Transient Motion Groups for Interactive Visualization of Time-Varying Point Clouds

Markus Broecker *Member, IEEE,* and Kevin Ponto *Member, IEEE*

Abstract—Physical simulations provide a rich source of time-variant three-dimensional data. Unfortunately, the data generated from these types of simulation are often large in size and are thereby only experienced through pre-rendered movies from a fixed viewpoint. Rendering of large point cloud data sets is well understood, however the data requirements for rendering a sequence of such data sets grow linearly with the number of frames in the sequence. Both GPU memory and upload speed are limiting factors for interactive playback speeds. While previous techniques have shown the ability to reduce the storage sizes, the decompression speeds for these methods are shown to be too time and computation-intensive for interactive playback.

This article presents a compression method which detects and describes group motion within the point clouds over the temporal domain. High compression rates are achieved through careful re-ordering of the points within the clouds and the implicit group movement information. The presented data structures enable efficient storage, fast decompression speeds and high rendering performance.

We test our method on four different data sets confirming our method is able to reduce storage requirements, increase playback performance while maintaining data integrity unlike existing methods which are either only able to reduce file sizes or lose data integrity.

Index Terms—Data structures, Data compression, Point clouds, Computer Graphics

1 Introduction

TIME-variant three-dimensional point clouds are a rich source of of information that can be explored, annotated and interacted with in virtual reality environments. While physical simulations, for example n-body or SPH simulations, generate these types of information, the complexity of the calculations precludes them from running in real-time. The results of these simulations are often very large in size, which hinders interactive visualization of the data. In this regard, the common method for viewing these types of simulation is through pre-rendered animations.

Unfortunately, pre-rendered movies do not provide a foundation for immersive exploration of the data. The forced perspective inhibits a user's ability to experience the data in interactive 3D environments. Beyond the user's inability to control the perspective of the data, the user can also not dynamically control the colors and shading of the individual projected points. On the other hand, compressing point cloud data on a per-frame basis leads to very low play back rates due to the high uncompression cost.

The challenge of creating immersive 3D animations from time-varying point clouds comes almost exclusively from the large file size. The data requirements are so high that even using state of the art locally connected solid state drives, the data can not be shown at standard animation rates (as shown in Section 4.3). Therefore, in order to enable real-time playback, the data must be compressed in a GPU-friendly manner.

1.1 Previous Work

The most straight-forward approach to compress a sequence of time-variant point clouds is to compress each frame individually and reconstruct the sequence from the individual frames. Extensive research into compression of static point cloud data exists. Schnabel et al. compress static point clouds by creating and compressing displacement maps over geometric primitives [1]. A RANSAC-based shape detection method segments the initial point cloud into distinct geometric shapes which can be efficiently encoded using only few parameters Additional detail is achieved by storing point offsets as heightmaps with different levels of detail. Compression of this texture data is achieved through vector quantization. This method works well for point cloud models created from physical surfaces, for example LiDAR scans of statues. However it is unclear if this method can be applied to point clouds which do not represent surfaces, such as n-body or SPH simulations. Detecting localized groups within unordered point clouds using RANSAC model estimation is very similar to the method proposed in this article. However, we will apply the RANSAC group detection in the temporal domain, whereas Schnabel et al. apply it in the spatial domain within a static point cloud.

Octree partitioning is commonly used for spatial organization of point clouds but can also be used for compression. If the tree's leaf size is chosen small enough so that a single point of the initial point maps to a single leaf, reconstruction of the initial point clouds is possible from the octree structure [2] alone. Potentially, the tree structure is able to be compressed and stored more efficiently than the underlying point cloud. However, there are two drawbacks to this method: first the points of the initial point cloud do not retain their original position but are represented by their respective leaf nodes' center. Secondly, while octrees are able to describe the elements of a point cloud implicitly, the space requirement of octree structures grow exponentially with every tree level of non-empty nodes. If an accurate representation of the underlying point cloud is desired a very detailed, and therefore deep, tree must be constructed.

• *Both authors are with the Living Environments Lab at the University of Wisconsin-Madison, Madison, WI 53701.*
E-mail: broecker@wisc.edu, kbponto@wisc.edu

Manuscript received February 19, 2015.

The point cloud library (PCL) [3] offers a built-in lossy compression mechanism targeted towards streaming point cloud data from sensor devices. The underlying compression mechanism is based on a double-buffered octree structure [4]. Differences in the octree between two frames are encoded efficiently through range encoding. This methods provides excellent compression ratios at a high quality. While this method is targeted towards real-time streaming of point cloud data created by depth cameras, we found the performance lacking with larger point clouds. Section 4.3 discusses this method in detail and contrasts it with our approach.

Closely related to general point clouds are LiDAR based scans of surfaces or terrain. The space requirement of these scans is high and compression is desirable. Previous efforts by Isenburg et al. [5], [6] and Mongus and Žalik [7] focus on the lossless compression of such data for archival purposes. Most often, these data sets are created from airborne LiDAR scans and represent vast patches of ground surface. However in these cases, the point clouds resemble mostly heightmaps which leads to wrong assumptions taken during compression (for example, offering only a limited range in the 'up' component of a position).

Time-varying volumetric data sets are often created from time-variant simulation data and point clouds. Zhao et al. recently investigated time-varying point cloud data to visualize volumetric data sets [8]. However, their input data represented volume data, not point clouds. The authors acknowledge that out-of-core rendering is necessary due to the high memory requirements of the whole data set and present a user-controlled additive level-of-detail mechanism in which only few particles are drawn if the animation is playing, while a static view composites additional renders of the point cloud to achieve an overall dense image. This method cannot be used in immersive, tracked environments in which the camera moves every frame as it would invalidate the accumulated contents of the frame buffer.

Following the idea of representing geometry through texture images [9] and utilizing existing image compression techniques to lower the memory footprint, there have also been many efforts to extend this idea to create "geometric videos". Alexa and Müller proposed using Principal Component Analysis along the temporal axis as a means of data compression [10] while Lengyel and Briceño et al. utilized prediction methods of projected 3D surface data [11]. Similarly, another approach is to store positional information as color channels and utilize existing video compression software [12], [13]. While these approaches are intended for meshes it is trivial to use these methods also for point clouds, especially as the number of vertices is limited and unchanging in the presented methods. While movie compression is optimized for fast decompression speed, it introduces fundamental errors which makes it unsuitable for point cloud data compression: first, the input data must be a low dynamic range image stream which introduces severe quantization errors into the data set. Second, channel responses and sampling in video compression codecs are non-linear, a three-dimensional position is cannot without loss be interpreted as, for example, a YUV color coordinate. Finally, additional error is introduces when converting between color spaces which are often furthermore built on principles of human perception. Applying a movie compression scheme on a quantized data set therefore changes the initial data set beyond what can be accepted as 'lossy compression'.

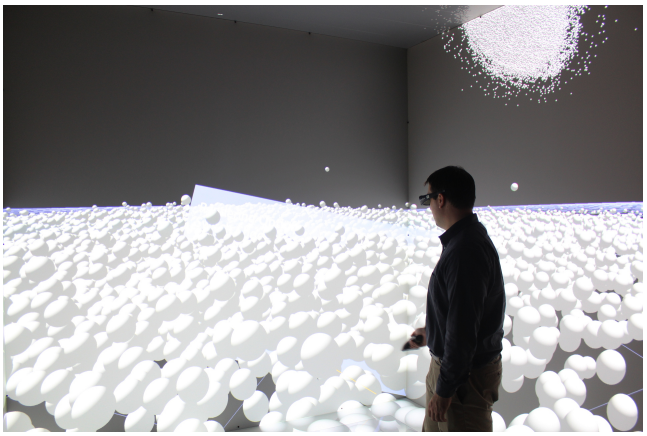


Fig. 1: A user navigating the animated 'Galaxy' data set in the CAVE.

1.2 Approach

This paper presents a novel compression method for time-varying point clouds generated by physics simulations. Our proposed method compresses space requirements by detecting and storing group motion and behavior in point clouds instead of trying store the motion of single elements in the point cloud. Group motion can be expressed by a transformation matrix and a list of indices referencing a subset of the initial points. This information requires less memory than either a list of new positions or velocities for points, thus achieving compression of the initial data set. We accept a small error distance ε between reconstructed and original data, which we can specify in *absolute* coordinates. We consider all points within a point cloud and between two frames that follow the same transformation and whose resulting position lie within the ε error bound to lie in the same *motion group*. For our purposes, we collect these groups and follow their motion through consecutive frames. During later frames the groups might be split again if different motions are detected for their constituent points. If the number of points in a group drops below a certain threshold or no transformation with an error smaller ε can be found for a point, it is designated as an *outlier* and not further tracked but stored using its coordinates explicitly. The *outlier ratio* of each frame, that is the number of outlier points divided by the number of total points, describes how well a frame can be represented using the found transformations. This ratio can be accumulated over multiple frames and rises monotonically. Once the ratio reaches a predefined threshold is reached, we group the previous frames into a *block*, in which the initial point set becomes the *key frame* and all frames (including the first) storing transformations and outliers are *delta frames* containing groups and outliers. The end result is an optimized compressed data structure which can be easily rendered and explored, for example in immersive display environments, as seen in Figure 1.

The goals of this work lie in the efficient compression of time-varying point cloud data for rendering. The bottleneck in rendering is usually the data upload to the GPU. We therefore seek to minimize the size of the upload required. A second requirement is low disk read and high decompression speed. Both are required to maintain a constant stream of data to be rendered in sequence. While many compression techniques are focused on lossless storage, our method takes inspiration from movie compression codecs, which are optimized for display. We accept a small loss in

precision of the data set for a interactive playback experience built on small file sizes and a high decompression speed.

The following sections of the paper will be laid out as follows. Section 2 will discuss the specifics of our compression method. Section 3 will discuss the empirical evaluation of parameter space for our compression method. Section 4 will compare and evaluate our compression method against other approaches. In Section 5 we will discuss the results of our method, design decisions and future work before concluding in Section 6.

2 METHODS

One of the core functions of our algorithm is to detect group motion within point clouds. We use a RANSAC approach [14] to solve this problem. We consider two point clouds as the same point cloud at two frames of the animation. Using subsets of points, we estimate transformations that would transform the first onto the second frame and select the transformation with the best fit, that is the highest number of points, for which the transformation succeeds. These points are grouped and the calculated transformation is stored with the group.

By repeating this procedure for a new frame, a tree structure is created over the temporal domain. Each previously found group is used as the basis for further split and motion detection operations. This refines the movement of groups over multiple frames and also supports movements in which large parts of a point cloud follow the same trajectory until they break away. For example, Figure 2 shows one of our data sets with groups highlighted in different colors.

While the tree structure segments the data set into groups with same movement characteristics, it does not compress the data. To achieve this goal, we regroup the tree and use it to rearrange the underlying point cloud. The regrouped tree structure enables implicit indexing while the motions group substitute explicit point data with implicit reconstruction. The results are stored in a tightly-packed block structure which has a small memory footprint and can be quickly written to and read from storage.

Upon loading from disk, the blocks need to be unpacked for rendering. Each frame is rendered by re-using data from the key frame, and the compressed per-frame data, while outliers are considered small self-contained per-frame point clouds which are rendered directly. This very direct mapping from the memory structure of a block to the data structure on the GPU enables rendering with high frame rates. Additionally, it is still possible to extract a self-contained point cloud for each frame of a block, for example to support selection.

The presented method proves to require less memory for storage and loading, while at the same time being faster for rendering and loading from disk, as Section 4.3 will show. The reduced file size mainly improves the read and playback speed while the block structure groups consecutive frames together for better cache coherence compared to storing point clouds for each frame on the disk.

The presented compression technique is lossy in precision with a user-defined upper bound, however it does not discard individual points or collapse multiple points into one, as observed in other compression methods. The maximum amount of error loss is able to be quantitatively defined to an ε value which does not drift over time. Additionally, our algorithm does not use quantization (i.e. bit reduction). As shown in Section 4.1, the resulting error in both position and velocity is in practice on average much lower than the user defined tolerance. Contrary to the usual practice of quantization of the initial data set into integers, we have chosen to use 32-bit single-precision and 16-bit half-precision floating point numbers (following the IEEE 754 standard) for data representation, depending on the maximum spatial extend of the data set and the chosen maximum permissible error.

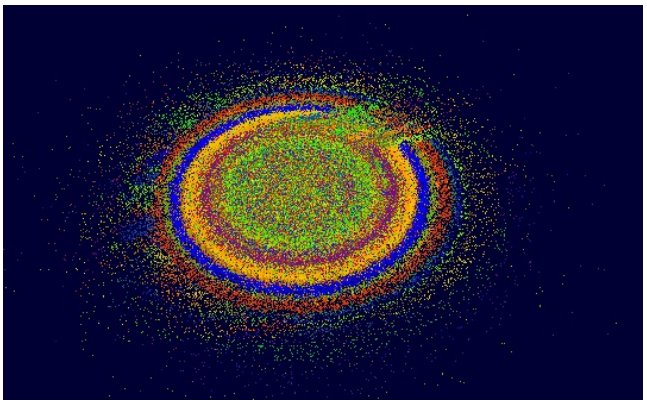


Fig. 2: Shows detected groups, each with a unique color, in the 'Galaxy' data set. As shown, the algorithm is able to detect rings of similar motion for the rotating cluster of stars.

The methods for our algorithm are divided into three distinct parts: splitting, gathering and rendering. The splitting section discusses how groups of points are found and formed. The gathering section describes how these groups are structured to be grouped into blocks and written to disk. Finally, the rendering section describes how the data is read from disk and rendered in an immersive display environment. The splitting and gather phases are illustrated in Figure 3.

2.1 Split Procedure

The goal of the splitting operation is to find groups of points which have the same motion. To do so, frames are inspected in pairs in temporal order (i.e. Frames 1 and 2 are inspected together, followed by Frames 2 and 3, and so on). The algorithm assumes that the data maintains order between frames, meaning that the *nth* index represents the same point at different times during the animation. Figure 2 shows the different groups of a single frame of the 'Galaxy' data set in different colors.

2.1.1 Inputs

The split procedure does not work directly on point clouds but on motion groups which form part of the tree structure. The group for the first split, that is the first frame, contains the initial point cloud as well as the identity transformation.

2.1.2 Outputs

The output of the splitting algorithm is grouping of points based on transformations. As groupings are only tested internally, the result of this algorithm is structured as a tree as shown in Figure 3 on the left. Points which do not correlate to any transformation group are put into an outlier group.

2.1.3 Parameters

The splitting procedure has four user definable parameters:

Maximum Iterations defines the maximum number of attempts

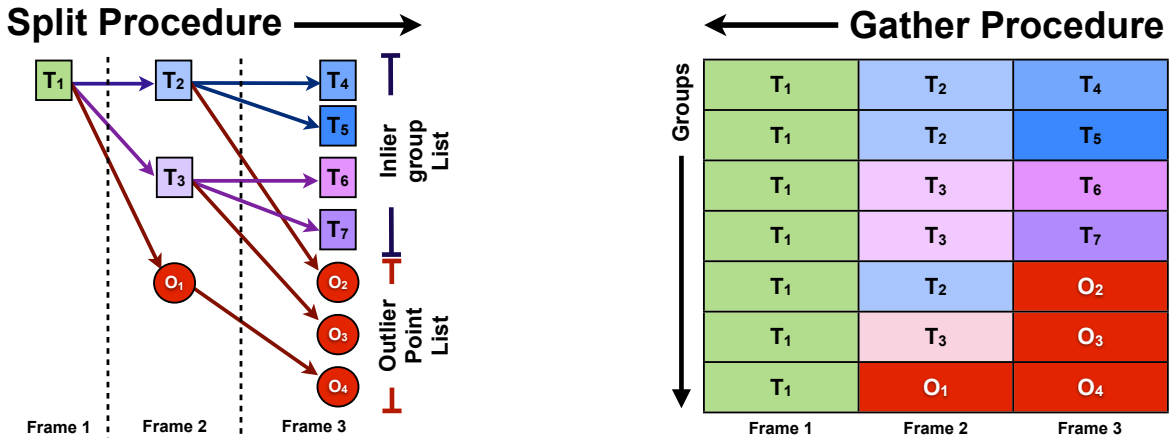


Fig. 3: An example of the split and gather procedures. The split procedure creates a tree structure over multiple frames, with each node representing a subgroup of the point cloud with common motion characteristics. This tree structure is expanded during the gather phase to rearrange the point cloud for efficient rendering.

to find groups of motion and therefore also the upper bound of groups per frame. Note that the total number of groups *per block* is related exponentially to the number of iterations, as our algorithm splits groups; 10 groups in the second frame will potentially become 100 groups in the third frame.

Maximum Error defines the maximum permissible *error* ε and determines visual quality. A larger error allows points to deviate further from their original positions as more points will be considered 'inliers' for a given transformation. A high error also reduces the number of groups while at the same time increases the number of points within a group. In turn, the higher group size and inlier rate leads to a higher compression ratio.

Maximum Outlier Ratio defines the maximum permissible ratio of outliers to inliers within a frame. If this threshold is crossed, the block will be considered finalized and will be forwarded to the gather procedure. The higher this ratio within a frame, the more space is required because the outliers are stored as points.

Minimum Group Size determines the minimum number of points required to fall within a group. If a motion group does not meet this constraint, the points in this group will be stored in the outlier group. A large group size is desirable as it can represent many points efficiently.

2.1.4 Procedure

As both the ideal groups and motion are unknown, we utilize a RANSAC style approach to find the groups of similar motions. This process consists of a series of steps to determine the motion groups between two frames (A and B) as shown in Figure 4. For each group in Frame A we perform the following steps:

1) Select a random sampling of points from point clouds A and B.
2) Find a transformation $\mathbf{T}$ which maps all points from A to B in the subset chosen in 1.
3) Apply this transformation to all points in group A.
4) Determine how many transformed points from A are within distance ε of their corresponding point in B.
5) Repeat steps 1–4 for *max iterations* and select the transformation which results in the largest amount of inliers.
6) Extract the inliers from the largest group and repeat steps 1–5 on the remaining points until either all points are classified into inliers or a maximum number of iterations has passed.
7) Classify all remaining points as outliers.

We use the Mersenne Twister pseudo-random number generator [15] to determine our random subset of six corresponding points from the two frames. The selected points were fed directly into an SVD-based pose estimation algorithm found in PCL. The detected transformation is then applied to all of the points from the first frame and compared to all of the points in the second frame. All points within the user defined ε distance are put into a *inlier* group while all points which fall outside of this distance are put into an *outlier* group.

To combat drifting error tolerance over time, we do not use the original but the reconstructed point cloud for A. The point cloud is reconstructed by transforming the key frame vertices by the transformation of group A. Similarly, although the transformation between A and B is calculated, the stored transformation is between B and the initial point cloud. This transformation matrix $\mathbf{T}_B$ can be calculated thus:

$$\mathbf{T}_B = \mathbf{T}_A \cdot \mathbf{T}$$

with $\mathbf{T}_A$ being the known transformation matrix of group A and $\mathbf{T}$ the estimated transformation matrix from A to B.

Transformation estimation is repeated for N iterations after which the transformation with the largest number of *inliers* is selected as the most suitable transformation. A new motion group is created using this transformation and all points which have been classified as 'inliers' with this transformation. All *outliers* are reused to find other transformations and are used as input for the next iteration of the splitting process, which is repeated until either all points were assigned to a motion group, a maximum number of iterations has taken place or no transformation to create a new subgroup could be found. Remaining points are assigned to the single *outlier* group of this split and stored in the next frame.

This per-frame splitting procedure is repeated for each pair of frames until a split results in an outlier ratio higher than the chosen *max ratio* threshold or no more frames are left to process.

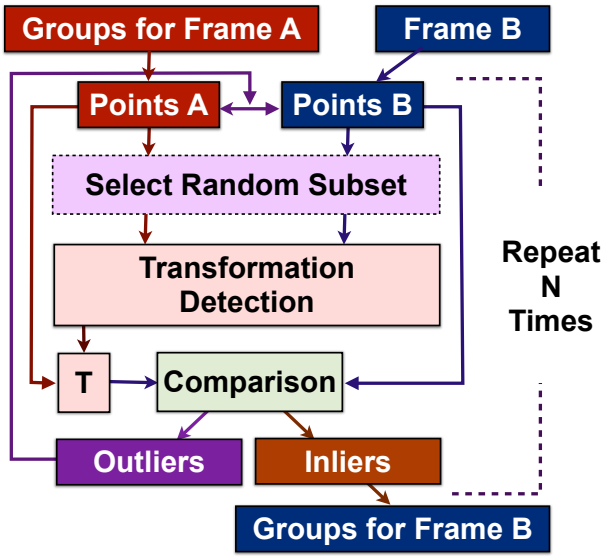


Fig. 4: Block diagram of the split method for two Frames A and B. The algorithm uses subsets of the points from the frames to find potential transformations. This transformation is then applied to all of the points from Frame A's group and compared to the points in Frame B. All points which fit the transformation are grouped as inliers while those that do not are put into the outlier group.

Figure 2 shows the output of the split phase on one frame of the 'Galaxy' data set and Figure 6 the detected groups of multiple frames in the 'Snowball' data set. Different groups are colored uniquely in these screen shots while outliers a drawn in red. The algorithm clearly detects the rotating motion of the galaxy in the foreground and creates different groups with unique rotation speeds, as observed in the different rings. Additionally, note that the groups are often not spatially coherent but spread over the whole data set, as observed in the randomly colored noise in large parts.

2.2 Gather Procedure

The goal of the gather step is to transform the temporal tree, created using repetitive splits over groups, into a tightly packed structure storing groups and outlier points for each frame. Re-arranging the initial point cloud order using a temporary matrix makes indexing unnecessary and the resulting block structure is both optimized for rendering and file storage. In detail, the gather step follows these sub steps:

1) Create a matrix containing the nodes over all frames.
2) Fill in the cells of the matrix from right to left by following a node's parent references.
3) Split the group's contents according to their location in the matrix. Each row contains the same points at this step.
4) Re-arrange the point cloud accordingly.

2.2.1 Inputs

The input for the gather procedure is the tree of inlier and outlier nodes created during the split phase. Each tree's root node is the initial point cloud stored in a group with the identity transform. Each level of the tree can contain both inlier and outlier nodes. The order of the nodes is important: inlier nodes are listed first, followed by the outlier nodes.

2.2.2 Outputs

The output of a gather phase is a block structure in memory which can be efficiently written to and read from disk and which has a lower memory footprint than individual point clouds per frame have. Additionally, this output structure maps easily onto rendering inputs and therefore requires little to no decompression, as will be shown in Section 4.3.

2.2.3 Procedure

A rectangular matrix storing tree nodes is created. The number of frames in the tree gives the width of the matrix, while the number of both inlier and outlier nodes in the last frame of the tree dictates the height of the matrix. To fill the other cells in the matrix, we walk from the rightmost frame left, filling in each node's parent in the cell left of it. Nodes with the same parent will have the same value in the entry left of it. As a result, the left-most column is always filled with the initial, single block which has the identity transform.

The resulting matrix now has different sized groups as line entries, with some of the groups being entered more than once. Two cases can be observed while following a single group over multiple frames by reading a line in the matrix from left to right:

1) Each group is followed by another group until the last frame. This happens only in the top sections of the matrix.
2) A group is followed in some frame by an outlier node. All following entries on the same line in the matrix store outliers from this point on. This case is observed only in the bottom rows of the matrix.

The first case compresses the data significantly. Each point in each of the nodes can be represented by the initial points and a single transformation matrix with little error. We therefore do not have to store the points of this group explicitly. The second case lists all points that fall outside this maximum error bound during one split. We have to store these points explicitly. Note that once a point is classified as an outlier it stays an outlier node for the rest of the block. Within the matrix a triangle structure appears through cases in which a group splits into smaller groups and finally into outlier nodes with the outlier nodes forming the lower half of the triangle.

After filling the matrix, we are able to rearrange the initial point cloud so that we can address all inlier points, described through groups, implicitly; outlier nodes will be discussed afterwards. Each cell entry corresponds to a single group of points, represented by a list of indices to the original points and a reference to a transformation. We note that each group can be divided in such a way that a single line in the matrix stores the same indices in each inlier group. To do so, we look at the indices in the rightmost group and advance to the left. As each cell entry to the left is a parent group of the one to its right, it must contain these indices. Furthermore, the group size along a line does not change. We can therefore use the indices stored in the last column of the array to rearrange the initial point cloud accordingly so that the new point cloud order reflects the indices in the last column.

This rearrangement enables us to reconstruct any inlier point implicitly by its position in the array only: the frame to be rendered fixes the column to look at and its index from the start indicates the group it is in. Once this group is found, the group's transformation matrix can be used to transform the initial point from the key frame to the end position with an error less or equal than ε.

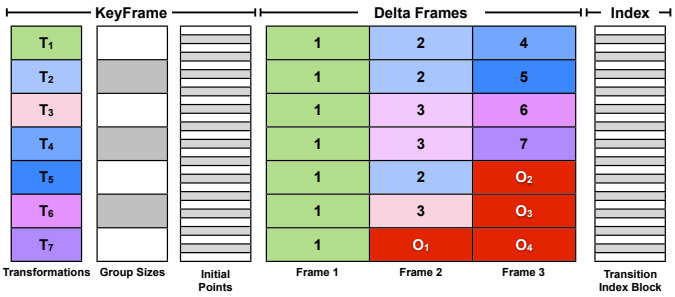


Fig. 5: The resulting block structure from the split and gather phases. The temporal tree of split nodes is re-arranged into linear arrays of the same subgroups. Each row in thus block references the same points in the key frame.

For example, consider Figures 3 and 5. Assume that group **T4** and **T5** each represent 100 points. If the 250th point of frame 2 is to be reconstructed, for example for rendering, the point would be found by looking at the second column in the third column, as the first 200 points are distributed in the first two cells of **T2**. The transformation matrix of **T3** multiplied with the 250th key frame point would reconstruct the point for this time frame.

Outlier nodes are handled differently than inlier nodes. Their group size does not change along a line of the matrix, as each outlier groups does not get split. Each outlier block along a line is followed by another outlier block. The outlier nodes store point data directly, which 'overwrites' the initial point data and does not need a transformation to do so. Additionally, the outlier nodes still store their original positions in the point cloud which allows re-ordering of the initial point cloud.

To continue the example given above, assume groups **T4**–**T7** represent 400 points total and outlier blocks **O2** and **O3** store 50 points each. If we want to reconstruct the 460th point in Frame 3, we skip the first 400 points in Frame 3, and outlier block **O2**. We then read the 10th point $(460-400-50=10)$ directly from outlier block **O3**.

2.3 Block Structure

Figure 5 shows the resulting block structure after the point re-arrangement. The block structure formats how the data is kept in memory and written to disk. The structure contains three components:

The **Key Frame** of a block stores the reusable data for all the frames in a block. In detail, it contains the initial point cloud, a list of all transformations and a list of the group sizes. Transformations are stored as a 4×3 matrices of floating point numbers. The group size list is stored as a list of integer values. This list enables indexing points indirectly, as the size of the groups along a line of the matrix is constant.

Delta frames are stored on a per frame basis. Each delta frame contains a list of references to transformations stored in the Key Frame. The group for which these transformation is applied is made implicit through the structure of the data. All outlier points are stored as a series of floating point values at the end of each frame.

The **Transition Index Block** is stored at the end of a block. As our method rearranges the order of the points, there is no guaranteed consistency of index for a point between blocks. In order to ensure that a point can be tracked between blocks, the point's original position is stored as an integer value to specify its mapping between blocks.

2.4 Rendering

A major advantage of the block structure over the initial point cloud data is the reduced amount of information that has to be transferred to the GPU. To draw frames of an unpacked point cloud, every point must be uploaded for each frame. While this is straight-forward to implement, it does not utilize GPU memory and bandwidth efficiently.

Using the block structure for rendering, the key frame's points and all transformation matrices are uploaded and stored on the GPU while frames of this block are rendered. A frame is drawn with only two draw calls: first all inliers are rendered by drawing the key frame's points from $[0..N]$, where N is the number of inliers in this frame. An index for each point provides the correct transformation matrix index used for this point. The transformation buffer is indexed into and a shader transforms the resulting vertex into clip space:

$$pos_{clip} = M_{mvp} \cdot M_{transform,i} \cdot vertex_{keyframe},$$

where pos_{clip} is the clip-space position of the vertex, M_{mvp} is the current ModelView-Projection matrix, $M_{transform,i}$ is the transformation matrix for the index i and $vertex_{keyframe}$ is the currently rendered vertex.

The second draw call renders all outliers for a frame directly; their data is uploaded to the GPU in advance after a block was loaded with each frame storing its outliers in a separate vertex buffer. This data does not change either for a frame and the buffer is rendered using a single draw call as well.

We render the points of the cloud as point sprites with variable radii. Rendering distance and a user-controlled variable control the radius of point sprites. A shader shades the particles to simulate spheres and discards fragments accordingly. Some data sets, such as 'Galaxy', simulate nebulae or gaseous objects. In these cases, the point sprites are alpha-blended. Rendering of liquids could also be implemented using surface description methods, such as marching cubes.

2.5 Interaction

The major advantage of our compression algorithm over rendered movies is that it preserves the nature of unordered three-dimensional point clouds, thus allowing the exploration of these and observing them from novel viewpoints. Interaction is implemented through data exploration and annotation. Data exploration is possible using wand navigation in immersive environments (such as the CAVE) and using mouse and keyboard commands on the desktop. Immersive environments, such as the CAVE, track the user's position continuously to adjust the view parameters – interactive frame rates are of utmost importance to produce immersive and pleasant experience.

Figure 7 shows a user interacting with the 'Dam break' data set in the CAVE. The wand is used for navigation, playback control and selection while other interaction settings, such as setting render options or changing point colors, are implemented using a 3D GUI.

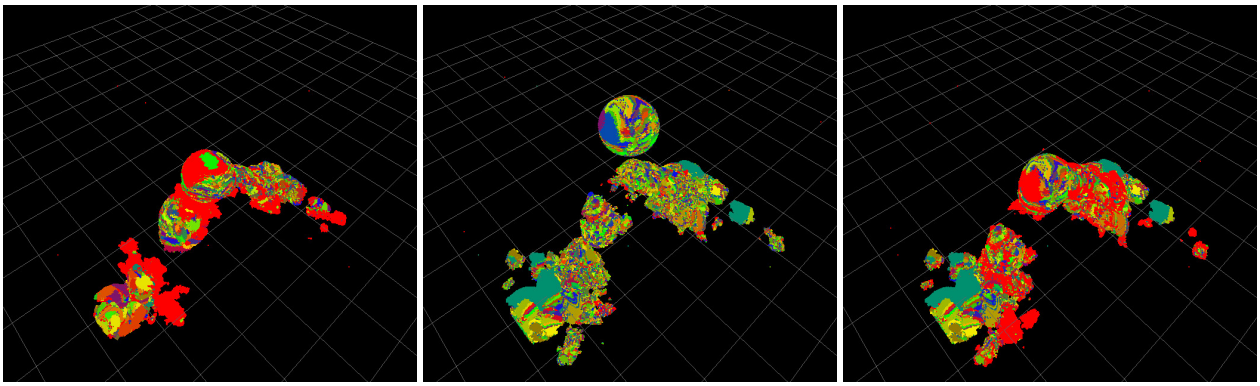


Fig. 6: Three frames in the 'Snowball' data set. Each group is depicted with a unique a random color while outliers are colored red. The algorithm detects large groups of particles with similar motion, outliers in one block can be assigned to groups in later frames.

Playback control allows data sets to be played forwards and backwards at higher or lower speeds. To enable this kind of control, we implemented playback using a sliding window technique with a fixed buffer of preloaded blocks in either direction. Playback moves the current buffer contents to either side while the block in the center is considered the 'active' one and is the only one rendered. A separate, dedicated thread is used for file loading and keeping the buffer filled. We found a buffer size of five blocks sufficiently big enough for uninterrupted playback of our data sets. Within a buffer the currently active frame is referenced and advanced based on a timer.

2.5.1 Selection

When visualizing large point clouds from physics simulations, it is desirable that some points can be selected and tagged so that their location can be easily followed through the simulation. We implemented tagging through a ray-point distance selection mechanic. A ray is cast and the distance of all points in the current frame to the ray is calculated. All points whose distance is less than a user-selectable threshold are considered selected. The selection ray is created in the CAVE from the tracked wand's position and orientation. In a desktop setting, the same beam is created by projecting the mouse cursor on the near and far planes of the view frustum. Unfortunately, our compression method does not provide an inherent spatial data ordering structure, such as an octree, which is often used for picking calculations. Our selection method therefore has to query each point in the data set. We improved the selection performance by testing points in parallel and achieved satisfactory results using 8 threads on millions of points.

Once a point is selected its color can be set or overwritten. Colors are supported by adding an additional color array to the block's key frame. A one-to-one mapping exists between a point and a color in this array. We assume for the selection mechanic that a point's color does not change between frames of a block or between blocks, unless overwritten by a new annotation. This color data is uploaded to the GPU only if changes occurred to the selection. Annotation colors are stored on a per-block basis. When a new block is loaded during rendering, the selection colors of the old block are copied to the new one. As the gather phase re-arranges the order of the points within each block, the colors cannot be copied directly. To overcome this limitation, the redirection index block is used which stores the original indices for each point in each block. It can therefore be used to transform all selection color data from a block into the order of the original point cloud. Both blocks have this redirection block which allows to transform the colors first into the common original order and from there into the order of the new block.

2.6 Implementation

We implemented the algorithm in C++ using standard libraries such as boost, OpenMP and OpenGL. Multiple programs and tools were created to compile data sets, display the data, extract information from the packed blocks, etc.

Large parts of the algorithm can be parallelized, for example transform detection, inlier counting or node splitting. We implemented both task and data parallel implementations of the split phase. In the task-parallel case, each group is handled in a dedicated thread and each point set contained therein is handled single-threaded (by the group thread). This performs poorly early in a block's creation where few blocks contain large amounts of data. The data-parallel implementation splits each groups in the master thread but uses multi-threading to parse and transform the point cloud contained within each group. This method performs well early on but might suffer with smaller groups. Overall, we found that the data-parallel implementation outperformed the task-parallel implementation.

Performance measures for compression, decompression, read, upload and draw speed were performed on a typical desktop PC with a hyper-threaded, Quadcore Intel I7 CPU, 16 GB of RAM, an NVidia GTX 750 GPU and a Samsung SSD.

3 Empirical Evaluation of the Parameter Space

Our algorithm has three main parameters: maximum permissible error, outlier ratio threshold and minimum group size. We determined optimal settings for each of the parameters by running series of compression tests on three different data sets. One parameter was varied while the other two were fixed. We did not test for different maximum iteration counts, but let the algorithm continue until it would not find any new groups for three consecutive iterations.

3.1 Data Sets

We investigated four data sets created from physics simulations, which are listed in Table 1. File size is the initial size of the data

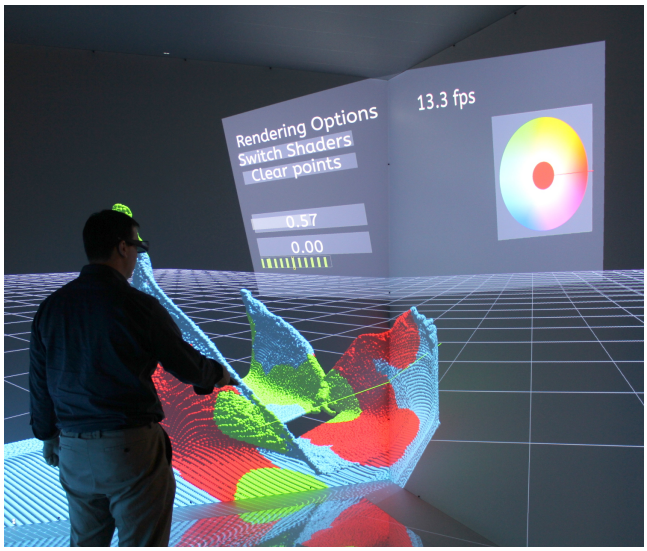


Fig. 7: A user annotating the dam break data set in the CAVE.

set, usually stored as a sequence of ASCII encoded files, while the bounding box span is the average (over all frames) length of the bounding box diagonal.

The 'Bullet' data set contains a simulation of more than forty thousand incompressible spheres arranged in a cylinder shape and falling into a small basin. The data set is created by the Bullet physics engine [16]. The data set shows orderly motion in the beginning, with the whole column of spheres falling uniformly devolving into chaotic movement once the first spheres collide with the reservoir.

The 'Galaxy' data set displays the collision of two galaxies and was generated from GADGET cosmological simulations [17], [18]. The first half of the data set shows very uniform motion, as the both galaxies rotate around their central axis and move towards each other. Once their positions join most of the movement is of chaotic nature as gravity tears them apart. Figure 2 shows the groups found in one of the galaxies during an early frame in the data set while Figure 8 shows three different frames at the beginning, the center and the end of the data set.

The 'Dambreak' data set simulates an initially static block of liquid crashing against a single pillar using the DualSPHysics Engine [19] and can be seen in Figure 7. The data set starts with a wall of liquid filling roughly a quarter of the simulated volume on one side and ready to crash into a simulated pillar. The latter part of the data set has very chaotic movement, turbulence and wave breaks and crowns as the liquid impacts the obstacle and flows around it.

The 'Snowball' data set simulates a series of snowballs colliding with a static object. The sticky nature of snow is simulated – snow balls are able to break apart but large chunks stick together. This data set was generated with the Chrono Physics Engine [20]. Of note is the steadily increasing number of points in the scene, as a new snowball containing $75,000$ points is created every 30 frames. Figure 6 shows the groups in three frames from the sequence.

3.2 Error

The maximum permissible error ε is the fundamental variable in the presented algorithm. A larger error results in larger group sizes and in fewer outliers as more points are classified as inliers. This increases the number of delta frames created until a new key frame and block is needed. Error also directly influences the visual quality of the result; a larger error distorts motion of some points.

Setting the error in absolute coordinates has the advantage that the permissible error can be tuned to the requirements of the display and the simulation: a quick 'preview' compression might use a larger error, while the final build compression might run against a very small error. The dimensions of the data set also influence viable range of the maximum error. For example, a maximum permissible error of 0.1 represents a relative error of less than 1% in relation to the span of the smallest data set, 'Snowball', and less than 0.1% error for the 'Galaxy' data set.

Please note that our algorithm preserves the 3D structure of the data set and enables navigation. When either zoomed in close enough or navigated in proximity to these points, a small error will be perceived larger.

	Bullet	Galaxy	Dambreak	Snowball
Frames	1000	2,500	126	694
Points (10^6)	0.04	1.4	1.3	0.08 – 1.7
Size/frame (MB)	0.5	22	50	$\sim$ 22
File size (GB)	0.5	54.0	8.0	72.0
Bounding box span	172.2	93.3	18.3	17.5

TABLE 1: The four data sets we used to test our method.

3.3 Outlier Ratio

Within a block, the outlier to inlier ratio changes for each frame. As soon as this ratio exceeds a previously set threshold, a new block is created. We consider the outlier ratio an important metric describing how well a point cloud is represented by the calculated transformations: a frame with a low ratio stores most of its points in the groups transformations with little space requirements, while a high ratio is the result of many points being stored as outliers and requiring much more space.

The initial cost of a key frame is high and the initial cost for delta frames low, as most points lie within groups and only few points are classified as outliers. As the algorithm progresses more points are moved into the outlier group and the cost of a delta frame grows. We are therefore interested at which threshold the cost of the outliers outweigh the cost of a new key frame and block. We investigated different values of cut-off by compressing all data sets with the same compression settings, only changing the maximum outlier ratio threshold and averaging the resulting values.

Outlier ratio	0.2	0.3	0.4	0.5	0.6	0.7
Delta frames	5.5	6.4	7.1	7.7	8.6	9.5
Compression rate	0.27	0.28	0.30	0.31	0.34	0.35
Run time (hrs)	4.61	2.97	2.97	3.33	3.55	3.48

TABLE 2: Influence of different outlier ratio thresholds on compression and run time. Aggregate results from all data sets.

Table 2 lists average aggregate compression values for all data sets for different outlier ratio thresholds while the other compression factors have been kept constant. Compression ratio is calculated by dividing the resulting file sizes of all blocks by the file sizes of the input frames in raw binary format. Run time was measured using internal CPU clocks. We found that for all data sets a threshold value between 0.25 and 0.35 provides the best compression ratio and processing time.

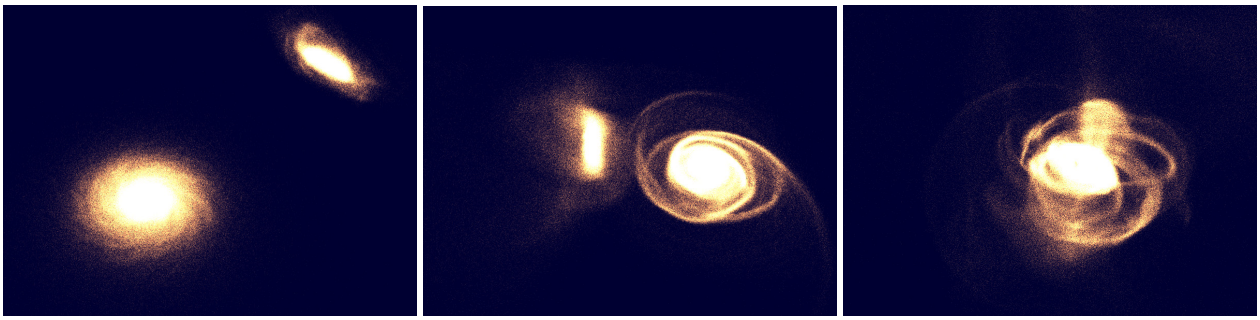


Fig. 8: Three frames in the 'Galaxy' data set rendered using additive blending for particles. Normally this data is viewed through prerendered animations. However, using our algorithm, we are able to interactively visualize this data set at rates greater than 30 frames per second.

3.4 Group Size

During the splitting phase the algorithm collects all points that are fitting the transformation to the preset error. The groups is only stored if the number of points found is above a minimum threshold. This group size influences the algorithm's behavior. A large group size results in potentially significant savings, however it requires that at least a minimum number of points can be found fitting the transformation. This number can be significantly lower during later splits which results in large number of points being classified as 'outliers'. Using a too small group size results in less savings, as the storage requirement for each transformation may take up more space than storing the points directly would.

We calculated the break-even point for space-savings on the group size as following: each transformation is stored as 12 `floats` whereas each point requires 3 `floats` as storage. We therefore break even with a group size as small as 4 – that is: as soon as more than 4 points are replaced with a group, memory has been saved. However, the group indices and sizes require extra space; a group might also be split during the gathering step of the algorithm and be recorded more than once per frame. Splitting and gathering of groups is data dependent and the number of occurrences can not be calculated in advance. We determined the optimal group size therefore through experimentation.

Group size	15	25	35	45	85	105
Delta frames	9.0	8.3	7.7	7.3	6.5	6.4
Compression rate	0.25	0.25	0.25	0.26	0.27	0.28
Run time (hrs)	14.1	6.7	5.7	7.4	6.8	6.7

TABLE 3: Influence of group size on resulting compression and file size in the 'Galaxy' data set for frames 1000–2000.

Tests of different group sizes with the 'Galaxy' data set are shown in Table 3. We found in this and other data sets that a minimum group size of 30 leads to the best compression results with a short processing time. The time required to process the whole data set grows sharply with small group sizes. Note that the group size only defines the lower boundary, if more points are found that can be expressed by the transformation found, the actual group size will be much bigger.

4 Results

In this section, we analyze our methods in terms of error, size and playback speed. For testing purposes, the parameters $\varepsilon = 0.1$, unlimited maximum iterations, max entropy of 0.35 and a minimum group size of 35 were chosen in accordance with the previous section's results.

4.1 Error

The maximum permissible error ε is an upper bound during splitting and reconstruction. Outliers decrease the overall error, as their position is unchanged from their original position. We measured the mean positional error of all points for all frames of the data sets by measuring the distance between the reconstructed and the original point. The data sets were compressed with $\varepsilon = 0.1$ and a maximum outlier ratio $outlier_{max} = 0.35$. We therefore expected to see a maximum error of

$$\varepsilon_{max} = \varepsilon \times (1 - outlier_{max}) = 0.1 \times (1 - 0.35) = 0.065.$$

Table 4 shows the mean positional errors, as well as the relative error, which is the mean error divided by the span of the data set's bounding box to give an indication of quality. All errors listed are well below the expected threshold and represent relative errors of less than 0.3% compared to the largest extend of the data set.

	Bullet	Galaxy	Dam break	Snowball
Mean positional error	0.010	0.022	0.056	0.025
Mean relative error	<0.001	<0.001	<0.003	<0.002

TABLE 4: Mean positional error for all data sets compressed at $\varepsilon = 0.1$.

4.2 Compression

We chose to compare our compression method to standard lossless data compression techniques. Single frames were compressed in sequence for comparison purposes to our method and PCL's compression. The following compression mechanisms were selected:

- **Raw** describes the tightly packed, binary, uncompressed 32-bit floating point numbers for each data set.
- **LZMA** is an improved Lempel-Ziv compression algorithm [21] and is implemented in many tools such as 7zip.
- **MG4** is a commercial LiDAR data compressor developed by LizardTech [22].
- **LAZ** (LASZip) is an open-source LAS LiDAR data compressor introduced by Isenburg et al. [6].
- **PCL** is PCL's built-in octree-based point cloud compression method for streaming, based on work by Kammerl et al. [4].

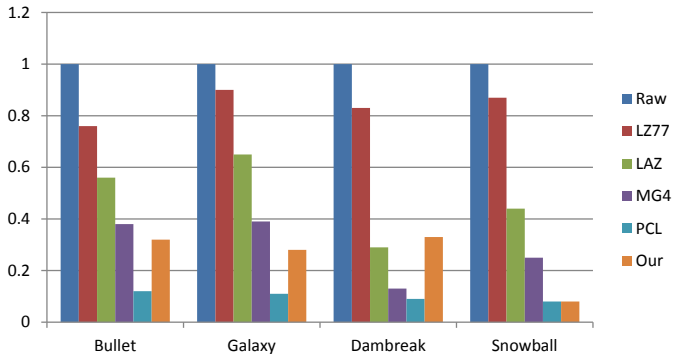


Fig. 9: Average compression ratio for the three data sets. As shown, our method is able to achieve similar compression rates compared to previous methods.

As shown in Table 5 and Figure 9, our method is able to substantially reduce the data sizes beyond what traditional lossless compression techniques can achieve.

	Galaxy	Dam break	Snowball	Bullet
Raw	40.0	2.4	7.1	0.5
LZMA	36.0 (0.90)	2.0 (0.83)	6.2 (0.87)	0.4 (0.76)
LAZ	26.1 (0.65)	0.7 (0.29)	3.1 (0.44)	0.3 (0.56)
MG4	15.6 (0.39)	0.3 (0.13)	1.8 (0.25)	0.2 (0.38)
PCL	4.3 (0.11)	0.2 (0.09)	0.6 (0.08)	0.07 (0.12)
Our method	11.0 (0.28)	0.8 (0.33)	0.6 (0.08)	0.16 (0.32)

TABLE 5: Compression of the data sets. Size is given in GB for the total file size of a data set with the compression rate relative to the raw size in parentheses.

4.3 Performance

One prime motivation to develop this algorithm was the previous inability to play back time-varying point clouds at interactive frame rates. The presented algorithm was tested in this regard by comparing the average playback speed of our method to the playback speed achieved by loading and rendering raw point clouds and the same sequence compressed by different methods. Interactive frame rates require high data throughput which requires both high read speed of files as well as low transfer times to the GPU. The former is achieved through small file sizes, whereas the latter is achieved by efficiently re-using data in our case. Point cloud data compressed with previous methods cannot be used directly on the GPU and has to be decompressed first, thereby increasing required data size and upload time.

We measure the time per frame (T_{frame}) as the sum of the time to read a frame of the sequence from disk (T_{read}), the time to decode the frame into a usable format (T_{decode}), the time to upload the data to the graphics card (T_{upload}) and finally the time it requires to render this data (T_{draw}):

$$T_{frame} = T_{read} + T_{decode} + T_{upload} + T_{draw}$$

. T_{read} is proportionate to the file-size on disk, T_{decode} is dependent on the complexity of the compression mechanism, and T_{upload} is proportionate to the amount of information being transferred to the graphics card. Table 6 shows the results, including the expected and observed frame rate for all cases. We normalized the measured times to a per-frame basis.

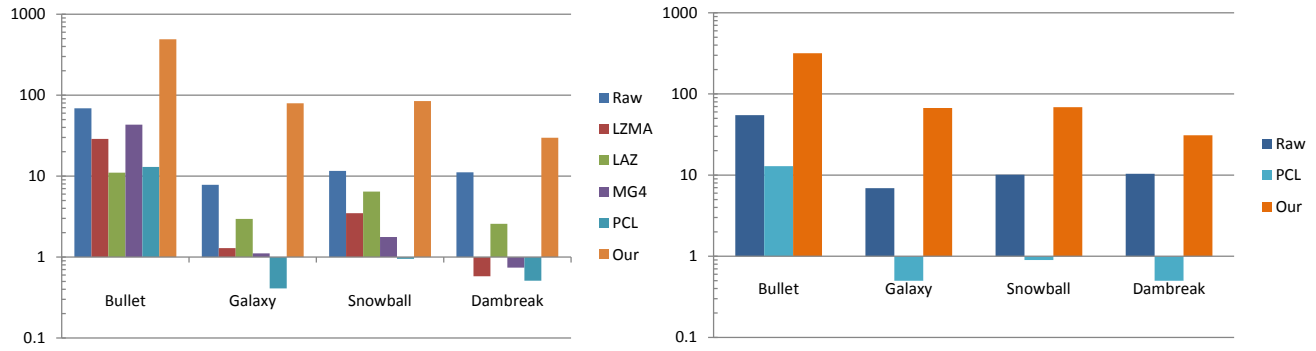
Many decompression tools exist only as external command line tools. In these cases we took measurements using `time` commands and subtracted read and write speed on the input and output files which we measured in a separate program. The OS file cache was cleared between runs. In case of these external commands, we were not able to measure GPU upload speed or draw time directly. Instead we used the values of the PCL decompression as a representative sample, as the data has to be converted into a GPU-friendly float buffer and uploaded to the GPU, a process similar for many of our other cases.

Bullet						
Method	Read	Unpack	Upload	Draw	Total	FPS (exp.)
Raw	14.2	0.0	0.3	0.1	14.6	68.7
LZMA	5.1	28.1	1.5	0.1	34.7	28.8
LAZ	5.0	84.0	1.5	0.1	90.6	11.0
MG4	1.5	20.1	1.5	0.1	23.1	43.3
PCL	0.9	74.6	1.5	0.1	77.0	13.0
Our	1.4	0.0	0.6	0.1	2.0	490.1
Galaxy						
Method	Read	Unpack	Upload	Draw	Total	FPS (exp.)
Raw	116.9	0.0	10.9	0.5	128.3	7.8
LZMA	49.0	710.0	13.1	0.6	772.5	1.3
LAZ	37.0	287.0	13.1	0.6	337.7	3.0
MG4	31.0	835.0	13.1	0.6	897.5	1.1
PCL	16.4	2,396.0	13.1	0.6	2,426.1	0.4
Our	6.8	0.0	5.6	0.2	12.6	79.4
Dambreak						
Method	Read	Unpack	Upload	Draw	Total	FPS (exp.)
Raw	80.1	0.0	9.4	0.3	89.8	11.1
LZMA	66.5	1,643.0	11.6	0.5	1,721.6	0.6
LAZ	19.5	358.0	11.6	0.5	389.6	2.6
MG4	19.5	1,311.0	11.6	0.5	1,342.6	0.7
PCL	11.9	1,926.0	11.6	0.5	1,950.3	0.5
Our	19.4	0.0	13.9	0.2	33.6	29.8
Snowball						
Method	Read	Unpack	Upload	Draw	Total	FPS (exp.)
Raw	78.7	0.0	7.3	0.3	86.4	11.6
LZMA	23.4	256.3	7.1	0.2	287.1	3.5
LAZ	17.5	130.3	7.1	0.2	155.0	6.5
MG4	13.1	544.2	7.1	0.2	564.7	1.8
PCL	9.7	1,035.9	7.1	0.2	1,052.9	0.9
Our	4.6	0.0	7.0	0.2	11.8	84.6

TABLE 6: Averaged per-frame decompression and upload performance of different methods for the all data sets.

Data extracted from the presented compression methods results in a flat point array which stores all points of the point cloud frame sequentially. We measured upload of such a 'raw' buffer to the GPU and applied this time to all decompression methods including 'raw' file reading. Our approach presents the data in a more compact form, resulting in a much lower upload time. Measure of rendering-related performance numbers is not straightforward. Modern GPUs gain much of their performance through pipelining, parallelization and bundling of instructions. Creating breakpoints to measure performance interrupts the workflow of the GPUs and introduces an additional performance loss. Lux [23] provides a good introduction measuring performance in OpenGL rendering applications using calls to the native rendering API. However, as the performance measurement is the same for both methods, it can still act as a guideline for performance comparison. This model does not take into account buffering or multi-threaded loading and decompression which can improve loading and decompression times, however it acts as a good comparison metric between methods. A lower total time results in a higher potential frame rate.

Block compression results in fewer files which in turn leads to lower read speeds, especially after per-frame normalization. Higher compression ratio of PCL results in a lower read speed, as



(a) Projected (b) Measured

Fig. 10: Projected and observed frame rates for different compression methods displayed on a logarithmic scale.

Method	Bullet	Galaxy	Snowball	Dambreak
Raw (exp.)	68.7	7.8	11.6	11.1
Raw (obs.)	54.9	6.9	10.1	10.4
PCL (exp.)	13.0	0.4	0.9	0.5
PCL (obs.)	12.9	0.5	0.9	0.5
Our (exp.)	490.1	79.4	84.6	29.8
Our (obs.)	317.0	67.1	68.6	31.0

TABLE 7: Projected and observed playback rate in frames-per-second for all data sets.

this is a function of file size. However, our method has the lowest decompression and GPU upload time. Decompression, in our case, is just the expansion of the read matrices into 4×4 matrices and the creation of a per-vertex index array, referencing the correct transformation group. No conversion of the data into flat float buffers is necessary. Figures 10a and 10b show the projected and measured fastest playback rate for all data sets.

5 Discussion

In this section we will discuss the results of the evaluation above as well as the advantages, limitations, and future work for our presented method.

5.1 Results

The data sets in this paper represent a small selection of possible data sets. We also tested our algorithm against other data sets which yielded similar results to the numbers presented here.

5.1.1 Comparison to Previous Methods

Many of the initial data sets are stored in ASCII format with additional data from their physics simulations, such as particle velocity, pressure gradient, point IDs or similar. We stripped these data sets of all the extraneous information our algorithm does not yet support and wrote out the 'raw' binary data as a large block of floating point numbers.

We compared our method to currently existing point cloud compression methods for static data. Compressing a sequence with these methods would entail compressing each frame individually. However, the major concern of most compression methods is storage space, while our goal was to improve rendering speed. We accept the trade-off of accuracy for fast decompression. However, we also noted that while most methods claim to provide 'lossless' compression, this is only true to a certain resolution after which data either gets discarded or not is reconstructed properly. Compressing and decompressing often leads to different point clouds, both in precision and in point ordering. Additionally, many of these compression methods are found in the field of geospatial images where certain assumptions about the data can be made (for example, treating it as a heightmap), which do not hold for more general point cloud data.

Although many compression methods result in very small file sizes and thus low read speed, the requirement to unpack the data before uploading it to the GPU results in a very high per-frame cost. The time required to decompress data can grow significantly with the size of the data set. As a result, reading point cloud sequences in an uncompressed form results always in a better throughput and frame rate than compressing them.

Disk read speed is another factor that influences frame rate. In our experiments we have noted an almost three-fold performance increase from switching from conventional hard drives to solid state disks. Accessing multiple smaller files (one for each frame) vs a single large file that stores many frames also bears an additional overhead, as the operating system must lock the resources. We attribute the per-frame performance increase of our method compared to the single-file compressed versions to the overhead of opening multiple frames, even though the overall file size and therefore the data read is larger in our case. Similarly, caching of files provides a significant speed increase during loading. However, due to the large amounts of data, caching is not always possible or controllable, as there are different low-level cache mechanisms built into the operating system and the hardware itself. We disabled caching as best as we could for the performance measurements.

Special consideration must be paid to PCL's compression method. While it consistently delivered the best compression rates, its performance, especially with large data sets, did not allow for interactive frame rates. Considering only play rate, not compressing the data at all would lead to better performance. We think there are two reasons for this behavior: Firstly, PCL compression works really well for small data sets, as it constructs an octree for each frame. The depth of the octree and therefore construction time primarily depends on the size of the input point cloud. However, we noticed a severe increase in construction for larger point clouds which indicates that this method is better suited for smaller data sets. For example, a depth camera with a sensor resolution of 640×480 pixels (eg the Kinect), creates at best a point cloud of only $307,200$ points – a fraction of the size of simulation data. Secondly, we believe that the data created in physics simulations is not well suited for the PCL compression

method. At its core it compares differences between octrees with the assumption that only parts of an octree change, for example having only few moving objects in a largely static scene. However, data sets such as 'Galaxy' represent millions of points which are independently moving at all times.

5.1.2 Playback

Through our initial testing, users were able to easily play, rewind and annotate time-varying point clouds inside of an immersive display environment as shown in Figures 1 and 7. Playback speed is a major factor in understanding time-varying data sets. We found that data sets played back at only 2-3 frames a second lose coherency and the viewer has trouble following the general motion of the points. This is exacerbated by non-uniform playback speeds. We achieve high frame rates on desktop machines also by employing solid state hard drives. However, it is often not easy or possible to change the hardware of a large immersive VR system, such as a CAVE. In our system, the files are read from standard hard drives on shared servers which are in addition accessed over ethernet from the render nodes. We therefore attain a much lower performance than what is possible on the desktop, as seen in the frame rate counter in Figure 7. However, we tested our algorithm with both solid state drives and conventional hard drives and found that while the solid state drives provide a three-fold read speed improvement our algorithm still provided a larger playback speed increased on standard hard drives compared to playing back raw files from an SSD. Hardware upgrades alone therefore do not solve the initial problem.

5.1.3 Discontinuity

One disadvantage of the presented method lies in the possibility of creating discontinuities between blocks. The algorithm exhibits the following behavior if individual frames or the block structure and compared to their original counterparts which were used as input: the first frame of the block will map directly onto the key frame points with an identity transformation, thereby exhibiting no error. However, following frames will map the key frame point clouds using estimated transformations and the frame's overall error will grow with the error of the individual groups. Once a group's error becomes too large, the group will be converted into an outlier group, its points will be stored directly and conversely, the error will shrink. However, the overall error will always increase while staying well below the maximum error threshold set by the user, as seen in Table 4.

A discontinuity can be detected between two blocks when the last frame of the current block does not map without a noticeable error onto the first frame of the next block. This is true both for static and moving points but more visible in the former. Figure 11 shows this behavior over multiple consecutive blocks. Note that the mean error (in blue) at first increases within a block before continuosly dropping. At the same time, more and more points are classified as outliers (in red) and are stored directly, thereby lowering the mean error of a block.

While there is still a discontinuity, as the error of all points is not 0, it is barely noticeable in point clouds in which all particles are in motion however it can manifest itself in scenes in which large parts are stationary. These stationary areas seem to jitter or jump slightly between two consecutive blocks. As the error between two blocks decreases with the number of outliers present, ease-in interpolation is naturally achieved as the number of outlier points stored in the last couple of frames in a block is increased

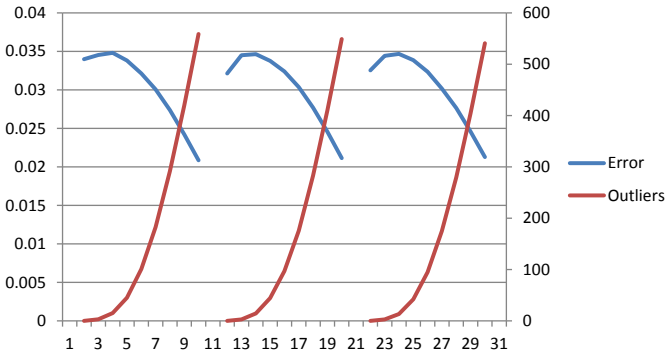


Fig. 11: The mean aggregate error and the number of outlier points over three consecutive blocks.

at the cost of space saving. While the discontinuity between two blocks is a fundamental problem of the presented is approach, we do not believe it to be compromising the basic idea of achieving compression from tracking motion groups.

5.2 Advantages

The presented method has a number of advantages over previous presentation and compression methods. First, we store true 3D data, as opposed to movies rendered from a fixed perspective, thus our algorithm lets the user explore and interact with the data.

Second, the algorithm enables the user to choose an error rate in absolute coordinates. This is a more direct form of quality control than the 'quality percentage' sliders found on many compression methods. The absolute error can be tuned to conform to the error bounds of the bounding box or the physical simulation, therefore presenting a true representation of the data within the error limits. We note that the calculated error was smaller than the user set maximum error for all cases.

Third, the underlying data representation is not based on quantization and therefore allows a high degree of accuracy while preserving the appearance of uniform sampling. However, we note that the description of a motion group is in effect the description of the bounding box's motion of a small subset of point within the initial keyframe. As such, point cloud compression methods, such as octree compression can be applied to the key frame or the outlier point data.

Fourth, decompression of data is trivial and upload to the GPU is very low. While the initial frame bears the highest cost, it is quickly amortized over the run of multiple frames within a block. Previous approaches achieve high compression rates but require a costly decompression and data conversion step for each frame, thus reducing possible frame rates significantly.

Finally, our method of storing delta frames in blocks results in a very robust data storage. Each frame and its groups depends only on the key frame but not on preceding frames. This allows us to play back the data in both directions. Delta frames can also be dropped (for example, during transmission) without influencing other block data or compromising image quality of the remaining animation sequence as long as the initial point cloud is unchanged.

5.3 Limitations

This work on the algorithm lays a foundation onto which future extensions and improvement can be built. We acknowledge the following limitations of our method in its current state: first, we

are currently unable to store color or any per-point per-frame changing data. While our method is able to handle colors (for example, for annotations), these colors are assigned per block, not per frame. Similarly, the reordering of the points, which enables significant space savings, also interferes with copying per-point attributes from block to block; we therefore have to introduce the redirect block at a fixed cost which stores the original point order and enables a mapping from block-to-block.

Second, while the algorithm enables less data to uploaded to the graphics card over a series of frames, the initial upload data requirement is much greater. Furthermore, as the data is structured for motion groups as opposed to being structured spatially, the entire point cloud is naively rendered every frame, as the bounding boxes of motion groups often span the entire data set, as points within a motion group are not spatially coherent. This can be problematic when dealing with limited hardware. The addition of spatial data structures could help in both the rendering and annotation components for interactive viewing.

Third, using RANSAC to find common features or models is time-consuming and, given the greedy nature of the algorithm, may not yield an optimal solution. The heavy reliance on random sampling also makes the compression non-deterministic for a given input cloud.

Finally, while our algorithm allows users to specify the maximum allowable error, this in turn requires the user to have an spatial understanding of their data set. An acceptable error for a simulation of galaxies would likely be unacceptable for a simulation of molecules. These limitations motivate future research directions, as outlined below.

5.4 Future Work

We believe this algorithm lays the foundation for future expansions of this work, including the support of per-point colors, real-time capture and compression of point clouds and refinements to the compression method. There is also potential to use this method for transient feature detection in animated data sets.For example, Figure 2 shows a disturbance in the otherwise symmetrically motion groups of the 'Galaxy' data set. The presence of the second galaxy (not seen in this figure) and its gravitational influence disturbs the motion of these particles in this part of the data set enough that they are assigned to different motion groups.

5.4.1 Quality Metrics

The methods presented in this paper aim to create an algorithm which is lossy with a user-definable lower error bound. One of the reasons for choosing this approach is due to the lack of clarity of how loss of information will be perceived.

For example, all of the point information is stored in full 32-bits in our current approach. Reducing the bit depth, for instance to a 16-bit float value, cuts the largest storage requirement in our method by half. In testing we have found that the 'Galaxy' data set can be reduced to approximately 5 GB when using 16-bit float values and removing the transition index block, compared to the 11 GB in the standard compression or the 40 GB of initial raw float data. However, as the ramifications of this bit reduction are still unknown, we have chosen to only present the findings from our current approach using 32-bit floats.

Also, while the presented algorithm enables the user specify an absolute error, it is not clear how this error will be perceived. Research into the visual perception of error in the data set could help determine maximum and optimal settings for compression. An important factor to this error perception is also the role outliers and groups play and what the visual impact is. Future work will aim to enable the user to define quality metrics (such as seen in image and video encoding). The goal of this work will be to enable a 'maximum permissible error' to achieve maximum compression of the data set given a quality setting. Some video and audio compression methods work similarly by relying on a perceptual model (for example, psychoacoustics for audio) in which the less noticeable errors are removed.

5.4.2 Transformation Detection and Transition Indices

In the split phase we use transformation estimation methods, such as pose estimation, to calculate the transformation matrix between two time steps. These methods usually contain constraints relevant for the application – for example, pose estimation assumes rigid body transformations without scaling. However, we do not require these constraints for the transformation description as long as a valid 4×4 transformation matrix is created. For example, it would be possible, although inefficient, to create this matrix using a random number generator as the RANSAC approach will guarantee that only the best-suited matrix is chosen.

One large, although fixed, cost for each frame comes from the transition index block. The algorithm changes the order of the points within the point clouds to build the groups. However, to maintain coherence between blocks for interpolation or annotation purposes, the original point order has to be preserved. We do so by storing the original point's position in the transition index block which can be used to relate the points in one block to another.

We believe that a future research direction could include replacing the transition index block by a just-in-time evaluation of blocks and the creation of such a block. This is related to the problem of transform estimation between two point sets but must also include a time-based predictive step, as two consecutive frames – the last frame of the current and the first frame of the next block – are two separate steps in the animation and not the same point cloud.

6 Conclusion

This article introduces a novel compression method for time-varying point cloud data. A high compression ratio is achieved by tracking and describing group motion. This results in a significant decrease in disk and memory usage. The data layout is in addition optimized for rendering with little to no decompression required which in turn improves playback performance. The spatial structure of point clouds is preserved which allows the immersive exploration at interactive frame rates and interaction methods such as tagging.

It is important to note that his method does not try to achieve maximum compression but rather tries to maximize playback performance. Therefore it should be rather viewed as a 'movie codec' compression for point cloud sequences than a compression method used for archiving purposes.

Future work will extend this algorithm to support user-defined quality metrics and support more general, unstructured, time-varying point cloud data structures such as gathered from 3D camera.

ACKNOWLEDGMENTS

The authors would like to thank Ross Tredinnick and Patricia F. Brennan for their support on this project. This project was supported by grant number R01HS022548 from the Agency for Healthcare Research and Quality. The content is solely the responsibility of the authors and does not necessarily represent the official views of the Agency for Healthcare Research and Quality.

REFERENCES

[1] R. Schnabel, S. Möser, and R. Klein, "A parallelly decodeable compression scheme for efficient point-cloud rendering." in *SPBG*. Citeseer, 2007, pp. 119–128.
[2] R. Schnabel and R. Klein, "Octree-based point-cloud compression." in *SPBG*, 2006, pp. 111–120.
[3] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–4.
[4] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach, "Real-time compression of point cloud streams," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 778–785.
[5] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of predicted floating-point geometry," *Computer-Aided Design*, vol. 37, no. 8, pp. 869–877, 2005.
[6] M. Isenburg, "Laszip," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, pp. 209–217, 2013.
[7] D. Mongus and B. Žalik, "Efficient method for lossless lidar data compression," *International journal of remote sensing*, vol. 32, no. 9, pp. 2507–2518, 2011.
[8] K. Zhao, J. Nishimura, N. Sakamoto, and K. Koyamada, "A new framework for visualizing a time-varying unstructured grid dataset with pbvr," in *Advanced Methods, Techniques, and Applications in Modeling and Simulation*. Springer, 2012, pp. 506–516.
[9] X. Gu, S. J. Gortler, and H. Hoppe, "Geometry images," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 355–361.
[10] M. Alexa and W. Müller, "Representing animations by principal components," in *Computer Graphics Forum*, vol. 19, no. 3. Wiley Online Library, 2000, pp. 411–418.
[11] H. M. Briceño, P. V. Sander, L. McMillan, S. Gortler, and H. Hoppe, "Geometry videos: a new representation for 3d animations," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 2003, pp. 136–146.
[12] S.-Y. Kim and Y.-S. Ho, "Mesh-based depth coding for 3d video using hierarchical decomposition of depth maps," in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 5. IEEE, 2007, pp. V–117.
[13] T. Matsuyama, S. Nobuhara, T. Takai, and T. Tung, "3d video encoding," in *3D Video and Its Applications*. Springer, 2012, pp. 315–341.
[14] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.
[15] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.
[16] E. Coumans *et al.*, "Bullet physics library," bulletphysics.org, online; accessed: 16-February 2015.
[17] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, 2005.
[18] V. Springel, N. Yoshida, and S. D. White, "Gadget: a code for collisionless and gasdynamical cosmological simulations," *New Astronomy*, vol. 6, no. 2, pp. 79–117, 2001.
[19] X. Y. Ni and W. B. Feng, "Numerical simulation of wave overtopping based on dualsphysics," *Applied Mechanics and Materials*, vol. 405, pp. 1463–1471, 2013.
[20] T. Heyn, H. Mazhar, A. Pazouki, D. Melanz, A. Seidl, J. Madsen, A. Bartholomew, D. Negrut, D. Lamb, and A. Tasora, "Chrono: A parallel physics library for rigid-body, flexible-body, and fluid dynamics," in *ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers, 2013, pp. V07BT10A050–V07BT10A050.
[21] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.
[22] "Lizardtech lidar compression," www.lizardtech.com, online; accessed 16-February 2015.
[23] C. Lux, "The opengl timer query," in *OpenGL Insights*, C. Patrick and C. Riccio, Eds. CRC Press, 2012.

Markus Broecker received his Dipl.-Inf in Computer Visualistics from the University of Koblenz in 2009 and his PhD from the University of South Australia in 2013. He is currently as postdoc at the Living Environments Lab at the University of Wisconsin-Madison. His research interests include Virtual and Augmented Reality and point cloud rendering methods.

Kevin Ponto received a B.S. degree in computer engineering from the University of Wisconsin-Madison, a M.S. degree from the Arts Computation Engineering program at the University of California, Irvine and a Ph.D. in computer Science from the University of California, San Diego. He is currently an Assistant Professor at the University of Wisconsin - Madison, jointly appointed between the Living Environments Laboratory at the Wisconsin Institutes for Discovery and the Design Studies Department in the School of Human Ecology. He also has affiliate appointments in the Department of Computer Sciences and the Arts Institute. His research interests include Virtual and Augmented Reality and Wearable Computing.