SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

November 2014, Vol. 57 112105:1–112105:19
doi: 10.1007/s11432-014-5116-6

Fast and compact dynamic data compression based on composite rigid body construction

MA ZhiQiang[1], WANG LiLi[1]*, ZHANG BoNing[1], KE Wei[2] & ZHAO QinPing[1]

[1]*State Key Laboratory of Virtual Reality Technology and Systems, School of Computer Science and Engineering, Beihang University, Beijing* 100191*, China;*
[2]*Macao Polytechnic Institute, Macao* 999078*, China*

Received March 24, 2014; accepted April 24, 2014; published online June 30, 2014

Abstract 3D dynamic datasets compression still poses two challenges. One is high time cost due to growing data and complex computation of compression algorithms. The other is low compression factor because of complex motions of dynamic scenes and unknown motion equations. In this paper, composite rigid body construction for fast and compact compression of 3D dynamic datasets is proposed to solve these two problems. It accelerates the compression with a fast rigid body decomposition based on disjoint union, and avoids serial searching, comparing and merging of the rigid body decomposition. To increase the compression factor, composite rigid body is introduced with consideration of motion consistency among rigid bodies at different time periods. The results of the experiments show that our algorithm compresses dynamic datasets quickly and achieves a high compression factor.

Keywords time-varying datasets, dynamic datasets compression, rigid body decomposition,disjoint union, composite rigid body construction

Citation Ma Z Q, Wang L L, Zhang B N, et al. Fast and compact dynamic data compression based on composite rigid body construction. Sci China Inf Sci, 2014, 57: 112105(19), doi: 10.1007/s11432-014-5116-6

1 Introduction

With the advent of the Internet era, as well as the wide use of mobile computing platforms, such as laptops, tablets, and smart phones, remote transmission and visualization have helped people make full use of data on the Internet, and this has become an irresistible trend. Whereas the growth of data obtained through observations and simulations is increasing much faster than the network bandwidth, data traffic with limited bandwidth has become the bottleneck of transmission and visualization. Data compression has provided a solution to this problem. There are two categories of data compression methods for remote transmission and visualization. One is based on image compression, which renders the scenes on the server, and sends the result images compressed with image or video coding to the clients. These methods pose the risk of long delay due to variable visualization parameters, such as viewpoints and looking directions, required by a huge number of clients. The other is based on 3D scenes compression. The term node is used to represent both a vertex in triangle-based scenes (Figure 1 row 1) and a particle in point-based scenes (row 2). For static scenes, these methods encode the positions and

*Corresponding author (email: wanglily@buaa.edu.cn)

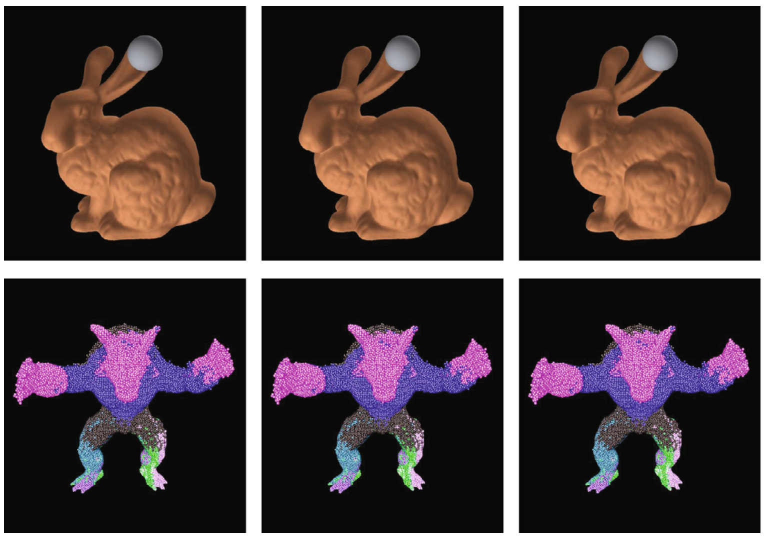

Figure 1 Our results (left), Rosen's RBD results (middle) and the original dataset (right) for comparison. The dataset sizes are 10 m × 9.8 m × 14.3 m for *Bunny*, 10 m × 8.3 m × 7.5 m for *Armadillo*, the maximum node error is 10 mm. The compression factors of our method are 49.8:1 and 9.3:1, and the time costs are 18 and 67 s for *Bunny* and *Armadillo*, while Rosen's are 10.2:1 and 2.9:1, 770 and 987 s. The different colors of *Armadillo* indicate the different composite rigid bodies generated by our method.

colors of nodes on the server, and reconstruct the 3D scenes on clients. They usually simplify the geometry for complex scenes before encoding. Clients have 3D scenes, and can render them with variable parameters, thus avoiding frequent interactions between clients and server when visualization parameters change. There are some other methods focusing on the node trajectory compression for time-varying data, but fast and compact compression is still a big challenge in the visualization field because of complex motions and unknown motion equations of the scenes.

Rigid body decomposition (RBD) [1] is proposed to merge nodes into rigid bodies according to motion consistency, and has achieved good compression factors for many kinds of dynamic scenes. Because of serial searching, comparing, and merging, RBD takes a long time for the scenes with hundreds of thousands of nodes. Another problem of RBD is missing motion consistency among the rigid bodies at different time periods. For some datasets, the number of rigid bodies is large, thus the compression factor is low.

In this paper we propose composite rigid body construction (CRBC) to provide a solution to both the problems above. CRBC can process all kinds of time-varying datasets handled by RBD. The first technique is rigid body decomposition based on disjoint union (DU-RBD). To avoid the serial searching, comparing and merging of RBD, DU-RBD employs parallel generation of small rigid bodies (SRBs) and a fast SRB combination based on disjoint union. DU-RBD accelerates the process of RBD. The second technique takes advantage of motion consistency for a period of time, and merges the rigid bodies (RBs) outputted from DU-RBD into composite rigid bodies (CRBs) according to their transformations. The use of CRBs reduces the number of transformation matrices that need to be transmitted and makes the compression more compact.

This method is tested on both triangle- and point-based scenes, which include significant movements of the elements. The compressed datasets including CRBs and node colors are transmitted to the client, where they are decoded to reconstruct the 3D dynamic scene. Figure 1 shows the reconstructed triangle- and point-based scenes. The maximum node error threshold is controlled to 0.1% (10 mm/10 m). Compared with Rosen's RBD method, CRBC increases the compression factor to 5 and 3.2 times, and reduces the time cost to 1/42 and 1/15 for *Bunny* and *Armadillo*. We also refer the reader to the attached video.

The remainder of the paper is organized as follows. In Section 2, the prior related work is discussed. The definition of CRB is given in Section 3 and its construction is discussed in detail in Section 4. Section 5 presents and discusses the results. Section 6 concludes the paper and sketches directions for future work.

2 Related work

A variety of compression methods for time-varying data have been developed. These methods can be classified as trajectory analysis, basis function methods, skinning decomposition, and motion vector methods.

Trajectory analysis methods analyze the features of the joints or vertices trajectories, and approximate them with parametric curves or polylines, which are clustered later for compact representations. Arikan [2] fits 3D cubic Bezier curves to the trajectory of points, and clusters the curves with clustered principle component analysis. Individual trajectory simplification [1] is proposed to simplify the trajectory curves into polylines and to cluster the trajectory with a greedy algorithm. Lange et al. [3] also use a continuous piecewise linear function to simplify the objects' trajectory in moving objects databases. Three alignment kernels such as shortest sequence alignment, edit distance alignment, and dynamic time warping alignment are provided in [4], which cluster the trajectories to be more compact. Using trajectories analysis methods on a scene with less coherence of trajectories, the compression factor is low.

Basis function methods project the positions of all vertices or clusters of vertices on some bases, and reveal the correlations of sets both in space and on the time axis. Then the data can be represented with the basis functions and only part of weights. Principal component analysis (PCA) [5–7] is used to project the dynamic data into a linear subspace, and to get the principal component bases to reduce the dimension of the data. Akhter et al. [8] present a bilinear spatiotemporal basis model to represent data as a linear combination of spatiotemporal sequences consisting of shape modes oscillating over time at key frequencies. It can quickly and efficiently label and denoise large databases of dense facial motion capture data. Another option of the basis function is the wavelet. Wavelet transformation for 3D mesh sequences and selection for optimized coefficients are proposed in [9–11].

The problem of approximating mesh animation with skinning decomposition was first addressed by James and Twigg [12]. It identifies the near-rigid structure of the data with mean shift method, and associates the triangles to the core bones. Vertices' trajectories are estimated from the bones' transformations. Landreneau and Schaefer [13] propose poisson optimization to reduce the number of the control points for deformation. Kavan et al. [14] introduce iterative coordinate descent optimization to quickly construct high-quality skinned approximations of arbitrary mesh animations. The weight reduction techniques [12–14] cannot get trade-off well between accuracy (skinning error) and performance. Le and Deng [15] propose a novel smooth skinning decomposition with rigid bone method to extract both rigid bone transformations and a sparse, convex bone vertex weight map from a set of example poses. Le and Deng [16] also introduce a two-layer structure including master bone and virtual bone to reduce the dense weights to sparse ones without quality decreased.

Motion vector methods partition the vertices or meshes into sets manually or automatically, and compute the motion vectors or transformation matrices, which are used to transfer the vertices or meshes from one time step to another. Lengyel [17] splits the vertices of the time-varying geometry stream into sets, and uses affine transformation to approximate the trajectories of the vertices. Shamir and Pascucci [18] present a multi-resolution structure based on time-dependent directed acyclic graph (TDAG) to simply deforming meshes. It separates the temporal deformation into low and high frequency motions, and simplification is acquired by extracting high frequency transformation matrices. Shlafman [19] decomposes the models into a small number of patches, and uses parameterizations method to map the patches onto a disk or cylinder, which can represent the deformation of the scene. Motion vector computed for each vertex is introduced in [20]. The positions of neighborhood within a distance of a vertex are used to predict its motion vector. Ibarria and Rossignac [21] also predict vertex's motion vector by its relative coordinates from the previous frame. They introduce two space-time predictors to exploit the inter-frame coherence. An octree-based motion representation method is proposed in [22], which can represent the dynamic sequence of the scene with a reduced set of motion vectors. Rigid body decomposition is the technique that clusters the vertices into groups and uses sequences of rigid body transformations to approximate the motions of the clusters [1]. RBD does compact compression even for the scene with large scale deformations or motions, but the global searching and merging for RB construction are rather time-consuming. Besides, RBD doesn't consider the motion consistency among the vertices at different time periods, which would help us improve the compression factor further.

Our method belongs to the motion vector methods and is based on RBD. We change the global searching and merging of RBD into local ones, and construct SRBs in parallel. The SRBs are combined into RBs with disjoint union, which change time complexity from $O(m^2)$ to linear, where m is the number of nodes. We also introduce a method for considering motion consistency at different time periods, and generating a more compact representation named composite rigid body.

3 Composite rigid body definition

The definition and construction of RB in RBD algorithm are first introduced, then we define CRB.

3.1 RB definition

RBD algorithm [1] groups nodes with similar trajectory into rigid bodies. A rigid body is a cluster of at least 3 nodes whose motions can be approximated with a single sequence of rigid body transformations. Instead of storing each node's trajectory, rigid body only stores the initial position of each node and the sequence of transformations. So a rigid body with v nodes ($v \geqslant 3$) is defined as RB(P_1,Q):

• P_1=$\{(x_1, y_1, z_1)_1, (x_2, y_2, z_2)_1, \dots, (x_v, y_v, z_v)_1\}$is the set of initial positions of v nodes;

• Q=$\{q_2, q_3, \dots, q_s\}$ is a set of $s-1$ transformations of the rigid body, and s is the number of time steps.

The rigid body decomposition approximates the position of node i in the rigid body rb_r at time step f according to

$$P_i^f = q_f(x_i, y_i, z_i)_1^{\mathrm{T}}, \tag{1}$$

where q_f is rb_r's transformations at time step f, and the initial position of P_i is $(x_i, y_i, z_i)_1$.

3.2 RB construction

A dynamic scene including m nodes is shown in Figure 2 left. For a node i, RBD first tests if i can be merged into any existing rigid body. Eq. (2) is used to decide if the node i can be merged into a RB rb$_r$.

$$\mathrm{rb}_r \cup \{i\} = \begin{cases} 1, \text{if } \mathrm{Err}(i, Q_r) \leqslant \varepsilon, \\ 0, \text{if } \mathrm{Err}(i, Q_r) > \varepsilon, \end{cases} \tag{2}$$

where ε is the error threshold, Q_r is rb_r's transformations over s time steps and $\mathrm{Err}(i, Q_r)$ returns maximum node position error of i under Q_r, which can be computed by

$$\mathrm{Err}(i, Q_r) = \max(\mathrm{fabs}(P_i^f - P_i^{f'})) \quad (1 < f \leqslant s), \tag{3}$$

where $P_i^{f'}$ is node i's original position at the time step f, and P_i^f is i's position computed with rb$_r$'s transformation q_f in (1). If $\mathrm{Err}(i, Q_r)$ is not bigger than ε, node i can be merged into the RB rb$_r$. If not, other two nodes j and k in the remaining nodes need to be found to construct a original RB with three nodes. The node j is first to be found with the condition that the distance between i and j over s time steps remains approximately the same, which means the distance variations of ij between the first time step and other time steps are always smaller than the client-defined error threshold ε. Then RBD finds node k whose distance variations to the nodes i and j also remain within ε over s time steps. Once the original RB with three nodes i, j, k is generated, RBD computes the RB's sequence transformation matrices $q_2, q_3, \dots, q_s$ corresponding to the initial 3D positions of nodes in the RB. In Figure 2 (right), the RB(i, j, k) moves to 3D positions (i', j', k') at the time step f. A transformation q_f is computed by taking i's position to i''s position, with a rotation that aligns the planes of triangle ijk and $i'j'k'$, and with a rotation about the normal of the triangle plane that aligns edges ij and $i'j'$.

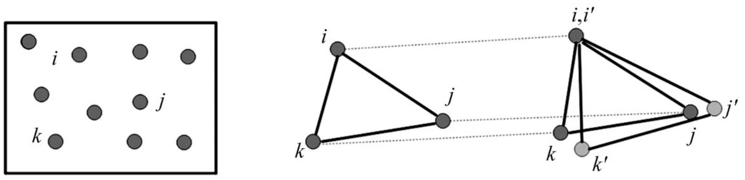

Figure 2 RB construction in Rosen's RBD algorithm.

The search of two nodes j and k in original RB construction is a kind of serial search. If adjacent nodes have low motion consistency, a lot of time will be used to construct the original RB, especially if no nodes j, k can be found.

3.3 CRB definition

There is still some motion redundancy in rigid body organization. Nodes in the same rigid body are required to have motion consistency over all s time steps. However, in many cases, nodes in different rigid bodies also have motion consistency at different time periods within s time steps. The merging of rigid bodies with motion consistency in a period of time can improve compression factor efficiently. For example, n rigid bodies have motion consistency from time step 1 to $s/2$. Without rigid body merging, $n{\times}(s-1)$ transformations are needed to be stored over s time steps. Whereas with the merging, only $(s/2-1)+n\times s/2$ transformations are needed, i.e., from time step 1 to $s/2$, there are only $s/2-1$ stored transformations with rigid body merging compared with $n\times(s/2-1)$ stored transformations without rigid body merging. To make rigid body more compact, we introduce composite rigid body to merge rigid bodies according to their motion consistency at different time periods within s time steps. CRB construction uses 3D node positions of all time steps as inputs, which are the same as that of RBD. The outputs are composite rigid bodies. We define a composite rigid body as CRB(t_s,t_e,RBset,Q):

- t_s is the index of starting time step for the CRB;
- t_e is the index of ending time step for the CRB;
- RBset=$\{\mathrm{rb}_1,\mathrm{rb}_2,\ldots,\mathrm{rb}_n\}$ is a set of RBs whose motions can be approximated with the transformations Q of rb_1 from t_s to t_e;
- Q=$\{q_{t_s},\ldots,q_{t_e}\}$ is a set of (t_e-t_s+1) sequences of rb_1's transformations from t_s to t_e.

4 Composite rigid body construction

The composite rigid body is constructed with the following two major steps: 1) rigid body decomposition based on disjoint union; 2) composite rigid body generation.

Figure 3 shows the overview of composite rigid body construction. Firstly, rigid body decomposition based on disjoint union is introduced to accelerate the rigid body generation of the scene. Multiple RBDs are explored on the nodes located on the local regions of the scene in parallel, and the output SRBs are combined to generate the RBs quickly with disjoint union. The details of the first step are discussed in Subsection 4.1. Secondly, a CRB generation algorithm is proposed to consider motion consistency among the rigid bodies at different time periods. All rigid bodies of the scene are initialized as one CRB at the first time step. For a given time step f, we check whether all rigid bodies in the same CRB of the last time step $f-1$ can still maintain the rigid body motion. If not, this CRB will be divided into some children CRBs at the time step f. In the process of CRB generation, each CRB has its own lifetime from t_s to t_e. These CRBs give us partitions of RBs, and their lifetimes define the starting and ending time steps of the CRBs. We illustrate the second step with details in Subsection 4.2. The nodes that cannot be combined into the existing SRBs or RBs are called unassigned nodes(UNs). UN compressions 1 and 2 are used to make UNs more compact and to improve compression factor.

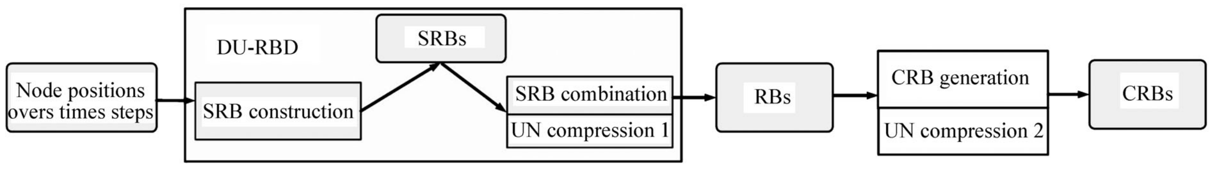

Figure 3 Overview of composite rigid body construction.

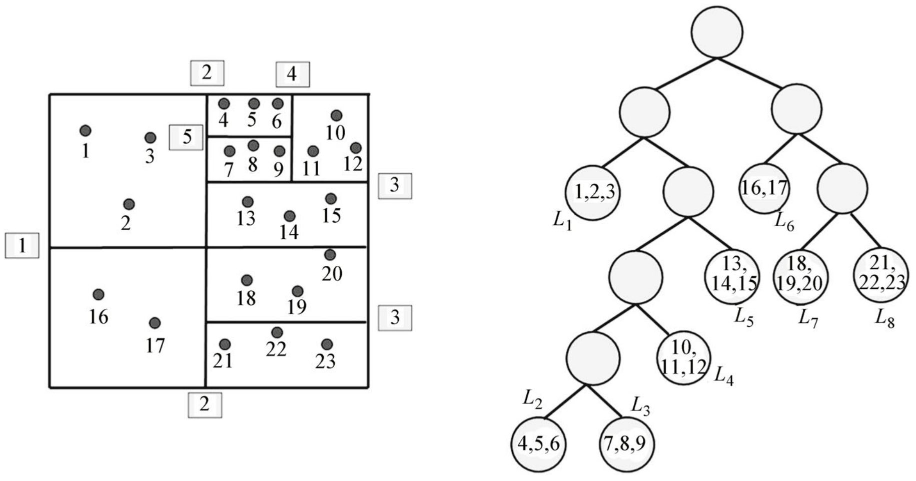

Figure 4 KD-tree division for 3D dynamic scene.

4.1 Rigid body decomposition based on disjoint union

Rigid body decomposition based on disjoint union is introduced to accelerate the process of rigid body decomposition [1]. RBD generates rigid bodies of scenes with a brute force method. If the scene has m nodes, which can be decomposed into n rigid bodies, it takes $O(nm)$ time to determine whether the nodes can be combined into the existing RBs or a new RB needs to be generated. Moreover, it takes $O(m^2)$ time to construct a new rigid body with three nodes. We accelerate RBD with two sub-steps: 1) SRB construction, in which the SRBs of the scene are generated in parallel, avoiding a lot of global searches in RBD. 2) SRB combination, which uses a fast algorithm based on disjoint union to combine those SRBs into the RBs of the scene quickly. The details of these two sub-steps are introduced in the following sections.

4.1.1 *Small rigid body construction*

Small rigid body is a rigid body that has the nodes only in local region of the scene. SRB construction is the process to generate SRBs for the scene in parallel. The small rigid body is constructed with the following two steps: 1) KD-tree creation for 3D dynamic scene; 2) small rigid body construction.

Step 1 A KD-tree is constructed according to the 3D positions of the nodes at the first time step. A predefined constant k is used to limit the maximum number of nodes in each leaf of the tree, and balance the sizes of the leaves. We first create axis aligned bounding box (AABB) for the dynamic scene at the first time step. Then the 3D scene is divided with KD-tree until the number of nodes in each leaf is smaller than k. Figure 4 (left) shows the 2D result of KD-tree division. k is set to 3, and the numbers from 1 to 5 in box represent division times. A KD-tree is constructed in Figure 4 (right). We don't use octree to divide the scene because when compared with KD-tree, octree possibly divides the scene into more small leaves with less than three nodes, and generates more UNs that should be constructed into SRBs. More UNs lower compression speed.

Step 2 After building a KD-tree for 3D dynamic scene, we construct SRBs in the leaves that have at least three nodes. In Figure 4 (right), the number of nodes in leaf L_6 is less than 3, so L_6 cannot be used to construct a SRB and its nodes are defined as UNs. The SRB constructions for leaves are independent,

Algorithm 1 Merge( )

```
Input: $\mathrm{srb}_i$, $\mathrm{srb}_j$
Output: Flag
1 Flag=true;
2 for each node $w$ in $\mathrm{srb}_j$ do
3     if !($\mathrm{srb}_i \cup \{w\}$) do
4         Flag=false;
5     end
6 end
7 if Flag then
8     $\mathrm{srb}_i$=$\mathrm{srb}_i \cup \mathrm{srb}_j$;
9     $\mathrm{srb}_i.v$=$\mathrm{srb}_i.v$+$\mathrm{srb}_i.u$;
10 end
11 return Flag
```

so they can be done in parallel. We implement SRB construction in parallel on CUDA. Each leaf with three or more nodes is allocated a thread. In each thread, we first initialize an original RB with three nodes. If the original RB can be constructed, each remaining node in the thread is merged into the RB by using (2) and (3) to output a SRB. If the RB construction fails, all the nodes in the leaf are set to UNs. After SRBs construction, we can get SRBs and some UNs. The structure of the SRB is the same as that of the RB discussed in Section 3.

SRB construction is faster than RBD because of two reasons. One is that SRB construction generates SRBs for leaves in parallel, whereas RBD generates RBs in turn. The other is that the construction of original RB in RBD, which needs to find three nodes in all the remaining nodes of the entire scene rather than our one leaf needs plenty of time.

4.1.2 *SRB combination*

SRB construction is fast, but the large number of output SRBs with transformation matrices will decrease the compression factor. Since many SRBs still have motion consistency, SRB combination is introduced to merge these output SRBs into RBs quickly to increase compression factor. SRB combination is executed with the following steps: 1) mapping creation between SRB leaves of KD-tree and uniform grid cells; 2) small rigid body combination based on disjoint union; 3) unassigned nodes compression 1.

Step 1 A 3D uniform grid is introduced into SRB combination. If we combine SRB by traversing the leaves of KD-tree directly, the adjacency relations of SRBs at some dimensions are lost. Compared with 3D uniform grid, disjoint union on KD-tree's leaf leads to more RBs and decrease compression factor when adjacent SRBs have high motion consistency. Thus, before feeding the SRBs into the combination process, we relate the SRBs with the cells of a 3D uniform grid ($64\times64\times64$ grid cells are used in our experiments), which organizes the SRBs into a straightforward way to show their 3D distributions and provides a simple way to do the operations of disjoint union on these SRBs while considering their adjacent relations.

To relate SRBs to the cells of the grid, the mapping from the leaves of the KD-tree to the grid cells is created. The first node of the SRB inside a leaf is taken as the representative node of the leaf. The cell where the representative node locates is mapped to the leaf, i.e. the SRB inside this leaf is related to the cell. If there are more than one SRB related to the same cell, the first SRB is kept, and the other SRBs are tested to see if they can be merged into the first one. Function Merge( ) in Algorithm 1 is used to test if one SRB srb_j can be merged into the other SRB srb_i. The SRB srb_j including u nodes can be merged into srb_i including v nodes with the condition that all nodes in srb_j can be merged into srb_i. Because SRB has the same structure with RB, (2) and (3) are also used to merge a node w in srb_j into srb_i. If Merge( ) returns true, all u nodes of srb_j are added into srb_i. Otherwise, the nodes in srb_j are discarded as unassigned nodes.

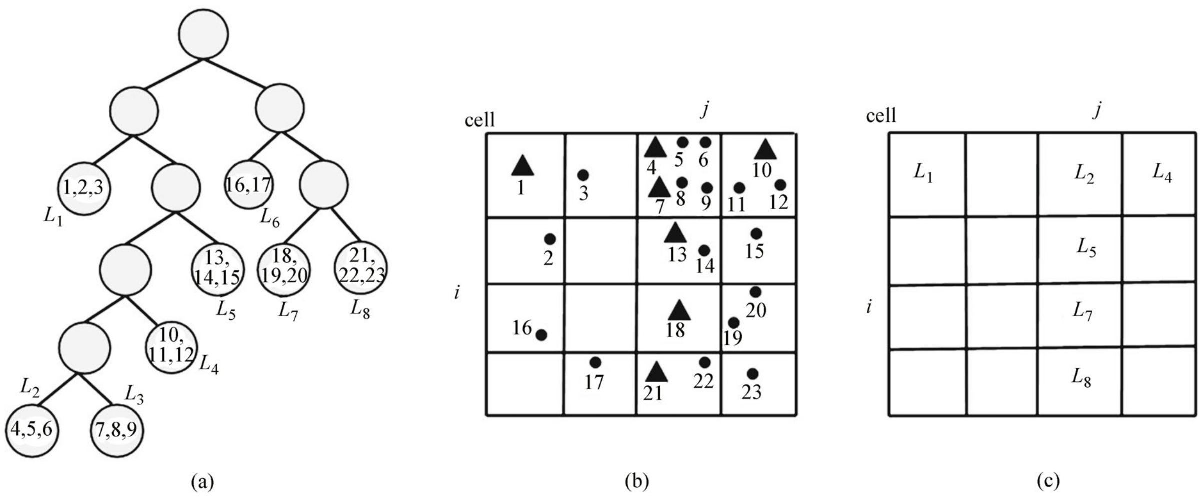

Figure 5 The creation of mapping between the SRB leaves of KD-tree and the cells of uniform grid.

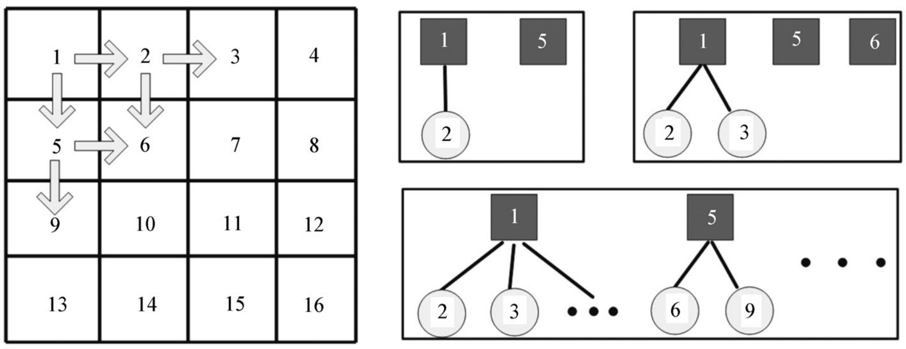

Figure 6 A uniform grid for SRB combination is shown (left), and each cell contains one SRB. The arrow indicates the right and bottom neighbors of srb_1, srb_2, and srb_5. SRB combination is executed when we move on to srb_1, srb_2, and srb_5 (right). The final quadrilateral SRBs are the RBs by SRB combination.

Figure 5 shows the mapping creation from the SRB leaves of KD-tree to the uniform grid cells. We map the leaves with SRB to the grid cells according to the positions of the representative nodes in the leaves (the first number in Figure 5(a) and triangle representation in Figure 5(b)). Because node 1 is the representative node of L_1 and locates in cell(1,1) (Figure 5(b)), L_1 is mapped to cell(1,1) (Figure 5(c)). Likewise, L_4 maps to cell(1,4), L_5 maps to cell(2,3), L_7 maps to cell(3,3), and L_8 maps to cell(4,3). Representative nodes 4 of L_2 and 7 of L_3 both locate in cell(1,3). But only the mapping between L_2 and cell(1,3) is kept because the first related SRB is in L_2. Function Merge( ) is used to test if the SRB in L_3 can be merged into the SRB of L_2. If Merge() returns true, all nodes in L_3's SRB are added into L_2's SRB. Otherwise, the nodes in L_3's SRB are discarded as UNs. L_6 contains only 2 nodes, which cannot be used to generate a SRB. So nodes 16 and 17 in L_6 are considered as UNs.

Step 2 In SRB combination, a fast algorithm based on disjoint union is proposed to combine SRBs into the RBs for the scene quickly. Figure 6 shows a 2D example of SRB combination. The left figure is the uniform grid generated with the SRB construction in Step 1. To be more comprehensible, each cell is supposed to map to a leaf of KD-tree and relates one SRB, from srb_1 to srb_{16}. The neighbor finding and union operations are executed in row major order for SRBs. For each cell with srb_i, we find its right and bottom cells to see if their srb_j can be combined with srb_i. Starting from srb_1 in Figure 6, first we find its right neighbor srb_2, and use Function Merge( ) to test whether srb_2 can be merged into srb_1. If Merge( ) returns true, srb_2 is merged into srb_1, also taking srb_1 as its parent. After that, the bottom neighbor srb_5 is tested. If srb_5 cannot be merged, it would be left. Then we move on to srb_2. When we test if srb_3 can be merged with srb_2, we compare srb_3 with srb_2's parent srb_1. If srb_3 can be merged with

Algorithm 2 SRB combination

```
Input: cells of the grid
Output: rigid body array RB
1 for each cell($i, j, k$) do
2   if cell($i, j, k$).srb!=null then
3     cell($i, j, k$).parentID=($i, j, k$);
4     cell($i, j, k$).rank=cell($i, j, k$).srbNodeNum;
5   end
6 end
7 for each cell($i, j, k$) do
8     if cell($i, j, k$).srb!=null then
9     Union(cell($i, j, k$), cell($i + 1, j, k$)); Union(cell($i, j, k$), cell($i, j + 1, k$)); Union(cell($i, j, k$), cell($i, j, k + 1$));
10    end
11 end
12 Save(RB,cell);
```

srb$_1$, srb$_3$' parent is also set to srb$_1$. If srb$_6$ cannot be merged with srb$_2$, it would be left. When moving on to srb$_5$, its right neighbor srb$_6$ is tested for merging. srb$_6$ is combined with srb$_5$ if Merge( ) returns true. The same is for srb$_9$. In Figure 6 (right), the final quadrilateral SRBs are the RBs generated by SRB combination.

Given cells of the grid, the SRB combination is computed with Algorithm 2. Each cell is a structure with four elements (srb, srbNodeNum, parentID, rank):

- srb is the merged SRB and its initial value is the cell's related SRB;
- srbNodeNum is a constant and its value is the node number of cell's related SRB;
- parentID is the index vector that refers to the cell's parent;
- rank is the node number of the merged SRB srb.

The algorithm considers each cell in turn. It first makes the cell's parentID point to itself, and initializes its rank with srbNodeNum. Then, for each cell with SRB inside, the algorithm combines the cell with the cells on the right, back, and bottom. After all cells are processed by these union operations, Save( ) function copies the cells' srb with non-zero rank to RB array for final results.

Union( ) function of two cells is defined in Algorithm 3. For two given cells, Union( ) first compares the cells' parentID. If they are the same, which indicates that these two cells are already merged with union operations, it returns directly. Otherwise, we need to determine whether the srb in the cell with smaller rank minrank_root can be merged into the srb in the cell with a bigger rank maxrank_root by using the function Merge( ). If Merge( ) returns true, it means that the srb of maxrank_root has been updated to contain all the nodes of srb inside minrank_root. The rank of maxrank_root is also updated to add the rank of minrank_root, and the rank of minrank_root changes to zero. The cell with smaller rank takes the parentID of maxrank_root as its parentID.

Step 3 The number of unassigned nodes produced with DU-RBD is much larger than the result of RBD, which is due to two reasons. One is that only the nodes in the same leaf of the KD-tree are tested to generate SRB, the other is that some SRBs are discarded in SRB combination, whose nodes are left as unassigned nodes. To reduce the number of unassigned nodes, we first merge them into the existing RBs. Then the nodes still unassigned are processed with RBD to generate new RBs. To accelerate this process, for a given node, a predefined radius is used to limit the searching space when the rigid body is constructed. After this, the number of unassigned nodes is almost the same as the results of Rosen's RBD method. We refer this process to reduce the unassigned nodes generated from DU-RBD as UN compression 1 in this paper.

Figure 7 shows the SRBs and RBs for the deformable bunny dataset. Because SRB is generated with the nodes inside a leaf of the KD-tree, there are a lot of SRBs generated in small sizes (left). Therefore time consumption reduces because of the parallel construction for these small SRBs. But the large number

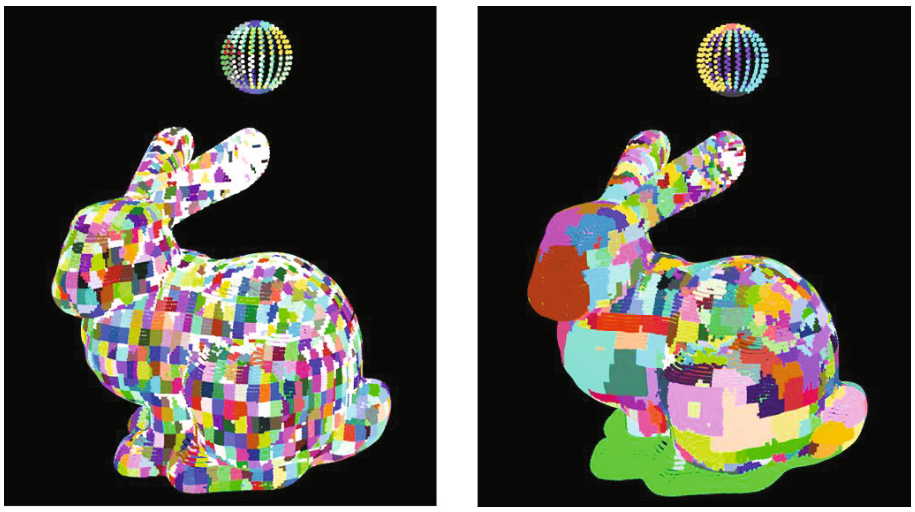

Figure 7 SRBs constructed with KD-tree(left) and RBs generated with SRB combination(right) are shown with random colors.

Algorithm 3 Union(cell(i, j, k), cell(x, y, z))

```
1 $(a_1, b_1, c_1)$=cell($i, j, k$).parentID;
2 $(a_2, b_2, c_2)$=cell($x, y, z$).parentID;
3 root1= cell($a_1, b_1, c_1$);
4 root2=cell($a_2, b_2, c_2$);
5 if root1== root2 then
6     return
7 end
8 if cell($x, y, z$).srb!= null then
9     minrank_root=minRank(root1,root2);
10    maxrank_root=maxRank(root1,root2);
11    if (maxrank_root.srb).Merge(minrank_root.srb) then
12        maxrank_root.rank=maxrank_root.rank+minrank_root.rank;
13        minrank_root.rank=0;
14        minrank_root.parentID=maxrank_root.parentID;
15    end
16 end
```

of SRB will decrease the compression factor, and a fast SRB combination is used to merge SRB into RB (right), which make the representation of the data more compact.

4.2 Composite rigid body generation

In RBD method [1], even if the two groups of nodes only have a different transformation at the last time step, they are considered as two separate RBs. The same transformations of the two RBs of the time steps before the last time step are transferred twice. To avoid this problem, composite rigid bodies are generated to consider the motion consistencies of RBs at different time periods.

Consider n rigid bodies constructed with DU-RBD. The goal of composite rigid body generation is to classify the rigid bodies into groups by motion consistency at different time periods, i.e., from a starting time to an ending time. The motion of the rigid bodies in the same group can be approximated well with a sequence of rigid body transformations.

The idea of our CRB generation algorithm is explained with an example in Figure 8. In this case, DU-RBD outputs 8 RBs, which are referred as rb_1 to rb_8. At the first time step, all these 8 RBs are considered as one CRB crb_1, whose starting time is 1. crb_1 remains the first three time steps, the trajectories

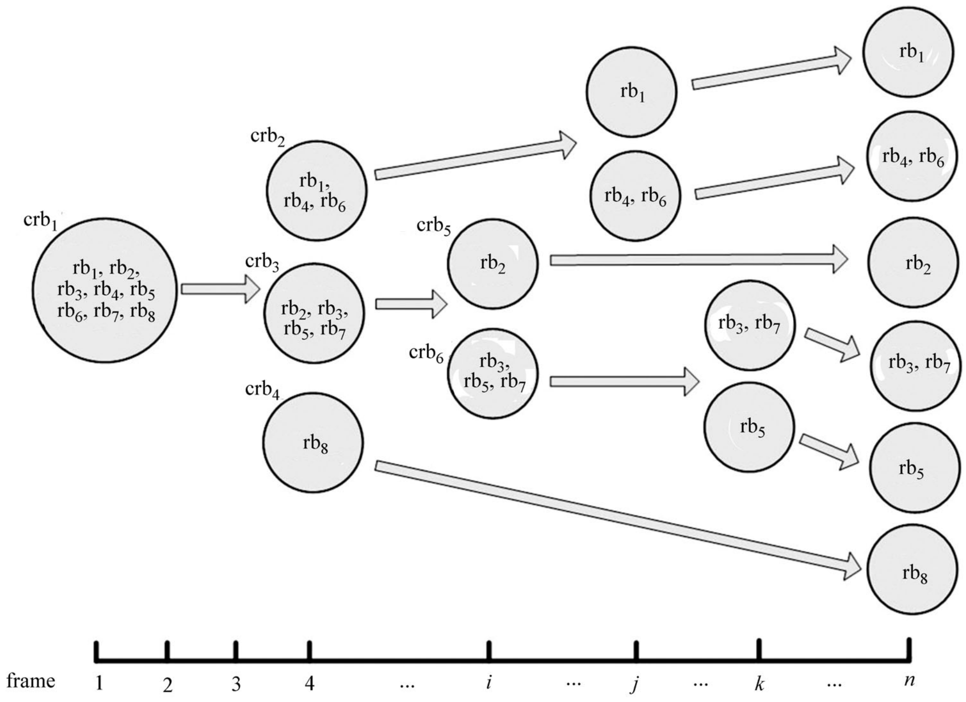

Figure 8 An example of division function for composite rigid body generation.

of rb_1 to rb_8 can be approximated with their positions in the first time step and rigid body transformation matrices of rb_1. Then at the time step 4, this CRB is divided into 3 new CRBs, namely crb_2, crb_3, and crb_4, which means the position errors of some RBs such as rb_2, rb_3, rb_5, rb_7 and rb_8, are bigger than predefined thresholds when we still use rigid body transformation matrices of rb_1 to compute their positions. Thus crb_1's ending time is set to 4, and three new CRBs' starting time steps are set to 4. This process recurs to each CRB until the total number of existing CRBs becomes 8 or the process reaches the last time step.

The algorithm of composite rigid body generation is in Algorithm 4. rb_1, rb_2, $\ldots$, rb_n refer to n rigid bodies, and frameNum refers to the total number of the time steps. In the initialization part, a CRB crb_1 is used to contain all RBs inputs, and its starting time is set to the first time step. Then, crb_1 is added into a queue q1. In the main process, for each CRB in q1, we check whether it can be divided into some new CRBs. There are two cases that the CRB does not need to be divided. One is when this CRB has only one RB inside, and the other is when the current time step is the last time step input. When either of these happens, this CRB is added into CRB_set directly, and its ending time is set to the last time step. Otherwise, this CRB enters the division process CRB_division() with its position information of the next time step. The function CRB_division() returns the number of the new CRBs generated, and an array newCRB is used to store the new CRBs. Consequently, if the number of the new CRBs is more than one, which means the CRB starts to be divided, all those new CRBs are added into a new queue q2, and their starting time steps are set to the next time step $f + 1$. At the same time, the original CRB is added into CRB_set, also its ending time is set to $f + 1$. This main process repeats for every time step. It will stop early only if the number of new CRB reaches to n. Figure 9 shows the CRBs in random colors of the corresponding time steps.

The CRB is divided and the new CRBs are constructed using Algorithm 5 and Algorithm 6. Algorithm 5 constructs the first new CRB with the first RB of the input CRB. Then, for other RBs inside the input CRB, we need to test whether they can be merged into the new CRBs generated. The first RB is used as the representation of a new CRB. Therefore, if the function (newCRB.rb(1)). Merge( ) returns true, which means the position of undetermined RB at time step f can be approximated by the transformation matrix of the first RB of this new CRB, the undetermined RB is added to this new CRB. Then we move on to the other remaining RBs. For any given RB, if it cannot be merged into any one of the new CRBs, a new CRB will be constructed to contain this RB. The algorithm puts all new CRBs into the array

Algorithm 4 Composite rigid body generation

```
Input: $\mathrm{rb}_1,\ldots,\mathrm{rb}_n$, frameNum
Output: CRB_set
1 $\mathrm{crb}_1$.RBset={$\mathrm{rb}_1,\ldots,\mathrm{rb}_n$};    $\mathrm{crb}_1.t_s = 1$;
2 q1 = null;    CRB_set={ };
3 q1.push_back($\mathrm{crb}_1$);    q1.number = 1;
4 for $f$ = 1 to frameNum do
5   q2 = null;
6   wihle q1.number > 0 do
7     crb=q1.pop_front();
8     q1.number = q1.number −1;
9     if crb.rbNo==1 || f==frameNum then
10        crb.$t_e$=frameNum;
11        CRB_set+={crb};
12    end
13    else
14        CRBNo=CRB_division(crb,$f$+1,&newCRB);
15        q2= Newqueue(CRBNo, newCRB);
16        CRB_set+={crb};
17      end
18    end
19    q1 =q2;
20 end
```

Algorithm 5 CRB_division(crb,f,&newCRB)

```
1 classfy=true;    newCRBno=0;
2 constructNewCRB(newCRBno,1);
3 for $i$=2 to crb.rbNo do
4   for $j$ =1 to newCRBno do
5     if (newCRB($j$).rb(1)).Merge(crb.rb($i$), $f$) then
6       $t$=++newCRB($j$).rbNo;
7       newCRB($j$).rb($t$)=crb.rb($i$);
8       classfy=false;
9       break;
10    end
11  end
12  if classfy then
13    constructNewCRB(newCRBno,$i$);
14  end
15 end
16 return newCRBno
```

Algorithm 6 constructNewCRB(newCRBno,i)

```
1 newCRBno=newCRBno+1;
2 newCRB(newCRBno).rb(1)=crb.rb($i$);
3 newCRB(newCRBno).rbNo=1;
```

newCRB, and the number of these new CRBs as an output is returned.

To improve the compression factor, we merge the remaining UNs into CRBs after CRB generation. We refer this process as UN compression 2. For each unassigned node p, first we determine whether it can be merged into crb_1 during its lifetime (see in Figure 8). If it could be merged, we store the index 1 for p, and test whether p could be merged into the CRBs that are divided from crb_1, such as crb_2, crb_3,

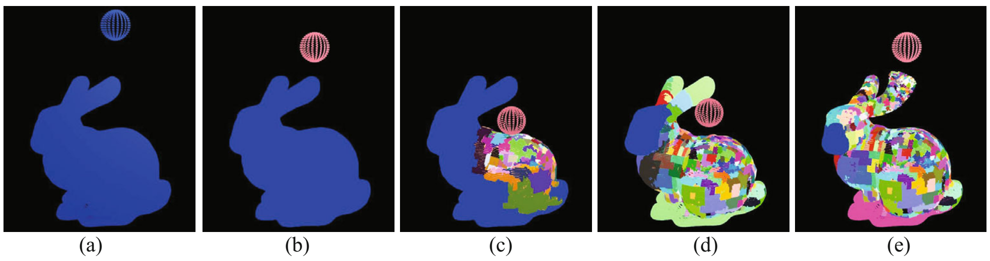

Figure 9 At the first time step, all RBs are in one CRB with blue color. At time step 190, the ball falls off, and one CRB is divided into two for the bunny and the ball respectively. In (c), the ball hits the back of the bunny, and the back of the bunny deforms according to FEA computation. Some small CRBs are generated at local region of the back. The deformations are transferred to other parts of the bunny, such as the whole body, the head and the ears. Consequently, more and more CRBs are generated in the last two figures on the right. (a) Time step=1; (b) time step=190; (c) time step=316; (d) time step=322; (e) time step=369.

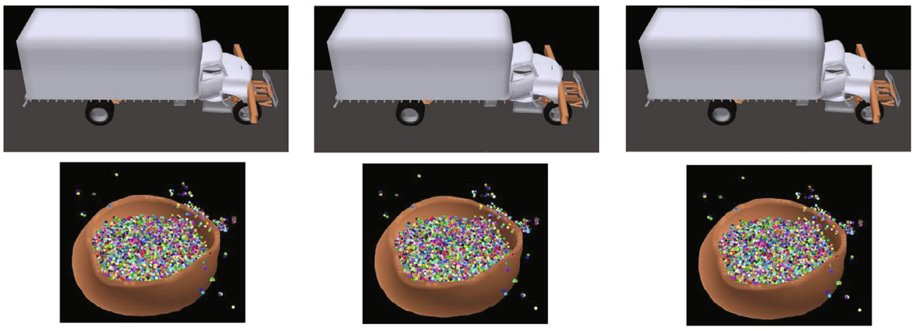

Figure 10 The *Truck* and *Bowl* scenes constructed by our algorithm (left) and Rosen's RBD algorithm (middle) are compared with the original dataset (right).

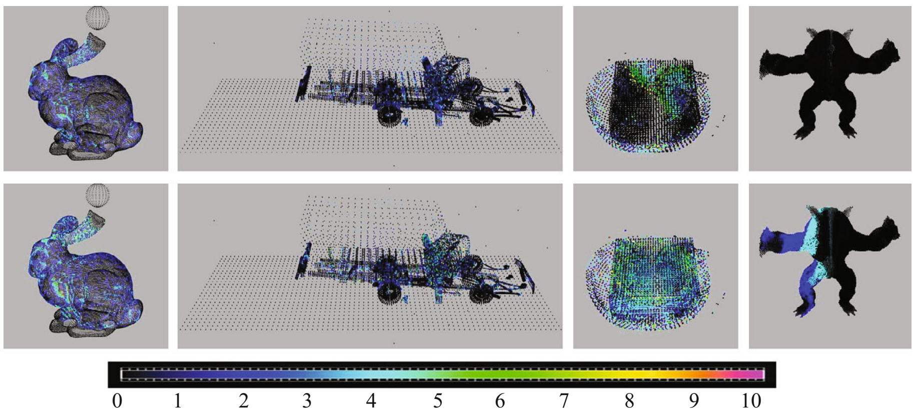

Figure 11 Visualization of error on the node itself between its original and reconstructed positions with our algorithm (row 1) and Rosen's RBD+ITS algorithm (row 2). Error values within 10 mm for each node corresponding to different colors (row 3).

and crb_4. If p can be merged into one of those CRBs, for example, crb_3, the index 3 is recorded for p too, and then we move on to the children of crb_3. This process recurs until p cannot be merged into any descendants of this branch at time step i, such as crb_5 and crb_6, then the 3D positions of p are stored from ith time step.

To improve the compression factor, we merge the remaining UNs into CRBs after CRB generation. We refer this process as UN compression 2. For each unassigned node p, first we determine whether it can be merged into crb_1 during its lifetime (see in Figure 8). If it could be merged, we store the index 1 for p, and test whether p could be merged into the CRBs that are divided from crb_1, such as crb_2, crb_3,
and crb_4. If p can be merged into one of those CRBs, for example, crb_3, the index 3 is recorded for p too, and then we move on to the children of crb_3. This process recurs until p cannot be merged into any

descendants of this branch at time step i, such as crb_5 and crb_6, then the 3D positions of p are stored from ith time step.

We store the final compressed 3D dynamic datasets with the following structure. 1) The number of nodes in the scene. 2) The initial position and color of each node in the scene. 3) The connection relations for triangle-based scenes. 4) The number of RBs. 5) Each RB stores the number of nodes and the global index of node. 6) The number of CRBs. 7) Each CRB stores the starting and ending time steps t_s, t_e, transformations from t_s to t_e, the number of RBs and the global index of RB. 8) The number of UNs. 9) Each UN stores the global index of the nodes, the indexes of CRBs, 3D positions of the node from t_e+1 of its last indexed CRB to the last time step.

5 Result and discussion

We apply our CRBC algorithm to four datasets: *Bunny* (Figure 1, top), *Armadillo* (Figure 1, bottom), *Truck* (Figure 10, top), and *Bowl* (Figure 10, bottom). The deformation of the bunny, the truck, and the bowl in these dataset are computed with finite element analysis (FEA). The motions of armadillo-shaped fluid and the water are simulated with smoothed particle hydrodynamics method (SPH). *Bunny* has 35 000 nodes and 381 time steps. The size of its AABB is 10 m $\times$ 9.8 m $\times$ 14.3 m. *Armadillo* has 72 000 nodes and 81 time steps. The AABB of it is about 9.9 m $\times$ 8.3 m $\times$ 7.5 m. *Truck* has 29 000 nodes and also 81 time steps. Its AABB occupies 15 m $\times$ 5 m $\times$ 3.3 m. The last dataset is a scene with fluid-solid coupling. It has 19 000 nodes and 171 time steps, and its AABB is about 6.7 m $\times$ 6.4 m $\times$ 10.5 m. All performance measurements reported in this paper were recorded on a 3.4 GHz Intel(R) Core(TM) i7-2600 CPU PC with 4 GB of RAM and an NVIDIA GeForce GTX 580 graphics card.

5.1 Compression quality

Figure 10 shows another two datasets we used to test our algorithm. The top row is *Truck* and the bottom row is *Bowl*. The figures in the left column are the results of our algorithm, and the figures on the middle column are the outputs of Rosen's RBD algorithm, and the right column shows the ground truth of the original dataset without any compression. Although there is almost no obvious difference between the original and the reconstructed scenes by using our or Rosen's RBD algorithms, the average errors of nodes with our algorithm is lesser than those of Rosen's RBD algorithm.

Table 1 gives average errors of four datasets with variable error thresholds using our method and Rosen's methods. The average error is computed as the average of the differences between the positions of all nodes for the reconstructed and the original data over all the time steps. Figure 11 shows the error of the node between the reconstructed position and the original one at a time step. We can see more nodes with smaller error colors, such as black and blue, existing in different models with our algorithm (row 1) compared with Rosen's RBD+ITS method (row 2). The accompanying video also shows the position errors of nodes over time steps for different scenes. Our average error is smaller than Rosen's RBD mainly because of UN compression. The position of a reconstructed UN is obtained by the linear interpolation with Rosen's ITS algorithm [1], whereas by CRB's transformations or their original datasets with our algorithm.

The average error ratio between our method and Rosen's RBD+ITS is also given in Table 1. The best case is the *Bowl* scene with error threshold 10 mm and its error ratio is 0.360, while the worst one is the *Bunny* scene with error threshold 100 mm, and its error ratio is 0.999. This is because the best case has a lot of UNs left for UN Compression, and the worst has almost none. We also give node number-position error histograms for a time step of those two cases in Figure 12. In the histogram of the best case (Figure 12(a)), almost 95% nodes have very small position errors between 0 and 0.5 mm by using our method. In contrast, using Rosen's RBD+ITS method, there are only 45% nodes with error from 0 to 0.5 mm, 10% nodes with error from 0.5 to 10 mm. There are still about 45% nodes that cannot be merged into RBs. ITS is used to compress the trajectories of them, thus leading to the bigger position error above 10 mm. Even in the worst case (Figure 12(b)), the node number with smaller error from 0 to 5 mm of our method is higher than that of Rosen's method.

Table 1 The average error of our method compared with Rosen's

Model	Error threshold (mm)	Average error (mm)				
		Rosen's ITS	Rosen's TC+ITS	Rosen's RBD+ITS	Our method	Error ratio
Bunny	10	8.26	8.25	3.31	2.88	0.870
	50	25.46	23.49	12.73	12.56	0.987
	100	41.92	41.95	23.82	23.81	0.999
Truck	10	8.03	8.03	1.90	1.72	0.905
	50	37.50	37.30	6.93	6.81	0.983
	100	63.70	63.50	15.72	15.63	0.994
Armadillo	10	8.51	8.52	5.28	3.14	0.595
	50	36.94	36.95	23.15	15.97	0.690
	100	71.77	71.85	44.71	36.42	0.815
Bowl	10	8.93	8.95	6.33	2.28	0.360
	50	42.83	42.84	27.23	14.76	0.542
	100	76.95	76.96	51.62	29.14	0.565

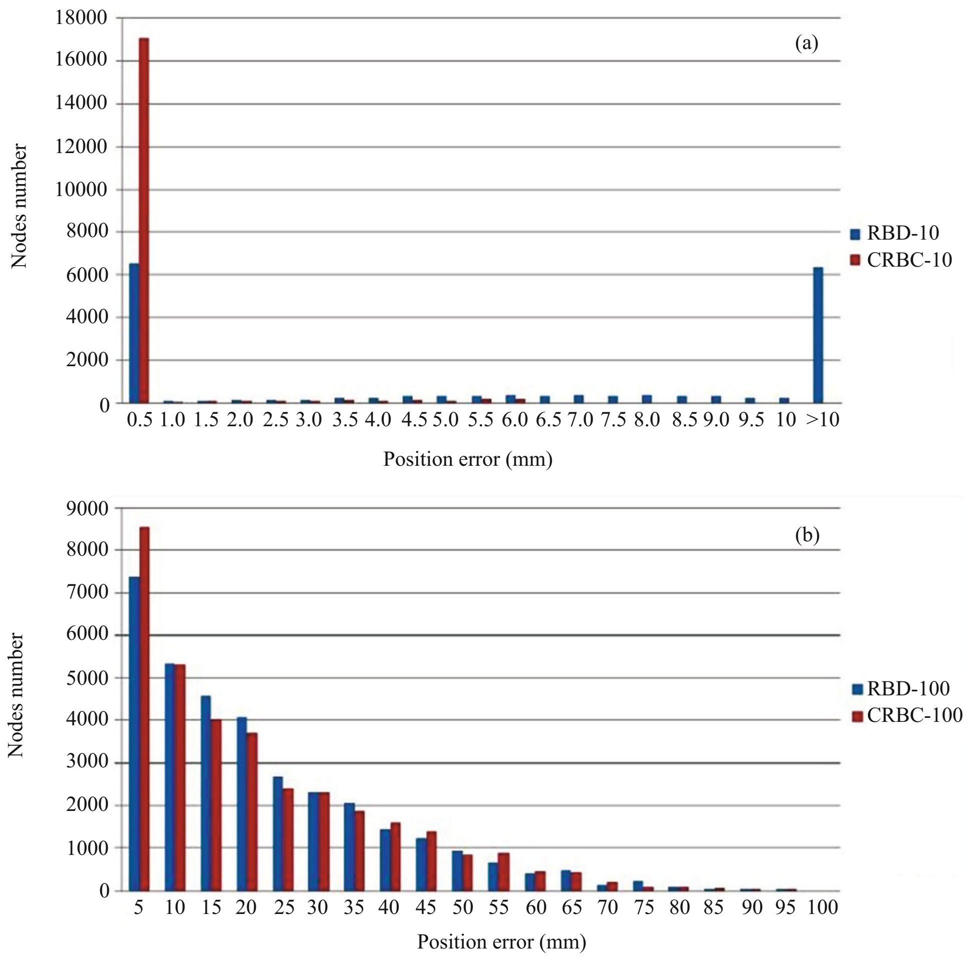

Figure 12 The histogram of node distribution according to its position error. (a) *Bowl* scene with error threshold 10 mm; (b) *Bunny* scene with error threshold 100 mm at a given time step.

We also give the average errors of Rosen's ITS and ITS+TC [1] algorithms in Table 1. ITS reconstructs nodes' trajectories by linear interpolation with the stored positions at key time steps, and TC simplifies nodes trajectories into clusters by minimizing cluster entropy. From Table 1, we can see that ITS and TC get bigger average error than our and Rosen's RBD+ITS algorithms.

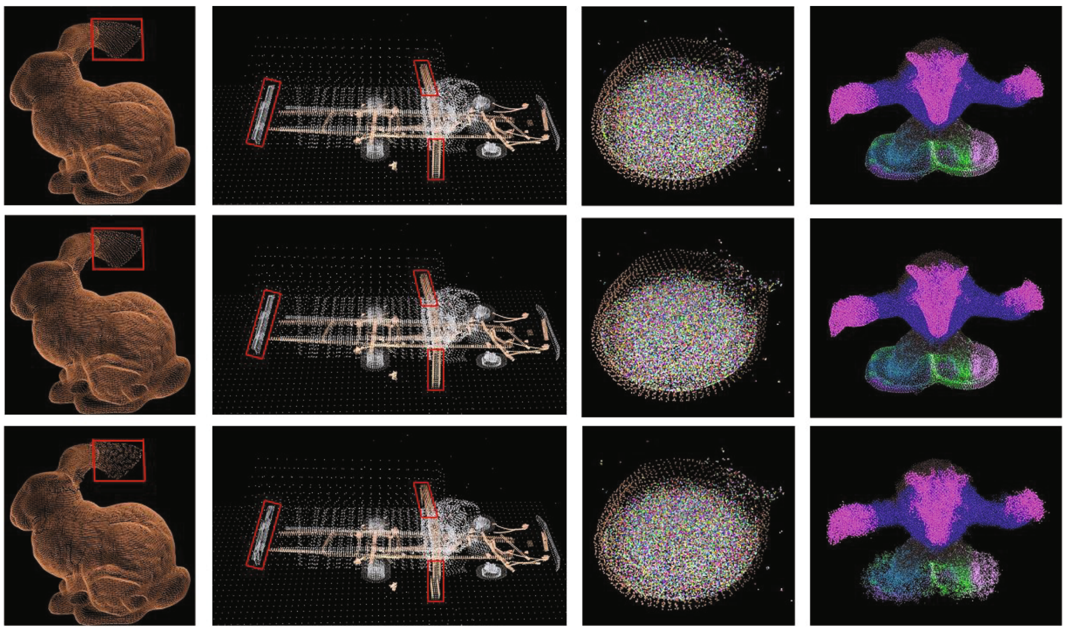

Figure 13 The points based on the rendering for the four datasets. Compared with the original datasets (row 1), the reconstructed datasets with error<10 mm (row 2) are very similar in vision. However, there are some big differences between the original datasets and the results with error<100 mm (row 3).

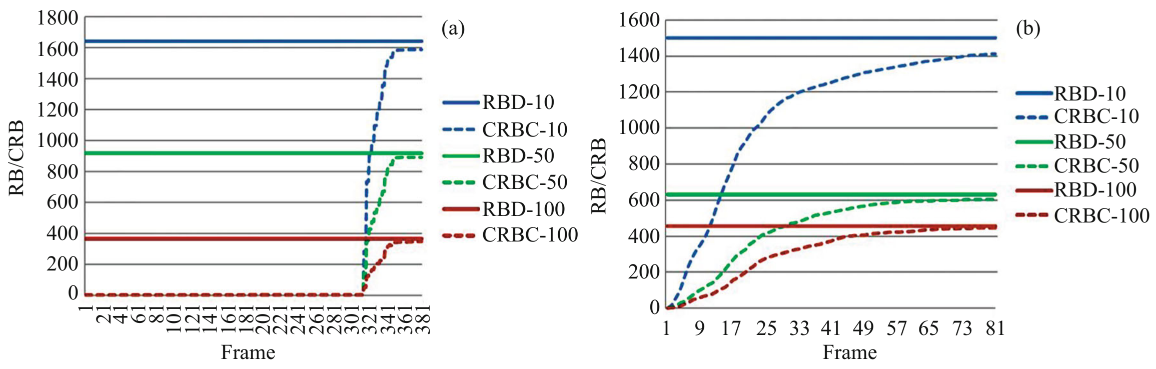

Figure 14 The CRB number is compared with the RB number of Rosen's method for time steps. (a) Deformable bunny; (b) truck traffic.

In our method, large error thresholds lead to fewer RBs, fewer CRBs, and fewer UNs, thus increasing the compression factor. But the reconstruction quality decreases according to the increase of error thresholds. For better visualization, Figure 13 shows point-based rendering for four datasets. No obvious difference can be noticed from the original datasets (row 1) to the results with error<10 mm (row 2). But for the error<100 m cases (row 3), there are some large differences at the bunny ears, the obstacle ends and the feet of armadillo. For the *Bowl* scene, there are still some differences between the original and the constructed scene with error<100 mm. But the differences are covered by the distribution of SPH particles.

5.2 Compression performance

We evaluate our compression performance by compression factor and compression time.

5.2.1 *Compression factor*

In Table 2, the compression factors of our method are in average 3.3 times of those of Rosen's RBD+ITS method. The best compression factor of our method is from the *Bunny* dataset with error 100 mm, and is 6 times of the Rosen's, whereas the worst is from the *Truck* dataset with error 10 mm. The results

Table 2 The compression performance of our method compared with Rosen's

Model	Error threshold (mm)	Compression factor					Compression time (s)	
		Rosen's ITS	Rosen's TC+ITS	Rosen's RBD+ITS	Our method	Zlib	Rosen's RBD+ITS	Our method
Bunny [1.53 GB]	10	5.3	8.7	10.2	49.8	3.2	769.95	18.38
	50	10.9	17.2	17.7	95.2		441.53	6.53
	100	30.1	46.9	47.4	283.6		173.62	2.42
Truck [365 MB]	10	6.5	9.8	11.4	14.3	3.8	567.87	17.2
	50	15.3	24.8	25.1	37.5		375.68	4.38
	100	22.1	33.8	33.4	56.9		282.91	1.97
Armadillo [330 MB]	10	2.7	3.0	2.9	9.3	3.9	986.92	67.24
	50	4.0	4.7	4.6	15.4		552.17	26.13
	100	7.8	8.5	8.4	21.5		258.24	9.68
Bowl [286 MB]	10	2.2	2.5	2.3	6.5	4.1	1269.53	155.38
	50	2.9	3.4	3.1	11.6		956.27	109.61
	100	4.5	5.2	4.9	17.5		763.46	94.93

tioned above are caused by the motion patterns of objects in these two datasets. Deformable *Bunny* dataset has 381 time steps, in which the bunny rotates from time step 1 to 180. Then it stays and the ball falls off without touching the bunny from 181 to 315. At last the ball hits the bunny, bounces up, and the bunny deforms from 316 to 381. That is, in the first 315 time steps, two objects maintain their rigid body motions respectively, then the bunny deforms. Our method works well in this case because CRB takes motion consistency at different time periods among RBs into consideration. The *Truck* dataset has 81 time steps, simulates the truck collision at the first time step, and then deformation and the crushing from time step 2 to 81. Even for this motion pattern, the compression factor of our method is still higher than Rosen's RBD+ITS because the number of CRB for each time step is smaller than or equal to the RB number of Rosen's method. Figure 14 compares the CRB number of our method (dot lines) with the RB number of Rosen's RBD method (solid lines) for time steps. The color blue, green, and red indicate the variable error thresholds: 10, 50, and 100 mm. In the deformable *Bunny* dataset (Figure 14(a)), the CRB number keeps almost as a constant of 1 or 2 from time step 1 to 315, then quickly goes up and stops at the value which is less than the number of RBs of Rosen's method. In Figure 14(b), the truck deforms on all 81 time steps after collision, therefore, the CRB number increases gradually to approach the RB number.

Users can segment the trajectory of a dynamic scene into pieces manually according to the motion consistency at different time periods. RBD can be used on each piece separately to increase the com pression factor. For example, the *Bunny* trajectory can be segmented into three pieces. We use RBD to compress the trajectory at time intervals from 1 to 180, 181 to 315, and 316 to 381 individually. Even in this case, the compressed number of CRBs in each time subset is closer or equal to Rosen's RB number. Thus the compression factor of our method is almost the same as that of RBD algorithm. For the *Truck* scene, manual segmentation for trajectory is difficult and results in a lot of trajectory pieces. For each piece, the initial positions of nodes have to be stored with RBD algorithm. Whereas our CRB method just needs to store the initial positions of nodes once over the entire s time steps. In this situation, RBD results in much lower compression factor than ours.

Like Rosen's RBD algorithm, our compression algorithm is also not based on the motion consistency of adjacent nodes. If the nodes in the KD-tree cannot be constructed into a small rigid body, UN compression 1 is used to compress the remaining unassigned nodes. Thus we can get closer compression factors with that of Rosen's method. The *Bowl* and the *Bunny* scenes with different motion consistency of adjacent nodes are taken as examples. In the *Bowl* scene, water nodes break up at last few time steps, so adjacent nodes maintain almost no motion consistency over all 171 time steps. If we only compress nodes of the *Bowl* scene based on DU-RBD without CRB generation, the final compression factors are 2.25, 3.02, 4.86 with corresponding error thresholds 10, 50, and 100 mm, which are almost the same as

Table 3 The time consumed for each step of our algorithm(s)

Model	Error (mm)	SRB construction	SRB combination	UN compression 1	CRB construction	UN compression 2
Bunny	10	0.45	0.33	3.65	5.23	6.72
	50	0.43	0.29	1.15	1.24	3.42
	100	0.39	0.23	0.48	0.48	0.94
Truck	10	0.54	0.21	1.22	3.54	11.89
	50	0.49	0.14	0.30	0.62	2.83
	100	0.41	0.06	0.14	0.33	1.03
Armadillo	10	1.52	0.05	45.56	0.56	19.55
	50	1.06	0.11	18.12	0.88	5.96
	100	0.42	0.28	5.62	1.11	2.25
Bowl	10	1.97	0.05	123.52	0.58	29.26
	50	1.89	0.06	82.69	0.59	20.28
	100	1.82	0.07	77.16	0.61	15.33

those of RBD method in Table 2. In the *Bunny* scene, although the adjacent nodes have high motion consistency, the final compression factors are 10.02, 16.8, and 46.9 without CRB generation, which are also closer to those of RBD method. The results above demonstrate that without CRB generation, whether the motion consistency of adjacent nodes is high or not, DU-RBD achieves almost the same compression factor as that of RBD algorithm. CRB generation improves the compression factor in our method compared with Rosen's RBD method.

We also compare our method with Rosen's ITS, TC+ITS, and a standard compression algorithm Zlib in Table 2. Compared with Rosen's ITS and TC+ITS, we still have advantage in compression factor. Compared with Zlib, the results show that in the *Bunny* and the *Truck* scenes, our compression factors are at least 3.7 times of Zlib's because most nodes can be organized into CRBs. However, our compression factors are at most 5.5 times of those of Zlib's for the *Armadillo* and the *Bowl* scenes. The reason is that *Armadillo* and *Bowl* have many fluid particles, and motion consistency of nodes is very low, which leads to few CRBs.

5.2.2 *Compression time*

The last two columns of Table 2 show time consumed with our CRBC and Rosen's RBD+ITS methods. Our compressions for these four datasets are from 7 to 143 times faster than Rosen's. The highest speedup case is the *Truck* dataset with error 100 mm, whereas the lowest one is the *Bowl* dataset with error 10 and 100 mm. The time cost at all steps of our method is shown in Table 3. From Table 3, the average time cost of SRB generation, SRB combination, and CRB construction for all examples is 2.42 s, while two UN compressions take 40.24 s, which means the UN compressions are the most time-consuming parts. In the high speedup case, the time spent for 2 UN compressions is 1.17 s, while the lowest uses 152.78 s. This is because in the *Truck* scene, the higher motion consistency adjacent nodes have, the higher success ratio of SRB construction each leaf obtains. More SRBs generation leads to smaller number of UNs, which results in less compression time for UNs. The *Bowl* scene includes adjacent nodes with low motion consistency, and suffers a long UN compression.

5.3 Limitations

Our method compresses the trajectories of the nodes, thus it suffers the same limitation of the node-based compression methods. It does not compress the topological relations for the vertices from triangles and polygons. Thus to reconstruct the mesh-based scene, that topological relations are still need to be transferred.

Both our method and Rosen's RBD method are not guaranteed to generate the best solutions of rigid body construction because of the main frame of greedy algorithm. Both methods get low compression factors when the nodes have almost no motion consistency in scenes.

6 Conclusion and future work

We have presented the techniques for composite rigid body construction, which accelerates compression with rigid body decomposition based on disjoint union, and increases the compression factor by merging the rigid bodies that have period transformation consistency into composite rigid bodies. The results of the experiments show that our algorithm compresses dynamic datasets quickly and compactly.

In terms of future work, for some large datasets, the data may still be too large to be sent even after compression. We will extend our method from nodes in 3D space to samples in the screen space, thus providing solutions to these cases. The view parameters from the clients are used to determine the samples required.

Acknowledgements

This work was supported by National High Technology Research and Development Program of China (863) (Grant No. 2013AA01A604), National Natural Science Foundation of China (Grant Nos. 61272349, 61190121 and 61190125), Macao Science and Technology Development Fund (Grant No. 043/2009/A2).

References

1 Rosen P, Popescu V. Simplification of node position data for interactive visualization of dynamic datasets. IEEE Trans Vis Comput Graph, 2012, 18: 1537–1548
2 Arikan O. Compression of motion capture databases. ACM Trans Graph, 2006, 25: 890–897
3 Lange R, Farrell T, Durr F, et al. Remote real-time trajectory simplification. In: Proceedings of 2009 IEEE International Conference on Pervasive Computing and Communications. Piscataway: IEEE, 2009. 1–10
4 Vries, G D, Someren M V. Clustering vessel trajectories with alignment kernels under trajectory compression. In: Proceedings of European Conference on Machine Learning and Knowledge Discovery in Databases. Berlin: Springer, 2010. 296–311
5 Alexa M, Mueller W. Representing animations by principal components. Comput Graph Forum, 2000, 19: 411–418
6 Karni Z, Gotsman C. Compression of soft-body animation sequences. Comput Graph, 2004, 28: 24–34
7 Sattler M, Sarlette R, Klein R. Simple and efficient compression of animation sequences. In: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer animation. New York: ACM, 2005. 209–217
8 Akhter I, Simon T, Khan S, et al. Bilinear spatiotemporal basis models. ACM Trans Graph, 2012, 31: 17–29
9 Guskov I, Khodakovsky A. Wavelet compression of parametrically coherent mesh sequences. In: Proceedings of the 2004 ACM Siggraph/Eurographics Symposium on Computer Animation, Switzerland, 2004. 183–192
10 Payan F, Antonini M. Wavelet-based compression of 3d mesh sequences. In: Proceedings of the Second International Conference on Machine Intelligence. Piscataway: IEEE, 2005. 32–40
11 Beaudoin P, Poulin P, van de Panne M. Adapting wavelet compression to human motion capture clips. In: Proceedings of Graphics Interface. New York: ACM, 2007. 355–364
12 James D L, Twigg C D. Skinning mesh animations. ACM Trans Graph, 2005, 24: 399–407
13 Landreneau E, Schaefer S. Poisson-based weight reduction of animated meshes. Comput Graph Forum, 2010, 29: 1945–1954
14 Kavan L, Sloan P P, O'Sullivan C. Fast and efficient skinning of animated meshes. Comput Graph Forum, 2010, 29: 327–336
15 Le B H, Deng Z. Smooth skinning decomposition with rigid bones. ACM Trans Graph, 2012, 31: 199–209
16 Le B H, Deng Z. Two-layer sparse compression of dense-weight blend skinning. ACM Trans Graph, 2013, 32: 258–266
17 Lengyel J E. Compression of time-dependent geometry. In: Proceedings of ACM Symposium on Interactive 3D Graphics. New York: ACM, 1999. 89–95
18 Shamir A, Pascucci V. Temporal and spatial level of details for dynamic meshes. In: Proceedings of ACM Symposium on Virtual Reality Software and Technology. New York: ACM, 2001. 77–84
19 Shlafman S, Tal A, Katz S. Metamorphosis of polyhedral surfaces using decomposition. Comput Graph Forum, 2002, 21: 219–228
20 Yang J H, Kim C S, Lee S U. Compression of 3-D triangle mesh sequences based on vertex-wise motion vector prediction. IEEE Trans Circuit Syst Video Technol, 2002, 12: 1178–1184
21 Ibarria L, Rossignac J. Dynapack: space-time compression of the 3D animations of triangle meshes with fixed connectivity. In: Proceedings of the ACM Siggraph/Eurographics Symposium on Computer Animation, Switzerland, 2003. 126–135
22 Zhang J, Owen C B. Octree-based animated geometry compression. Comput Graph, 2007, 31: 463–479