoding the unassigned nodes with 0 error. For example the average error for RBD alone on the Airplane dataset with a threshold of 1mm is 0.079mm, almost an order of magnitude below the 0.76mm of ITS and TC+ITS. This is explained by the fact that simulations are divergent and a rigid body that barely passes the error test for the last state is likely to have much smaller errors at the beginning of the simulation. Table 1 also reports the number of rigid bodies which, as expected, decreases as the error threshold increases.

For the error thresholds that provide a good visualization, RBD did not find any rigid bodies in the Liquid and the Fusion

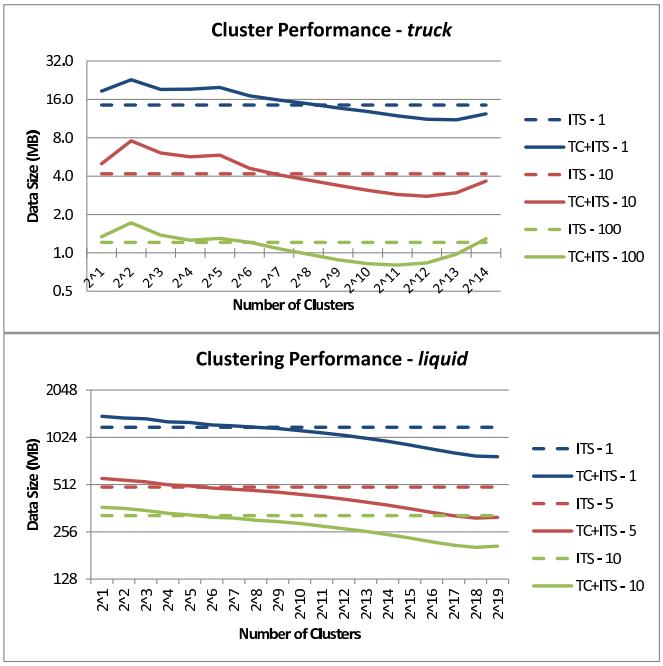


Fig. 9. Trajectory clustering performance as a function of the number of clusters.

datasets. This is expected, it is unlikely that 3 nodes move as a rigid triangle in these long and unstable simulations. TC+ITS did simplifying these datasets by allowing for occasional larger errors in clusters, which were subsequently eliminated by the ITS step on trajectory residuals. Running TC with a termination condition based on the error threshold would result in single trajectory clusters, which does not bring compression. In order to use RBD on the Liquid and Fusion datasets, one option is to run RBD with a larger threshold and then to reduce the error in a subsequent ITS step on the residuals, akin to TC+ITS. We chose a different option: we segmented the Liquid dataset into sequences of 10 states, as shorter sequences reduce the motion complexity. Figure 2 shows some of the rigid bodies found for an error threshold of 0.1% (1mm). As a rigid body decomposition stores the initial state of a sequence, the upper limit on the compression factor is 10:1 when 10-state sequences are used. The actual compression factor was 8.5:1, which is worse than the 12:1 that achieved by TC+ITS, but with a superior average error of 0.26mm as opposed to 0.94mm.

5.4 Limitations

Scientists and engineers are concerned about errors within their simulations since those errors can lead to incorrect decision making. Although all the techniques we have described are lossy, all of them provide a way of strictly controlling the error introduced.

All the algorithms described proceed in greedy fashion and are not guaranteed to produce the optimal solution. However, one cannot find the optimal solution to individual trajectory simplification, to trajectory clustering and to rigid body decomposition in a reasonable amount of time. We have

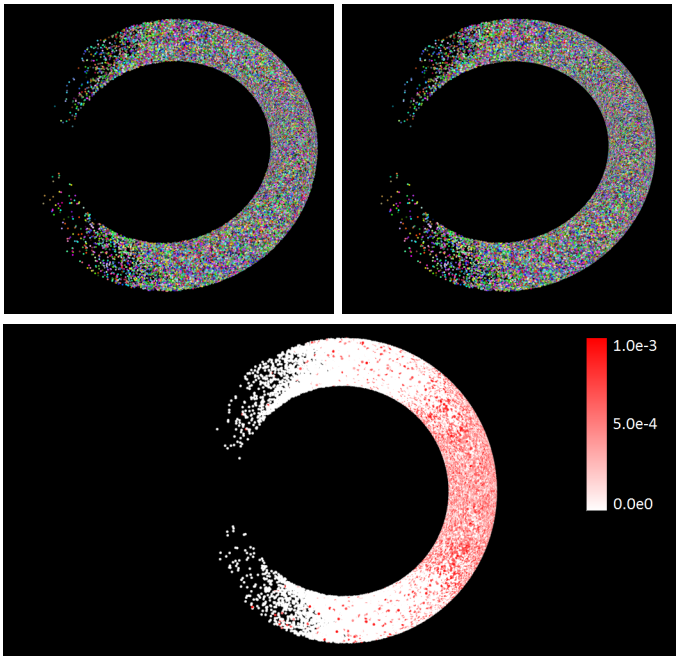


Fig. 10. Trajectory clustering simplification of Fusion dataset (top, left) and original (bottom, right). The error per node is show (bottom) for a compression factor of 5:1 with a maximum relative error of 0.4%.

shown that our algorithms produce good approximate solutions quickly on a variety of datasets.

Our work so far has been limited to node positions and has not considered additional node data (associate scalars, vectors, etc.), which could represent an important fraction of the total dataset size.

Finally, in their current form, our methods cannot be used to generate the best simplification of a dataset that fits a given storage resource.

6 Conclusions and Future Work

We have presented general techniques for simplifying node position data that achieve good compression factors with strictly-enforced, user-specified error thresholds. The encoding time is a fraction of the time it took to compute the simulations. Decoding is straightforward and well suited for graphics hardware. Visualizations based on the simplified datasets are virtually indistinguishable from those produced from the original datasets.

In terms of future work, these techniques can be readily integrated into applications to improve the interactivity of visualizations by reducing the storage and memory needs of large datasets. This can lead to interactive in-core visualization of datasets too large to fit in memory or significantly reduce the disk and network access needs of out-of-core visualizations. These techniques are also useful for sharing datasets, where simplified versions of large datasets can be transmitted across the Internet in relatively small amounts of time.

Another direction of future work is to improve the compression factors achieved by our techniques by integrating the benefits of complementary approaches such as unstructured point cloud and floating-point compression.

Finally, the core trajectory clustering and rigid body decomposition algorithms developed here in the context of compression and visualization could prove useful as a simulation data analysis tool.

Acknowledgments

Magnetic field data courtesy of Scott Kruger, TechX Corp and the DOE SciDAC Center of Extended MHD Modeling. Fluid simulation data courtesy of Adam Bargteil. This work was funded in part by Purdue Universitys PLM Center for Excellence, DoE VACET, and the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10.

References

[1] C. Johnson, T. Munzner, R. Moorhead, H. Pfister, P. Rheingans, and T. Yoo, "Nih-nsf visualization research challenges report summary," *IEEE Computer Graphics and Applications*, vol. 26, no. 2, pp. 20 – 24, 2006.

[2] M. Deering, "Geometry compression," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 13–20.

[3] S. Gumhold and W. Strasser, "Real time compression of triangle mesh connectivity," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 133–140.

[4] F. Kalberer, K. Polthier, U. Reitebuch, and M. Wardetzky, "Freelence - coding with free valences," in *Computer Graphics Forum (Eurographics 2005)*, 2005, pp. 469–478.

[5] Z. Karni and C. Gotsman, "Spectral compression of mesh geometry," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 279–286.

[6] J. Rossignac, "Edgebreaker: connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 47 –61, 1999.

[7] G. Taubin and J. Rossignac, "Geometric compression through topological surgery," *ACM Transactions on Graphics*, vol. 17, pp. 84–115, 1998.

[8] C. Touma and C. Gotsman, "Triangle mesh compression," in *Proceedings of the 24th annual graphics interface conference*, ser. GI '98, 1998, pp. 26–34.

[9] M. Isenburg and P. Alliez, "Compressing polygon mesh geometry with parallelogram prediction," in *IEEE Visualization*, ser. VIS 2002, 2002, pp. 141 –146.

[10] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schröder, "Near-optimal connectivity encoding of 2-manifold polygon meshes," *Graphics Models*, vol. 64, pp. 147–168, 2002.

[11] L. C. Aleardi, O. Devillers, and G. Schaeffer, "Optimal succinct representations of planar maps," in *Proceedings of the twenty-second annual symposium on Computational geometry*, ser. SCG '06. New York, NY, USA: ACM, 2006, pp. 309–318.

[12] P. Alliez and M. Desbrun, "Progressive compression for lossless transmission of triangle meshes," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 195–202.

[13] M. Ben-Chen and C. Gotsman, "On the optimality of spectral compression of mesh data," *ACM Transactions on Graphics*, vol. 24, pp. 60–80, 2005.

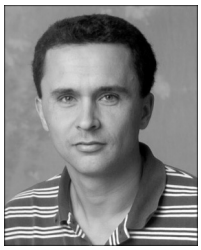
[14] Z. Karni, A. Bogomjakov, and C. Gotsman, "Efficient compression and rendering of multi-resolution meshes," in *Proceedings of the conference on Visualization '02*, ser. VIS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 347–354.

[15] U. Bischoff and J. Rossignac, "Tetstreamer: compressed back-to-front transmission of delaunay tetrahedra meshes," in *Data Compression Conference*, ser. DCC 2005, 2005, pp. 93 – 102.

[16] S. Gumhold, S. Guthe, and W. Strasser, "Tetrahedral mesh compression with the cut-border machine," in *Proceedings of the conference on Visualization '99: celebrating ten years*, ser. VIS '99. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 51–58.

[17] M. Isenburg and P. Alliez, "Compressing hexahedral volume meshes," in *10th Pacific Conference on Computer Graphics and Applications*, 2002, pp. 284 – 293.
[18] D. Chen, Y.-J. Chiang, and N. Memon, "Lossless compression of point-based 3d models," in *Pacific Graphics*, 2005, pp. 124–126.
[19] R. Lange, T. Farrell, F. Durr, and K. Rothermel, "Remote real-time trajectory simplification," in *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.
[20] T. W. Liao, "Clustering of time series data–a survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857 – 1874, 2005.
[21] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory clustering: a partition-and-group framework," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 593–604. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247546
[22] G. de Vries and M. van Someren, "Clustering vessel trajectories with alignment kernels under trajectory compression," in *European conference on Machine learning and knowledge discovery in databases*, ser. ECML PKDD'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 296–311.
[23] J. E. Lengyel, "Compression of time-dependent geometry," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, ser. I3D '99. New York, NY, USA: ACM, 1999, pp. 89–95.
[24] T. Winkler, J. Drieseberg, A. Hasenfuß, B. Hammer, and K. Hormann, "Thinning mesh animations," in *Proceedings of Vision, Modeling, and Visualization 2008*, 2008, pp. 149–158.
[25] M. Sattler, R. Sarlette, and R. Klein, "Simple and efficient compression of animation sequences," in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ser. SCA '05. New York, NY, USA: ACM, 2005, pp. 209–217.
[26] S. Shlafman, A. Tal, and S. Katz, "Metamorphosis of polyhedral surfaces using decomposition," *Computer Graphics Forum*, vol. 21, no. 3, pp. 219–228, 2002.
[27] D. L. James and C. D. Twigg, "Skinning mesh animations," *ACM Transactions on Graphics (SIGGRAPH 2005)*, vol. 24, pp. 399–407, July 2005.
[28] M. Alexa and W. Mller, "Representing animations by principal components," *Computer Graphics Forum*, vol. 19, no. 3, pp. 411–418, 2000.
[29] Z. Karni and C. Gotsman, "Compression of soft-body animation sequences," *Computers and Graphics*, vol. 28, pp. 25–34, 2004.
[30] E. Landreneau and S. Schaefer, "Simplification of articulated meshes," in *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 28, no. 2, 2009, pp. 347–353.
[31] A. Shamir and V. Pascucci, "Temporal and spatial level of details for dynamic meshes," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '01. New York, NY, USA: ACM, 2001, pp. 77–84.
[32] I. Guskov and A. Khodakovsky, "Wavelet compression of parametrically coherent mesh sequences," in *In Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2004.
[33] F. Payan and M. Antonini, "Wavelet-based compression of 3d mesh sequences," in *In Proceedings of the Second International Conference on Machine Intelligence*, 2005.
[34] J. Woodring and H.-W. Shen, "Multiscale time activity data exploration via temporal clustering visualization spreadsheet," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 123 –137, jan.-feb. 2009.
[35] L. Ibarria and J. Rossignac, "Dynapack: space-time compression of the 3d animations of triangle meshes with fixed connectivity," in *In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2003, pp. 126–135.
[36] O. Devillers and P.-M. Gandoin, "Geometric compression for interactive transmission," in *Proceedings of the conference on Visualization '00*, ser. VIS '00. Los Alamitos, CA, USA: IEEE Computer Society Press, 2000, pp. 319–326.
[37] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of predicted floating-point geometry," *Computer Aided Design*, vol. 37, pp. 869–877, 2005.
[38] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 1245–1250, 2006.
[39] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Data Compression Conference*, ser. DCC 2006, 2006, pp. 133 – 142.
[40] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, pp. 520–540, 1987.
[41] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244 – 256, 1972.
[42] D. Douglas and T. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.
[43] S. P. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, no. 28, pp. 129–137, 1982.

Paul Rosen completed his Ph.D. in computer science at Purdue University in 2010. He is currently a Research Assistant Professor in the Scientific Computing and Imaging Institute at the University of Utah. His research interests lie primarily in camera model design and its applications in computer graphics and visualization, but also computer vision and computer-human interaction. His interests also include high-performance computing, visualization of large-data simulations, software visualization, and uncertainty visualization.



Voicu Popescu received a B.S. degree in computer science from the Technical University of Cluj-Napoca, Romania in 1995, and a Ph.D. degree in computer science from the University of North Carolina at Chapel Hill, USA in 2001. He is an associate professor with the Computer Science Department of Purdue University. His research interests lie in the areas of computer graphics, computer vision, and visualization. His current projects include camera model design, perceptual evaluation of rendered imagery and 3D displays, research and development of 3D scene acquisition systems, and research, development, and assessment of next generation distance learning systems.