Methods for the Interactive Analysis and Playback of Large Body Simulations

Markus Broecker
Living Environments Laboratory
Wisconsin Institutes for Discovery
Room B1142, 330 N Orchard St.
University of Wisconsin-Madison
Madison, WI 53715
Phone: 608-316-4690
broecker@wisc.edu

Kevin Ponto
Living Environments Laboratory
Wisconsin Institutes for Discovery
Room 3176, 330 N Orchard St.
University of Wisconsin-Madison
Madison, WI 53715
Phone: 608-316-4330
kbponto@wisc.edu

***Abstract*— Simulations, such as n-body or smoothed particle hydrodynamics, create large amounts of time variant data that can generally only be visualized non-interactively by rendering the resulting interaction into movies from a fixed viewpoint. This is due to the large amount of unstructured data that make these data incapable of either loading into graphics memory or streaming from disk. The presented approach aims to preserve the interactively navigable 3D structure of the underlying data by reordering and compressing the data in such a way as to optimize playback rates. Additionally, the presented method is able to detect and describe coherent group motion between frames which might be used for analysis.**

TABLE OF CONTENTS

1. INTRODUCTION

Time-variant three-dimensional point clouds are a rich source of information that can be explored, annotated and interacted with in virtual reality environments. While physical simulations, for example n-body or smoothed particle hydrodynamics (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 decompression cost.

Figure 1: A user navigating the 'Galaxy' data set in the CAVE. This data set contains 1.5 millions of individual particles moving through 2,500 frames.

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 7). Therefore, in order to enable real-time playback, for example in an immersive VR environment, such as a CAVE (Figure 1), the data must be compressed in a GPU-friendly manner.

2. 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. A RANSAC-based shape detection method segments the initial point cloud into distinct geometric shapes which can be efficiently encoded using only a few parameters. Additional detail is achieved by storing

978-1-4673-7676-1/16/$31.00

point offsets as heightmaps with different levels of detail. Compression of this texture data is achieved through vector quantization [1]. 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 work. 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.

The point cloud library (PCL) [3] offers a built-in lossy compression mechanism targeted towards streaming point cloud data from sensor devices 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 7 discusses this method in detail and contrasts it with our approach.

Lengyel introduced the idea of compressing time-variant vertex data based clustering based on a description of clustered transformations between frames [5]. Clusters are created based on local proximity. The vertex-based approach is easily extendable to point cloud data sets. However, his approach requires solving and optimizing the whole animation sequence for all vertices over all frames at the same time which is infeasible for larger data sets.

Compression methods for time-varying data from physical simulations usually consider only volumetric data. Previous work extended the concept of octrees into the temporal dimension [6], [7]. The initial data is first quantized and separate octrees are built for each time step. A second step collects all created octrees; sub-trees of multiple time steps are re-used if no or only little change was detected thus compressing the data. Additionally, rendering over multiple frames uses compositing to reuse previously rendered views for parts of the data set that are unchanging over multiple time steps. Our method is designed around unstructured point cloud data.

Following the idea of representing geometry through texture images [8] 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".

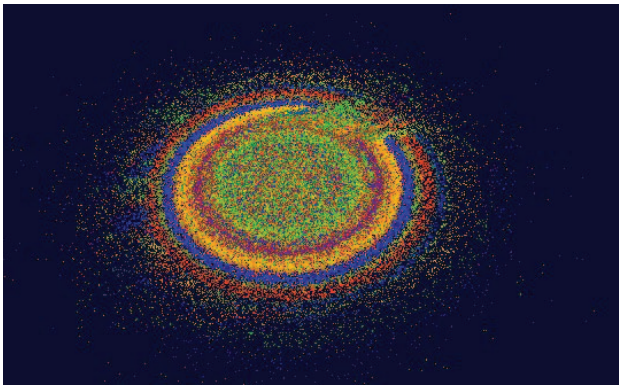

Figure 2: Shows detected groups, each with a unique color, in the 'Galaxy' data set. As shown, the algorithm is able to detect rings of similar motion for the rotating cluster of stars.

Alexa and Müller proposed using Principal Component Analysis along the temporal axis as a means of data compression [9] while Lengyel and Briceño et al. utilized prediction methods of projected 3D surface data [10]. Similarly, another approach is to store positional information as color channels and utilize existing video compression software [11], [12]. 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. Movie compression is optimized for fast decompression speed but 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 (few high-dynamic range movie compressors exist). 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 introduced when converting between color spaces which are often furthermore built on principles of human perception and not linear representation. Applying a movie compression scheme on a quantized data set therefore changes the initial data set beyond what can be accepted as 'lossy compression'.

Given the previous work, our goals are to

1. **Detect transient motion Groups for data analysis:** This novel kind of clustering enables efficient representation of common group motion within data sets. This group motion can be exploited for effective compression and initial visual analysis of data sets. Figure 2 shows some detected motion groups in a data set of a simulated galaxy simulation.
2. **Enable interactive bi-directional playback:** A block structure with independent frames allows the effective reconstruction of points at any point in the data stream, thereby allowing the user to play back animations and data sets both forwards and backwards which is different from the current video compression methods in which a frame is calculated from a key frame and a uni-directional sequence of changes to it.
3. **Enable user defined annotations:** We define meaningful interaction with the data set as both interactive exploration as well as annotation of the data, which is currently impossible with standard videos.

We employ a two-stage algorithm to do so. The first stage is the *split phase* in which common motion groups are detected

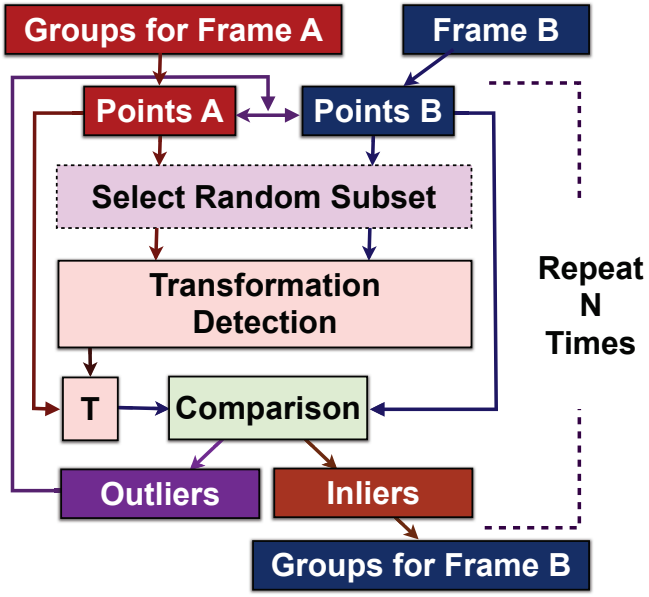

Figure 3: The algorithm.

between frames and followed over a sequence of multiple frames. The algorithm recursively splits the motion groups in the temporal domain, creating a tree-like hierarchy to best describe the motion of individual particles over frames.

The second stage is the *gather phase* in which the original data set is re-ordered based on the detected transformation hierarchy. This operation effectively compresses the data set while at the same time optimizing it for GPU transfer.

3. Motion Groups

The 'split phase' detects common motion groups in the data set between consecutive frames. Given a sequence $(F_0, F_1, F_2, \ldots, F_n)$ of point cloud frames that are registered, the algorithm finds the best clustering which describes motion groups from F_0 to F_n. The input point clouds are index-matched, that is the $n-th$ point is 'the same' over all frames.

Detection Motion Between Two Frames

We assume that the motion of a single particle is coherent (and not jerky) through multiple frames. We also assume that many particles from one frame have a similar linear motion that results int their positions in the next frame and similarly from the second to the third frame and so on. However, we do not impose spatial or localized coherence on this motion. Linear motion between positions can be described effectively using a transformation matrix that maps the input positions onto the output positions. Points with the same transformation are grouped into motion groups which contains the points as well as the corresponding matrix.

A greedy algorithm is used to create motion groups between two frames. A method similar to the Iterative Closest Point algorithm [13] is used to detect a transformation between the points of the first and second frame that minimizes the transformation error (over all points). If a rigid motion between frames is assumed with no scaling, an SVD-based 3D pose (or camera orientation) estimation method can be used to determine this transformation. A fixed number of randomly selected points in the input and output point cloud are used to determine the transformation matrix which is applied to all input points. Each point so transformed is then classified as either an inlier or an outlier, based on a user-chosen threshold and the linear distance between transformed point and original input.

The motion estimation coupled with the inlier/outlier segmentation is encapsulated in a RANSAC loop. Multiple instances of the ICP detector are run in each iteration; the best result is chosen according to the number of inliers and overall lowest error. A motion group is then created, storing the detected matrix and the corresponding points which are removed from the input point cloud. This algorithm is repeated until either the input set becomes empty or a fixed number of iterations have passed in which the input set did not get reduced. Figure 3 shows the algorithm in detail.

Motion Over Multiple Frames

The algorithm splits each frame into a number of motion groups and outlier groups. Each of those motion groups will have a parent motion group in a previous frame. The initial group used to initialize the algorithm contains all points of the first frame as well as the identity transformation.

Multiple consecutive frames with their motion group trees and outliers are grouped into separate blocks, each block storing all information necessary to play back this sequence of frames. The ratio between inlier and outlier points defines a user-selectable cutoff when a new block is started, as this ratio indirectly controls the possible compression rate of the data. Figure 4 shows on the left the transformation tree built from motion groups and outlier nodes. The tree's nodes are sorted, motion groups are inserted at the top while outlier nodes are always inserted at the bottom thus creating a clear separation each frame.

While this method was originally designed to handle time-variant point cloud data sets of fixed size, it is easy to extend it to variable size data sets, such as the 'snowball' data set. In this data set, the criterion to create a new block and motion group tree was the creation of a new snowball and associated points every 30 seconds. More chaotic point creation/destruction can be handled by marking these points as outliers.

Reconstruction

If a point lies within a motion group, it can be reconstructed by following the transformation chain of their group to their unique initial point:

$$p_n = T_n \times T_{n-1} \times \ldots \times T_1 \times p_0.$$

This transformation chain is usually stored pre-multiplied in each motion group (as it is unchanging) and enables quick reconstruction of points. This independence of each group within a frame and withing the sequence of multiple frames makes the representation more versatile than video-codec inspired compression methods, as it allows seeking in both direction and each frame is independent of the previous one. If a point was deemed an outlier at any stage of the algorithm, it is stored directly with XYZ coordinates; reconstruction is not necessary.

4. COMPRESSION

Compression is achieved through reordering of the point cloud based on the hierarchical transformation tree (see Figure 4). We call this reordering the 'gather phase' of the algorithm.

As points in a motion group T_n in frame $2 \dots m$ can be reconstructed from the initial point and the groups transformation, the only information needed for reconstruction is the initial point start index, the size of the group and the accumulated transformation matrix of this group. Outliers are stored directly. A large number of outliers per frame therefore decreases compression efficiency. The presented method affords a high decompression speed by reordering the initial data set and inferring point positions by through their transformation chain and initial point.

Redirection List

Compression of the data set reorders the initial point cloud in the first frame of each block. However, texturing and indexing requires consistent indices between blocks. An optional redirection list within each block achieves this by storing the original indices for each point as a single integer. This list is not needed for pure playback but only if coloring and selection is wanted. As the number of points do not change between frames/blocks this list of a fixed size and cost.

5. ANALYSIS

We investigated three data sets created from physics simulations, which are summarized in Table 1. File size is the initial size of the data 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. Note that while the file size of the data sets is below some of the larger simulation data sets, it is still too big to fully load into GPU memory simultaneously for the whole data set.

The 'Galaxy' data set displays the collision of two galaxies and was generated from GADGET cosmological simulations [14]. 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 5 shows three different frames at the beginning, the middle and towards 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 [15] and can be seen in Figure 6. 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 [16]. 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 7 shows the groups in three frames from the sequence.

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

	Galaxy	Dambreak	Snowball
Frames	2,500	126	694
Points (10^6)	1.4	1.3	0.08 – 1.7
Size/frame (MB)	22	50	~ 22
File size (GB)	54.0	8.0	72.0
Bounding box span	93.3	18.3	17.5
1% error	0.93	0.183	0.175

6. DISPLAY

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 motion groups 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, storing all motion group's matrices, 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 non-solid particle objects, liquids or gases. In these cases, the point sprites are alpha-blended. Rendering of liquids could also be implemented using runtime surface reconstruction methods, such as marching cubes.

Interaction and Selection

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.

Figure 6 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.

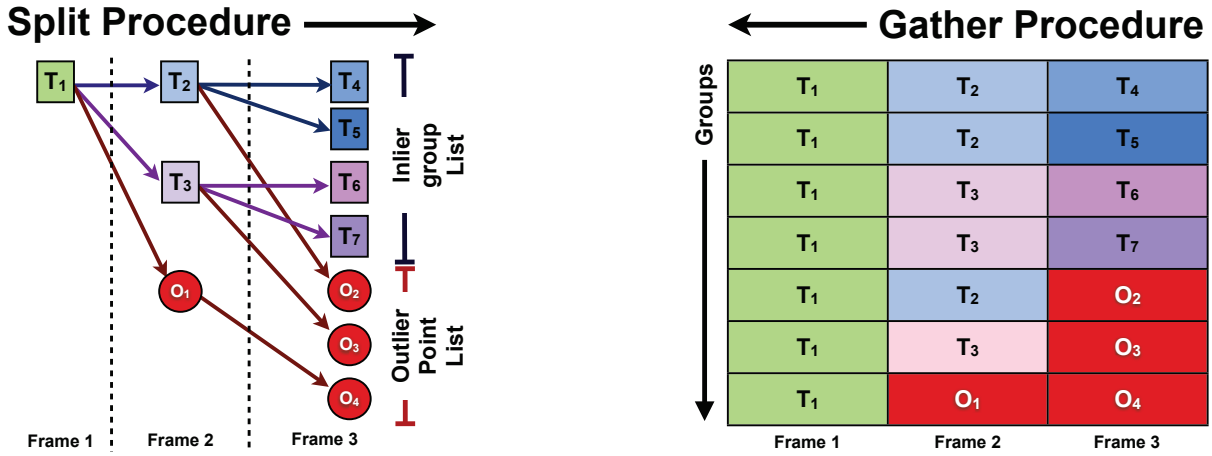

Figure 4: Splitting the point cloud into motion groups in the 'Split Phase' and reordering the initial points based on the hierarchy to efficiently save space in the 'Gather Phase'.

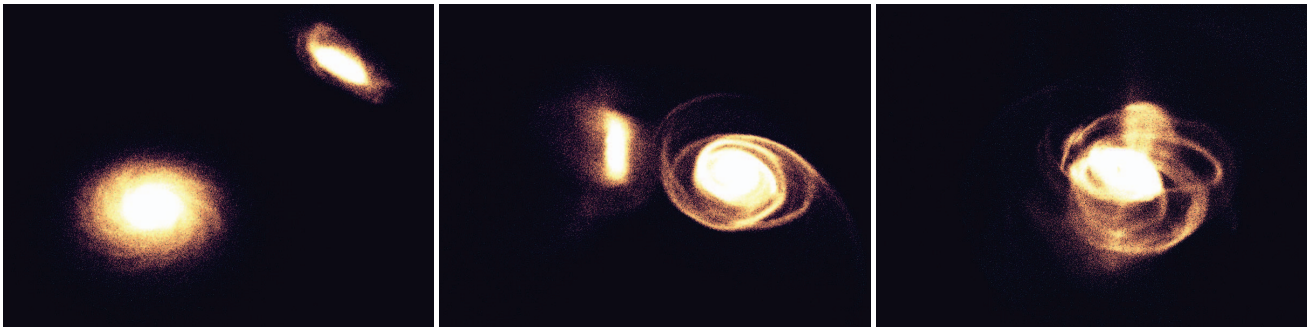

Figure 5: Three frames in the 'Galaxy' data set rendered using additive blending for particles. Normally this data is viewed through prerendered animations. However, using our algorithm, we are able to interactively visualize this data set at rates greater than 30 frames per second.

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 thread is used for file loading and keeping the buffer filled.

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.

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. The tools were implemented on a standard desktop PC featuring with a quadcore Intel I7 CPU, 16GB of RAM an NVidia GTX 750 GPU with 2GB of VRAM and a Samsung SSD. Large parts of the algorithm can be parallelized, for example transform detection, inlier counting, node splitting or selection which we did using boost threads and OpenMP.

7. Results

In this section, we analyze our methods in terms of error, compression and decompression/playback speed on the three data sets introduced in Table 1. In particular, we were interested in the actually achieved error, the compressed size of the data sets, decompression speed and resulting display rate.

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 $\epsilon = 0.1$ and which is less than a 1% relative positional error, as seen in Table 1. The maximum outlier ratio before a new block was started was set to 0.35 which was determined to be optimal for these data sets through empirical analysis.

With an absolute error of $\epsilon = 0.1$ we expected to see a maximum actual error of

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

Table 2 shows the mean positional errors, as well as the relative error, which is the mean error divided by the span of

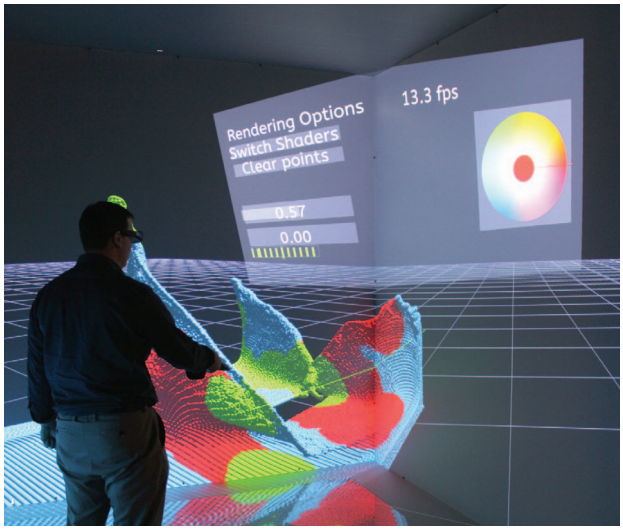

Figure 6: A user annotating the dam break data set in the CAVE.

the data set's bounding box to give an indication of quality. The data sets were compressed with a maximumg error threshold of $\epsilon = 0.1$. 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.

Table 2: Mean positional error.

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

Compression

We chose to compare our compression method to standard 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 [17] and is implemented in many tools such as 7zip.
MG4 is a commercial LiDAR data compressor developed by LizardTech [18].
LAZ (LASZip) is an open-source LAS LiDAR data compressor introduced by Isenburg et al.[19].
PCL is PCL's built-in octree-based point cloud compression method for streaming, based on work by Kammerl et al. [4].

As shown in Table 3 and Figure 8, our method is able to substantially reduce the data sizes beyond what traditional lossless compression techniques can achieve. In Table 3, the compressed size of the data sets are given in GB with the compression rate relative to the raw size in parentheses.

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

Table 3: Compression of the data sets.

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

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. Table 4 shows the averaged per-frame decompression and upload performance of different methods for all decompression methods; all times are in milliseconds.

Table 4: Per-frame cost details.

Galaxy					
Method	Read	Unpack	Upload	Draw	Total
Raw	116.9	0.0	10.9	0.5	128.3
LZMA	49.0	710.0	13.1	0.6	772.5
LAZ	37.0	287.0	13.1	0.6	337.7
MG4	31.0	835.0	13.1	0.6	897.5
PCL	16.4	2,396.0	13.1	0.6	2,426.1
Our	6.8	0.0	5.6	0.2	12.6
Dambreak					
Method	Read	Unpack	Upload	Draw	Total
Raw	80.1	0.0	9.4	0.3	89.8
LZMA	66.5	1,643.0	11.6	0.5	1,721.6
LAZ	19.5	358.0	11.6	0.5	389.6
MG4	19.5	1,311.0	11.6	0.5	1,342.6
PCL	11.9	1,926.0	11.6	0.5	1,950.3
Our	19.4	0.0	13.9	0.2	33.6
Snowball					
Method	Read	Unpack	Upload	Draw	Total
Raw	78.7	0.0	7.3	0.3	86.4
LZMA	23.4	256.3	7.1	0.2	287.1
LAZ	17.5	130.3	7.1	0.2	155.0
MG4	13.1	544.2	7.1	0.2	564.7
PCL	9.7	1,035.9	7.1	0.2	1,052.9
Our	4.6	0.0	7.0	0.2	11.8

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.

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

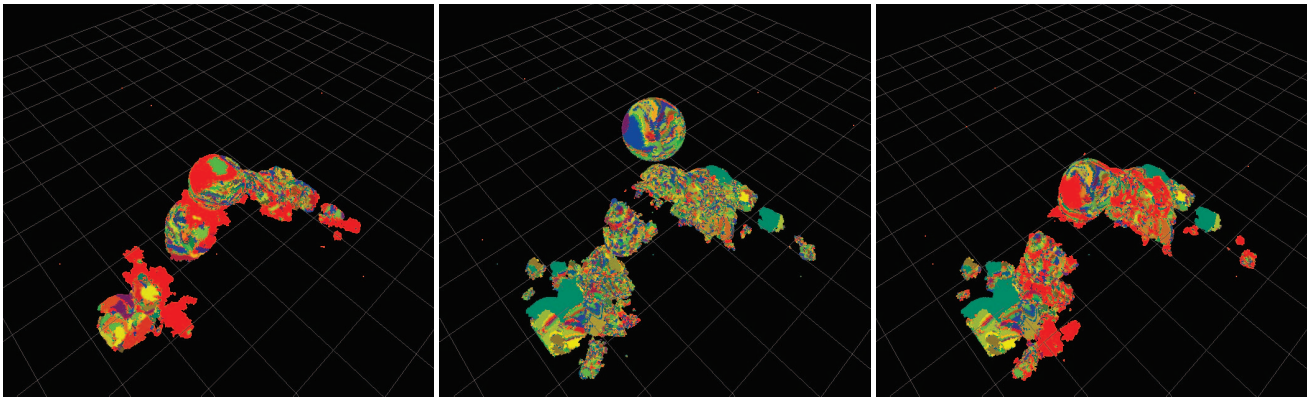

Figure 7: Three frames in the 'Snowball' data set. Each group is depicted with a unique a random color while outliers are colored red. The algorithm detects large groups of particles with similar motion, outliers in one block can be assigned to groups in later frames.

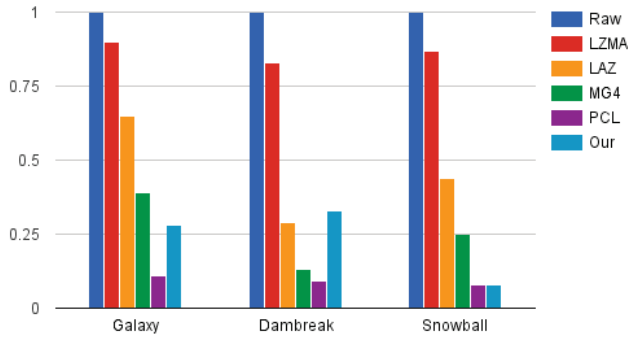

Figure 8: Average compression ratio for the three data sets. As shown, our method is able to achieve similar compression rates compared to previous methods.

approach presents the data in a more compact form, resulting in a much lower upload time. Measure of rendering-related performance numbers is not straight-forward. 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 [20] 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.

Table 5: Playback rate in fps.

Method	Galaxy	Snowball	Dambreak
Raw (exp.)	7.8	11.6	11.1
Raw (obs.)	6.9	10.1	10.4
PCL (exp.)	0.4	0.9	0.5
PCL (obs.)	0.5	0.9	0.5
Our (exp.)	79.4	84.6	29.8
Our (obs.)	67.1	68.6	31.0

Block compression results in fewer files which in turn leads to a reduced file read time, especially after per-frame normalization. Higher compression ratio of PCL results in a lower read time as well, as the time required to read a file is a function of file size. However, our method has no actual decompression requirements. 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. Furthermore, we are able to optimize disk storage to make it highly GPU-upload friendly which is lacking. Table 5 shows the projected and observed playback rates for these data sets. The projected value is derived from the measured performance values in Table 4. The measured playback rate investigates included the whole pipeline: from reading the file from disk to rendering an image on screen.

8. 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.

Results

The data sets in this paper represent a small selection of possible data sets. We also tested our algorithm against other data sets, such as animated mesh vertex and motion capture data, which yielded similar results to the numbers presented here.

Comparison to Previous Methods—We compared our method to currently existing point cloud compression methods for static data. 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.

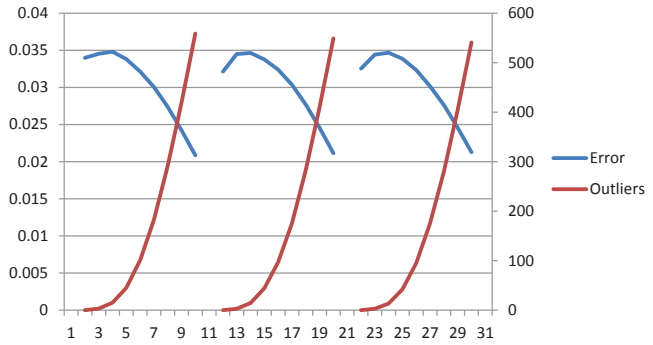

Figure 9: The mean aggregate error and the number of outlier points over three consecutive blocks.

One could make the argument that streaming raw data using a multi-threaded buffering would yield similar results to our method. However, the application is ultimately bounded by file read times and decompression speed. If the time it takes to read and decompress a file is longer than the time it takes to display that frame (eg 33ms for a 30Hz refresh rate) the buffer will run empty. A small file read and decompression time of our method leads to smaller buffers which also require less runtime memory.

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 2.

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

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. Quantization, as observed in other compression methods, is especially noticeable during animated sequences. As points can only take up discrete positions, any appearance of individual motion is quickly lost in dense data sets. 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.

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, the current method focuses on vertex positions and we are currently unable to store color or any per-point per-frame changing data in the blocks themselves. The reordering of the points, which enables significant space savings, 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. Once this is established, however, is is possible to use these indices as unique texture coordinates into textures which can be compressed using video codecs (see below).

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.

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, but compare it to Figure 5, left) 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.

Colors—The optional per-block redirection list affords consistent indexing of each point and can act as texture coordinate for each point. Per-frame color textures can become large and expensive; we therefore suggest the use of movie compression on these textures. The lossy nature of these compressors, while detrimental to geometric data, is not necessarily noticeable for color data as most compressors are build around models of perception.

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. 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 motion groups play and what the visual impact is (eg are groups or outliers the defining feature of data sets). 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 conversion) in which the less noticeable errors are removed.

Transformation Detection—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.

9. 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 viewed as a 'movie codec' compression for point cloud sequences rather 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 cameras or other depth sensors.

Acknowledgments

Support for this research was provided by the University of Wisconsin–Madison, Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation.

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] J. E. Lengyel, "Compression of time-dependent geometry," in *Proceedings of the 1999 symposium on Interactive 3D graphics*. ACM, 1999, pp. 89–95.

[6] K.-L. Ma and H.-W. Shen, "Compression and accelerated rendering of time-varying volume data," in *Proceedings of the 2000 International Computer Symposium-Workshop on Computer Graphics and Vir-*

tual Reality, 2000, pp. 82–89.

[7] H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree," in *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, ser. VIS '99. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 371–377. [Online]. Available: http://dl.acm.org/citation.cfm?id=319351.319434

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] T. Matsuyama, S. Nobuhara, T. Takai, and T. Tung, "3d video encoding," in *3D Video and Its Applications*. Springer, 2012, pp. 315–341.

[13] P. Besl and N. D. McKay, "A method for registration of 3-d shapes," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 14, no. 2, pp. 239–256, Feb 1992.

[14] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, 2005.

[15] 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.

[16] 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.

[17] 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.

[18] "Lizardtech lidar compression," www.lizardtech.com, online; accessed 16-February 2015.

[19] M. Isenburg, "Laszip," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, pp. 209–217, 2013.

[20] C. Lux, "The opengl timer query," in *OpenGL Insights*, C. Patrick and C. Riccio, Eds. CRC Press, 2012.

BIOGRAPHY

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 works currently as an assistant scientist 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.*